The Idris Reference

Release 0.9.19.1

The Idris Community

January 31, 2016

Contents

This is the reference guide for the Idris Language. It documents the language specification and internals. This will tell you how Idris works, for using it you should read the Idris Tutorial.

CHAPTER 1

Documenting Idris Code

Idris documentation comes in two major forms: comments, which exist for a reader's edification and are ignored by the compiler, and inline API documentation, which the compiler parses and stores for future reference. To consult the documentation for a declaration `f`, write `:doc f` at the REPL or use the appropriate command in your editor (`C-c C-d` in Emacs, `<LocalLeader>h` in Vim).

1.1 Comments

Use comments to explain why code is written the way that it is. Idris's comment syntax is the same as that of Haskell: lines beginning with `--` are comments, and regions bracketed by `{-` and `-}` are comments even if they extend across multiple lines. These can be used to comment out lines of code or provide simple documentation for the readers of Idris code.

1.2 Inline Documentation

Idris also supports a comprehensive and rich inline syntax for Idris code to be generated. This syntax also allows for named parameters and variables within type signatures to be individually annotated using a syntax similar to Javadoc parameter annotations.

Documentation always comes before the declaration being documented. Inline documentation applies to either top-level declarations or to constructors. Documentation for specific arguments to constructors, type constructors, or functions can be associated with these arguments using their names.

The inline documentation for a declaration is an unbroken string of lines, each of which begins with `|||` (three pipe symbols). The first paragraph of the documentation is taken to be an overview, and in some contexts, only this overview will be shown. After the documentation for the declaration as a whole, it is possible to associate documentation with specific named parameters, which can either be explicitly name or the results of converting free variables to implicit parameters. Annotations are the same as with Javadoc annotations, that is for the named parameter `(n : T)`, the corresponding annotation is `||| @ n Some description` that is placed before the declaration.

Documentation is written in Markdown, though not all contexts will display all possible formatting (for example, images are not displayed when viewing documentation in the REPL, and only some terminals render italics correctly). A comprehensive set of examples is given below.

```
||| Modules can also be documented.
module Docs

||| Add some numbers.
|||
||| Addition is really great. This paragraph is not part of the overview.
||| Still the same paragraph. Lists are also nifty:
||| * Really nifty!
```

```
||| * Yep!
||| * The name `add` is a **bold** choice
||| @ n is the recursive param
||| @ m is not
add : (n, m : Nat) -> Nat
add Z     m = m
add (S n) m = S (add n m)


||| Append some vectors
||| @ a the contents of the vectors
||| @ xs the first vector (recursive param)
||| @ ys the second vector (not analysed)
appendV : (xs : Vect n a) -> (ys : Vect m a) -> Vect (add n m) a
appendV []      ys = ys
appendV (x::xs) ys = x :: appendV xs ys

||| Here's a simple datatype
data Ty =
  ||| Unit
  UNIT |
  ||| Functions
  ARR Ty Ty

||| Points to a place in a typing context
data Elem : Vect n Ty -> Ty -> Type where
  Here : {ts : Vect n Ty} -> Elem (t::ts) t
  There : {ts : Vect n Ty} -> Elem ts t -> Elem (t'::ts) t

||| A more interesting datatype
||| @ n the number of free variables
||| @ ctxt a typing context for the free variables
||| @ ty the type of the term
data Term : (ctxt : Vect n Ty) -> (ty : Ty) -> Type where

  ||| The constructor of the unit type
  ||| More comment
  ||| @ ctxt the typing context
  UnitCon : {ctxt : Vect n Ty} -> Term ctxt UNIT

  ||| Function application
  ||| @ f the function to apply
  ||| @ x the argument
  App : {ctxt : Vect n Ty} -> (f : Term ctxt (ARR t1 t2)) -> (x : Term ctxt t1) -> Term ctxt t2

  ||| Lambda
  ||| @ body the function body
  Lam : {ctxt : Vect n Ty} -> (body : Term (t1::ctxt) t2) -> Term ctxt (ARR t1 t2)

  ||| Variables
  ||| @ i de Bruijn index
  Var : {ctxt : Vect n Ty} -> (i : Elem ctxt t) -> Term ctxt t

||| A computation that may someday finish
codata Partial : Type -> Type where

  ||| A finished computation
  ||| @ value the result
  Now : (value : a) -> Partial a

  ||| A not-yet-finished computation
  ||| @ rest the remaining work
  Later : (rest : Partial a) -> Partial a
```

```
||| We can document records, including their fields and constructors
record Yummy where
  ||| Make a yummy
  constructor MkYummy
  ||| What to eat
  food : String
```

CHAPTER 2

Packages

Idris includes a simple system for building packages from a package description file. These files can be used with the Idris compiler to manage the development process of your Idris programmes and packages.

2.1 Package Descriptions

A package description includes the following:

- A header, consisting of the keyword package followed by the package name.
- Fields describing package contents, `<field> = <value>`

At least one field must be the modules field, where the value is a comma separated list of modules. For example, a library test which has two modules `foo.idr` and `bar.idr` as source files would be written as follows:

```
package foo

modules = foo, bar
```

Other examples of package files can be found in the `libs` directory of the main Idris repository, and in third-party libraries.

2.1.1 Common Fields

Other common fields which may be present in an `ipkg` file are:

- `sourcedir = <dir>`, which takes the directory (relative to the current directory) which contains the source. Default is the current directory.
- `executable = <output>`, which takes the name of the executable file to generate.
- `main = <module>`, which takes the name of the main module, and must be present if the executable field is present.
- `opts = "<idris options>"`, which allows options to be passed to Idris.
- `pkgs = <pkg name> (',' <pkg name>)+`, a comma separated list of package names that the Idris package requires.

2.1.2 Binding to C

In more advanced cases, particularly to support creating bindings to external `C` libraries, the following options are available:

- `makefile = <file>`, which specifies a `Makefile`, to be built before the Idris modules, for example to support linking with a `C` library.

- `libs = <libs>`, which takes a comma separated list of libraries which must be present for the package to be usable.
- `objs = <objs>`, which takes a comma separated list of additional object files to be installed, perhaps generated by the `Makefile`.

2.1.3 Testing

For testing Idris packages there is a rudimentary testing harness, run in the `IO` context. The `iPKG` file is used to specify the functions used for testing. The following option is available:

- `tests = <test functions>`, which takes the qualified names of all test functions to be run.

Important: The modules containing the test functions must also be added to the list of modules.

2.2 Using Package files

Given an Idris package file `text.ipkg` it can be used with the Idris compiler as follows:

- `idris --build test.ipkg` will build all modules in the package
- `idris --install test.ipkg` will install the package, making it accessible by other Idris libraries and programs.
- `idris --clean test.ipkg` will delete all intermediate code and executable files generated when building.
- `idris --mkdoc test.ipkg` will build HTML documentation for your package in the folder `test_doc` in your project's root directory.
- `idris --checkpkg test.ipkg` will type check all modules in the package only. This differs from build that type checks **and** generates code.
- `idris --testpkg test.ipkg` will compile and run any embedded tests you have specified in the `tests` parameter.

Once the test package has been installed, the command line option `--package test` makes it accessible (abbreviated to `-p test`). For example:

```
idris -p test Main.idr
```

CHAPTER 3

Uniqueness Types

Uniqueness Types are an experimental feature available from Idris 0.9.15. A value with a unique type is guaranteed to have *at most one* reference to it at run-time, which means that it can safely be updated in-place, reducing the need for memory allocation and garbage collection. The motivation is that we would like to be able to write reactive systems, programs which run in limited memory environments, device drivers, and any other system with hard real-time requirements, ideally while giving up as little high level conveniences as possible.

They are inspired by linear types, Uniqueness Types in the Clean programming language, and ownership types and borrowed pointers in the Rust programming language.

Some things we hope to be able to do eventually with uniqueness types include:

- Safe, pure, in-place update of arrays, lists, etc
- Provide guarantees of correct resource usage, state transitions, etc
- Provide guarantees that critical program fragments will *never* allocate

3.1 Using Uniqueness

If `x : T` and `T : UniqueType`, then there is at most one reference to `x` at any time during run-time execution. For example, we can declare the type of unique lists as follows:

```
data UList : Type -> UniqueType
     Nil   : UList a
     (::)  : a -> UList a -> UList a
```

If we have a value `xs : UList a`, then there is at most one reference to `xs` at run-time. The type checker preserves this guarantee by ensuring that there is at most one reference to any value of a unique type in a pattern clause. For example, the following function definition would be valid:

```
umap : (a -> b) -> UList a -> UList b
umap f [] = []
umap f (x :: xs) = f x :: umap f xs
```

In the second clause, `xs` is a value of a unique type, and only appears once on the right hand side, so this clause is valid. Not only that, since we know there can be no other reference to the `UList a` argument, we can reuse its space for building the result! The compiler is aware of this, and compiles this definition to an in-place update of the list.

The following function definition would not be valid (even assuming an implementation of ++), however, since `xs` appears twice:

```
dupList : UList a -> UList a
dupList xs = xs ++ xs
```

This would result in a shared pointer to `xs`, so the typechecker reports:

```
unique.idr:12:5:Unique name xs is used more than once
```

If we explicitly copy, however, the typechecker is happy:

```
dup : UList a -> UList a
dup [] = []
dup (x :: xs) = x :: x :: dup xs
```

Note that it's fine to use `x` twice, because `a` is a `Type`, rather than a `UniqueType`.

There are some other restrictions on where a `UniqueType` can appear, so that the uniqueness property is preserved. In particular, the type of the function type, `(x : a) -> b` depends on the type of `a` or `b` - if either is a `UniqueType`, then the function type is also a `UniqueType`. Then, in a data declaration, if the type constructor builds a `Type`, then no constructor can have a `UniqueType`. For example, the following definition is invalid, since it would embed a unique value in a possible non-unique value:

```
data BadList : UniqueType -> Type
     Nil  : {a : UniqueType} -> BadList a
     (::) : {a : UniqueType} -> a -> BadList a -> BadList a
```

Finally, types may be polymorphic in their uniqueness, to a limited extent. Since `Type` and `UniqueType` are different types, we are limited in how much we can use polymorphic functions on unique types. For example, if we have function composition defined as follows:

```
(.) : {a, b, c : Type} -> (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

And we have some functions over unique types:

```
foo : UList a -> UList b
bar : UList b -> UList c
```

Then we cannot compose `foo` and `bar` as `bar . foo`, because `UList` does not compute a `Type`! Instead, we can define composition as follows:

```
(.) : {a, b, c : Type*} -> (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

The `Type*` type stands for either unique or non-unique types. Since such a function may be passed a `UniqueType`, any value of type `Type*` must also satisfy the requirement that it appears at most once on the right hand side.

3.1.1 Borrowed Types

It quickly becomes obvious when working with uniqueness types that having only one reference at a time can be painful. For example, what if we want to display a list before updating it?

```
showU : Show a => UList a -> String
showU xs = "[" ++ showU' xs ++ "]" where
  showU' : UList a -> String
  showU' [] = ""
  showU' [x] = show x
  showU' (x :: xs) = show x ++ ", " ++ showU' xs
```

This is a valid definition of `showU`, but unfortunately it consumes the list! So the following function would be invalid:

```
printAndUpdate : UList Int -> IO ()
printAndUpdate xs = do putStrLn (showU xs)
                       let xs' = umap (*2) xs -- xs no longer available!
                       putStrLn (showU xs')
```

Still, one would hope to be able to display a unique list without problem, since it merely *inspects* the list; there are no updates. We can achieve this, using the notion of *borrowing*. A Borrowed type is a Unique type which can be inspected at the top level (by pattern matching, or by *lending* to another function) but no further. This ensures that the internals (i.e. the arguments to top level patterns) will not be passed to any function which will update them.

`Borrowed` converts a `UniqueType` to a `BorrowedType`. It is defined as follows (along with some additional rules in the typechecker):

```
data Borrowed : UniqueType -> BorrowedType where
     Read : {a : UniqueType} -> a -> Borrowed a

implicit
lend : {a : UniqueType} -> a -> Borrowed a
lend x = Read x
```

A value can be "lent" to another function using `lend`. Arguments to `lend` are not counted by the type checker as a reference to a unique value, therefore a value can be lent as many times as desired. Using this, we can write `showU` as follows:

```
showU : Show a => Borrowed (UList a) -> String
showU xs = "[" ++ showU' xs ++ "]" where
  showU' : Borrowed (UList a) -> String
  showU' [] = ""
  showU' [x] = show x
  showU' (Read (x :: xs)) = show x ++ ", " ++ showU' (lend xs)
```

Unlike a unique value, a borrowed value may be referred to as many times as desired. However, there is a restriction on how a borrowed value can be used. After all, much like a library book or your neighbour's lawnmower, if a function borrows a value it is expected to return it in exactly the condition in which it was received!

The restriction is that when a `Borrowed` type is matched, any pattern variables under the `Read` which have a unique type may not be referred to at all on the right hand side (unless they are themselves `lent` to another function).

Uniqueness information is stored in the type, and in particular in function types. Once we're in a unique context, any new function which is constructed will be required to have unique type, which prevents the following sort of bad program being implemented:

```
foo : UList Int -> IO ()
foo xs = do let f = \x : Int => showU xs
            putStrLn $ free xs
            putStrLn $ f 42
            return ()
```

Since `lend` is implicit, in practice for functions to lend and borrow values merely requires the argument to be marked as `Borrowed`. We can therefore write `showU` as follows:

```
showU : Show a => Borrowed (UList a) -> String
showU xs = "[" ++ showU' xs ++ "]" where
  showU' : Borrowed (UList a) -> String
  showU' [] = ""
  showU' [x] = show x
  showU' (x :: xs) = show x ++ ", " ++ showU' xs
```

3.1.2 Problems/Disadvantages/Still to do...

This is a work in progress, there is lots to do. The most obvious problem is the loss of abstraction. On the one hand, we have more precise control over memory usage with `UniqueType` and `BorrowedType`, but they are not in general compatible with functions polymorphic over `Type`. In the short term, we can start to write reactive and low memory systems with this, but longer term it would be nice to support more abstraction.

We also haven't checked any of the metatheory, so this could all be fatally flawed! The implementation is based to a large extent on Uniqueness Typing Simplified, by de Vries et al, so there is reason to believe things should be

fine, but we still have to do the work.

Much as there are with linear types, there are some annoyances when trying to prove properties of functions with unique types (for example, what counts as a use of a value). Since we require *at most* one use of a value, rather than *exactly* one, this seems to be less of an issue in practice, but still needs thought.

CHAPTER 4

New Foreign Function Interface

Ever since Idris has had multiple backends compiling to different target languages on potentially different platforms, we have had the problem that the foreign function interface (FFI) was written under the assumption of compiling to C. As a result, it has been hard to write generic code for multiple targets, or even to be sure that if code compiles that it will run on the expected target.

As of 0.9.17, Idris will have a new foreign function interface (FFI) which is aware of multiple targets. Users who are working with the default code generator can happily continue writing programs as before with no changes, but if you are writing bindings for an external library, writing a back end, or working with a non-C back end, there are some things you will need to be aware of, which this page describes.

4.1 The `IO'` monad, and `main`

The `IO` monad exists as before, but is now specific to the C backend (or, more precisely, any backend whose foreign function calls are compatible with C.) Additionally, there is now an `IO'` monad, which is parameterised over a FFI descriptor:

```
data IO' : (lang : FFI) -> Type -> Type
```

The Prelude defines two FFI descriptors which are imported automatically, for C and JavaScript/Node, and defines `IO` to use the C FFI and `JS_IO` to use the JavaScript FFI:

```
FFI_C  : FFI
FFI_JS : FFI

IO : Type -> Type
IO a = IO' FFI_C a

JS_IO : Type -> Type
JS_IO a = IO' FFI_JS a
```

As before, the entry point to an Idris program is `main`, but the type of `main` can now be any instance of `IO'`, e.g. the following are both valid:

```
main : IO ()
main : JS_IO ()
```

The FFI descriptor includes details about which types can be marshalled between the foreign language and Idris, and the "target" of a foreign function call (typically just a String representation of the function's name, but potentially something more complicated such as an external library file or even a URL).

4.2 FFI descriptors

An FFI descriptor is a record containing a predicate which holds when a type can be marshalled, and the type of the target of a foreign call:

```
record FFI where
     constructor MkFFI
     ffi_types : Type -> Type
     ffi_fn : Type
```

For C, this is:

```
||| Supported C integer types
data C_IntTypes : Type -> Type where
     C_IntChar   : C_IntTypes Char
     C_IntNative : C_IntTypes Int
     ... -- more integer types

||| Supported C foreign types
data C_Types : Type -> Type where
     C_Str   : C_Types String
     C_Float : C_Types Float
     C_Ptr   : C_Types Ptr
     C_MPtr  : C_Types ManagedPtr
     C_Unit  : C_Types ()
     C_Any   : C_Types (Raw a)
     C_IntT  : C_IntTypes i -> C_Types i

FFI_C : FFI
FFI_C = MkFFI C_Types
              String -- the name of the C function
```

4.3 Foreign calls

To call a foreign function, the `foreign` function is used. For example:

```
do_fopen : String -> String -> IO Ptr
do_fopen f m
   = foreign FFI_C "fileOpen" (String -> String -> IO Ptr) f m
```

The `foreign` function takes an FFI description, a function name (the type is given by the `ffi_fn` field of `FFI_C` here), and a function type, which gives the expected types of the remaining arguments. Here, we're calling an external function `fileOpen` which takes, in the C, a `char*` file name, a `char*` mode, and returns a file pointer. It is the job of the C back end to convert Idris `String` to C `char*` and vice versa.

The argument types and return type given here must be present in the `fn_types` predicate of the `FFI_C` description for the foreign call to be valid.

Note The arguments to `foreign` *must* be known at compile time, because the foreign calls are generated statically. The `%inline` directive on a function can be used to give hints to help this, for example a shorthand for calling external JavaScript functions:

```
%inline
jscall : (fname : String) -> (ty : Type) ->
         {auto fty : FTy FFI_JS [] ty} -> ty
jscall fname ty = foreign FFI_JS fname ty
```

4.3.1 FFI implementation

In order to write bindings to external libraries, the details of how `foreign` works are unnecessary — you simply need to know that `foreign` takes an FFI descriptor, the function name, and its type. It is instructive to look a little deeper, however:

The type of `foreign` is as follows:

```
foreign : (ffi : FFI)
       -> (fname : ffi_fn f)
       -> (ty : Type)
       -> {auto fty : FTy ffi [] ty}
       -> ty
```

The important argument here is the implicit `fty`, which contains a proof (`FTy`) that the given type is valid according to the FFI description `ffi`:

```
data FTy : FFI -> List Type -> Type -> Type where
     FRet : ffi_types f t -> FTy f xs (IO' f t)
     FFun : ffi_types f s -> FTy f (s :: xs) t -> FTy f xs (s -> t)
```

Notice that this uses the `ffi_types` field of the FFI descriptor — these arguments to `FRet` and `FFun` give explicit proofs that the type is valid in this FFI. For example, the above `do_fopen` builds the following implicit proof as the `fty` argument to `foreign`:

```
FFun C_Str (FFun C_Str (FRet C_Ptr))
```

4.4 Compiling foreign calls

(This section assumes some knowledge of the Idris internals.)

When writing a back end, we now need to know how to compile `foreign`. We'll skip the details here of how a `foreign` call reaches the intermediate representation (the IR), though you can look in `IO.idr` in the `prelude` package to see a bit more detail — a `foreign` call is implemented by the primitive function `mkForeignPrim`. The important part of the IR as defined in `Lang.hs` is the following constructor:

```
data LExp = ...
          | LForeign FDesc -- Function descriptor
                     FDesc -- Return type descriptor
                     [(FDesc, LExp)]
```

So, a `foreign` call appears in the IR as the `LForeign` constructor, which takes a function descriptor (of a type given by the `ffi_fn` field in the FFI descriptor), a return type descriptor (given by an application of `FTy`), and a list of arguments with type descriptors (also given by an application of `FTy`).

An `FDesc` describes an application of a name to some arguments, and is really just a simplified subset of an `LExp`:

```
data FDesc = FCon Name
           | FStr String
           | FUnknown
           | FApp Name [FDesc]
```

There are corresponding structures in the lower level IRs, such as the defunctionalised, simplified and bytecode forms.

Our `do_fopen` example above arrives in the `LExp` form as:

```
LForeign (FStr "fileOpen") (FCon (sUN "C_Ptr"))
         [(FCon (sUN "C_Str"), f), (FCon (sUN "C_Str"), m)]
```

(Assuming that `f` and `m` stand for the `LExp` representations of the arguments.) This information should be enough for any back end to marshal the arguments and return value appropriately.

Note: When processing `FDesc`, be aware that there may be implicit arguments, which have not been erased. For example, `C_IntT` has an implicit argument `i`, so will appear in an `FDesc` as something of the form `FApp (sUN "C_IntT") [i, t]` where `i` is the implicit argument (which can be ignored) and `t` is the descriptor of the integer type. See `CodegenC.hs`, specifically the function `toFType`, to see how this works in practice.

4.5 JavaScript FFI descriptor

The JavaScript FFI descriptor is a little more complex, because the JavaScript FFI supports marshalling functions. It is defined as follows:

```
mutual
  data JsFn t = MkJsFn t

  data JS_IntTypes  : Type -> Type where
       JS_IntChar   : JS_IntTypes Char
       JS_IntNative : JS_IntTypes Int

  data JS_FnTypes : Type -> Type where
       JS_Fn     : JS_Types s -> JS_FnTypes t -> JS_FnTypes (s -> t)
       JS_FnIO   : JS_Types t -> JS_FnTypes (IO' l t)
       JS_FnBase : JS_Types t -> JS_FnTypes t

  data JS_Types : Type -> Type where
       JS_Str   : JS_Types String
       JS_Float : JS_Types Float
       JS_Ptr   : JS_Types Ptr
       JS_Unit  : JS_Types ()
       JS_FnT   : JS_FnTypes a -> JS_Types (JsFn a)
       JS_IntT  : JS_IntTypes i -> JS_Types i
```

The reason for wrapping function types in a `JsFn` is to help the proof search when building `FTy`. We hope to improve proof search eventually, but for the moment it works much more reliably if the indices are disjoint! An example of using this appears in IdrisScript when setting timeouts:

```
setTimeout : (() -> JS_IO ()) -> (millis : Int) -> JS_IO Timeout
setTimeout f millis = do
  timeout <- jscall "setTimeout(%0, %1)"
                    (JsFn (() -> JS_IO ()) -> Int -> JS_IO Ptr)
                    (MkJsFn f) millis
  return $ MkTimeout timeout
```

CHAPTER 5

Syntax Guide

Examples are mostly adapted from the Idris tutorial.

5.1 Source File Structure

Source files consist of:

1. An optional *Module Header*.
2. Zero or more *Imports*.
3. Zero or more declarations, e.g. *Variables*, *Data types*, etc.

For example:

```
module MyModule   -- module header

import Data.Vect  -- an import

%default total    -- a directive

foo : Nat         -- a declaration
foo = 5
```

5.1.1 Module Header

A file can start with a module header, introduced by the `module` keyword:

```
module Semantics
```

Module names can be hierarchical, with parts separated by `.`:

```
module Semantics.Transform
```

Each file can define only a single module, which includes everything defined in that file.

Like with declarations, a *docstring* can be used to provide documentation for a module:

```
||| Implementation of predicate transformer semantics.
module Semantics.Transform
```

5.1.2 Imports

An `import` makes the names in another module available for use by the current module:

```
import Data.Vect
```

All the declarations in an imported module are available for use in the file. In a case where a name is ambiguous — e.g. because it is imported from multiple modules, or appears in multiple visible namespaces — the ambiguity can be resolved using *Qualified Names*. (Often, the compiler can resolve the ambiguity for you, using the types involved.)

Imported modules can be given aliases to make qualified names more compact:

```
import Data.Vect as V
```

Note that names made visible by import are not, by default, re-exported to users of the module being written. This can be done using `import public`:

```
import public Data.Vect
```

5.2 Variables

A variable is always defined by defining its type on one line, and its value on the next line, using the syntax

```
<id> : <type>
<id> = <value>
```

Examples

```
x : Int
x = 100
hello : String
hello = "hello"
```

5.3 Types

In Idris, types are first class values. So a type declaration is the same as just declaration of a variable whose type is `Type`. In Idris, variables that denote a type need not be capitalised. Example:

```
MyIntType : Type
MyIntType = Int
```

a more interesting example:

```
MyListType : Type
MyListType = List Int
```

While capitalising types is not required, the rules for generating implicit arguments mean it is often a good idea.

5.3.1 Data types

Idris provides two kinds of syntax for defining data types. The first, Haskell style syntax, defines a regular algebraic data type. For example

```
data Either a b = Left a | Right b
```

or

```
data List a = Nil | (::) a (List a)
```

The second, more general kind of data type, is defined using Agda or GADT style syntax. This syntax defines a data type that is parameterised by some values (in the `Vect` example, a value of type `Nat` and a value of type `Type`).

```
data Vect : Nat -> Type -> Type where
  Nil  : Vect Z a
  (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
```

5.4 Operators

5.4.1 Arithmetic

```
x + y
x - y
x * y
x / y
(x * y) + (a / b)
```

5.4.2 Equality and Relational

```
x == y
x /= y
x >= y
x > y
x <= y
x < y
```

5.4.3 Conditional

```
x && y
x || y
not x
```

5.5 Conditionals

5.5.1 If Then Else

```
if <test> then <true> else <false>
```

5.5.2 Case Expressions

```
case <test> of
    <case 1>  => <expr>
    <case 2>  => <expr>
    ...
    otherwise => <expr>
```

5.6 Functions

5.6.1 Named

Named functions are defined in the same way as variables, with the type followed by the definition.

```
<id> : <argument type> -> <return type>
<id> arg = <expr>
```

Example

```
plusOne : Int -> Int
plusOne x = x + 1
```

Functions can also have multiple inputs, for example

```
makeHello : String -> String -> String
makeHello first last = "hello, my name is " ++ first ++ " " ++ last
```

Functions can also have named arguments. This is required if you want to annotate parameters in a docstring. The following shows the same `makeHello` function as above, but with named parameters which are also annotated in the docstring

```
||| Makes a string introducing a person
||| @first The person's first name
||| @last The person's last name
makeHello : (first : String) -> (last : String) -> String
makeHello first last = "hello, my name is " ++ first ++ " " ++ last
```

Like Haskell, Idris functions can be defined by pattern matching. For example

```
sum : List Int -> Int
sum []        = 0
sum (x :: xs) = x + (sum xs)
```

Similarly case analysis looks like

```
answerString : Bool -> String
answerString False = "Wrong answer"
answerString True = "Correct answer"
```

5.6.2 Dependent Functions

Dependent functions are functions where the type of the return value depends on the input value. In order to define a dependent function, named parameters must be used, since the parameter will appear in the return type. For example, consider

```
zeros : (n : Nat) -> Vect n Int
zeros Z     = []
zeros (S k) = 0 :: (zeros k)
```

In this example, the return type is `Vect n Int` which is an expression which depends on the input parameter `n`. ### Anonymous Arguments in anonymous functions are separated by comma.

```
(\x => <expr>)
(\x, y => <expr>)
```

5.6.3 Modifiers

Visibility

```
public
abstract
private
```

Totality

```
total
implicit
partial
covering
```

Options

```
%export
%hint
%no_implicit
%error_handler
%error_reverse
%assert_total
%reflection
%specialise [<name list>]
```

5.7 Misc

5.7.1 Qualified Names

If multiple declarations with the same name are visible, using the name can result in an ambiguous situation. The compiler will attempt to resolve the ambiguity using the types involved. If it's unable — for example, because the declarations with the same name also have the same type signatures — the situation can be cleared up using a *qualified name*.

A qualified name has the symbol's namespace prefixed, separated by a `.`:

```
Data.Vect.length
```

This would specifically reference a `length` declaration from `Data.Vect`.

Qualified names can be written using two different shorthands:

1. Names in modules that are *imported* using an alias can be qualified by the alias.
2. The name can be qualified by the *shortest unique suffix* of the namespace in question. For example, the `length` case above can likely be shortened to `Vect.length`.

5.7.2 Comments

```
-- Single Line
{- Multiline -}
||| Docstring (goes before definition)
```

5.7.3 Multi line String literals

```
foo = """
this is a
string literal"""
```

5.8 Directives

```
%lib <path>
%link <path>
%flag <path>
%include <path>
%hide <function>
%freeze <name>
%access <accessibility>
%default <totality>
%logging <level 0--11>
%dynamic <list of libs>
%name <list of names>
%error_handlers <list of names>
%language <extension>
```

CHAPTER 6

Erasure By Usage Analysis

This work stems from this feature proposal (obsoleted by this page). Beware that the information in the proposal is out of date — and sometimes even in direct contradiction with the eventual implementation.

6.1 Motivation

Traditional dependently typed languages (Agda, Coq) are good at erasing *proofs* (either via irrelevance or an extra universe).

```
half : (n : Nat) -> Even n -> Nat
half Z EZ = Z
half (S (S n)) (ES pf) = S (half n pf)
```

For example, in the above snippet, the second argument is a proof, which is used only to convince the compiler that the function is total. This proof is never inspected at runtime and thus can be erased. In this case, the mere existence of the proof is sufficient and we can use irrelevance-related methods to achieve erasure.

However, sometimes we want to erase *indices* and this is where the traditional approaches stop being useful, mainly for reasons described in the original proposal.

```
uninterleave : {n : Nat} -> Vect (n * 2) a -> (Vect n a, Vect n a)
uninterleave [] = ([] , [])
uninterleave (x :: y :: rest) with (unzipPairs rest)
  | (xs, ys) = (x :: xs, y :: ys)
```

Notice that in this case, the second argument is the important one and we would like to get rid of the `n` instead, although the shape of the program is generally the same as in the previous case.

There are methods described by Brady, McBride and McKinna in *[BMM04]* to remove the indices from data structures, exploiting the fact that functions operating on them either already have a copy of the appropriate index or the index can be quickly reconstructed if needed. However, we often want to erase the indices altogether, from the whole program, even in those cases where reconstruction is not possible.

The following two sections describe two cases where doing so improves the runtime performance asymptotically.

6.1.1 Binary numbers

- O(n) instead of O(log n)

Consider the following `Nat`-indexed type family representing binary numbers:

```
data Bin : Nat -> Type where
  N : Bin 0
  O : {n : Nat} -> Bin n -> Bin (0 + 2*n)
  I : {n : Nat} -> Bin n -> Bin (1 + 2*n)
```

These are supposed to be (at least asymptotically) fast and memory-efficient because their size is logarithmic compared to the numbers they represent.

Unfortunately this is not the case. The problem is that these binary numbers still carry the *unary* indices with them, performing arithmetic on the indices whenever arithmetic is done on the binary numbers themselves. Hence the real representation of the number 15 looks like this:

```
I -> I -> I -> I -> N
S    S    S    Z
S    S    Z
S    S
S    Z
S
S
S
Z
```

The used memory is actually *linear*, not logarithmic and therefore we cannot get below O(n) with time complexities.

One could argue that Idris in fact compiles `Nat` via GMP but that's a moot point for two reasons:

- First, whenever we try to index our data structures with anything else than `Nat`, the compiler is not going to come to the rescue.
- Second, even with `Nat`, the GMP integers are *still* there and they slow the runtime down.

This ought not to be the case since the `Nat` are never used at runtime and they are only there for typechecking purposes. Hence we should get rid of them and get runtime code similar to what a idris programmer would write.

6.1.2 U-views of lists

- O(n^2) instead of O(n)

Consider the type of U-views of lists:

```
data U : List a -> Type where
  nil : U []
  one : (z : a) -> U [z]
  two : {xs : List a} -> (x : a) -> (u : U xs) -> (y : a) -> U (x :: xs ++ [y])
```

For better intuition, the shape of the U-view of `[x0,x1,x2,z,y2,y1,y0]` looks like this:

```
x0   y0    (two)
x1   y1    (two)
x2   y1    (two)
   z       (one)
```

When recursing over this structure, the values of `xs` range over `[x0,x1,x2,z,y2,y1,y0]`, `[x1,x2,z,y2,y1]`, `[x2,z,y2]`, `[z]`. No matter whether these lists are stored or built on demand, they take up a quadratic amount of memory (because they cannot share nodes), and hence it takes a quadratic amount of time just to build values of this index alone.

But the reasonable expectation is that operations with U-views take linear time — so we need to erase the index `xs` if we want to achieve this goal.

6.2 Changes to Idris

Usage analysis is run at every compilation and its outputs are used for various purposes. This is actually invisible to the user but it's a relatively big and important change, which enables the new features.

Everything that is found to be unused is erased. No annotations are needed, just don't use the thing and it will vanish from the generated code. However, if you wish, you can use the dot annotations to get a warning if the thing is accidentally used.

"Being used" in this context means that the value of the "thing" may influence run-time behaviour of the program. (More precisely, it is not found to be irrelevant to the run-time behaviour by the usage analysis algorithm.)

"Things" considered for removal by erasure include:

- function arguments
- data constructor fields (including record fields and dictionary fields of class instances)

For example, `Either` often compiles to the same runtime representation as `Bool`. Constructor field removal sometimes combines with the newtype optimisation to have quite a strong effect.

There is a new compiler option `--warnreach`, which will enable warnings coming from erasure. Since we have full usage analysis, we can compile even those programs that violate erasure annotations – it's just that the binaries may run slower than expected. The warnings will be enabled by default in future versions of Idris (and possibly turned to errors). However, in this transitional period, we chose to keep them on-demand to avoid confusion until better documentation is written.

Case-tree elaboration tries to avoid using dotted "things" whenever possible. (NB. This is not yet perfect and it's being worked on: https://gist.github.com/ziman/10458331)

Postulates are no longer required to be collapsible. They are now required to be *unused* instead.

6.3 Changes to the language

You can use dots to mark fields that are not intended to be used at runtime.

```
data Bin : Nat -> Type where
  N : Bin 0
  O : .{n : Nat} -> Bin n -> Bin (0 + 2*n)
  I : .{n : Nat} -> Bin n -> Bin (1 + 2*n)
```

If these fields are found to be used at runtime, the dots will trigger a warning (with `--warnreach`).

Note that free (unbound) implicits are dotted by default so, for example, the constructor `O` can be defined as:

```
O : Bin n -> Bin (0 + 2*n)
```

and this is actually the preferred form.

If you have a free implicit which is meant to be used at runtime, you have to change it into an (undotted) `{bound : implicit}`.

You can also put dots in types of functions to get more guarantees.

```
half : (n : Nat) -> .(pf : Even n) -> Nat
```

and free implicits are automatically dotted here, too.

6.4 What it means

Dot annotations serve two purposes:

- influence case-tree elaboration to avoid dotted variables
- trigger warnings when a dotted variable is used

However, there's no direct connection between being dotted and being erased. The compiler erases everything it can, dotted or not. The dots are there mainly to help the programmer (and the compiler) refrain from using the values they want to erase.

6.5 How to use it

Ideally, few or no extra annotations are needed – in practice, it turns out that having free implicits automatically dotted is enough to get good erasure.

Therefore, just compile with `--warnreach` to see warnings if erasure cannot remove parts of the program.

However, those programs that have been written without runtime behaviour in mind, will need some help to get in the form that compiles to a reasonable binary. Generally, it's sufficient to follow erasure warnings (which may be sometimes unhelpful at the moment).

6.6 Benchmarks

- source: https://github.com/ziman/idris-benchmarks
- results: http://ziman.functor.sk/erasure-bm/

It can be clearly seen that asymptotics are improved by erasure.

6.7 Shortcomings

You can't get warnings in libraries because usage analysis starts from `Main.main`. This will be solved by the planned `%default_usage` pragma.

Usage warnings are quite bad and unhelpful at the moment. We should include more information and at least translate argument numbers to their names.

There is no decent documentation yet. This wiki page is the first one.

There is no generally accepted terminology. We switch between "dotted", "unused", "erased", "irrelevant", "inaccessible", while each has a slightly different meaning. We need more consistent and understandable naming.

If the same type is used in both erased and non-erased context, it will retain its fields to accommodate the least common denominator – the non-erased context. This is particularly troublesome in the case of the type of (dependent) pairs, where it actually means that no erasure would be performed. We should probably locate disjoint uses of data types and split them into "sub-types". There are three different flavours of dependent types now: `Sigma` (nothing erased), `Exists` (first component erased), `Subset` (second component erased).

Case-tree building does not avoid dotted values coming from pattern-matched constructors (https://gist.github.com/ziman/10458331). This is to be fixed soon. (Fixed.)

Higher-order function arguments and opaque functional variables are considered to be using all their arguments. To work around this, you can force erasure via the type system, using the `Erased` wrapper: https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Data/Erased.idr

Typeclass methods are considered to be using the union of all their implementations. In other words, an argument of a method is unused only if it is unused in every implementation of the method that occurs in the program.

6.8 Planned features

- Fixes to the above shortcomings in general.
- **Improvements to the case-tree elaborator so that it properly avoids** dotted fields of data constructors. Done.
- **Compiler pragma `%default_usage` `used/unused` and per-function** overrides `used` and `unused`, which allow the programmer to mark the return value of a function as used, even if the function is not used in `main` (which is the case when writing library code). These annotations

will help library writers discover usage violations in their code before it is actually published and used in compiled programs.

6.9 Troubleshooting

6.9.1 My program is slower

The patch introducing erasure by usage analysis also disabled some optimisations that were in place before; these are subsumed by the new erasure. However, in some erasure-unaware programs, where erasure by usage analysis does not exercise its full potential (but the old optimisations would have worked), certain slowdown may be observed (up to ~10% according to preliminary benchmarking), due to retention and computation of information that should not be necessary at runtime.

A simple check whether this is the case is to compile with `--warnreach`. If you see warnings, there is some unnecessary code getting compiled into the binary.

The solution is to change the code so that there are no warnings.

6.9.2 Usage warnings are unhelpful

This is a known issue and we are working on it. For now, see the section *How to read and resolve erasure warnings*.

6.9.3 There should be no warnings in this function

A possible cause is non-totality of the function (more precisely, non-coverage). If a function is non-covering, the program needs to inspect all arguments in order to detect coverage failures at runtime. Since the function inspects all its arguments, nothing can be erased and this may transitively cause usage violations. The solution is to make the function total or accept the fact that it will use its arguments and remove some dots from the appropriate constructor fields and function arguments. (Please note that this is not a shortcoming of erasure and there is nothing we can do about it.)

Another possible cause is the currently imperfect case-tree elaboration, which does not avoid dotted constructor fields (see https://gist.github.com/ziman/10458331). You can either rephrase the function or wait until this is fixed, hopefully soon. Fixed.

6.9.4 The compiler refuses to recognise this thing as erased

You can force anything to be erased by wrapping it in the `Erased` monad. While this program triggers usage warnings,

```
f : (g : Nat -> Nat) -> .(x : Nat) -> Nat
f g x = g x  -- WARNING: g uses x
```

the following program does not:

```
f : (g : Erased Nat -> Nat) -> .(x : Nat) -> Nat
f g x = g (Erase x)  -- OK
```

6.10 How to read and resolve erasure warnings

6.10.1 Example 1

Consider the following program:

```
vlen : Vect n a -> Nat
vlen {n = n} xs = n

sumLengths : List (Vect n a) -> Nat
sumLengths       []  = 0
sumLengths (v :: vs) = vlen v + sumLengths vs

main : IO ()
main = print . sumLengths $ [[0,1],[2,3]]
```

When you compile it using `--warnreach`, there is one warning:

```
Main.sumLengths: inaccessible arguments reachable:
  n (no more information available)
```

The warning does not contain much detail at this point so we can try compiling with `--dumpcases cases.txt` and look up the compiled definition in `cases.txt`:

```
Main.sumLengths {e0} {e1} {e2} =
  case {e2} of
  | Prelude.List.::({e6}) => LPlus (ATInt ITBig)({e0}, Main.sumLengths({e0}, ____, {e6}))
  | Prelude.List.Nil() => 0
```

The reason for the warning is that `sumLengths` calls `vlen`, which gets inlined. The second clause of `sumLengths` then accesses the variable `n`, compiled as `{e0}`. Since `n` is a free implicit, it is automatically considered dotted and this triggers the warning.

A solution would be either making the argument `n` a bound implicit parameter to indicate that we wish to keep it at runtime,

```
sumLengths : {n : Nat} -> List (Vect n a) -> Nat
```

or fixing `vlen` to not use the index:

```
vlen : Vect n a -> Nat
vlen [] = Z
vlen (x :: xs) = S (vlen xs)
```

Which solution is appropriate depends on the usecase.

6.10.2 Example 2

Consider the following program manipulating value-indexed binary numbers.

```
data Bin : Nat -> Type where
    N : Bin Z
    O : Bin n -> Bin (0 + n + n)
    I : Bin n -> Bin (1 + n + n)

toN : (b : Bin n) -> Nat
toN  N = Z
toN (O {n} bs) = 0 + n + n
toN (I {n} bs) = 1 + n + n

main : IO ()
main = print . toN $ I (I (O (O (I N))))
```

In the function `toN`, we attempted to "cheat" and instead of traversing the whole structure, we just projected the value index `n` out of constructors `I` and `O`. However, this index is a free implicit, therefore it is considered dotted.

Inspecting it then produces the following warnings when compiling with `--warnreach`:

```
Main.I: inaccessible arguments reachable:
  n from Main.toN arg# 1
Main.O: inaccessible arguments reachable:
  n from Main.toN arg# 1
```

We can see that the argument `n` of both `I` and `O` is used in the function `toN`, argument 1.

At this stage of development, warnings only contain argument numbers, not names; this will hopefully be fixed. When numbering arguments, we go from 0, taking free implicits first, left-to-right; then the bound arguments. The function `toN` has therefore in fact two arguments: `n` (argument 0) and `b` (argument 1). And indeed, as the warning says, we project the dotted field from `b`.

Again, one solution is to fix the function `toN` to calculate its result honestly; the other one is to accept that we carry a `Nat` with every constructor of `Bin` and make it a bound implicit:

```
O : {n : Nat} -> Bin n -> Bin (0 + n + n)
I : {n : Nat} -> bin n -> Bin (1 + n + n)
```

6.11 References

CHAPTER 7

The IDE Protocol

The Idris REPL has two modes of interaction: a human-readable syntax designed for direct use in a terminal, and a machine-readable syntax designed for using Idris as a backend for external tools.

7.1 Protocol Overview

The communication protocol is of asynchronous request-reply style: a single request from the client is handled by Idris at a time. Idris waits for a request on its standard input stream, and outputs the answer or answers to standard output. The result of a request can be either success, failure, or intermediate output; and furthermore, before the result is delivered, there might be additional meta-messages.

A reply can consist of multiple messages: any number of messages to inform the user about the progress of the request or other informational output, and finally a result, either `ok` or `error`.

The wire format is the length of the message in characters, encoded in 6 characters hexadecimal, followed by the message encoded as S-expression (sexp). Additionally, each request includes a unique integer (counting upwards), which is repeated in all messages corresponding to that request.

An example interaction from loading the file `/home/hannes/empty.idr` looks as follows on the wire::

```
00002a((:load-file "/home/hannes/empty.idr") 1)
000039(:write-string "Type checking /home/hannes/empty.idr" 1)
000025(:set-prompt "/home/hannes/empty" 1)
000032(:return (:ok "Loaded /home/hannes/empty.idr") 1)
```

The first message is the request from idris-mode to load the specific file, which length is hex 2a, decimal 42 (including the newline at the end). The request identifier is set to 1. The first message from Idris is to write the string `Type checking /home/hannes/empty.idr`, another is to set the prompt to `*/home/hannes/empty`. The answer, starting with `:return` is `ok`, and additional information is that the file was loaded.

There are three atoms in the wire language: numbers, strings, and symbols. The only compound object is a list, which is surrounded by parenthesis. The syntax is:

```
A ::= NUM | '"' STR '"' | ':' ALPHA+
S ::= A | '(' S* ')' | nil
```

where `NUM` is either 0 or a positive integer, `ALPHA` is an alphabetical character, and `STR` is the contents of a string, with `"` escaped by a backslash. The atom `nil` is accepted instead of `()` for compatibility with some regexp pretty-printing routines.

The state of the Idris process is mainly the active file, which needs to be kept synchronised between the editor and Idris. This is achieved by the already seen `:load-file` command.

The available commands include:

`(:load-file FILENAME)` Load the named file

`(:interpret STRING)` Interpret `STRING` at the Idris REPL, returning a highlighted result

(:repl-completions STRING) Return the result of tab-completing `STRING` as a REPL command

(:type-of STRING) Return the type of the name, written with Idris syntax in the `STRING`. The reply may contain highlighting information.

(:case-split LINE NAME) Generate a case-split for the pattern variable `NAME` on program line `LINE`. The pattern-match cases to be substituted are returned as a string with no highlighting.

(:add-clause LINE NAME) Generate an initial pattern-match clause for the function declared as `NAME` on program line `LINE`. The initial clause is returned as a string with no highlighting.

(:add-proof-clause LINE NAME) Add a clause driven by the <== syntax.

(:add-missing LINE NAME) Add the missing cases discovered by totality checking the function declared as `NAME` on program line `LINE`. The missing clauses are returned as a string with no highlighting.

(:make-with LINE NAME) Create a with-rule pattern match template for the clause of function `NAME` on line `LINE`. The new code is returned with no highlighting.

(:proof-search LINE NAME HINTS) Attempt to fill out the holes on `LINE```named ``NAME` by proof search. `HINTS` is a possibly-empty list of additional things to try while searching.

(:docs-for NAME) Look up the documentation for `NAME`, and return it as a highlighted string.

(:metavariables WIDTH) List the currently-active holes, with their types pretty-printed with `WIDTH` columns.

(:who-calls NAME) Get a list of callers of `NAME`

(:calls-who NAME) Get a list of callees of `NAME`

(:browse-namespace NAMESPACE) Return the contents of `NAMESPACE`, like `:browse` at the command-line REPL

(:normalise-term TM) Return a highlighted string consisting of the results of normalising the serialised term `TM` (which would previously have been sent as the `tt-term` property of a string)

(:show-term-implicits TM) Return a highlighted string consisting of the results of making all arguments in serialised term `TM` (which would previously have been sent as the `tt-term` property of a string) explicit.

(:hide-term-implicits TM) Return a highlighted string consisting of the results of making all arguments in serialised term `TM` (which would previously have been sent as the `tt-term` property of a string) follow their usual implicitness setting.

(:elaborate-term TM) Return a highlighted string consisting of the the core language term corresponding to serialised term `TM` (which would previously have been sent as the `tt-term` property of a string).

(:print-definition NAME) Return the definition of `NAME` as a highlighted string

Possible replies include a normal final reply::

```
(:return (:ok SEXP [HIGHLIGHTING]))
(:return (:error String [HIGHLIGHTING]))
```

A normal intermediate reply::

```
(:output (:ok SEXP [HIGHLIGHTING]))
(:output (:error String [HIGHLIGHTING]))
```

Informational and/or abnormal replies::

```
(:write-string String)
(:set-prompt String)
(:warning (FilePath (LINE COL) (LINE COL) String [HIGHLIGHTING]))
```

Proof mode replies::

```
(:start-proof-mode)
(:write-proof-state [String] [HIGHLIGHTING])
(:end-proof-mode)
(:write-goal String)
```

7.2 Output Highlighting

Idris mode supports highlighting the output from Idris. In reality, this highlighting is controlled by the Idris compiler. Some of the return forms from Idris support an optional extra parameter: a list mapping spans of text to metadata about that text. Clients can then use this list both to highlight the displayed output and to enable richer interaction by having more metadata present. For example, the Emacs mode allows right-clicking identifiers to get a menu with access to documentation and type signatures.

A particular semantic span is a three element list. The first element of the list is the index at which the span begins, the second element is the number of characters included in the span, and the third is the semantic data itself. The semantic data is a list of lists. The head of each list is a key that denotes what kind of metadata is in the list, and the tail is the metadata itself.

The following keys are available:

`name` gives a reference to the fully-qualified Idris name

`implicit` provides a Boolean value that is True if the region is the name of an implicit argument

`decor` describes the category of a token, which can be `type`, `function`, `data`, `keyword`, or `bound`.

`source-loc` states that the region refers to a source code location. Its body is a collection of key-value pairs, with the following possibilities:

`filename` provides the filename

`start` provides the line and column that the source location starts at as a two-element tail

`end` provides the line and column that the source location ends at as a two-element tail

`text-formatting` provides an attribute of formatted text. This is for use with natural-language text, not code, and is presently emitted only from inline documentation. The potential values are `bold`, `italic`, and `underline`.

`link-href` provides a URL that the corresponding text is a link to.

`quasiquotation` states that the region is quasiquoted.

`antiquotation` states that the region is antiquoted.

`tt-term` A serialised representation of the Idris core term corresponding to the region of text.

7.3 Source Code Highlighting

Idris supports instructing editors how to colour their code. When elaborating source code or REPL input, Idris will locate regions of the source code corresponding to names, and emit information about these names using the same metadata as output highlighting.

These messages will arrive as replies to the command that caused elaboration to occur, such as `:load-file` or `:interpret`. They have the format::

```
(:output (:ok (:highlight-source POSNS)))
```

where `POSNS` is a list of positions to highlight. Each of these is a two-element list whose first element is a position (encoded as for the `source-loc` property above) and whose second element is highlighting metadata in the same format used for output.

CHAPTER 8

Semantic Highlighting & Pretty Printing

Since `v0.9.18` Idris comes with support for semantic highlighting. When using the `REPL` or IDE support, Idris will highlight your code accordingly to its meaning within the Idris structure. A precursor to semantic highlighting support is the pretty printing of definitions to console, LaTeX, or HTML.

The default styling scheme used was inspired by Conor McBride's own set of stylings, informally known as *Conor Colours*.

8.1 Legend

The concepts and their default stylings are as follows:

Idris Term	HTML	LaTeX	IDE/REPL
Bound Variable	Purple	Magenta	
Keyword	Bold	Underlined	
Function	Green	Green	
Type	Blue	Blue	
Data	Red	Red	
Implicit	Italic Purple	Italic Magenta	

8.2 Pretty Printing

Idris also supports the pretty printing of code to HTML and LaTeX using the commands:

- `:pp <latex|html> <width> <function name>`
- `:pprint <latex|html> <width> <function name>`

8.3 Customisation

If you are not happy with the colours used, the VIM and Emacs editor support allows for customisation of the colours. When pretty printing Idris code as LaTeX and HTML, commands and a CSS style are provided. The colours used by the REPL can be customised through the initialisation script.

8.4 Further Information

Please also see the Idris Extras project for links to editor support, and pre-made style files for LaTeX and HTML.

CHAPTER 9

DEPRECATED: Tactics and Theorem Proving

Warning: The interactive theorem-proving interface documented here has been deprecated in favor of *Elaborator Reflection*.

Idris supports interactive theorem proving, and the analyse of context through holes. To list all unproven holes, use the command `:m`. This will display their qualified names and the expected types. To interactively prove a holes, use the command `:p name` where `name` is the hole. Once the proof is complete, the command `:a` will append it to the current module.

Once in the interactive prover, the following commands are available:

9.1 Basic commands

- `:q` - Quits the prover (gives up on proving current lemma).
- `:abandon` - Same as :q
- `:state` - Displays the current state of the proof.
- `:term` - Displays the current proof term complete with its yet-to-be-filled holes (is only really useful for debugging).
- `:undo` - Undoes the last tactic.
- `:qed` - Once the interactive theorem prover tells you "No more goals," you get to type this in celebration! (Completes the proof and exits the prover)

9.2 Commonly Used Tactics

9.2.1 Compute

- `compute` - Normalises all terms in the goal (note: does not normalise assumptions)

```
----------                 Goal:                 ----------
 (Vect (S (S Z + (S Z) + (S n))) Nat) -> Vect (S (S (S (S n)))) Nat
-lemma> compute
----------                 Goal:                 ----------
 (Vect (S (S (S (S n)))) Nat) -> Vect (S (S (S (S n)))) Nat
-lemma>
```

9.2.2 Exact

- `exact` - Provide a term of the goal type directly.

```
----------                    Goal:                    ----------
 Nat
-lemma> exact Z
lemma: No more goals.
-lemma>
```

9.2.3 Refine

- `refine` - Use a name to refine the goal. If the name needs arguments, introduce them as new goals.

9.2.4 Trivial

- `trivial` - Satisfies the goal using an assumption that matches its type.

```
----------                Assumptions:                ----------
 value : Nat
----------                    Goal:                   ----------
 Nat
-lemma> trivial
lemma: No more goals.
-lemma>
```

9.2.5 Intro

- `intro` - If your goal is an arrow, turns the left term into an assumption.

```
----------                    Goal:                    ----------
 Nat -> Nat -> Nat
-lemma> intro
----------                Assumptions:                ----------
 n : Nat
----------                    Goal:                   ----------
 Nat -> Nat
-lemma>
```

You can also supply your own name for the assumption:

```
----------                    Goal:                    ----------
Nat -> Nat -> Nat
-lemma> intro number
----------                Assumptions:                ----------
 number : Nat
----------                    Goal:                   ----------
Nat -> Nat
```

9.2.6 Intros

- `intros` - Exactly like intro, but it operates on all left terms at once.

```
----------                    Goal:                    ----------
 Nat -> Nat -> Nat
-lemma> intros
----------                Assumptions:                ----------
 n : Nat
 m : Nat
----------                    Goal:                   ----------
 Nat
-lemma>
```

9.2.7 let

- `let` - Introduces a new assumption; you may use current assumptions to define the new one.

```
----------                Assumptions:                ----------
 n : Nat
----------                   Goal:                    ----------
 BigInt
-lemma> let x = toIntegerNat n
----------                Assumptions:                ----------
 n : Nat
  x = toIntegerNat n: BigInt
----------                   Goal:                    ----------
 BigInt
-lemma>
```

9.2.8 rewrite

- `rewrite` - Takes an expression with an equality type (x = y), and replaces all instances of x in the goal with y. Is often useful in combination with 'sym'.

```
----------                Assumptions:                ----------
 n : Nat
 a : Type
 value : Vect Z a
----------                   Goal:                    ----------
 Vect (mult n Z) a
-lemma> rewrite sym (multZeroRightZero n)
----------                Assumptions:                ----------
 n : Nat
 a : Type
 value : Vect Z a
----------                   Goal:                    ----------
 Vect Z a
-lemma>
```

9.2.9 induction

- `induction` - (`Note that this is still experimental` and you may get strange results and error messages. We are aware of these and will finish the implementation eventually!) Prove the goal by induction. Each constructor of the datatype becomes a goal. Constructors with recursive arguments become induction steps, while simple constructors become base cases. Note that this only works for datatypes that have eliminators: a datatype definition must have the `%elim` modifier.

9.2.10 sourceLocation

- `sourceLocation` - Solve the current goal with information about the location in the source code where the tactic was invoked. This is mostly for embedded DSLs and programmer tools like assertions that need to know where they are called. See `Language.Reflection.SourceLocation` for more information.

9.3 Less commonly-used tactics

- `applyTactic` - Apply a user-defined tactic. This should be a function of type `List (TTName, Binder TT) -> TT -> Tactic`, where the first argument represents the proof context and the second represents the goal. If your tactic will produce a proof term directly, use the `Exact` constructor from `Tactic`.

- `attack` - ?
- `equiv` - Replaces the goal with a new one that is convertible with the old one
- `fill` - ?
- `focus` - ?
- `mrefine` - Refining by matching against a type
- `reflect` - ?
- `solve` - Takes a guess with the correct type and fills a hole with it, closing a proof obligation. This happens automatically in the interactive prover, so `solve` is really only relevant in tactic scripts used for helping implicit argument resolution.
- `try` - ?

CHAPTER 10

The Idris REPL

Idris comes with a `REPL`.

10.1 Evaluation

Being a fully dependently typed language, Idris has two phases where it evaluates things, compile-time and run-time. At compile-time it will only evaluate things which it knows to be total (i.e. terminating and covering all possible inputs) in order to keep type checking decidable. The compile-time evaluator is part of the Idris kernel, and is implemented in Haskell using a HOAS (higher order abstract syntax) style representation of values. Since everything is known to have a normal form here, the evaluation strategy doesn't actually matter because either way it will get the same answer, and in practice it will do whatever the Haskell run-time system chooses to do.

The REPL, for convenience, uses the compile-time notion of evaluation. As well as being easier to implement (because we have the evaluator available) this can be very useful to show how terms evaluate in the type checker. So you can see the difference between:

```
Idris> \n, m => (S n) + m
\n => \m => S (plus n m) : Nat -> Nat -> Nat

Idris> \n, m => n + (S m)
\n => \m => plus n (S m) : Nat -> Nat -> Nat
```

10.2 Customisation

Idris supports initialisation scripts.

10.2.1 Initialisation scripts

When the Idris REPL starts up, it will attempt to open the file repl/init in Idris's application data directory. The application data directory is the result of the Haskell function call `getAppUserDataDirectory "idris"`, which on most Unix-like systems will return $HOME/.idris and on various versions of Windows will return paths such as `C:/Documents And Settings/user/Application Data/appName`.

The file repl/init is a newline-separate list of REPL commands. Not all commands are supported in initialisation scripts — only the subset that will not interfere with the normal operation of the REPL. In particular, setting colours, display options such as showing implicits, and log levels are supported.

10.2.2 Example initialisation script

```
:colour prompt white italic bold
:colour implicit magenta italic
```

10.3 The REPL Commands

The current set of supported commands are:

Command	Arguments	Purpose
<expr>		Evaluate an expression
:t :type	<expr>	Check the type of an expression
:core	<expr>	View the core language representation of a term
:miss :missing	<name>	Show missing clauses
:doc	<name>	Show internal documentation
:mkdoc	<namespace>	Generate IdrisDoc for namespace(s) and dependencies
:apropos	[<package list>] <name>	Search names, types, and documentation
:s :search	[<package list>] <expr>	Search for values by type
:wc :whocalls	<name>	List the callers of some name
:cw :callswho	<name>	List the callees of some name
:browse	<namespace>	List the contents of some namespace
:total	<name>	Check the totality of a name
:r :reload		Reload current file
:l :load	<filename>	Load a new file
:cd	<filename>	Change working directory
:module	<module>	Import an extra module
:e :edit		Edit current file using $EDITOR or $VISUAL
:m :metavars		Show remaining proof obligations (holes)
:p :prove	<hole>	Prove a hole
:a :addproof	<name>	Add proof to source file
:rmproof	<name>	Remove proof from proof stack
:showproof	<name>	Show proof
:proofs		Show available proofs
:x	<expr>	Execute IO actions resulting from an expression using the interpreter
:c :compile	<filename>	Compile to an executable [codegen] <filename>
:exec :execute	[<expr>]	Compile to an executable and run
:dynamic	<filename>	Dynamically load a C library (similar to %dynamic)
:dynamic		List dynamically loaded C libraries
:? :h :help		Display this help text
:set	<option>	Set an option (errorcontext, showimplicits, originalerrors, autosolve, nobanner,
:unset	<option>	Unset an option
:color :colour	<option>	Turn REPL colours on or off; set a specific colour
:consolewidth	auto|infinite|<number>	Set the width of the console
:printerdepth	<number-or-blank>	Set the maximum pretty-printing depth, or infinite if nothing specified
:q :quit		Exit the Idris system
:w :warranty		Displays warranty information
:let	(<top-level-declaration>)...	Evaluate a declaration, such as a function definition, instance implementation,
:unlet :undefine	(<name>)...	Remove the listed repl definitions, or all repl definitions if no names given
:printdef	<name>	Show the definition of a function
:pp :pprint	<option> <number> <name>	Pretty prints an Idris function in either LaTeX or HTML and for a specified wi

10.4 Using the REPL

10.4.1 Getting help

The command `:help` (or `:h` or `:?`) prints a short summary of the available commands.

10.4.2 Quitting Idris

If you would like to leave Idris, simply use `:q` or `:quit`.

10.4.3 Evaluating expressions

To evaluate an expression, simply type it. If Idris is unable to infer the type, it can be helpful to use the operator `the` to manually provide one, as Idris's syntax does not allow for direct type annotations. Examples of `the` include:

```
Idris> the Nat 4
4 : Nat
Idris> the Int 4
4 : Int
Idris> the (List Nat) [1,2]
[1,2] : List Nat
Idris> the (Vect _ Nat) [1,2]
[1,2] : Vect 2 Nat
```

This may not work in cases where the expression still involves ambiguous names. The name can be disambiguated by using the `with` keyword:

```
Idris> sum [1,2,3]
When elaborating an application of function Prelude.Foldable.sum:
        Can't disambiguate name: Prelude.List.::,
                                 Prelude.Stream.::,
                                 Prelude.Vect.::
Idris> with List sum [1,2,3]
6 : Integer
```

10.4.4 Adding let bindings

To add a let binding to the REPL, use `:let`. It's likely you'll also need to provide a type annotation. `:let` also works for other declarations as well, such as `data`.

```
Idris> :let x : String; x = "hello"
Idris> x
"hello" : String
Idris> :let y = 10
Idris> y
10 : Integer
Idris> :let data Foo : Type where Bar : Foo
Idris> Bar
Bar : Foo
```

10.4.5 Getting type information

To ask Idris for the type of some expression, use the `:t` command. Additionally, if used with an overloaded name, Idris will provide all overloadings and their types. To ask for the type of an infix operator, surround it in parentheses.

```
Idris> :t "foo"
"foo" : String
Idris> :t plus
Prelude.Nat.plus : Nat -> Nat -> Nat
Idris> :t (++)
Builtins.++ : String -> String -> String
Prelude.List.++ : (List a) -> (List a) -> List a
Prelude.Vect.++ : (Vect m a) -> (Vect n a) -> Vect (m + n) a
Idris> :t plus 4
plus (Builtins.fromInteger 4) : Nat -> Nat
```

You can also ask for basic information about typeclasses with `:doc`:

```
Idris> :doc Monad
Type class Monad

Parameters:
    m

Methods:
    (>>=) : Monad m => m a -> (a -> m b) -> m b

        infixl 5

Instances:
    Monad id
    Monad PrimIO
    Monad IO
    Monad Maybe

...
```

Other documentation is also available from `:doc`:

```
Idris> :doc (+)
Prelude.Classes.+ : (a : Type) -> (Num a) -> a -> a -> a

infixl 8

Arguments:
        Class constraint Prelude.Classes.Num a
        __pi_arg : a
        __pi_arg1 : a
```

```
Idris> :doc Vect
Data type Prelude.Vect.Vect : Nat -> Type -> Type

Arguments:
        Nat
        Type

Constructors:

Prelude.Vect.Nil : (a : Type) -> Vect 0 a


Prelude.Vect.:: : (a : Type) -> (n : Nat) -> a -> (Vect n a) -> Vect (S n) a

infixr 7

Arguments:
        a
        Vect n a
```

```
Idris> :doc Monad
Type class Prelude.Monad.Monad
Methods:

Prelude.Monad.>>= : (m : Type -> Type) -> (a : Type) -> (b : Type) -> (Monad m) -> (m a) -> (a ->

infixl 5

Arguments:
        Class constraint Prelude.Monad.Monad m
        __pi_arg : m a
        __pi_arg1 : a -> m b
```

10.4.6 Finding things

The command `:apropos` searches names, types, and documentation for some string, and prints the results. For example:

```
Idris> :apropos eq
eqPtr : Ptr -> Ptr -> IO Bool


eqSucc : (left : Nat) -> (right : Nat) -> (left = right) -> S left = S right
S preserves equality

lemma_both_neq : ((x = x') -> _|_) -> ((y = y') -> _|_) -> ((x, y) = (x', y')) -> _|_


lemma_fst_neq_snd_eq : ((x = x') -> _|_) -> (y = y') -> ((x, y) = (x', y)) -> _|_


lemma_snd_neq : (x = x) -> ((y = y') -> _|_) -> ((x, y) = (x, y')) -> _|_


lemma_x_eq_xs_neq : (x = y) -> ((xs = ys) -> _|_) -> (x :: xs = y :: ys) -> _|_


lemma_x_neq_xs_eq : ((x = y) -> _|_) -> (xs = ys) -> (x :: xs = y :: ys) -> _|_


lemma_x_neq_xs_neq : ((x = y) -> _|_) -> ((xs = ys) -> _|_) -> (x :: xs = y :: ys) -> _|_


prim__eqB16 : Bits16 -> Bits16 -> Int

prim__eqB16x8 : Bits16x8 -> Bits16x8 -> Bits16x8

prim__eqB32 : Bits32 -> Bits32 -> Int

prim__eqB32x4 : Bits32x4 -> Bits32x4 -> Bits32x4

prim__eqB64 : Bits64 -> Bits64 -> Int

prim__eqB64x2 : Bits64x2 -> Bits64x2 -> Bits64x2

prim__eqB8 : Bits8 -> Bits8 -> Int

prim__eqB8x16 : Bits8x16 -> Bits8x16 -> Bits8x16

prim__eqBigInt : Integer -> Integer -> Int
```

```
prim__eqChar : Char -> Char -> Int

prim__eqFloat : Float -> Float -> Int

prim__eqInt : Int -> Int -> Int

prim__eqString : String -> String -> Int

prim__syntactic_eq : (a : Type) -> (b : Type) -> (x : a) -> (y : b) -> Maybe (x = y)

sequence : Traversable t => Applicative f => (t (f a)) -> f (t a)


sequence_ : Foldable t => Applicative f => (t (f a)) -> f ()


Eq : Type -> Type
The Eq class defines inequality and equality.

GTE : Nat -> Nat -> Type
Greater than or equal to

LTE : Nat -> Nat -> Type
Proofs that n is less than or equal to m

gte : Nat -> Nat -> Bool
Boolean test than one Nat is greater than or equal to another

lte : Nat -> Nat -> Bool
Boolean test than one Nat is less than or equal to another

ord : Char -> Int
Convert the number to its ASCII equivalent.

replace : (x = y) -> (P x) -> P y
Perform substitution in a term according to some equality.

sym : (l = r) -> r = l
Symmetry of propositional equality

trans : (a = b) -> (b = c) -> a = c
Transitivity of propositional equality
```

`:search` does a type-based search, in the spirit of Hoogle. See Type-directed search (:search) for more details. Here is an example:

```
Idris> :search a -> b -> a
= Prelude.Basics.const : a -> b -> a
Constant function. Ignores its second argument.

= assert_smaller : a -> b -> b
Assert to the totality checker than y is always structurally
smaller than x (which is typically a pattern argument)

> malloc : Int -> a -> a


> Prelude.pow : Num a => a -> Nat -> a


> Prelude.Classes.(*) : Num a => a -> a -> a
```

```
> Prelude.Classes.(+) : Num a => a -> a -> a
... (More results)
```

`:search` can also look for dependent types:

```
Idris> :search plus (S n) n = plus n (S n)
< Prelude.Nat.plusSuccRightSucc : (left : Nat) ->
                                  (right : Nat) ->
                                  S (left + right) = left + S right
```

10.4.7 Loading and reloading Idris code

The command `:l File.idr` will load File.idr into the currently-running REPL, and `:r` will reload the last file that was loaded.

10.4.8 Totality

All Idris definitions are checked for totality. The command `:total <NAME>` will display the result of that check. If a definition is not total, this may be due to an incomplete pattern match. If that is the case, `:missing` or `:miss` will display the missing cases.

10.4.9 Editing files

The command `:e` launches your default editor on the current module. After control returns to Idris, the file is reloaded.

10.4.10 Invoking the compiler

The current module can be compiled to an executable using the command `:c <FILENAME>` or `:compile <FILENAME>`. This command allows to specify codegen, so for example JavaScript can be generated using `:c javascript <FILENAME>`. The `:exec` command will compile the program to a temporary file and run the resulting executable.

10.4.11 IO actions

Unlike GHCI, the Idris REPL is not inside of an implicit IO monad. This means that a special command must be used to execute IO actions. `:x tm` will execute the IO action `tm` in an Idris interpreter.

10.4.12 Dynamically loading C libraries

Sometimes, an Idris program will depend on external libraries written in C. In order to use these libraries from the Idris interpreter, they must first be dynamically loaded. This is achieved through the `%dynamic <LIB>` directive in Idris source files or through the `:dynamic <LIB>` command at the REPL. The current set of dynamically loaded libraries can be viewed by executing `:dynamic` with no arguments. These libraries are available through the Idris FFI in *type providers* and `:exec`.

10.5 Colours

Idris terms are available in amazing colour! By default, the Idris REPL uses colour to distinguish between data constructors, types or type constructors, operators, bound variables, and implicit arguments. This feature is available on all POSIX-like systems, and there are plans to allow it to work on Windows as well.

If you do not like the default colours, they can be turned off using the command

```
:colour off
```

and, when boredom strikes, they can be re-enabled using the command

```
:colour on
```

To modify a colour, use the command

```
:colour <CATEGORY> <OPTIONS>
```

where <CATEGORY is one of `keyword`, `boundvar`, `implicit`, `function`, `type`, `data`, or `prompt`, and is a space-separated list drawn from the colours and the font options. The available colours are `default`, `black`, `yellow`, `cyan`, `red`, `blue`, `white`, `green`, and `magenta`. If more than one colour is specified, the last one takes precedence. The available options are `dull` and `vivid`, `bold` and `nobold`, `italic` and `noitalic`, `underline` and `nounderline`, forming pairs of opposites. The colour `default` refers to your terminal's default colour.

The colours used at startup can be changed using REPL initialisation scripts.

Colour can be disabled at startup by the `--nocolour` command-line option.

CHAPTER 11

Compilation and Logging

This section provides highlights of the Idris compilation process, and provides details over how you can follow the process through logging.

11.1 Compilation Process

Idris follows the following compilation process:

1. Parsing
2. Type Checking
   (a) Elaboration
   (b) Coverage
   (c) Unification
   (d) Totality Checking
   (e) Erasure
3. Code Generation
   (a) Defunctionalisation
   (b) Inlining
   (c) Resolving variables
   (d) Code Generation

11.2 Type Checking Only

With Idris you can ask it to terminate the compilation process after type checking has completed. This is achieved through use of either:

- The command line options
  - `--check` for files
  - `--checkpkg` for packages
- The REPL command: `:check`

Use of this option will still result in the generation of the Idris binary `.ibc` files, and is suitable if you do not wish to generate code from one of the supported backends.

11.3 Logging Output

The internal operation of Idris is captured using a category based logger. Currently, the logging infrastructure has support for the following categories:

- Parser
- Elaborator
- Code generation
- Erasure
- Coverage Checking
- IBC generation

These categories are specified using the command-line option: `--logging-categories CATS`, where `CATS` is a quoted colon seperated string of the categories you want to see. By default if this option is not specified all categories are allowed. Sub-categories have yet to be defined but will be in the future, especially for the elaborator.

Further, the verbosity of logging can be controlled by specifying a logging level between: 1 to 10 using the command-line option: `--log <level>`.

- Level 0: Show no logging output. Default level
- Level 1: High level details of the compilation process.
- Level 2: Provides details of the coverage checking, and further details the elaboration process specifically: Class, Clauses, Data, Term, and Types,
- Level 3: Provides details of compilation of the IRTS, erasure, parsing, case splitting, and further details elaboration of: Instances, Providers, and Values.
- Level 4: Provides further details on: Erasure, Coverage Checking, Case splitting, and elaboration of clauses.
- Level 5: Provides details on the prover, and further details elaboration (adding declarations) and compilation of the IRTS.
- Level 6: Further details elaboration and coverage checking.
- Level 7:
- Level 8:
- Level 9:
- Level 10: Further details elaboration.

CHAPTER 12

Core Language Features

- Full-spectrum dependent types
- Strict evaluation (plus `Lazy : Type -> Type` type constructor for explicit laziness)
- Lambda, Pi (forall), Let bindings
- Pattern matching definitions
- Export modifiers `public`, `abstract`, `private`
- Function options `partial`, `total`
- `where` clauses
- "magic with"
- Implicit arguments (in top level types)
- "Bound" implicit arguments `{n : Nat} -> {a : Type} -> Vect n a`
- "Unbound" implicit arguments — `Vect n a` is equivalent to the above in a type, `n` and `a` are implicitly bound. This applies to names beginning with a lower case letter in an argument position.
- 'Tactic' implicit arguments, which are solved by running a tactic script or giving a default argument, rather than by unification.
- Unit type `()`, empty type `Void`
- Tuples (desugaring to nested pairs)
- Dependent pair syntax `(x : T ** P x)` (there exists an `x` of type `T` such that `P x`)
- Inline `case` expressions
- Heterogeneous equality
- `do` notation
- Idiom brackets
- Interfaces (like type classes), supporting default methods and dependencies between methods
- `rewrite` prf `in` expr
- Metavariables
- Inline proof/tactic scripts
- Implicit coercion
- `codata`
- Also `Inf : Type -> Type` type constructor for mixed data/codata. In fact `codata` is implemented by putting recursive arguments under `Inf`.
- `syntax` rules for defining pattern and term syntactic sugar

- these are used in the standard library to define `if ... then ... else` expressions and an Agda-style preorder reasoning syntax.
- Uniqueness typing using the `UniqueType` universe.
- Partial evaluation by `[static]` argument annotations.
- Error message reflection
- Eliminators
- Label types `'name`
- `%logging n`
- `%unifyLog`

CHAPTER 13

Language Extensions

13.1 Type Providers

Idris type providers are a way to get the type system to reflect observations about the world outside of Idris. Similarly to F# type providers, they cause effectful computations to run during type checking, returning information that the type checker can use when checking the rest of the program. While F# type providers are based on code generation, Idris type providers use only the ordinary execution semantics of Idris to generate the information.

A type provider is simply a term of type `IO (Provider t)`, where `Provider` is a data type with constructors for a successful result and an error. The type `t` can be either `Type` (the type of types) or a concrete type. Then, a type provider `p` is invoked using the syntax `%provide (x : t) with p`. When the type checker encounters this line, the IO action `p` is executed. Then, the resulting term is extracted from the IO monad. If it is `Provide y` for some `y : t`, then `x` is bound to `y` for the remainder of typechecking and in the compiled code. If execution fails, a generic error is reported and type checking terminates. If the resulting term is `Error e` for some string `e`, then type checking fails and the error `e` is reported to the user.

Example Idris type providers can be seen at this repository. More detailed descriptions are available in David Christiansen's WGP '13 paper and M.Sc. thesis.

CHAPTER 14

Elaborator Reflection

The Idris elaborator is responsible for converting high-level Idris code into the core language. It is implemented as a kind of embedded tactic language in Haskell, where tactic scripts are written in an *elaboration monad* that provides error handling and a proof state. For details, see Edwin Brady's 2013 paper in the Journal of Functional Programming.

Elaborator reflection makes the elaboration type as well as a selection of its tactics available to Idris code. This means that metaprograms written in Idris can have complete control over the elaboration process, generating arbitrary code, and they have access to all of the facilities available in the elaborator, such as higher-order unification, type checking, and emitting auxiliary definitions.

14.1 The Elaborator State

The elaborator state contains information about the ongoing elaboration process. In particular, it contains a *goal type*, which is to be filled by an under-construction *proof term*. The proof term can contain *holes*, each of which has a scope in which it is valid and a type. Some holes may additionally contain *guesses*, which can be substituted in the scope of the hole. The holes are tracked in a *hole queue*, and one of them is *focused*. In addition to the goal type, proof term, and holes, the elaborator state contains a collection of unsolved unification problems that can affect elaboration.

The elaborator state is not directly available to Idris programs. Instead, it is modified through the use of *tactics*, which are operations that affect the elaborator state. A tactic that returns a value of type `a`, potentially modifying the elaborator state, has type `Elab a`. The default tactics are all in the namespace `Language.Reflection.Elab.Tactics`.

14.2 Running Elaborator Scripts

On their own, tactics have no effect. The meta-operation `%runElab script` runs `script` in the current elaboration context. For example, the following script constructs the identity function at type `Nat`:

```
idNat : Nat -> Nat
idNat = %runElab (do intro (Just (UN "x"))
                     fill (Var (UN "x"))
                     solve)
```

On the right-hand side, the Idris elaborator has the goal `Nat -> Nat`. When it encounters the `%runElab` directive, it fulfills this goal by running the provided script. The first tactic, `intro`, constructs a lambda that binds the name `x`. The name argument is optional because a default name can be taken from the function type. Now, the proof term is of the form `\x : Nat => {hole}`. The second tactic, `fill`, fills this hole with a guess, giving the term `\x : Nat => {holex}`. Finally, the `solve` tactic instantiates the guess, giving the result `\x : Nat => x`.

Because elaborator scripts are ordinary Idris expressions, it is also possible to use them in multiple contexts. Note that there is nothing `Nat`-specific about the above script. We can generate identity functions at any concrete type using the same script:

```
mkId : Elab ()
mkId = do intro (Just (UN "x"))
          fill (Var (UN "x"))
          solve

idNat : Nat -> Nat
idNat = %runElab mkId

idUnit : () -> ()
idUnit = %runElab mkId

idString : String -> String
idString = %runElab mkId
```

14.3 Interactively Building Elab Scripts

You can build an `Elab` script interactively at the REPL. Use the command `:metavars`, or `:m` for short, to list the available holes. Then, issue the `:elab <hole>` command at the REPL to enter the elaboration shell.

At the shell, you can enter proof tactics to alter the proof state. You can view the system-provided tactics prior to entering the shell by issuing the REPL command `:browse Language.Reflection.Elab.Tactics`. When you have discharged all goals, you can complete the proof using the `:qed` command and receive in return an elaboration script that fills the hole.

The interactive elaboration shell accepts a limited number of commands, including a subset of the commands understood by the normal Idris REPL as well as some elaboration-specific commands.

General-purpose commands:

- `:eval <EXPR>`, or `:e <EXPR>` for short, evaluates the provided expression and prints the result.
- `:type <EXPR>`, or `:t <EXPR>` for short, prints the provided expression together with its type.
- `:search <TYPE>` searches for definitions having the provided type.
- `:doc <NAME>` searches for definitions with the provided name and prints their documentation.

Commands for viewing the proof state:

- `:state` displays the current state of the term being constructed. It lists both other goals and the current goal.
- `:term` displays the current proof term as well as its yet-to-be-filled holes.

Commands for manipulating the proof state:

- `:undo` undoes the effects of the last tactic.
- `:abandon` gives up on proving the current lemma and quits the elaboration shell.
- `:qed` finishes the script and exits the elaboration shell. The shell will only accept this command once it reports, "No more goals." On exit, it will print out the finished elaboration script for you to copy into your program.

14.4 Failure

Some tactics may *fail*. For example, `intro` will fail if the focused hole does not have a function type, `solve` will fail if the current hole does not contain a guess, and `fill` will fail if the term to be filled in has the wrong type. Scripts can also fail explicitly using the `fail` tactic.

To account for failure, there is an `Alternative` instance for `Elab`. The `<|>` operator first tries the script to its left. If that script fails, any changes that it made to the state are undone and the right argument is executed. If the first argument succeeds, then the second argument is not executed.

14.5 Querying the Elaboration State

`Elab` includes operations to query the elaboration state, allowing scripts to use information about their environment to steer the elaboration process. The ordinary Idris bind syntax can be used to propagate this information. For example, a tactic that solves the current goal when it is the unit type might look like this:

```
triv : Elab ()
triv = do compute
          g <- getGoal
          case (snd g) of
            `(() : Type) => do fill `(() : ())
                               solve
            otherGoal => fail [ TermPart otherGoal
                              , TextPart "is not trivial"
                              ]
```

The tactic `compute` normalises the type of its goal with respect to the current context. While not strictly necessary, this allows `triv` to be used in contexts where the triviality of the goal is not immediately apparent. Then, `getGoal` is used, and its result is bound to `g`. Because it returns a pair consisting of the current goal's name and type, we case-split on its second projection. If the goal type turns out to have been the unit type, we fill using the unit constructor and solve the goal. Otherwise, we fail with an error message informing the user that the current goal is not trivial.

Additionally, the elaboration state can be dumped into an error message with the `debug` tactic. A variant, `debugMessage`, allows arbitrary messages to be included with the state, allowing for a kind of "`printf` debugging" of elaboration scripts. The message format used by `debugMessage` is the same for errors produced by the error reflection mechanism, allowing the re-use of the Idris pretty-printer when rendering messages.

14.6 Changing the Global Context

`Elab` scripts can modify the global context during execution. Just as the Idris elaborator produces auxiliary definitions to implement features such as `where`-blocks and `case` expressions, user elaboration scripts may need to define functions. Furthermore, this allows `Elab` reflection to be used to implement features such as type class deriving. The operations `declareType`, `defineFunction`, and `addInstance` allow `Elab` scripts to modify the global context.

14.7 Using Idris's Features

The Idris compiler has a number of ways to automate the construction of terms. On its own, the `Elab` state and its interactions with the unifier allow implicits to be solved using unification. Additional operations use further features of Idris. In particular, `resolveTC` solves the current goal using type class resolution, `search` invokes the proof search mechanism, and `sourceLocation` finds the context in the original file at which the elaboration script is invoked.

14.8 Recursive Elaboration

The elaboration mechanism can be invoked recursively using the `runElab` tactic. This tactic takes a goal type and an elaboration script as arguments and runs the script in a fresh lexical environment to create an inhabitant of

the provided goal type. This is primarily useful for code generation, particularly for generating pattern-matching clauses, where variable scope needs to be one that isn't the present local context.

14.9 Learn More

While this documentation is still incomplete, elaboration reflection works in Idris today. As you wait for the completion of the documentation, the list of built-in tactics can be obtained using the `:browse` command in an Idris REPL or the corresponding feature in one of the graphical IDE clients to explore the `Language.Reflection.Elab.Tactics` namespace. All of the built-in tactics contain documentation strings.

CHAPTER 15

Miscellaneous

Things we have yet to classify, or are two small to justify their own page.

15.1 The Unifier Log

If you're having a hard time debugging why the unifier won't accept something (often while debugging the compiler itself), try applying the special operator `%unifyLog` to the expression in question. This will cause the type checker to spit out all sorts of informative messages.

15.2 Namespaces and type-directed disambiguation

Names can be defined in separate namespaces, and disambiguated by type. An expression `with NAME EXPR` will privilege the namespace `NAME` in the expression `EXPR`. For example:

```
Idris> with List [[1,2],[3,4],[5,6]]
[[1, 2], [3, 4], [5, 6]] : List (List Integer)

Idris> with Vect [[1,2],[3,4],[5,6]]
[[1, 2], [3, 4], [5, 6]] : Vect 3 (Vect 2 Integer)

Idris> [[1,2],[3,4],[5,6]]
Can't disambiguate name: Prelude.List.::, Prelude.Stream.::, Prelude.Vect.::
```

15.3 Alternatives

The syntax `(| option1, option2, option3, ... |)` type checks each of the options in turn until one of them works. This is used, for example, when translating integer literals.

```
Idris> the Nat (| "foo", Z, (-3) |)
0 : Nat
```

This can also be used to give simple automated proofs, for example: trying some constructors of proofs.

```
syntax Trivial = (| oh, refl |)
```

15.4 Totality checking assertions

All definitions are checked for *coverage* (i.e. all well-typed applications are handled) and either for *termination* (i.e. all well-typed applications will eventually produce an answer) or, if returning codata, for productivity (in practice, all recursive calls are constructor guarded).

Obviously, termination checking is undecidable. In practice, the termination checker looks for *size change* - every cycle of recursive calls must have a decreasing argument, such as a recursive argument of a strictly positive data type.

There are two built-in functions which can be used to give the totality checker a hint:

- `assert_total x` asserts that the expression `x` is terminating and covering, even if the totality checker cannot tell. This can be used for example if `x` uses a function which does not cover all inputs, but the caller knows that the specific input is covered.
- `assert_smaller p x` asserts that the expression `x` is structurally smaller than the pattern `p`.

For example, the following function is not checked as total:

```
qsort : Ord a => List a -> List a
qsort [] = []
qsort (x :: xs) = qsort (filter (<= x) xs) ++ (x :: qsort (filter (>= x) xs)))
```

This is because the checker cannot tell that `filter` will always produce a value smaller than the pattern `x :: xs` for the recursive call to `qsort`. We can assert that this will always be true as follows:

```
total
qsort : Ord a => List a -> List a
qsort [] = []
qsort (x :: xs) = qsort (assert_smaller (x :: xs) (filter (<= x) xs)) ++
                  (x :: qsort (assert_smaller (x :: xs) (filter (>= x) xs))))
```

15.5 Preorder reasoning

This syntax is defined in the module `Syntax.PreorderReasoning` in the `base` package. It provides a syntax for composing proofs of reflexive-transitive relations, using overloadable functions called `step` and `qed`. This module also defines `step` and `qed` functions allowing the syntax to be used for demonstrating equality. Here is an example:

```
import Syntax.PreorderReasoning
multThree : (a, b, c : Nat) -> a * b * c = c * a * b
multThree a b c =
  (a * b * c) ={ sym (multAssociative a b c) }=
  (a * (b * c)) ={ cong (multCommutative b c) }=
  (a * (c * b)) ={ multAssociative a c b }=
  (a * c * b) ={ cong {f = (*b)} (multCommutative a c) }=
  (c * a * b) QED
```

Note that the parentheses are required – only a simple expression can be on the left of `={ }=` or `QED`. Also, when using preorder reasoning syntax to prove things about equality, remember that you can only relate the entire expression, not subexpressions. This might occasionally require the use of `cong`.

Finally, although equality is the most obvious application of preorder reasoning, it can be used for any reflexive-transitive relation. Something like `step1 ={ just1 }= step2 ={ just2 }= end QED` is translated to `(step step1 just1 (step step2 just2 (qed end)))`, selecting the appropriate definitions of `step` and `qed` through the normal disambiguation process. The standard library, for example, also contains an implementation of preorder reasoning on isomorphisms.

15.6 Pattern matching on Implicit Arguments

Pattern matching is only allowed on implicit arguments when they are referred by name, e.g.

```
foo : {n : Nat} -> Nat
foo {n = Z} = Z
foo {n = S k} = k
```

or

```
foo : {n : Nat} -> Nat
foo {n = n} = n
```

The latter could be shortened to the following:

```
foo : {n : Nat} -> Nat
foo {n} = n
```

That is, `{x}` behaves like `{x=x}`.

15.7 Existence of an instance

In order to show that an instance of some typeclass is defined for some type, one could use the `%instance` keyword:

```
foo : Num Nat
foo = %instance
```

15.8 'match' application

`ty <== name` applies the function `name` in such a way that it has the type `ty`, by matching `ty` against the function's type. This can be used in proofs, for example:

```
plus_comm : (n : Nat) -> (m : Nat) -> (n + m = m + n)
-- Base case
(Z + m = m + Z) <== plus_comm =
    rewrite ((m + Z = m) <== plusZeroRightNeutral) ==>
            (Z + m = m) in refl

-- Step case
(S k + m = m + S k) <== plus_comm =
    rewrite ((k + m = m + k) <== plus_comm) in
    rewrite ((S (m + k) = m + S k) <== plusSuccRightSucc) in
        refl
```

15.9 Reflection

Including `%reflection` functions and `quoteGoal x by fn in t`, which applies `fn` to the expected type of the current expression, and puts the result in `x` which is in scope when elaborating `t`.

[BMM04] Edwin Brady, Conor McBride, James McKinna: Inductive families need not store their indices