Idris 语言文档

Version 1.3.1

Contents

CHAPTER 1

Idris 教程

本文档为 Idris 的教程，它简单介绍了如何用 Idris 语言编程。 文档中覆盖了该语言的核心特性，并假定你至少熟悉一门函数式编程语言，如 Haskell 或 OCaml。

1.1 引言

在传统编程语言中，**类型** 与 **值** 之间有明确的区分。例如在 Haskell 中, 下面这些类型分别表示整数、字符、字符列表 以及任意值的列表:

- `Int`, `Char`, `[Char]`, `[a]`

与此对应，下面这些值分别为上述类型的成员:

- `42`, `'a'`, `"Hello world!"`, `[2,3,4,5,6]`

然而，在带有 **依赖类型（Dependent Type）** 的语言中，它们的区别并不明显。 依赖类型允许类型「依赖」于值，换句话说，类型是 **一等（First-Class）** 的语言构造， 即它可以像值一样进行操作。标准范例就是带长度的列表类型[1] `Vect n a`， 其中 `a` 为元素的类型，而 `n` 为该列表的长度且可以任意长。

当类型包含了描述其性质的值（如列表的长度）时，它就能描述函数自身的性质了。 比如连接两个列表的操作，它拥有性质：结果列表的长度为两个输入列表的长度之和。 因此我们可以为 `app` 函数赋予如下类型，它用于连接向量（Vector）：

```
app : Vect n a -> Vect m a -> Vect (n + m) a
```

[1] 也许会令人困惑，它在依赖类型编程的文献中通常被称作「向量 (Vector) 」。

本教程介绍了 Idris，一个通用的依赖类型函数式编程语言。Idris 项目旨在为可验证的通用编程打造一个依赖类型的语言。 为此，Idris 被设计成了编译型语言，目的在于生成高效的可执行代码。 它还拥有轻量的外部函数接口，可与外部 `C` 库轻松交互。

1.1.1 目标受众

本教程面向已经熟悉函数式语言（如 Haskell 或 OCaml ）的读者，旨在简要地介绍该语言。 尽管大多数概念都会有简单的解释，读者仍须熟悉 Haskell 的语法。 我们亦假设读者有兴趣使用依赖类型来编写和验证系统软件。

对 Idris 更加深入的介绍，见 Edwin Brady 所著的《 Idris 类型驱动开发 》， 其进度要慢得多，涵盖了交互式程序开发以及更多的例子。本书可从 Manning 获取。

1.1.2 示例代码

本教程中的示例代码通过了 Idris 的测试。这些文件可在 Idris 发行版的 `samples` 目录中找到，因此您可以轻松使用它们。然而，我们强烈建议您不要只是载入后阅读， 而是亲自输入它们。

1.2 入门

1.2.1 前提需求

在安装 Idris 之前，你需要确认是否拥有所有必须的库和工具。你需要:

- 近期版本的 GHC 。目前测试过的最早版本为 7.10.3.
- **GNU 多精度运算库** (GMP) 可从 MacPorts/Homebrew 和所有主流 Linux 发行版中获取。

1.2.2 下载并安装

如果你满足所有的前提需求，那么安装 Idris 的最简方式就是在命令行输入:

```
cabal update; cabal install idris
```

这会安装 Hackage 中的最新版本及其所有依赖。如果你想要最新开发版的话， 可以在 GitHub 上找到它， 然后根据构建指令来安装。

如果你之前从未安装过使用 Cabal 的东西，Idris 可能不在你的 PATH 中。 如果 Idris 可执行文件无法找到，请将 `~/.cabal/bin` 添加到 `$PATH` 环境变量中。Mac OS X 用户则需要添加 `~/Library/Haskell/bin`，Windows 用户通常会发现 Cabal 将程序安装在 `%HOME%\AppData\Roaming\cabal\bin` 中。

为了检查安装是否成功，你需要编写第一个程序。请创建一个名为 `hello.idr` 的文件， 其中包含以下文本:

```
module Main

main : IO ()
main = putStrLn "Hello world"
```

如果你熟悉 Haskell，那会很清楚这段程序在做什么，是如何做的； 如果你对它感到陌生，我们稍后会详细地解释。你可以在命令行中输入 `idris hello.idr -o hello` 来将程序编译成可执行文件。 这会创建一个名为 `hello` 的可执行文件，你可以运行它：

```
$ idris hello.idr -o hello
$ ./hello
Hello world
```

请注意，美元符号 `$` 表示命令行。下面是一些常用的 Idris 命令行选项：

- `-o prog` 编译成名为 `prog` 的可执行文件。
- `--check` 无需启动交互式环境就对文件及其依赖进行类型检查。
- `--package pkg` 将包添加为依赖，例如通过 `--package contrib` 来使用 contrib 包。
- `--help` 显示用法摘要和命令行选项。

1.2.3 交互式环境

在命令行中输入 `idris` 来启动交互式环境。你会看到如下内容：

```
$ idris
    ____    __     _
   /  _/___/ /____(_)____
   / // __  / ___/ / ___/     Version 1.3.1
 _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
/___/\__,_/_/  /_/____/       Type :? for help

Idris>
```

它会提供一个 `ghci` 风格的界面，可以像类型检查那样求值表达式、进行定理证明、 编译、编辑、以及执行多种其它操作。命令 `:?` 会列出所支持的命令。在以下示例中， `hello.idr` 已被加载，`main` 的类型已通过检查，之后该程序被编译成了可执行的 `hello`。 在对某文件类型检查时，如果通过，就会创建它的的字节码版本（本例中为 `hello.ibc`） 以提升未来的加载速度。在源文件被修改之后，字节码会重新生成。

```
$ idris hello.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.3.1
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Type checking ./hello.idr
*hello> :t main
Main.main : IO ()
*hello> :c hello
*hello> :q
Bye bye
$ ./hello
Hello world
```

1.3 类型与函数

1.3.1 原语类型

Idris 定义了一些原语（Primitive）类型：`Int`、`Integer` 和 `Double` 用于数值类型，`Char` 和 `String` 用于文本操作，`Ptr` 则表示外部指针。库中还声明了一些数据类型，包括 `Bool` 及其值 `True` 和 `False`。我们可以用这些类型来声明常量。将以下内容保存到文件 `Prims.idr` 中，在命令行中输入 `idris Prims.idr` 以将其加载到 Idris 的交互式环境中：

```
module Prims

x : Int
x = 42

foo : String
foo = "Sausage machine"

bar : Char
bar = 'Z'

quux : Bool
quux = False
```

提示： 原语（Primitive）本意为不可分割的基本构件，原语类型即最基本的类型。

Idris 文件由一个可选的模块声明（此处为 `module Prims`），一个可选的导入列表，一组声明及其定义构成。在本例中并未指定导入。Idris 可由多个模块构成，每个模块中的每个定义都有它自己的命名空间。这点会在 模块与命名空间 (éąţ 29) 一节中进一步讨论。在编写 Idris 程序时，声明的顺序和缩进都很重要。函数和数据类型必须先定义再使用，每个定义都必须有类型声明，例如之前列出的 `x : Int` 和 `foo : String`。新声明的缩进级别必须与之前的声明相同。此外，分号 `;` 也可用于结束声明。

库模块 `Prelude` 会在每个 Idris 程序中自动导入，包括 IO 功能，算术运算，数据结构以及多种通用函数。Prelude 中定义了一些算术和比较运算符，我们可以在提示符中使用它们。在提示符中进行求值会给出一个答案及其类型。例如：

```
*prims> 6*6+6
42 : Integer
*prims> x == 6*6+6
True : Bool
```

Idris 为原语类型定义了所有的普通算术和比较运算。它们通过接口进行了重载，并可被扩展以适用于用户定义的类型，这点我们会在 接口 (éąţ 21) 一节中讨论。布尔表达式可通过 `if...then...else` 构造来测试，例如：

```
*prims> if x == 6 * 6 + 6 then "The answer!" else "Not the answer"
"The answer!" : String
```

1.3.2 数据类型

数据类型的声明方式和语法与 Haskell 非常相似。例如，自然数和列表的声明如下：

```
data Nat    = Z   | S Nat           -- 自然数（零与后继）
data List a = Nil | (::) a (List a) -- 多态列表
```

以上声明来自于标准库。一进制自然数要么为零（`Z`），要么就是另一个自然数的后继（`S k`）；列表要么为空（`Nil`），要么就是一个值被添加在另一个列表之前（`x :: xs`）。在 `List` 的声明中，我们使用了中缀操作符 `::`。像这样的新操作符可通过缀序声明（Fixity Declaration）来添加，如下所示：

```
infixr 10 ::
```

函数、数据构造器与类型构造器均可用名字作为中缀操作符。它们被括在括号中时， 也可作为前缀形式使用，例如 `(::)`。中缀操作符可使用下面的任何符号：

```
:+-*\/=.?|&><!@$%^~#
```

有些以这些符号构成的操作符无法被用户定义。它们是 `:`、`=>`、`->`、`<-`、`=`、`?=`、`|`、`**`、`==>`、`\`、`%`、`~`、`?` 以及 `!`。

1.3.3 函数

函数通过模式匹配（Pattern Matching）实现，语法同样和 Haskell 类似。 其主要的不同在于 Idris 的所有函数都需要类型声明，声明中使用单冒号 `:` （而非 Haskell 的双冒号 `::` ）。自然数的某些运算函数定义如下，同样来自标准库：

```
-- 一进制加法
plus : Nat -> Nat -> Nat
plus Z     y = y
plus (S k) y = S (plus k y)

-- 一进制乘法
mult : Nat -> Nat -> Nat
mult Z     y = Z
mult (S k) y = plus y (mult k y)
```

标准算术运算符 `+` 和 `*` 同样根据 `Nat` 的需要进行了重载， 它们使用上面的函数来定义。和 Haskell 不同的是，类型和函数名的首字母并无大小写限制。 函数名（前面的 `plus` 和 `mult` ），数据构造器（`Z`、`S`、`Nil` 和 `::`） 以及类型构造器（`Nat` 和 `List`）均属同一命名空间。不过按照约定， 数据类型和构造器的名字通常以大写字母开头。我们可以在 Idris 提示符中测试这些函数：

```
Idris> plus (S (S Z)) (S (S Z))
4 : Nat
Idris> mult (S (S (S Z))) (plus (S (S Z)) (S (S Z)))
12 : Nat
```

注解: 在显示一个 `Nat` 元素，如 `(S (S (S (S Z))))` 时，Idris 会将其显示为 `4`。 `plus (S (S Z)) (S (S Z))` 的结果实际上为 `(S (S (S (S Z))))`， 即自然数 `4`。这点可在 Idris 提示符中验证：

```
Idris> (S (S (S (S Z))))
4 : Nat
```

和算术运算符一样，整数字面也可通过接口重载，因此我们也能像下面这样测试函数：

```
Idris> plus 2 2
4 : Nat
Idris> mult 3 (plus 2 2)
12 : Nat
```

你可能会很好奇，既然计算机已经完美内建了整数运算，我们为何还需要一进制的自然数？ 主要的原因在于一进制数的结构非常便于推理，而且易与其它数据结构建立联系， 我们之后就会看到。尽管如此，我们并不希望以牺牲效率为代价获得这种便捷。幸运的是， Idris 知道 `Nat` （以及类似的结构化类型）和数之间的联系， 这意味着 Idris 可以优化它们的表示以及像 `plus` 和 `mult` 这样的函数。

where 从句

函数也可通过 where 从句来 **局部** 地定义。例如，要定义用来反转列表的函数， 我们可以使用辅助函数来累加新的，反转后的列表，并且它无需全局可见：

```
reverse : List a -> List a
reverse xs = revAcc [] xs where
  revAcc : List a -> List a -> List a
  revAcc acc [] = acc
  revAcc acc (x :: xs) = revAcc (x :: acc) xs
```

缩进是十分重要的，where 块中函数的缩进层次必须比外层函数更深。

注解： 作用域

任何外层作用域中可见，且没有被重新被定义过的名字，在 where 从句中也可见 （这里的 xs 被重新定义了）。若某个名字是某个类型的 **形参（Parameter）**，那么仅当它在类型中出现时才会在 where 从句的作用域中，即，它在整体结构中是固定不变的。

除函数外，where 块中也可包含局部数据声明，以下代码中的的 MyLT 就无法在 foo 的定义之外访问。

```
foo : Int -> Int
foo x = case isLT of
            Yes => x*2
            No => x*4
    where
       data MyLT = Yes | No

       isLT : MyLT
       isLT = if x < 20 then Yes else No
```

通常，where 从句中定义的函数和其它顶层函数一样，都需要类型声明。 然而，函数 f 的类型声明可在以下情况中省略：

- f 出现在顶层定义的右边
- f 的类型完全可以通过其首次应用来确定

因此，举例来说，以下定义是合法的：

```
even : Nat -> Bool
even Z = True
even (S k) = odd k where
  odd Z = False
  odd (S k) = even k

test : List Nat
test = [c (S 1), c Z, d (S Z)]
  where c x = 42 + x
        d y = c (y + 1 + z y)
              where z w = y + w
```

坑

Idris 程序中可以挖 **坑（Hole）** 来表示未完成的部分。例如，我们可以在「Hello world」 程序中为问候语 greeting 挖一个坑：

```
main : IO ()
main = putStrLn ?greeting
```

语法 ?greeting 挖了个坑，它表示程序中尚未写完的部分。这是个有效的 Idris 程序，你可以检查 greeting 的类型：

```
*Hello> :t greeting
--------------------------------------
greeting : String
```

检查坑的类型也会显示作用域中所有变量的类型。例如，给定一个未完成的 even 定义：

```
even : Nat -> Bool
even Z = True
even (S k) = ?even_rhs
```

我们可以检查 even_rhs 的类型，查看期望的返回类型，以及变量 k 的类型：

```
*Even> :t even_rhs
  k : Nat
--------------------------------------
even_rhs : Bool
```

坑非常有用，因为它能帮助我们 **逐步地** 编写函数。我们无需一次写完整个函数， 而是留下一些尚未编写的部分，让 Idris 告诉我们如何完成其定义。

提示： lhs（left hand side） 与 rhs（right hand side）分别表示等式中等号的左边和右边， 即左式和右式。

1.3.4 依赖类型

一等类型

在 Idris 中，类型是一等（First-Class）的，即它们可以像其它的语言构造那样被计算和操作 （以及传给函数）。例如，我们可以编写一个用来计算类型的函数：

```
isSingleton : Bool -> Type
isSingleton True = Nat
isSingleton False = List Nat
```

该函数可从一个 Bool 值计算出适当的类型，布尔值表示其类型是否为一个单例（Singleton）。 我们可以在任何能够使用类型的地方用该函数计算出一个类型。例如，它可用于计算返回类型：

```
mkSingle : (x : Bool) -> isSingleton x
mkSingle True = 0
mkSingle False = []
```

它也可拥有不同的输入类型。以下函数能够计算 Nat 列表之和，或者返回给定的 Nat，这取决于单例标记 single 是否为 True：

```
sum : (single : Bool) -> isSingleton single -> Nat
sum True x = x
sum False [] = 0
sum False (x :: xs) = x + sum False xs
```

向量

依赖类型的一个范例就是「带长度的列表」类型，在依赖类型的文献中，它通常被称作向量（Vector）。向量作为 Idris 库的一部分，可通过导入 `Data.Vect` 来使用，当然我们也自己声明它：

```
data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

注意我们使用了与 `List` 相同的构造器名。只要名字声明在不同的命名空间内（在实践中，通常在不同的模块内），Idris 就能接受像这样的特设（ad-hoc）名重载。有歧义的构造器名称通常可根据上下文来解决。

这里声明了一个类型族（Type Family），该声明的形式与之前的简单类型声明不太一样。我们显式地描述了类型构造器 `Vect` 的类型，它接受一个 `Nat` 和一个类型作为参数，其中 `Type` 表示类型的类型。我们说 `Vect` 通过 `Nat` 来 **索引**，并被 `Type` **参数化**。每个构造器会产生该类型家族的不同部分。`Nil` 只能用于构造零长度的向量，而 `::` 用于构造非零长度的向量。在 `::` 的类型中，我们显式地指定了一个类型为 `a` 的元素和一个类型为 `Vect k a` 的尾部（Tail）（即长度为 `k` 的向量），二者构成了一个长度为 `S k` 的向量。

同 `List` 以及 `Nat` 这类简单类型一样，我们可以通过模式匹配以同样的方式为 `Vect` 这样的依赖类型定义函数。作用于 `Vect` 的函数的类型能够描述所涉及向量的长度会如何变化。例如，下面定义的 `++` 用于连接两个 `Vect`：

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

`(++)` 的类型描述了结果向量的长度必须为输入向量的长度之和。如果我们以某种方式给出了错误的定义使其不成立，那么 Idris 就不会接受该定义。例如：

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

在经由 Idris 类型检查器检查时，它会给出以下结果：

```
$ idris VBroken.idr --check
VBroken.idr:9:23-25:
When checking right hand side of Vect.++ with expected type
        Vect (S k + m) a

When checking an application of constructor Vect.::::
        Type mismatch between
                Vect (k + k) a (Type of xs ++ xs)
        and
                Vect (plus k m) a (Expected type)

        Specifically:
                Type mismatch between
                        plus k k
                and
                        plus k m
```

该错误信息指出两个向量的长度不匹配：我们需要一个长度为 `k + m` 的向量，而你提供了一个长度为 `k + k` 的向量。

有限集

有限集，顾名思义，即元素有限的集合。它作为 Idris 库的一部分，可通过导入 Data.Fin 来使用，当然也可以像下面这样声明它：

```
data Fin : Nat -> Type where
   FZ : Fin (S k)
   FS : Fin k -> Fin (S k)
```

从它的签名中，我们可以看出该类型构造器接受一个 Nat，然后产生一个 **类型** 。因此，它不是一个「对象的容器」意义上的集合，而是个拥有无名元素的一般集合。举例来说，就是「存在一个包含五个元素的集合」的那种集合。实际上，它是一个捕获了所有落入零至 (n - 1) 范围内的整数的类型，其中 n 是用于实例化 Fin 类型的参数。例如，Fin 5 可被视作从 0 到 4 之间的整数的类型。

我们来仔细地观察一下它的构造器。

对于拥有 S k 个元素的有限集来说，FZ 是它的第零个元素，FS n 则是它的第 n+1 个元素。Fin 通过 Nat 来索引，它表示该集合中元素的个数。由于我们无法构造出属于空集的元素，因此也就无法构造出 Fin Z。

如之前提到的，Fin 家族的用途之一在于表示有界的自然数集。由于前 n 个自然数构成了一个含有 n 个元素的有限集，我们可以将 Fin n 视作大于等于零且小于 n 的整数集。

例如，下面的函数根据给定的有界索引 Fin n 找出 Vect 中的元素，它在 Prelude 中定义为：

```
index : Fin n -> Vect n a -> a
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

该函数从一个向量中找出给定位置的值。位置的边界由该向量的长度所界定（每种情况下都是 n），因此无需在运行时进行边界检查。类型检查器保证了 位置不会大于该向量的长度，当然也不会小于零。

注意这里也没有 Nil 的情况，因为这种情况不可能存在。由于没有类型为 Fin Z 且位置为 Fin n 的元素，因此 n 不可能是 Z。因此，如果你试图在一个空向量中查找元素，就会得到一个编译时的类型错误，因为这样做会强行令 n 为 Z。

隐式参数

我们再仔细观察一下 index 的类型：

```
index : Fin n -> Vect n a -> a
```

它接受两个参数：一个类型为 n 元素有限集的元素，以及一个类型为 a 的 n 元素向量。不过这里还有两个名字：n 和 a，它们未被显式地声明。index 包含了 **隐式** 参数。我们也可以将 index 的类型写作：

```
index : {a:Type} -> {n:Nat} -> Fin n -> Vect n a -> a
```

隐式参数在类型声明的大括号 {} 中给定，它们并没有在应用 index 时给出，因为它们的值可以从 Fin n 和 Vect n a 的参数类型中推导出来。任何以小写字母开头，在类型声明中作为形参和索引出现的名字都不会应用到任何实参上，它们 **总是** 会作为隐式参数被自动绑定。隐式参数仍然可以在应用时通过 {a=value} 和 {n=value} 来显式地给定，例如：

```
index {a=Int} {n=2} FZ (2 :: 3 :: Nil)
```

实际上，无论是隐式还是显式，任何参数都可以给定一个名称。我们可以将 index 声明成这样：

```
index : (i:Fin n) -> (xs:Vect n a) -> a
```

写不写它纯属偏好问题，不过有时它能让参数更加明确，有助于函数文档的记录。

此外， {} 在等号左边时可用作模式匹配，即 {var = pat} 获取一个隐式变量， 并试图对「pat」进行模式匹配。例如：

```
isEmpty : Vect n a -> Bool
isEmpty {n = Z} _   = True
isEmpty {n = S k} _ = False
```

「using」记法

有时为隐式参数提供类型会十分有用，特别是存在依赖顺序，或隐式参数本身含有依赖的情况下。 例如，我们可能希望在以下定义中指明隐式参数的类型，它为向量定义了前提（它也在 Data.Vect 的 Elem 下定义）：

```
data IsElem : a -> Vect n a -> Type where
   Here :  {x:a} ->   {xs:Vect n a} -> IsElem x (x :: xs)
   There : {x,y:a} -> {xs:Vect n a} -> IsElem x xs -> IsElem x (y :: xs)
```

IsElem x xs 的实例描述了 x 是 xs 中的一个元素。我们可以构造这样的谓词： 若所需的元素在向量的头部时为 Here，在向量的尾部中时则为 There。例如：

```
testVec : Vect 4 Int
testVec = 3 :: 4 :: 5 :: 6 :: Nil

inVect : IsElem 5 Main.testVec
inVect = There (There Here)
```

重要： 隐式参数与作用域

在类型签名中，类型检查器会将所有以小写字母开头 **并且** 没有应用到别的东西上的变量 视作隐式变量。要让上面的代码示例能够编译，你需要为 testVec 提供一个限定名。 在前面的例子中，我们假设该代码处于 Main 模块内。

如果大量使用相同的隐式参数，就会导致定义难以阅读。为避免此问题，可使用 using 块来为任何在块中出现的隐式参数指定类型和顺序：

```
using (x:a, y:a, xs:Vect n a)
  data IsElem : a -> Vect n a -> Type where
     Here  : IsElem x (x :: xs)
     There : IsElem x xs -> IsElem x (y :: xs)
```

注：声明顺序与 *mutual* 互用块

通常，函数与数据类型必须在使用前定义，因为依赖类型允许函数作为类型的一部分出现， 而类型检查会依赖于特定的函数如何定义（尽管这只对全函数成立，见 完全性检查 (éąţ 44)） 。 然而，此限制可通过使用 mutual 互用块来放宽，它允许数据类型和函数同时定义：

```
mutual
  even : Nat -> Bool
```

(äÿŃéąţçžğçżŋ)

(çżŋäÿŁéąţ)

```
even Z = True
even (S k) = odd k

odd : Nat -> Bool
odd Z = False
odd (S k) = even k
```

在 mutual 块中，首先所有的类型声明会被添加，然后是函数体。 因此，没有一个函数类型可以依赖于块中任何函数的归约行为。

1.3.5 I/O

如果计算机程序不能通过某种方式与用户或系统进行交互，那么它基本上没什么用。在 Idris 这样纯粹（Pure）的语言中，表达式没有副作用（Side-Effect）。而 I/O 的难点在于它本质上是带有副作用的。因此在 Idris 中，这样的交互被封装在 IO 类型中：

```
data IO a -- IO 操作返回一个类型为 a 的值
```

我们先给出 IO 抽象的定义，它本质上描述了被执行的 I/O 操作是什么， 而非如何去执行它们。最终操作则由运行时系统在外部执行。我们已经见过一个带 IO 的程序了：

```
main : IO ()
main = putStrLn "Hello world"
```

putStrLn 的类型描述了它接受一个字符串，然后通过 I/O 活动返回一个单元类型 () 的元素。它还有一个变体 putStr 用来输出字符串但不换行。

```
putStrLn : String -> IO ()
putStr   : String -> IO ()
```

我们可以从用户的输入中读取字符串：

```
getLine : IO String
```

Prelude 中定义了很多 I/O 操作，例如为了读写文件，需要包含：

```
data File -- abstract
data Mode = Read | Write | ReadWrite

openFile : (f : String) -> (m : Mode) -> IO (Either FileError File)
closeFile : File -> IO ()

fGetLine : (h : File) -> IO (Either FileError String)
fPutStr : (h : File) -> (str : String) -> IO (Either FileError ())
fEOF : File -> IO Bool
```

注意其中几个函数会返回 Either ，因为它们可能会失败。

1.3.6 「do」记法

I/O 程序通常需要串连起多个活动，将一个计算的输出送入下一个计算的输入中。 然而，IO 是一个抽象类型，因此我们无法直接访问一个计算的结果。 因此，我们用 do-记法来串连起操作：

```
greet : IO ()
greet = do putStr "What is your name? "
           name <- getLine
           putStrLn ("Hello " ++ name)
```

语法 `x <- iovalue` 执行 `IO a` 类型的 I/O 操作 `iovalue`，然后将类型为 `a` 的结果送入变量 `x` 中。在这种情况下，`getLine` 会返回一个 `IO String`，因此 `name` 的类型为 `String`。缩进十分重要： do 语句块中的每个语句都必须从同一列开始。`pure` 操作允许我们将值直接注入到 IO 操作中：

```
pure : a -> IO a
```

后面我们会看到，do-记法比这里的展示更加通用，并且可以被重载。

1.3.7 惰性

通常，函数的参数会在函数被调用前求值（也就是说，Idris 采用了 **及早（Eager）** 求值策略）。然而，这并不总是最佳的方式。考虑以下函数：

```
ifThenElse : Bool -> a -> a -> a
ifThenElse True  t e = t
ifThenElse False t e = e
```

该函数会使用参数 `t` 或 `e` 二者之一，而非二者都用（我们之后会看到其实它被用于实现 `if...then...else` 构造）。我们更希望 **只有** 用到的参数才被求值。为此， Idris 提供了 `Lazy` 数据类型，它允许暂缓求值：

```
data Lazy : Type -> Type where
     Delay : (val : a) -> Lazy a

Force : Lazy a -> a
```

类型为 `Lazy a` 的值只有通过 `Force` 强制求值时才会被求值。Idris 类型检查器知道 `Lazy` 类型，并会在必要时在 `Lazy a` 和 `a` 之间插入转换， 反之亦同。因此我们可以将 `ifThenElse` 写成下面这样，无需任何 `Force` 或 `Delay` 的显式使用：

```
ifThenElse : Bool -> Lazy a -> Lazy a -> a
ifThenElse True  t e = t
ifThenElse False t e = e
```

1.3.8 余数据类型

我们可以通过余数据类型，将递归参数标记为潜在无穷来定义无穷数据结构。对于一个余数据类型 `T`，其每个构造器中类型为 `T` 的参数都会被转换成类型为 `Inf T` 的参数。 这会让每个 `T` 类型的参数惰性化，使得类型为 `T` 的无穷数据结构得以构建。 余数据类型的一个例子为 `Stream`，其定义如下：

```
codata Stream : Type -> Type where
  (::) : (e : a) -> Stream a -> Stream a
```

它会被编译器翻译成下面这样：

```
data Stream : Type -> Type where
  (::) : (e : a) -> Inf (Stream a) -> Stream a
```

以下是如何用余数据类型 `Stream` 来构建无穷数据结构的例子。 在这里我们创建了一个 1 的无穷流：

```
ones : Stream Nat
ones = 1 :: ones
```

要重点注意：余数据类型不允许创建互用的无穷递归数据结构。 例如，以下代码会创建一个无穷循环并导致栈溢出：

```
mutual
  codata Blue a = B a (Red a)
  codata Red a = R a (Blue a)

mutual
  blue : Blue Nat
  blue = B 1 red

  red : Red Nat
  red = R 1 blue

mutual
  findB : (a -> Bool) -> Blue a -> a
  findB f (B x r) = if f x then x else findR f r

  findR : (a -> Bool) -> Red a -> a
  findR f (R x b) = if f x then x else findB f b

main : IO ()
main = do printLn $ findB (== 1) blue
```

为了修复它，我们必须为构造器参数的类型显式地加上 `Inf` 声明，因为余数据类型 不会将它添加到和正在定义的构造器类型 **不同** 的构造器参数上。例如，以下程序输出「1」。

```
mutual
  data Blue : Type -> Type where
   B : a -> Inf (Red a) -> Blue a

  data Red : Type -> Type where
   R : a -> Inf (Blue a) -> Red a

mutual
  blue : Blue Nat
  blue = B 1 red

  red : Red Nat
  red = R 1 blue

mutual
  findB : (a -> Bool) -> Blue a -> a
  findB f (B x r) = if f x then x else findR f r

  findR : (a -> Bool) -> Red a -> a
  findR f (R x b) = if f x then x else findB f b

main : IO ()
main = do printLn $ findB (== 1) blue
```

提示: 「归纳数据类型」和「余归纳数据类型」

余数据类型（Codata Type）的全称为余归纳数据类型（Coinductive Data Type）， 归纳数据类型和余归纳数据类型是对偶的关系。从语义上看， Inductive Type 描述了如何从更小的 term 构造出更大的 term；而 Coinductive Type 则描述了如何从更大的 term 分解成更小的 term。 二者即为塔斯基不动点

定理中的最大不动点（对应余归纳）和最小不动点（对应归纳）。 参考自 Belleve 的回答。

1.3.9 常用数据类型

Idris 包含了很多常用的数据类型和库函数（见发行版中的 `libs/` 目录及 文档 ）。本节描述了其中的一部分。 作为 `Prelude.idr` 的一部分，下面描述的函数都会被每个 Idris 程序自动导入，

List 与 Vect

我们已经见过 `List` 和 `Vect` 数据类型了：

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

data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

注意它们的构造器名称是相同的：只要构造器名称在不同的命名空间中声明， 它们就可以被重载（其实一般的名字都可以），并且通常会根据其类型来确定。 作为一种语法糖，任何带有 `Nil` 和 `::` 构造其名的类型都可被写成列表的形式。 例如：

- `[]` 表示 `Nil`
- `[1,2,3]` 表示 `1 :: 2 :: 3 :: Nil`

库中还定义了一些用于操作这些类型的函数。 `map` 对 `List` 和 `Vect` 都进行了重载，它将一个函数应用到列表或向量中的每个元素上。

```
map : (a -> b) -> List a -> List b
map f []        = []
map f (x :: xs) = f x :: map f xs

map : (a -> b) -> Vect n a -> Vect n b
map f []        = []
map f (x :: xs) = f x :: map f xs
```

例如，给定以下整数向量，以及一个将整数乘以 2 的函数：

```
intVec : Vect 5 Int
intVec = [1, 2, 3, 4, 5]

double : Int -> Int
double x = x * 2
```

函数 `map` 可像下面这样将该向量中的每个元素乘以二：

```
*UsefulTypes> show (map double intVec)
"[2, 4, 6, 8, 10]" : String
```

更多可用于 `List` 和 `Vect` 的函数的详情请参阅以下库文件：

- `libs/prelude/Prelude/List.idr`
- `libs/base/Data/List.idr`
- `libs/base/Data/Vect.idr`

- libs/base/Data/VectType.idr

其中包括过滤、追加、反转等函数。

题外话：匿名函数与操作符段

上面的表达式其实还有更加利落的写法。其中一种就是使用匿名函数（Anonymous Function）：

```
*UsefulTypes> show (map (\x => x * 2) intVec)
"[2, 4, 6, 8, 10]" : String
```

记法 `\x => val` 构造了一个匿名函数，它接受一个参数 `x` 并返回表达式 `val`。匿名函数可以接受多个参数，它们以逗号分隔，如 `\x, y, z => val`。参数也可以显式地给定类型，如 `\x : Int => x * 2`，也可使用模式匹配，如 `\(x, y) => x + y`。

```
*UsefulTypes> show (map (* 2) intVec)
"[2, 4, 6, 8, 10]" : String
```

`(*2)` 是将数字乘以 2 的函数的简写，它会被展开为 `\x => x * 2`。同样，`(2*)` 会被展开为 `\x => 2 * x`。

提示： 匿名函数在函数式编程中又称为 λ-表达式（Lambda Expression）。

Maybe

`Maybe` 描述了可选值，表示给定类型的值是否存在：

```
data Maybe a = Just a | Nothing
```

`Maybe` 是一种为可能失败的操作赋予类型的方式。例如，在 `List`（而非向量）中查找时可能会产生越界错误：

```
list_lookup : Nat -> List a -> Maybe a
list_lookup _     Nil         = Nothing
list_lookup Z     (x :: xs) = Just x
list_lookup (S k) (x :: xs) = list_lookup k xs
```

`maybe` 函数用于处理 `Maybe` 类型的值，如果值存在就对其应用一个函数，否则提供一个默认值：

```
maybe : Lazy b -> Lazy (a -> b) -> Maybe a -> b
```

注意前两个参数的类型被封装在 `Lazy` 内。由于二者只有其一会被使用，而计算完大型表达式之后就丢弃会造成浪费，因此我们将它们标记为 `Lazy`。

元组

值可以通过以下内建的数据类型构成序对（Pair）：

```
data Pair a b = MkPair a b
```

序对的语法糖可以写成 `(a, b)`，根据上下文，其意思为 `Pair a b` 或 `MkPair a b`。元组（Tuple）可包含任意个数的值，它通过嵌套的序对来表示：

```
fred : (String, Int)
fred = ("Fred", 42)

jim : (String, Int, String)
jim = ("Jim", 25, "Cambridge")
```

```
*UsefulTypes> fst jim
"Jim" : String
*UsefulTypes> snd jim
(25, "Cambridge") : (Int, String)
*UsefulTypes> jim == ("Jim", (25, "Cambridge"))
True : Bool
```

依赖序对

依赖序对允许序对第二个元素的类型依赖于第一个元素的值。

```
data DPair : (a : Type) -> (P : a -> Type) -> Type where
   MkDPair : {P : a -> Type} -> (x : a) -> P x -> DPair a P
```

同样，它也有语法糖。`(a : A ** P)` 表示由 A 和 P 构成的序对的类型，其中名字 `a` 可出现在 `P` 中。`( a ** p )` 会构造一个该类型的值。例如， 我们可以将一个数和一个特定长度的 `Vect` 构造成一个序对：

```
vec : (n : Nat ** Vect n Int)
vec = (2 ** [3, 4])
```

如果你喜欢，也可以把它写成较长的形式，二者完全等价：

```
vec : DPair Nat (\n => Vect n Int)
vec = MkDPair 2 [3, 4]
```

当然，类型检查器可以根据向量的的长度推断出第一个元素的值。 我们可以在希望类型检查器填写值的地方写一个下划线 `_`，这样上面的定义也可以写作：

```
vec : (n : Nat ** Vect n Int)
vec = (_ ** [3, 4])
```

有时我们也更倾向于省略该序对第一个元素的类型，同样，它也可以被推断出来：

```
vec : (n ** Vect n Int)
vec = (_ ** [3, 4])
```

依赖序对的一个用处就是返回依赖类型的值，其中的索引未必事先知道。例如， 若按照某谓词过滤出 `Vect` 中的元素，我们不会事先知道结果向量的长度：

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
```

如果 `Vect` 为空，结果很简单：

```
filter p Nil = (_ ** [])
```

在 `::` 的情况下，我们需要检查 `filter` 递归调用的结果来提取结果的长度和向量。 为此，我们使用 `with` 记法，它允许我们对中间值进行模式匹配：

```
filter p (x :: xs) with (filter p xs)
  | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

我们之后会看到 `with` 的更多详情。

依赖序对有时被称作「Sigma 类型」。

记录

记录（Record） 数据类型将多个值（记录的 **字段（Field）**）收集在一起。 Idris 提供了定义记录的语法，它也会自动生成用于访问和更新字段的函数。 和数据结构的语法不同，Idris 中的记录遵循与 Haskell 中看起来不同的语法。 例如，我们可以将一个人的名字和年龄用记录表示：

```
record Person where
    constructor MkPerson
    firstName, middleName, lastName : String
    age : Int

fred : Person
fred = MkPerson "Fred" "Joe" "Bloggs" 30
```

构造器名称由 `constructor` 关键字确定，**字段** 在 `where` 关键字之后的缩进块中给定（此处为 `firstName`、`middleName`、`lastName` 以及 `age`）。

```
*Record> firstName fred
"Fred" : String
*Record> age fred
30 : Int
*Record> :t firstName
firstName : Person -> String
```

我们也可以用字段名来更新一个记录（确切来说，会产生一个更新了给定字段的记录的副本）：

```
*Record> record { firstName = "Jim" } fred
MkPerson "Jim" "Joe" "Bloggs" 30 : Person
*Record> record { firstName = "Jim", age $= (+ 1) } fred
MkPerson "Jim" "Joe" "Bloggs" 31 : Person
```

语法 `record { field = val, ... }` 会生成一个更新了记录中给定字段的函数。`=` 为字段赋予新值，而 `$=` 通过应用一个函数来更新其值。

每个记录在它自己的命名空间中定义，这意味着字段名可在多个记录中重用。

记录以及记录中的字段可拥有依赖类型。更新允许更改字段的类型，前提是结果是良类型的。

```
record Class where
    constructor ClassInfo
    students : Vect n Person
    className : String
```

将 `students` 的字段更新为不同长度的向量是安全的，因为它不会影响该记录的类型：

```
addStudent : Person -> Class -> Class
addStudent p c = record { students = p :: students c } c


*Record> addStudent fred (ClassInfo [] "CS")
ClassInfo [MkPerson "Fred" "Joe" "Bloggs" 30] "CS" : Class
```

我们也可以用 $= 来更简洁地定义 addStudent:

```
addStudent' : Person -> Class -> Class
addStudent' p c = record { students $= (p ::) } c
```

嵌套记录的更新

Idris 也提供了便于访问和更新嵌套记录的语法。例如，若一个字段可以通过表达式 c (b (a x)) 访问，那么它就可通过以下语法更新:

```
record { a->b->c = val } x
```

这会返回一个新的记录，通过路径 a->b->c 访问的字段会被设置为 val。 该语法是一等的，即 record { a->b->c = val } 本身拥有一个函数类型。 与此对应，你也可以使用以下语法访问该字段:

```
record { a->b->c } x
```

$= 记法对嵌套记录的更新亦有效。

依赖记录

记录也可依赖于值。记录拥有 **形参**，它无法像其它字段那样更新。 形参作为结果类型的参数出现，写在记录类型名之后。例如，一个序对类型可定义如下:

```
record Prod a b where
    constructor Times
    fst : a
    snd : b
```

使用前面的 class 记录，班级的大小可用 Vect 及通过 size 参数化该记录的大小来限制其类型。例如:

```
record SizedClass (size : Nat) where
    constructor SizedClassInfo
    students : Vect size Person
    className : String
```

注意 它无法再使用之前的 addStudent 函数了，因为这会改变班级的大小。 现在用于添加学生的函数必须在类型中指定班级的大小加一。由于其大小用自然数指定， 新的值可使用 S 构造器递增:

```
addStudent : Person -> SizedClass n -> SizedClass (S n)
addStudent p c =  SizedClassInfo (p :: students c) (className c)
```

1.3.10 更多表达式

let 绑定

中间值可使用 let 绑定来计算:

```
mirror : List a -> List a
mirror xs = let xs' = reverse xs in
                xs ++ xs'
```

我们也可以在 let 绑定中进行简单的模式匹配。例如，我们可以按如下方式从记录中提取字段， 也可以通过顶层的模式匹配来提取字段:

```
data Person = MkPerson String Int

showPerson : Person -> String
showPerson p = let MkPerson name age = p in
                   name ++ " is " ++ show age ++ " years old"
```

列表推导

Idris 提供了 **推导** 记法作为构建列表的简便写法。一般形式为：

```
[ expression | qualifiers ]
```

它会根据逗号分隔的限定式 `qualifiers` 给定的条件，通过求值表达式 `expression` 产生的值来生成列表。例如，我们可以按如下方式构建构建勾股三角的列表：

```
pythag : Int -> List (Int, Int, Int)
pythag n = [ (x, y, z) | z <- [1..n], y <- [1..z], x <- [1..y],
                         x*x + y*y == z*z ]
```

`[a..b]` 是另一种构建从 `a` 到 `b` 的数列的简便记法。此外， `[a,b..c]` 以 `a`，`b` 之差为增量，构建从 `a` 到 `c` 的数列。 它可作用于 `Nat`、`Int` 以及 `Integer`，它们使用了 Prelude 中的 `enumFromTo` 与 `enumFromThenTo` 函数。

提示： 推导式

推导式（Comprehension）来源于集合的构建方法，即 `{x|x∈XΦ(x)}`，其中的 `Φ(x)` 即为限定式（Qualifier）。详情参见 维基百科 。

---

case 表达式

另一种检查 **简单** 类型的中间值的方法是使用 `case` 表达式。例如， 以下函数在给定的字符处将字符串分为两部分：

```
splitAt : Char -> String -> (String, String)
splitAt c x = case break (== c) x of
                  (x, y) => (x, strTail y)
```

`break` 是个库函数，它从给定的函数返回 true 的位置将字符串分为一个字符串的序对。 我们接着解构它返回的序对，并移除第二个字符串的第一个字符。

一个 `case` 表达式可匹配多种情况，例如去检查一个类型为 `Maybe a` 的中间值。 回想 `list_lookup`，它按索引查找列表中的元素，若索引越界则返回 `Nothing`。 我们可以用它来编写 `lookup_default`，该函数按索引查找元素，若索引越界则返回默认值：

```
lookup_default : Nat -> List a -> a -> a
lookup_default i xs def = case list_lookup i xs of
                               Nothing => def
                               Just x => x
```

若索引在界内，我们就会获得该索引对应的值，否则就会获得默认值：

```
*UsefulTypes> lookup_default 2 [3,4,5,6] (-1)
5 : Integer
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
*UsefulTypes> lookup_default 4 [3,4,5,6] (-1)
-1 : Integer
```

限制： `case` 构造用于对中间表达式进行简单的分析，以此避免编写辅助函数， 它也在内部用于实现 `let` 和 λ-绑定的模式匹配。它 **仅** 在以下情况中可用：

- 每个分支 **匹配** 一个相同类型的值，并 **返回** 一个相同类型的值。
- 结果的类型是「已知」的，即表达式的类型无需对该 `case` 表达式进行类型检查就能确定。

1.3.11 完全性

Idris 区分 **完全（全，Total）** 函数与 **部分（偏，Partial）** 函数。 全函数满足以下情况之一：

- 对于所有可能的输入都会终止，或
- 产生一个非空的，有限的，可能为无限结果的前缀

若一个函数是完全的，我们可以认为其类型精确描述了该函数会做什么。例如， 若我们有一个返回类型为 `String` 的函数，根据它是否完全，我们能知道的东西会有所不同：

- 若它是全函数，就会在有限的时间内返回一个类型为 `String` 的值；
- 若它是偏函数，那么只要它不崩溃或进入无限循环，就会返回一个 `String`。

Idris 对此作了区分，因此它知道在进行类型检查（正如我们在 一等类型 (éąţ 8) 一节所见）的时候，哪些函数可以安全地求值。毕竟，若它在类型检查时试图对一个不终止的函数求值， 那么类型检查将无法终止！因此，在类型检查时只有全函数才会被求值。偏函数仍然可在类型中使用， 但它们不会被进一步求值。

1.4 接口

我们经常想要定义能够跨多种不同数据类型工作的函数。例如，我们想要算术运算符至少可以作用于 `Int`、`Integer` 和 `Double`。我们想要 `==` 作用于大部分的数据类型。 我们想要以一种统一的方式来显示不同的类型。

为此，我们使用了 **接口（Interface）**，它类似于 Haskell 中的类型类（Typeclass）或 Rust 中的特性（Trait）。为了定义接口，我们提供了一组可重载的的函数。`Show` 接口就是个简单的例子， 它在 Prelude 中定义，并提供了将值转换为 `String` 的接口：

```
interface Show a where
    show : a -> String
```

它会生成一个类型如下的函数，我们称之为 `Show` 接口的 **方法（Method）**：

```
show : Show a => a -> String
```

我们可以把它读作：「在 `a` 实现了 `Show` 的约束下，该函数接受一个输入 `a` 并返回一个 `String`。」我们可以通过为它定义接口的方法来实现该接口。例如，`Nat` 的 `Show` 实现可定义为：

```
Show Nat where
    show Z = "Z"
    show (S k) = "s" ++ show k
```

```
Idris> show (S (S (S Z)))
"sssZ" : String
```

一个类型只能实现一次某个接口，也就是说，实现无法被覆盖。实现的声明自身可以包含约束。为此，实现的参数必须为构造器（即数据或类型构造器）或变量（也就是说，你无法为函数赋予实现）。例如，要为向量定义一个 Show 的实现，我们需要确认其元素实现了 Show，因为要用它来将每个元素都转换为 String：

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

提示： 译者更愿意将它读作：对于一个实现了 Show 的 a，Vect n a 对 Show 的实现为……

1.4.1 默认定义

库中定义了 Eq 接口，它提供了比较值是否相等的方法，所有内建类型都实现了它：

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool
```

要为类型实现一个接口，我们必须给出所有方法的定义。例如，为 Nat 实现 Eq：

```
Eq Nat where
    Z     == Z     = True
    (S x) == (S y) = x == y
    Z     == (S y) = False
    (S x) == Z     = False

    x /= y = not (x == y)
```

很难想象在哪些情况下 /= 方法不是应用了 == 方法的结果的否定。因此，利用接口声明中的某个方法为其它方法提供默认的定义会很方便：

```
interface Eq a where
    (==) : a -> a -> Bool
    (/=) : a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

Eq 的最小完整实现只需要定义 == 或 /= 二者之一，而不需要二者都定义。若缺少某个方法定义，且存在它的默认定义，那么就会使用该默认定义。

1.4.2 扩展接口

接口也可以扩展。逻辑上，相等关系 Eq 的下一步是定义排序关系 Ord。我们可以定义一个 Ord 接口，它除了继承 Eq 的方法外还定义了自己的方法：

```
data Ordering = LT | EQ | GT

interface Eq a => Ord a where
    compare : a -> a -> Ordering

    (<) : a -> a -> Bool
    (>) : a -> a -> Bool
    (<=) : a -> a -> Bool
    (>=) : a -> a -> Bool
    max : a -> a -> a
    min : a -> a -> a

sort : Ord a => List a -> List a
```

函数、接口和实现可拥有多个约束。多个约束的列表写在括号内，以逗号分隔，例如：

```
sortAndShow : (Ord a, Show a) => List a -> String
sortAndShow xs = show (sort xs)
```

注意：接口与 mutual 块

除 mutual 块外，Idris 严格遵循「先定义后使用」的规则。在 mutual 块中， Idris 会分两趟进行繁释（elaborate）：第一趟为类型，第二趟为定义。当互用块中包含接口声明时， 第一趟会繁释接口的头部而不繁释方法类型；第二趟则繁释方法类型以及所有的默认定义。

1.4.3 函子与应用子

目前，我们已经见过形参类型为 Type 的单形参接口了。通常，形参的个数可为任意个 （甚至零个），而形参也可为 **任意** 类型。若形参的类型不为 Type， 我们则需要提供显式的类型声明。例如，Prelude 中定义的函子接口 Functor 为：

```
interface Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b
```

函子允许函数应用到结构上，例如将一个函数应用到 List 的每一个元素上：

```
Functor List where
  map f []      = []
  map f (x::xs) = f x :: map f xs

Idris> map (*2) [1..10]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20] : List Integer
```

定义了 Functor 之后，我们就能定义应用子 Applicative 了， 它对函数应用的概念进行了抽象：

```
infixl 2 <*>

interface Functor f => Applicative (f : Type -> Type) where
    pure  : a -> f a
    (<*>) : f (a -> b) -> f a -> f b
```

1.4.4 单子与 do-记法

单子接口 Monad 允许我们对绑定和计算进行封装，它也是「*do*」记法 (éąţ 12) 一节中 do-记法的基础。单子扩展了前面定义的 Applicative，其定义如下：

```
interface Applicative m => Monad (m : Type -> Type) where
    (>>=)  : m a -> (a -> m b) -> m b
```

在 do 块中会应用以下语法变换：

- x <- v; e 变为 v >>= (\x => e)
- v; e 变为 v >>= (_ => e)
- let x = v; e 变为 let x = v in e

IO 实现了 Monad，它使用原语函数定义。我们也可以为 Maybe 定义实现，其实现如下：

```
Monad Maybe where
    Nothing  >>= k = Nothing
    (Just x) >>= k = k x
```

通过它，我们可以定义一个将两个 Maybe Int 相加的函数，并用单子来封装错误处理：

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = do x' <- x -- 从 x 中提取值
               y' <- y -- 从 y 中提取值
               pure (x' + y') -- 二者相加
```

若 x 和 y 均可用，该函数会从二者中提取出值；若其中一个或二者均不可用，则返回 Nothing（「fail-fast 速错原则」）。Nothing 的情况通过 >>= 操作符来管理，由 do-记法来隐藏。

```
*Interfaces> m_add (Just 20) (Just 22)
Just 42 : Maybe Int
*Interfaces> m_add (Just 20) Nothing
Nothing : Maybe Int
```

模式匹配绑定

有时我们需要立即对 do-记法中某个函数的结果进行模式匹配。例如，假设我们有一个函数 readNumber，它从控制台读取一个数，若该数有效则返回 Just x 形式的值，否则返回 Nothing：

```
readNumber : IO (Maybe Nat)
readNumber = do
  input <- getLine
  if all isDigit (unpack input)
     then pure (Just (cast input))
     else pure Nothing
```

如果接着用它来编写读取两个数，若二者均无效则返回 Nothing 的函数，那么我们可能想要对 readNumber 进行模式匹配：

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers =
  do x <- readNumber
     case x of
          Nothing => pure Nothing
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
          Just x_ok => do y <- readNumber
                          case y of
                               Nothing => pure Nothing
                               Just y_ok => pure (Just (x_ok, y_ok))
```

如果有很多错误需要处理，它的嵌套层次很快就会变得非常深！我们不妨将绑定和模式匹配组合成一行。例如，我们可以对 Just x_ok 的形式进行模式匹配：

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers =
  do Just x_ok <- readNumber
     Just y_ok <- readNumber
     pure (Just (x_ok, y_ok))
```

然而问题仍然存在，我们现在忽略了 Nothing 的情况，所以该函数不再是完全的了！ 我们可以把 Nothing 的情况添加回去：

```
readNumbers : IO (Maybe (Nat, Nat))
readNumbers =
  do Just x_ok <- readNumber | Nothing => pure Nothing
     Just y_ok <- readNumber | Nothing => pure Nothing
     pure (Just (x_ok, y_ok))
```

此版本的 readNumbers 效果与初版完全一样（实际上，它是初版的语法糖， 并且会直接被翻译回初版的形式）。每条语句的第一部分（Just x_ok <- 和 Just y_ok <-）给出了首选的绑定：若能匹配，do 块的剩余部分就会继续执行。 第二部分给出了候选的绑定，其中的绑定可以有不止一个。

!-记法

在很多情况下，do-记法会让程序不必要地嗦，在将值绑定一次就立即使用的情况下尤甚， 例如前面的 m_add。此时我们可以使用更加简短的方式：

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = pure (!x + !y)
```

!expr 记法表示 expr 应当在求值后立即被隐式绑定。从概念上讲，我们可以把 ! 看做拥有以下类型的前缀函数：

```
(!) : m a -> a
```

然而请注意，它并不是一个真的函数，而是一个语法！在实践中，子表达式 !expr 会在 expr 的当前作用域内尽可能地提升，将它绑定到一个全新的名字 x， 然后用它来代替 !expr。首先表达式会按从左到右的顺序深度优先地上升。在实践中，! 记法允许我们以更直接的方式来编程，同时该记法也标出了哪些表达式为单子。

例如，表达式：

```
let y = 42 in f !(g !(print y) !x)
```

会被提升为：

```
let y = 42 in do y' <- print y
                 x' <- x
                 g' <- g y' x'
                 f g'
```

单子推导式

我们之间在 更多表达式 (éąţ 19) 一节中看到的列表推导记法其实更通用， 它可应用于任何实现了 Monad 和 Alternative 的东西：

```
interface Applicative f => Alternative (f : Type -> Type) where
    empty : f a
    (<|>) : f a -> f a -> f a
```

通常，推导式形式为 [exp | qual1, qual2, ⋯, qualn] 其中 quali 可以为：

- 一个生成式 x <- e
- 一个 **守卫式（Guard）**，它是一个类型为 Bool 的表达式
- 一个 let 绑定 let x = e

要翻译一个推导式 [exp | qual1, qual2, ⋯, qualn]，首先任何作为 **守卫式** 的限定式 qual 会使用以下函数翻译为 guard qual：

```
guard : Alternative f => Bool -> f ()
```

接着该推导式会被转换为 do-记法：

```
do { qual1; qual2; ...; qualn; pure exp; }
```

使用单子推导式，m_add 的选取（alternative）定义为：

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = [ x' + y' | x' <- x, y' <- y ]
```

1.4.5 习语括号

do-记法为串连提供了另一种写法，而习语则为 **应用** 提供了另一种写法。 本节中的记法以及大量的例子受到了 Conor McBride 和 Ross Paterson 的论文 「带作用的应用子编程」[1] 的启发。

首先，让我们回顾一下前面的 m_add。它所做的只是将一个操作符应用到两个从 Maybe Int 中提取的值。我们可以抽象出该应用：

```
m_app : Maybe (a -> b) -> Maybe a -> Maybe b
m_app (Just f) (Just a) = Just (f a)
m_app _        _        = Nothing
```

我们可以用它来编写另一种 m_add，它使用了函数应用的选取概念，带有显式的 m_app 调用：

```
m_add' : Maybe Int -> Maybe Int -> Maybe Int
m_add' x y = m_app (m_app (Just (+)) x) y
```

与其到处插入 m_app，我们不如使用习语括号（Idiom Brackets）来做这件事。 为此，我们可以像下面这样为 Maybe 提供一个 Applicative 的实现，其中 <*> 的定义方式与前面的 m_app 相同（它已在 Idris 库中定义）：

```
Applicative Maybe where
    pure = Just
```

(äÿŃéąţçżğçżŋ)

[1] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. J. Funct. Program. 18, 1 (January 2008), 1-13. DOI=10.1017/S0956796807006326 http://dx.doi.org/10.1017/S0956796807006326

(çżŋäÿŁéąţ)

```
    (Just f) <*> (Just a) = Just (f a)
    _        <*> _        = Nothing
```

按照 <*> 的实现，我们可以像下面这样使用它，其中函数应用 [| f a1 ··· an |] 会被翻译成 pure f <*> a1 <*> ··· <*> an:

```
m_add' : Maybe Int -> Maybe Int -> Maybe Int
m_add' x y = [| x + y |]
```

错误处理解释器

习语记法在定义求值器时通常很有用。McBride 和 Paterson 就为下面这样的语言描述了求值器[1]：

```
data Expr = Var String      -- 变量
          | Val Int         -- 值
          | Add Expr Expr   -- 加法
```

求值会被根据上下文将变量（表示为 String）映射为 Int 值，且可能会失败。 我们定义了数据类型 Eval 来包装求值器:

```
data Eval : Type -> Type where
     MkEval : (List (String, Int) -> Maybe a) -> Eval a
```

将求值器包装在数据类型中意味着我们之后可以为它提供接口的实现。我们首先定义一个函数， 它在求值过程中从上下文取出值。

```
fetch : String -> Eval Int
fetch x = MkEval (\e => fetchVal e) where
    fetchVal : List (String, Int) -> Maybe Int
    fetchVal [] = Nothing
    fetchVal ((v, val) :: xs) = if (x == v)
                                then (Just val)
                                else (fetchVal xs)
```

在定义该语言的求值器时，我们会应用 Eval 上下文中的函数，这样它自然会为 Eval 提供 Applicative 的实现。在 Eval 能够实现 Applicative 之前，我们必须为 Eval 实现 Functor:

```
Functor Eval where
    map f (MkEval g) = MkEval (\e => map f (g e))

Applicative Eval where
    pure x = MkEval (\e => Just x)

    (<*>) (MkEval f) (MkEval g) = MkEval (\x => app (f x) (g x)) where
        app : Maybe (a -> b) -> Maybe a -> Maybe b
        app (Just fx) (Just gx) = Just (fx gx)
        app _         _         = Nothing
```

现在就可以在求值表达式时，通过应用的习语来处理错误了:

```
eval : Expr -> Eval Int
eval (Var x)   = fetch x
eval (Val x)   = [| x |]
eval (Add x y) = [| eval x + eval y |]
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
runEval : List (String, Int) -> Expr -> Maybe Int
runEval env e = case eval e of
    MkEval envFn => envFn env
```

1.4.6 命名实现

有时我们希望一个类型可以拥有一个接口的多个实现，例如为排序或打印提供另一种方法。 为此，实现可以像下面这样 **命名**:

```
[myord] Ord Nat where
   compare Z (S n)     = GT
   compare (S n) Z     = LT
   compare Z Z         = EQ
   compare (S x) (S y) = compare @{myord} x y
```

它像往常一样声明了一个实现，不过带有显式的名字 myord。语法 compare @{myord} 会为 compare 提供显式的实现，否则它会使用 Nat 的默认实现。我们可以用它来反向排序一个 Nat 列表。给定以下列表:

```
testList : List Nat
testList = [3,4,1]
```

我们可以在 Idris 提示符中用默认的 Ord 实现来排序，之后使用命名的实现 myord:

```
*named_impl> show (sort testList)
"[sO, sssO, ssssO]" : String
*named_impl> show (sort @{myord} testList)
"[ssssO, sssO, sO]" : String
```

有时，我们也需要访问命名的父级实现。例如 Prelude 中定义的半群 Semigroup 接口:

```
interface Semigroup ty where
  (<+>) : ty -> ty -> ty
```

接着又定义了幺半群 Monoid，它用「幺元」 neutral 扩展了 Semigroup:

```
interface Semigroup ty => Monoid ty where
  neutral : ty
```

我们可以为 Nat 定义 Semigroup 与 Monoid 的两种不同的实现， 一个基于加法，一个基于乘法:

```
[PlusNatSemi] Semigroup Nat where
  (<+>) x y = x + y

[MultNatSemi] Semigroup Nat where
  (<+>) x y = x * y
```

加法的幺元为 0，而乘法的幺元为 1。因此，我们在定义 Monoid 的实现时， 保证扩展了正确的 Semigroup 十分重要。我们可以通过 using 从句来做到这一点:

```
[PlusNatMonoid] Monoid Nat using PlusNatSemi where
  neutral = 0

[MultNatMonoid] Monoid Nat using MultNatSemi where
  neutral = 1
```

using PlusNatSemi 从句指明 PlusNatMonoid 应当扩展 PlusNatSemi。

1.4.7 确定形参

当接口的形参多于一个时，通过限定形参可帮助查找实现。例如：

```
interface Monad m => MonadState s (m : Type -> Type) | m where
  get : m s
  put : s -> m ()
```

在此接口中，查找该接口的实现只需要知道 m 即可，而 s 可根据实现来确定。它通过在接口声明之后添加 | m 来声明。我们将 m 称为 MonadState 接口的 **确定形参（Determining Parameter）**，因为它是用于查找实现的形参。

1.5 模块与命名空间

Idris 程序由一组模块构成。每个模块包含一个可选的 module 声明来命名模块，一系列 import 语句用于导入其它模块，还有一组类型、接口和函数的声明与定义。例如，以下模块定义了二叉树类型 BTree（在 Btree.idr 文件中）：

```
module Btree

public export
data BTree a = Leaf
             | Node (BTree a) a (BTree a)

export
insert : Ord a => a -> BTree a -> BTree a
insert x Leaf = Node Leaf x Leaf
insert x (Node l v r) = if (x < v) then (Node (insert x l) v r)
                                   else (Node l v (insert x r))

export
toList : BTree a -> List a
toList Leaf = []
toList (Node l v r) = Btree.toList l ++ (v :: Btree.toList r)

export
toTree : Ord a => List a -> BTree a
toTree [] = Leaf
toTree (x :: xs) = insert x (toTree xs)
```

修饰符 export 和 public export 说明了哪些名称在其它模块中可见。之后会详细解释。

这里给出了一个 main 程序（在 bmain.idr 文件中），它利用 Btree 模块对列表排序：

```
module Main

import Btree

main : IO ()
main = do let t = toTree [1,8,2,7,9,3]
          print (Btree.toList t)
```

同一名称可以在多个模块中定义：名称可以通过模块名来字 **限定（Qualified）**。Btree 模块中定义的全部名称如下：

- Btree.BTree
- Btree.Leaf
- Btree.Node
- Btree.insert
- Btree.toList
- Btree.toTree

如果名称可以区分，就没必要给出完整的限定名。通过明确的限定，或者根据它们的类型， 可以消除名称的歧义。

模块名和文件名之间没有正式的联系，尽管通常会建议使用同一名字。import 语句用于引用文件名，它通过 . 来分隔路径。例如，import foo.bar 会导入文件 foo/bar.idr，其中通常会有模块声明 module foo.bar。对模块名字的唯一要求是， 包含 main 函数的主模块必须命名为 Main，而文件名无需为 Main.idr。

1.5.1 导出修饰符

Idris 允许对模块内容的可见性进行细粒度的控制。默认情况下， 模块中定义的所有名称都是私有的。这有助于最小化接口并隐藏内部细节。Idris 允许将函数、类型和接口标记为 private、export 或 public export。 它们的一般含义如下:

- private 表示完全不被导出。此为默认情况。
- export 表示顶级类型是导出的。
- public export 表示整个定义都是导出的。

更改可见性的另一限制是，定义无法引用可见性更低的名称。例如， public export 的定义无法使用私有名称, 而 export 的类型也无法使用私有名称。 这是为了避免私有名称泄露到模块的接口中。

对于函数的意义

- export 表示导出类型
- public export 表示导出类型和定义, 导出之后的定义可被使用。 换言之, 定义本身被认为是模块接口的一部分。名字 public export 比较长是为了让你三思而后行。

注解: Idris 中的同义类型可通过编写函数来创建。在为模块设置可见性时， 如果同义类型要在模块外部使用，那么将它们声明为 public export 可能会更好。 否则，Idris 会无法获知该类型与谁同义。

既然 public export 意味着函数的定义是导出的，那么该函数的定义实际上也就成为了模块 API 的一部分。因此，除非你确实想要导出函数的完整定义，否则最好避免将其声明为 public export 。

对于数据类型的涵义

对于数据类型，其涵义如下:

- export 的类型构造器是导出的

- public export 的类型构造器和数据构造器是导出的

对于接口的涵义

对于接口，其涵义如下：

- export 的接口名字是导出的
- public export 的接口名、方法名和默认定义都是导出的

%access 指令

默认的导出模式可以通过 %access 指令更改，例如：

```
module Btree

%access export

public export
data BTree a = Leaf
             | Node (BTree a) a (BTree a)

insert : Ord a => a -> BTree a -> BTree a
insert x Leaf = Node Leaf x Leaf
insert x (Node l v r) = if (x < v) then (Node (insert x l) v r)
                                   else (Node l v (insert x r))

toList : BTree a -> List a
toList Leaf = []
toList (Node l v r) = Btree.toList l ++ (v :: Btree.toList r)

toTree : Ord a => List a -> BTree a
toTree [] = Leaf
toTree (x :: xs) = insert x (toTree xs)
```

在这种情况下，没有访问修饰符的任何函数均可以导出为 export，而不是 private。

内部模块 API 的传递

另外，通过对 import 使用 public 修饰符，可将模块内导入的模块再次导出。 例如：

```
module A

import B
import public C
```

模块 A 会导出名字 a, 以及模块 C 中所有公共或抽象的名称， 但无法再从模块 B 中导出任何东西。

1.5.2 显式命名空间

在定义模块的同时，也会隐式定义一个命名空间。然而，该命名空间也可以 **显式** 给出。当你想在同一模块内重载名称时，它会非常有用：

```
module Foo

namespace x
  test : Int -> Int
  test x = x * 2

namespace y
  test : String -> String
  test x = x ++ x
```

这个（明显人为设计的）模块使用完整的限定名 `Foo.x.test` 和 `Foo.y.test` 定义了两个函数，二者可以根据函数类型来消歧义：

```
*Foo> test 3
6 : Int
*Foo> test "foo"
"foofoo" : String
```

1.5.3 形参化的块

一组函数的多个参数可通过 `parameters` 声明进行形参化（Parameterise），例如：

```
parameters (x : Nat, y : Nat)
  addAll : Nat -> Nat
  addAll z = x + y + z
```

`parameters` 形参块的作用是为块中的每个函数、类型和数据构造器添加形参声明。 具体来说，就是将形参添加到参数列表的前面。在此块外，形参必须显式地给定。 因此在 REPL 中调用 `addAll` 函数时，它会拥有以下函数声明：

```
*params> :t addAll
addAll : Nat -> Nat -> Nat -> Nat
```

以及以下定义：

```
addAll : (x : Nat) -> (y : Nat) -> (z : Nat) -> Nat
addAll x y z = x + y + z
```

形参块可以嵌套。在为所有类型和数据构造器显式地添加形参时， 块中也可包含数据声明。它们也可以是带有隐式参数的依赖类型：

```
parameters (y : Nat, xs : Vect x a)
  data Vects : Type -> Type where
    MkVects : Vect y a -> Vects a

  append : Vects a -> Vect (x + y) a
  append (MkVects ys) = xs ++ ys
```

为了在形参块外使用 `Vects` 或者 `append`，我们必须给出参数 `xs` 和 `y`。 在这里，我们可以用占位符来代表类型检查器能推断出的值：

```
*params> show (append _ _ (MkVects _ [1,2,3] [4,5,6]))
"[1, 2, 3, 4, 5, 6]" : String
```

提示： 形参（Parameter）与实参（Argument）

在数学中，对于函数 `f(x)`，`f` 为函数名，`x` 则称作函数 `f` 的形式参数（Parameter），简称形参；在函数应用时，传输函数的参数则称作实际参数 （Argument），简称实参。例如 `f(2)` 中的 `2` 即为传入 `f(x)` 的实参。

在英文原文中，有时并不明确区分形参与实参，一般统译作「参数」。只有当需要明确区分时， 才分别译作「形参」与「实参」。

1.6 包

Idris 包括一套简单的构建系统，它会根据已命名的包描述文件来构建包以及可执行文件。 描述文件可以配合 Idris 编译器来管理开发过程。

1.6.1 包的描述

包的描述包含以下内容:

- 包头，由关键字 `package` 后跟一个包名构成。包名可以是任何有效的 Idris 标识符。 iPKG 格式也可包含一个带引号的 `version`，它接受任何有效的文件名。
- 描述包内容的字段，`<field> = <value>`

其中至少有一个 modules 字段，对应的值为逗号分隔的模块列表。例如， 给定一个 Idris 包 `maths`，包含 `Maths.idr` 、`Maths.NumOps.idr` `Maths.BinOps.idr` 和 `Maths.HexOps.idr` 几个模块，其相应的包文件应为:

```
package maths

modules = Maths
        , Maths.NumOps
        , Maths.BinOps
        , Maths.HexOps
```

其它包文件的例子可以在主 Idris 仓库的 `libs` 目录中以及 第三方库 中找到。

1.6.2 使用包文件

Idris 本身是知晓包的，还有一些专门的命令来帮助构建、安装以及清除包。 例如，对于前面给出的 `maths` 包，我们可以像下面这样使用 Idris:

- `idris --build maths.ipkg` 会构建包中的所有模块。
- `idris --install maths.ipkg` 会安装包，使其可以被其它 Idris 库和程序访问。
- `idris --clean maths.ipkg` 会删除构建时候产生的所有代码及可执行文件。

一旦 maths 包安装完成，命令行选项 `--package maths` （简写为 `-p maths`）就可以使用了。例如:

```
idris -p maths Main.idr
```

1.6.3 测试 Idris 包

集成的构建系统包含简单的测试框架。该框架会收集 tests 下的 ipkg 文件中列出的函数。所有的测试函数必须返回 IO ()。

当输入 idris --testpkg yourmodule.ipkg 后，构建系统会列出单个 main 函数下的 tests 函数，并在你机器上的全新环境中创建一个临时文件。它会将该临时文件编译成可执行文件并执行它。

测试本身负责报告它们的成功或失败。测试函数通常用 putStrLn 报告测试结果。测试框架不强加任何报告标准，因此也不会合计测试结果。

我们以下面的函数列表为例，它们在样本包 maths 中名为 NumOps 的模块内定义：

```
module Maths.NumOps

%access export -- to make functions under test visible

double : Num a => a -> a
double a = a + a

triple : Num a => a -> a
triple a = a + double a
```

一个限定名为 Test.NumOps 的简单测试模块可声明为：

```
module Test.NumOps

import Maths.NumOps

%access export  -- to make the test functions visible

assertEq : Eq a => (given : a) -> (expected : a) -> IO ()
assertEq g e = if g == e
    then putStrLn "Test Passed"
    else putStrLn "Test Failed"

assertNotEq : Eq a => (given : a) -> (expected : a) -> IO ()
assertNotEq g e = if not (g == e)
    then putStrLn "Test Passed"
    else putStrLn "Test Failed"

testDouble : IO ()
testDouble = assertEq (double 2) 4

testTriple : IO ()
testTriple = assertNotEq (triple 2) 5
```

函数 assertEq 和 assertNotEq 分别用于运行预期为通过和失败的相等性测试。实际上的测试为 testDouble 和 testTriple，它们在 maths.ipkg 文件中的声明如下

```
package maths

modules = Maths.NumOps
        , Test.NumOps

tests = Test.NumOps.testDouble
      , Test.NumOps.testTriple
```

测试框架可通过 idris --testpkg maths.ipkg 命令调用：

```
> idris --testpkg maths.ipkg
Type checking ./Maths/NumOps.idr
Type checking ./Test/NumOps.idr
Type checking /var/folders/63/np5g0d5j54x1s0z12rf41wxm0000gp/T/idristests144128232716531729.idr
Test Passed
Test Passed
```

注意当我们使用 `assertEq` 和 `assertNoEq` 函数测试时，它们是如何通过打印 `Test Passed` 来报告测试成功的。

1.6.4 在 Atom 中使用包依赖

如果你在使用 Atom 编辑器，并且依赖了另一个包，例如 `import Lightyear` 或者 `import Pruviloj`，那么你需要让 Atom 知道它应该加载什么。最简单的方式是通过 .ipkg 文件来完成。本教程在下一节中会描述一个 ipkg 文件通常会包含哪些内容， 不过这里先给出此平凡例子的简单步骤：

- 创建一个 myProject 文件夹。
- 添加一个 myProject.ipkg 文件，包含如下代码：

```
package myProject

pkgs = pruviloj, lightyear
```

- 在 Atom 中，使用文件菜单打开 myProject 文件夹。

更多信息

更多详情，包括可用字段的完整列表，可在参考手册的 *Packages* (ếąţ 151) 中找到。

1.7 示例：良类型的解释器

在本节中，我们使用已经见过的特性来编写一个更大的例子：一个简单的函数式语言解释器， 带有变量、函数应用、二元运算符和 `if...then...else` 构造。 我们用依赖类型系统来保证所有能够表示的程序都是良类型的。

1.7.1 语言的表示

首先，我们来定义语言中的类型。我们有整数、布尔和函数，用 `Ty` 来表示：

```
data Ty = TyInt | TyBool | TyFun Ty Ty
```

我们可以写一个函数将上面的表示转换为具体的 Idris 类型——要注意类型是一等的， 所以可以像其它值一样参与计算：

```
interpTy : Ty -> Type
interpTy TyInt       = Integer
interpTy TyBool      = Bool
interpTy (TyFun a t) = interpTy a -> interpTy t
```

我们将定义一种表示方法，它只能表示良类型的程序。我们通过表达式的类型 与 其局部变量的类型（上下文）来索引它。上下文可以用 `Vect` 数据类型来表达。 由于上下文需要被经常使用，因此它会被

表示为隐式参数。为此，我们在 using 块中定义了所有需要的东西。（注意后文中的代码需要缩进才能在 using 块中使用）：

```
using (G:Vect n Ty)
```

表达式通过局部变量的类型和表达式自身的类型来索引：

```
data Expr : Vect n Ty -> Ty -> Type
```

表达式的完整表示为：

```
data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type where
    Stop : HasType FZ (t :: G) t
    Pop  : HasType k G t -> HasType (FS k) (u :: G) t

data Expr : Vect n Ty -> Ty -> Type where
    Var : HasType i G t -> Expr G t
    Val : (x : Integer) -> Expr G TyInt
    Lam : Expr (a :: G) t -> Expr G (TyFun a t)
    App : Expr G (TyFun a t) -> Expr G a -> Expr G t
    Op  : (interpTy a -> interpTy b -> interpTy c) ->
          Expr G a -> Expr G b -> Expr G c
    If  : Expr G TyBool ->
          Lazy (Expr G a) ->
          Lazy (Expr G a) -> Expr G a
```

上述代码使用了 Idris 标准库中的 Vect 和 Fin 类型。它们不在 Prelude 中， 因此我们需要导入它们：

```
import Data.Vect
import Data.Fin
```

由于表达式通过其类型来索引，我们可以直接从构造器的定义中得出该语言的类型规则。 下面我们来逐一观察每个构造器。

我们为变量使用了不带名字的表示法 - 它们以 **de Bruijn** 法来索引。 变量以它们在上下文中从属关系的证明来表示： HasType i G T 是变量 i 在上下文 G 中拥有类型 T 的证明。它的定义如下：

```
data HasType : (i : Fin n) -> Vect n Ty -> Ty -> Type where
    Stop : HasType FZ (t :: G) t
    Pop  : HasType k G t -> HasType (FS k) (u :: G) t
```

我们把 **Stop** 当做最近定义的变量有良类型的证明，而 **Pop n** 则证明了 如果第 n 个最近定义的变量是良类型的，那么第 n+1 个也是。在实践中， 这意味着我们通过 Var 构造器，使用 Stop 来引用最近定义的变量， Pop Stop 来引用下一个，以此类推。

```
Var : HasType i G t -> Expr G t
```

所以，在表达式 \x. \y. x y 中，变量 x 的 de Bruijn 索引为 1， 可表示为 Pop Stop ；变量 y 的 de Bruijn 索引为 0，可表示为 Stop 。 我们通过计算定义和使用之间 λ 的数量来查找它们。

值携带了整数的具体表示：

```
Val : (x : Integer) -> Expr G TyInt
```

λ 用于创建函数。在类型为 a -> t 的函数的作用域内，有一个新的类型为 a 的局部变量，它用上下文索引来表示：

```
Lam : Expr (a :: G) t -> Expr G (TyFun a t)
```

函数应用接受一个从 a 到 t 的函数和一个类型为 a 的值，产生一个类型为 t 的值。

```
App : Expr G (TyFun a t) -> Expr G a -> Expr G t
```

我们允许任意的二元运算符，运算符的类型会告诉我们参数的类型：

```
Op : (interpTy a -> interpTy b -> interpTy c) ->
     Expr G a -> Expr G b -> Expr G c
```

最后，`If` 表达式根据给定的布尔值来做出选择，每个分支必须有相同的类型。 我们将对分支使用惰性求值，只有被选择的分支才会被求值：

```
If : Expr G TyBool ->
     Lazy (Expr G a) ->
     Lazy (Expr G a) ->
     Expr G a
```

1.7.2 编写解释器

当我们对一个 `Expr` 求值时，我们需要其作用域内的值及其类型。 `Env` 是一个根据其作用域中的类型来索引的环境。尽管环境和局部变量类型的向量有着很强联系， 它也只是列表的另一种形式。我们使用了一般的 `::` 和 `Nil` 构造器， 这样就能使用一般的列表语法了。给定一个变量在上下文中定义的证明，我们可以从环境中得到一个值：

```
data Env : Vect n Ty -> Type where
    Nil  : Env Nil
    (::) : interpTy a -> Env G -> Env (a :: G)

lookup : HasType i G t -> Env G -> interpTy t
lookup Stop    (x :: xs) = x
lookup (Pop k) (x :: xs) = lookup k xs
```

有了上述内容，解释器就是一个根据特定环境将 `Expr` 转换成一个具体的 Idris 值的函数了：

```
interp : Env G -> Expr G t -> interpTy t
```

作为参考，解释器的完整定义如下。对于每一个构造器，我们把它转换成对应的 Idris 值：

```
interp env (Var i)     = lookup i env
interp env (Val x)     = x
interp env (Lam sc)    = \x => interp (x :: env) sc
interp env (App f s)   = interp env f (interp env s)
interp env (Op op x y) = op (interp env x) (interp env y)
interp env (If x t e)  = if interp env x then interp env t
                                         else interp env e
```

我们来逐一观察每一种情况。对于一个变量，我们只要从环境中找出它：

```
interp env (Var i) = lookup i env
```

对于一个值，我们只需要返回其实际的表达方式：

```
interp env (Val x) = x
```

λ 则比较有意思。我们构造了一个解释 λ 内部作用域的函数，但其环境中带有一个新的值。 因此，目标语言的函数可以用如下的方法来转换成 Idris 函数：

```
interp env (Lam sc) = \x => interp (x :: env) sc
```

对于函数应用，我们解释函数及其参数，并直接将函数应用于参数。我们知道，根据其类型， 解释 f 必定会得到一个函数：

```
interp env (App f s) = interp env f (interp env s)
```

运算符和条件分支同样会转换成等价的 Idris 构造。对于运算符，我们将函数直接应用到 操作数上；对于 If，我们直接使用 Idris 的 if...then...else 构造。

```
interp env (Op op x y) = op (interp env x) (interp env y)
interp env (If x t e)  = if interp env x then interp env t
                                         else interp env e
```

1.7.3 测试

我们可以编写一个简单的测试函数。首先，将两个输入值加起来，\x. \y. y + x 可写成如下形式：

```
add : Expr G (TyFun TyInt (TyFun TyInt TyInt))
add = Lam (Lam (Op (+) (Var Stop) (Var (Pop Stop))))
```

更有趣的是阶乘函数 fact，如 \x. if (x == 0) then 1 else (fact (x-1) * x)， 可写成如下形式：

```
fact : Expr G (TyFun TyInt TyInt)
fact = Lam (If (Op (==) (Var Stop) (Val 0))
               (Val 1)
               (Op (*) (App fact (Op (-) (Var Stop) (Val 1)))
                       (Var Stop)))
```

1.7.4 运行

作为结束，我们编写一个 main 程序，它根据用户的输入来解释阶乘函数：

```
main : IO ()
main = do putStr "Enter a number: "
          x <- getLine
          printLn (interp [] fact (cast x))
```

此处的 cast 是一个被重载的函数，在可能的情况下，它将值转为另一种类型。在这里， 它将字符串转换成了整数，在输入不合法时返回 0 。在 Idris 的交互式环境运行中这个程序 会产生如下结果：

```
$ idris interp.idr
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 1.3.1
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Type checking ./interp.idr
*interp> :exec
Enter a number: 6
720
*interp>
```

题外话：cast

Prelude 中定义了一个 Cast 接口来实现类型之间的转换：

```
interface Cast from to where
    cast : from -> to
```

这是一个 **多参数** 接口，定义了转换的源类型和目标类型。在转换被应用的地方， 类型检查器必须能够推导出 **两个** 参数。原语类型间有意义的转换均已定义。

1.8 视角与「with」规则

1.8.1 依赖模式匹配

由于类型可以依赖于值，因此某些参数的形式可根据其它参数的值来确定。例如， 如果我们写出 (++) 的长度隐式参数，就会看出它的形式由向量是否为空来决定：

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) {n=Z}   []        ys = ys
(++) {n=S k} (x :: xs) ys = x :: xs ++ ys
```

假如 n 在 [] 的情况下是一个后继，或者在 :: 的情况下为零， 那么它的定义就不是良类型的。

1.8.2 with 规则：匹配中间值

我们经常需要匹配中间计算（Intermediate Computation）的结果。Idris 受到 Epigram[1] 中视角（View）的启发，为此提供了一种构造，即 with 规则， 它考虑到在依赖型的语言中匹配值会影响我们对其它值的形式的了解。在最简单的形式中， with 规则会为正在定义的函数附加一个额外的参数。

我们已经见过向量过滤函数了。这次我们用 with 来定义它：

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
filter p [] = ( _ ** [] )
filter p (x :: xs) with (filter p xs)
  filter p (x :: xs) | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

在这里，with 从句能让我们解构（deconstruct）“filter p xs“ 的结果。 该视角精化的参数模式 filter p (x :: xs) 位于 with 从句的下方， 之后是一条竖线 | 后面跟着解构的中间结果 (_ ** xs')。如果该视角精化的参数模式与原函数的参数模式相同，那么 | 左侧的就是多余的，可以省略：

```
filter p (x :: xs) with (filter p xs)
  | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

提示： 这个例子并不好，它省去了大量的细节，包括什么是视角，如何用视角来解构数据， 以及 with 从句帮你做了什么。参数 p 与结果 (p ** Vect p a) 中的 p 也毫无关系，这点可能会对初学者造成困扰。有条件的读者可参考 Type-Driven Development with Idris 第二部分中的 Views: extending pattern matching 一章，或者重新思考接下来这个例子。

---

with 从句还可以嵌套：

[1] Conor McBride and James McKinna. 2004. The view from the left. J. Funct. Program. 14, 1 (January 2004), 69-111. https://doi.org/10.1017/S0956796803004829

```
foo : Int -> Int -> Bool
foo n m with (succ n)
  foo _ m | 2 with (succ m)
    foo _ _ | 2 | 3 = True
    foo _ _ | 2 | _ = False
  foo _ _ | _ = False
```

如果中间计算本身具有依赖类型，那么其结果会影响其它参数的形式 — 我们可以通过测试 附加参数来了解一个值的形式。在这类情况下，视角精化的参数模式必须是显式的。 例如，一个 Nat 不是偶数就是奇数。如果它是偶数，那么它就等于两个 Nat 之和。 否则，它就是两个 Nat 之和再加一：

```
data Parity : Nat -> Type where
   Even : Parity (n + n)
   Odd  : Parity (S (n + n))
```

我们将 Parity 称为 Nat 的一个 **视角（View）**。它拥有一个 **覆盖函数（Covering Function）** 来测试 Nat 的奇偶性并构造相应的谓词：

```
parity : (n:Nat) -> Parity n
```

我们之后再看 parity 的定义。我们可以用它配合 with 规则来编写一个函数， 它将一个自然数转换成一系列二进制数字（False 为 0，True 为 1，低位在前）：

```
natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k with (parity k)
   natToBin (j + j)     | Even = False :: natToBin j
   natToBin (S (j + j)) | Odd  = True  :: natToBin j
```

parity k 的值影响了 k 的形式，因为 parity k 的结果取决于 k。 因此，除了 | 右侧的中间计算结果（Even 和 Odd）的模式外， 我们还可以写出该结果如何影响 | 左侧的其它模式。即：

- 当 parity k 求值为 Even 时，我们可以根据 Even 构造器的定义 Parity (n + n)，将原始参数 k 精化为模式 (j + j)。这样 (j + j) 就代替了 | 左侧的 k，而 Even 构造器则出现在右侧。精化模式中的自然数 j 会被用在 = 符号的两侧。
- 否则，当 parity k 求值为 Odd 时，根据 Odd 构造器的定义 Parity (S (n + n))，原始参数 k 会被精化为模式 S (j + j)， 它和 Odd 会出现在 | 的两侧，同样自然数 j 会被用在 = 符号的两侧。

注意，在精化模式的两个 j 之间有一个函数 (+)，它被允许是因为附加参数 已经确定了此模式的形式。

我们会在下一节 实践中的证明 (éąţ 42) 中回到 parity 上来完成它的定义。

1.8.3 with 与证明

要使用依赖模式匹配进行定理证明，有时必须根据匹配模式显式地构造出证明结果。为此，你可以为 with 从句加上 proof p 后缀，由模式匹配生成的证明会被命名为 p 并加入到作用域中。 例如：

```
data Foo = FInt Int | FBool Bool

optional : Foo -> Maybe Int
optional (FInt x) = Just x
optional (FBool b) = Nothing

isFInt : (foo:Foo) -> Maybe (x : Int ** (optional foo = Just x))
isFInt foo with (optional foo) proof p
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
isFInt foo | Nothing = Nothing           -- here, p : Nothing = optional foo
isFInt foo | (Just x) = Just (x ** Refl) -- here, p : Just x = optional foo
```

1.9 定理证明

1.9.1 相等性

Idris 可以声明命题的相等性，陈述并证明有关程序的定理。 相等性是内建的，不过这里给出其概念上的定义：

```
data (=) : a -> b -> Type where
   Refl : x = x
```

任何类型的任何值之间都可以判断相等性，然而构造相等性证明的唯一方式就是值确实相等。例如：

```
fiveIsFive : 5 = 5
fiveIsFive = Refl

twoPlusTwo : 2 + 2 = 4
twoPlusTwo = Refl
```

1.9.2 空类型

存在一个空类型 ⊥，它没有构造器。因此，在不使用部分定义或一般递归函数的情况下， 无法构造出空类型的元素（详见 完全性检查 (éąţ 44) 一节）。 由此，我们可以用空类型来证明某些命题是不可能的，例如零不可能是任何自然数的后继：

```
disjoint : (n : Nat) -> Z = S n -> Void
disjoint n p = replace {P = disjointTy} p ()
  where
    disjointTy : Nat -> Type
    disjointTy Z = ()
    disjointTy (S k) = Void
```

我们不必太关心该函数如何工作 — 本质上，它应用库函数 `replace`， 根据相等证明来变换谓词。在本例中，我们利用某些东西无法存在的证明， 将一个可以存在的类型（即空元组）的值，变换成了一个无法存在的类型的值。

一旦拥有了空类型的元素，我们就能证明任何东西。`void` 在库中定义，用于辅助反证法。

```
void : Void -> a
```

1.9.3 简单定理

当对依赖类型进行类型检查时，该类型就会被 **规范化（Normalized）**。 比如我们想要证明以下关于 `plus` 的归约行为（Reduction Behaviour）的定理：

```
plusReduces : (n:Nat) -> plus Z n = n
```

我们将该定理的陈述写成了类型，就像写出程序的类型一样。实际上证明和程序之间并没有本质的区

别。 就我们所关注的而言，证明不过是个程序，只是它的类型精确到足以保证满足我们关心的特殊性质。

我们不会在这里深入细节，Curry-Howard 同构[1] 解释了这种关系。该证明很平凡， 因为 plus 的定义将 plus Z n 规范化成了 n：

```
plusReduces n = Refl
```

如果我们换种方式证明该论点，那就有点难了，因为加法是对第一个参数递归定义的。 该证明同样也可以在 plus 的第一个参数 n 上递归：

```
plusReducesZ : (n:Nat) -> n = plus n Z
plusReducesZ Z = Refl
plusReducesZ (S k) = cong (plusReducesZ k)
```

cong 是库中定义的一个函数，它指明相等性也适用于函数应用：

```
cong : {f : t -> u} -> a = b -> f a = f b
```

我们可以对加法在后继上的递归行为做同样的事情：

```
plusReducesS : (n:Nat) -> (m:Nat) -> S (plus n m) = plus n (S m)
plusReducesS Z m = Refl
plusReducesS (S k) m = cong (plusReducesS k m)
```

即便对于如此平凡的定理，一口气构造出证明也有点棘手。当事情变得更复杂时， 需要考虑的就太多了，此时根本无法在这种「批量模式」下构造证明。

Idris 提供了交互式编辑的能力，它可以帮助构造证明。关于在编辑器中交互式构造证明的更多详情，参见 *Theorem Proving* (éąţ 136)。

1.9.4 实践中的证明

证明定理的需求可在实践中自然产生。例如，在之前的 视角与「*with*」规则 (éąţ 39) 一节中， 我们用函数 parity 实现了 natToBin：

```
parity : (n:Nat) -> Parity n
```

然而，我们并未提供 parity 的定义。我们可能觉得它看起来会像下面这样：

```
parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = Even {n=S j}
  parity (S (S (S (j + j)))) | Odd  = Odd {n=S j}
```

然而，它会因类型错误而无法编译：

```
在按照期望的类型
        Parity (S (S (j + j)))
检查 views.parity 中 with 块的右侧时

发现

                                                    (äÿŃéąţçżğçżŋ)
```

[1] Timothy G. Griffin. 1989. A formulae-as-type notion of control. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '90). ACM, New York, NY, USA, 47-58. DOI=10.1145/96709.96714 http://doi.acm.org/10.1145/96709.96714

(çżŋäÿŁéąţ)

```
        Parity (S j + S j)  (Even 的类型)
与
        Parity (S (S (plus j j)))  (期望的类型)
的类型不匹配
```

问题在于，在 Even 的类型中规范化 S j + S j 并不能得到我们需要的 Parity 右侧的类型。我们知道 S (S (plus j j)) 等于 S j + S j，但需要向 Idris 证明它。我们可以先为该定义挖一些 **坑** （见 坑 (éąţ 7)）：

```
parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = let result = Even {n=S j} in
                                      ?helpEven
  parity (S (S (S (j + j)))) | Odd  = let result = Odd {n=S j} in
                                      ?helpOdd
```

检查 helpEven 的类型会告诉我们需要为 Even 的情况证明什么：

```
  j : Nat
  result : Parity (S (plus j (S j)))
--------------------------------------
helpEven : Parity (S (S (plus j j)))
```

由此我们可以编写一个辅助函数，将它的类型 **重写** 为需要的形式：

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = rewrite plusSuccRightSucc j j in p
```

rewrite ... in 语法允许你根据相等性证明来改写它，以此改变表达式需要的类型。 在这里，我们使用了 plusSuccRightSucc，其类型如下：

```
plusSuccRightSucc : (left : Nat) -> (right : Nat) -> S (left + right) = left + S right
```

我们可以在 helpEven 的右侧挖个坑来看到 rewrite 的效果，然后一步一步地做。 我们从下面开始：

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = ?helpEven_rhs
```

先查看一下 helpEven_rhs 的类型：

```
  j : Nat
  p : Parity (S (plus j (S j)))
--------------------------------------
helpEven_rhs : Parity (S (S (plus j j)))
```

然后通过应用 plusSuccRightSucc j j 来进行 rewrite 重写， 它会给出等式 S (j + j) = j + S j，从而在类型中用 j + S j 取代 S (j + j) （在这里是 S (plus j j)，它由 S (j + j) 规约而来）：

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = rewrite plusSuccRightSucc j j in ?helpEven_rhs
```

现在检查 helpEven_rhs 的类型会告诉我们发生了什么，包括刚才所用的等式的类型 （即 _rewrite_rule 的类型）：

```
  j : Nat
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
  p : Parity (S (plus j (S j)))
  _rewrite_rule : S (plus j j) = plus j (S j)
--------------------------------------
helpEven_rhs : Parity (S (plus j (S j)))
```

对 Odd 的情况使用 rewrite 和另一个辅助函数，我们可以完成 parity：

```
helpEven : (j : Nat) -> Parity (S j + S j) -> Parity (S (S (plus j j)))
helpEven j p = rewrite plusSuccRightSucc j j in p

helpOdd : (j : Nat) -> Parity (S (S (j + S j))) -> Parity (S (S (S (j + j))))
helpOdd j p = rewrite plusSuccRightSucc j j in p

parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = helpEven j (Even {n = S j})
  parity (S (S (S (j + j)))) | Odd  = helpOdd j (Odd {n = S j})
```

rewrite 的完整细节超出了本入门教程的范围，不过定理证明教程 （见 *Theorem Proving* (éąţ 136)）中覆盖了它。

1.9.5 完全性检查

如果我们真的想要信任我们的证明，它们定义为 **全** 函数是十分重要的 — 也就是说， 一个函数为所有可能的输入情况定义，并且保证会终止。不然我们就能构造出一个空类型的元素， 以它开始我们可以证明任何东西：

```
-- 利用部分定义的「hd」
empty1 : Void
empty1 = hd [] where
    hd : List a -> a
    hd (x :: xs) = x

-- 不会终止
empty2 : Void
empty2 = empty2
```

Idris 会在内部检查所有函数的完全性，我们可在提示符中用 :total 命令来检查。 我们会看到上面的两个定义都是不完全的：

```
*Theorems> :total empty1
可能不完全，由于： empty1#hd
    不完全，因为有遗漏的情况
*Theorems> :total empty2
可能不完全，由于递归路径 empty2
```

注意这里用了「可能」一词 — 由于停机问题的不可判定性，完全性检查当然永远无法确定。 因此，该检查是保守的。我们也可以将函数标记为完全的，使其在完全性检查失败时产生编译期错误：

```
total empty2 : Void
empty2 = empty2
```

```
对 ./theorems.idr 进行类型检查
theorems.idr:25:empty2 可能不完全，由于递归路径 empty2
```

令人欣慰的是，我们在 空类型 (éąţ 41) 一节中对零与后继构造器分立的证明是完全的：

```
*theorems> :total disjoint
Total
```

完全性检查必然是保守的。要被记录为完全的，函数 `f` 必须：

- 覆盖所有可能的输入
- 是 **良基** 的 — 即，当一系列（可能互相）递归的调用再次到达 `f` 时，它必须能够表明其参数之一已经递减。
- 使用的数据类型必须 **严格为正（strictly positive）**
- 没有调用任何不完全的函数

完全性的指令与编译器参数

默认情况下，Idris 允许所有良类型的定义，无论是否完全。然而在理想情况下， 函数总是要尽可能地完全，因为这能保证它们可以在有限时间内， 对于所有可能的输入提供一个结果。我们可以要求函数是完全的，通过以下两种方式之一：

- 使用 `--total` 编译器参数。
- 为源文件添加 `%default total` 指令。在这之后的所有定义都会要求为完全的， 除非显式地标记为 `partial`。

在 `%default total` 声明 **之后** 的所有函数都会被要求是完全的。与此相应， `%default partial` 声明之后的要求则被放宽。

最后，编译器参数 `--warnpartial` 会为任何未声明完全性的偏函数打印一个警告。

完全性检查的问题

请注意，完全性检查器并不完美！首先，由于停机问题的不可判定性，它必然是保守的， 因此一些 **确实完全** 的程序不会被检测为完全的。其次，当前实现投入的精力有限， 因此它仍然有可能将不完全的函数当作完全的。你的证明请先不要依赖它！

完全性的提示

有时你确信一个程序是完全的，但 Idris 不这么认为，此时可以对检查器给出提示， 以此来给出终止参数的详情。检查器会确保所有的递归调用链最终都能导致其中一个参数递减到基本情况， 然而有些情况很难辨别。例如，以下定义无法被检查为 `total`，因为检查器无法确定 `filter (< x) xs` 一定小于 `(x :: xs)`：

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

prelude 中定义的 `assert_smaller` 旨在解决这个问题：

```
assert_smaller : a -> a -> a
assert_smaller x y = y
```

它简单地求值成第二个参数，但也会向完全性检查器断言 y 在结构上小于 x。 当检查器自己无法解决时，它可用于解释完全性的推理。现在上面的例子可重写为：

```
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)))
```

表达式 assert_smaller (x :: xs) (filter (<= x) xs) 断言 filter 的结果总是小于模式 (x :: xs) 。

在更极端的情况下，函数 assert_total 能将一个表达式标为总是完全的：

```
assert_total : a -> a
assert_total x = x
```

通常，你应当避免使用该函数，不过在对原语进行推理，或者在对外部定义的， 完全性可被外部参数展示的函数（例如 C 库中的）进行推理时，它会非常有用。

1.10 临时定义

在依赖类型编程中，有时类型检查器需要的类型会和我们编写的程序的类型不同 （在这里是它们的形式不同），然而尽管如此，它们在证明上依然是等价的。 例如，回想一下 parity 函数：

```
data Parity : Nat -> Type where
   Even : Parity (n + n)
   Odd  : Parity (S (n + n))
```

我们要把它实现为：

```
parity : (n:Nat) -> Parity n
parity Z     = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j)))     | Even = Even {n=S j}
  parity (S (S (S (j + j)))) | Odd  = Odd {n=S j}
```

它简单地指定了零为偶数，一为奇数，然后递归地说明了 k+2 的奇偶性与 k 相同。 显式地标出 n 是奇数还是偶数对于类型推断来说是必须的。然而，类型检查器却拒绝了它：

```
viewsbroken.idr:12:10: 在解析 ViewsBroken.parity 的右侧时：
    Parity (plus (S j) (S j))
与
    Parity (S (S (plus j j)))
的类型不匹配

具体为：
        plus (S j) (S j)
    与
        S (S (plus j j))
    的类型不匹配
```

类型检查器告诉我们 (j+1)+(j+1) 和 2+j+j 无法规范化为相同的值。这是因为 plus 是在第一个参数上递归定义的，而在第二个值中，有一个后继符号作用在第二个参数上， 因此它无法帮助归约。这些值明显相等 — 不过我们要如何重写程序来修复此问题?

1.10.1 临时定义

临时定义（Provisional Definition） 允许我们推迟证明的细节以帮助解决此问题。它主要有两个作用：

- 在 **成型（Prototyping）** 时，它可在所有的证明细节结束前测试程序。
- 在 **阅读** 程序时，推迟证明的细节通常会让过程更清晰，避免读者从底层算法中分心。

临时定义的写法和普通定义相同，只是它以 `?=` 而非 `=` 引入右式。我们将 `parity` 定义为：

```
parity : (n:Nat) -> Parity n
parity Z = Even {n=Z}
parity (S Z) = Odd {n=Z}
parity (S (S k)) with (parity k)
  parity (S (S (j + j))) | Even ?= Even {n=S j}
  parity (S (S (S (j + j)))) | Odd ?= Odd {n=S j}
```

当写成这种形式时，Idris 不会报告类型错误，而是在定理中挖一个坑，以此来修正类型错误。Idris 会告诉我们有两个证明义务，其名字根据模块和函数名生成：

```
*views> :m
Global holes:
        [views.parity_lemma_2,views.parity_lemma_1]
```

其中第一个坑的类型如下：

```
*views> :p views.parity_lemma_1

---------------------------------- (views.parity_lemma_1) --------
{hole0} : (j : Nat) -> (Parity (plus (S j) (S j))) -> Parity (S (S (plus j j)))

-views.parity_lemma_1>
```

它的两个参数为 `j`，一个是模式匹配作用域中的变量，另一个是 `value`，它是我们在临时定义右侧给出的值。我们的目标是重写类型以便让我们能使用该值。我们可以用 Prelude 中的以下定理来达到此目的：

```
plusSuccRightSucc : (left : Nat) -> (right : Nat) ->
  S (left + right) = left + (S right)
```

还要再用 `compute` 来展开 `plus` 的定义：

```
-views.parity_lemma_1> compute


---------------------------------- (views.parity_lemma_1) --------
{hole0} : (j : Nat) -> (Parity (S (plus j (S j)))) -> Parity (S (S (plus j j)))
```

在应用 `intros` 之后，我们有：

```
-views.parity_lemma_1> intros

  j : Nat
  value : Parity (S (plus j (S j)))
---------------------------------- (views.parity_lemma_1) --------
{hole2} : Parity (S (S (plus j j)))
```

接着，我们对称地对 `j` 和 `j` 应用 `plusSuccRightSucc` 重写规则，它会给出：

```
-views.parity_lemma_1> rewrite sym (plusSuccRightSucc j j)

  j : Nat
  value : Parity (S (plus j (S j)))
-------------------------------- (views.parity_lemma_1) --------
{hole3} : Parity (S (plus j (S j)))
```

sym 是一个在库中定义的函数，它可以反转重写的顺序：

```
sym : l = r -> r = l
sym Refl = Refl
```

我们可以用 trivial 策略来完成此证明，它会在前提中找到 value。 第二个引理的证明方式完全相同。

现在我们可以在提示符中测试 *with* 规则：匹配中间值 (ćąţ 39) 一节中的 natToBin 了。数字 42 的二进制为 101010。其二进制数字以逆序表示：

```
*views> show (natToBin 42)
"[False, True, False, True, False, True]" : String
```

1.10.2 暂且相信

Idris 在编译程序前需要完成证明（尽管在提示符中求值可以无需详细证明）。然而有时候， 特别在成型时，不去完成证明反而更容易。在尝试证明它们之前就测试程序甚至可能会更好， 如果测试找到了一个错误，你就会知道最好不要花时间去证明某些东西了！

因此，Idris 提供了一个内建的强迫（coercion）函数，它允许我们使用类型错误的值：

```
believe_me : a -> b
```

显然，它的使用必须要非常小心。在成型时它非常有用，在断言外部代码（可能在外部的 C 库中）的性质时也是可以用的。使用了它的 views.parity_lemma_1 的「证明」为：

```
views.parity_lemma_2 = proof {
    intro;
    intro;
    exact believe_me value;
}
```

exact 策略允许我们为该证明提供一个确切的值。在本例中，我们断言给出的值是正确的。

1.10.3 示例：二进制数

我们在前面通过 Parity 视角实现了 Nat 到二进制数的转换。在这里， 我们会展示如何用同样的视角来实现已验证的二进制转换。我们首先在与其等价的 Nat 上索引二进制数。这是一种通用的模式，即将它的表示（这里为 Binary）与其含义 （这里为 Nat）关联起来：

```
data Binary : Nat -> Type where
   BEnd : Binary Z
   BO : Binary n -> Binary (n + n)
   BI : Binary n -> Binary (S (n + n))
```

BO 和 BI 接受一个二进制数作为其参数并立即将它左移一位， 然后再加零或一作为新的最低位。索引 n + n 或 S (n + n) 描述了左移后再相加的结果与该数值的意义相同。它会产生低位在前的表示。

现在，将 Nat 转换为二进制的函数在其类型中描述了结果二进制数为原始 Nat 的正确表示：

```
natToBin : (n:Nat) -> Binary n
```

Parity 视角让定义变得相当简单：把数折半其实就是进行一次右移，尽管我们需要在 Odd 的情况下使用临时定义：

```
natToBin : (n:Nat) -> Binary n
natToBin Z = BEnd
natToBin (S k) with (parity k)
   natToBin (S (j + j))      | Even  = BI (natToBin j)
   natToBin (S (S (j + j))) | Odd  ?= BO (natToBin (S j))
```

Odd 情况的问题与 parity 定义中的相同，其证明过程也一样：

```
natToBin_lemma_1 = proof {
    intro;
    intro;
    rewrite sym (plusSuccRightSucc j j);
    trivial;
}
```

最后，我们来实现一个 main 程序，它读取用户输入的整数并输出为二进制：

```
main : IO ()
main = do putStr "Enter a number: "
          x <- getLine
          print (natToBin (fromInteger (cast x)))
```

当然，为了能让它工作，我们需要为 Binary n 实现 Show：

```
Show (Binary n) where
    show (BO x) = show x ++ "0"
    show (BI x) = show x ++ "1"
    show BEnd = ""
```

1.11 交互式编辑

目前，我们已经见过几个 Idris 的例子了，它的依赖类型系统可以在函数的 **类型** 中为期望的行为添加更加精确的描述，为函数的正确性增加额外的信心。 我们也见识过类型系统如何帮助开发 EDSL 了，它允许程序员描述目标语言的类型系统。 然而，精确的类型不止赋予了我们程序验证的能力，我们还可以利用类型来帮助 **构造** 出 **正确** 的程序。

Idris 的 REPL 提供了一些命令，可基于程序的类型来检查和修改程序的片段， 例如在模式变量中拆分情况，检查坑的类型，甚至还有基本的证明搜索机制。在本节中， 我们解释了如何在文本编辑器中，特别是如何在 Vim 中利用这些特性。 Emacs 的交互式模式也可以。

1.11.1 在 REPL 中编辑

REPL 提供了许多命令，我们稍后会介绍它们。它们基于当前加载的模块来生成新的程序片段。 其一般形式为：

:命令 [行号] [名称]

也就是说，每条命令都会作用在源码中特定行的特定名称上，然后输出一个新的程序片段。 每个命令都有一种原地 **更新** 源文件的形式：

```
:命令! [行号] [名称]
```

当 REPL 被加载时，它还会用 `idris --client` 在后台启动一个进程，该进程接受并响应 REPL 命令。例如，如果我们在某处运行着 REPL，就可以执行这样的命令：

```
$ idris --client ':t plus'
Prelude.Nat.plus : Nat -> Nat -> Nat
$ idris --client '2+2'
4 : Integer
```

文本编辑器可以利用此特性与编辑命令来提供交互式编辑。

1.11.2 编辑命令

:addclause

`:addclause n f` 命令，缩写为 `:ac n f`，它为第 `n` 行声明的函数 `f` 创建一个模版定义。例如，若从第 94 行开始的代码为：

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
```

那么 `:ac 94 vzipWith` 会给出：

```
vzipWith f xs ys = ?vzipWith_rhs
```

名称可根据程序员给定的提示来选择，必要时机器还会添加数字使其唯一。 我们可以像下面这样给出提示：

```
%name Vect xs, ys, zs, ws
```

它声明了 `Vect` 类型族的名字应按照 `xs`、`ys`、`zs`、`ws` 的顺序来选取。

:casesplit

`:casesplit n x` 命令，缩写为 `:cs n x`，它将第 `n` 行的模式变量 `x` 拆分为能构成它的多种模式，并移除任何由于一致性错误而不可能出现的情况。 例如，若从第 94 行开始的代码为：

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
vzipWith f xs ys = ?vzipWith_rhs
```

那么 `:cs 96 xs` 会给出：

```
vzipWith f [] ys = ?vzipWith_rhs_1
vzipWith f (x :: xs) ys = ?vzipWith_rhs_2
```

即，模式变量 `xs` 被拆分成了 `[]` 和 `x :: xs` 两种情况。与之前一样， 名称的选取启发自相同的规则。若我们（用 `:cs!`）更新文件后再拆分同一行的 `ys`，就会得到：

```
vzipWith f [] [] = ?vzipWith_rhs_3
```

即，模式变量 ys 被拆分成了 [] 这一个情况，因为 Idris 发现另一种可能的情况 y :: ys 会导致一致性错误。

:addmissing

:addmissing n f 命令，缩写为 :am n f，它为第 n 行的函数 f 添加能覆盖所有输入情况的从句。例如，若从第 94 行开始的代码为：

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
vzipWith f [] [] = ?vzipWith_rhs_1
```

那么 :am 96 vzipWith 会给出：

```
vzipWith f (x :: xs) (y :: ys) = ?vzipWith_rhs_2
```

即，它注意到不存在空向量的情况，然后生成了需要的从句，并消除了会导致不一致性错误的从句。

:proofsearch

:proofsearch n f 命令，缩写为 :ps n f，它试图通过证明搜索、尝试局部变量的值、递归调用以及所需类型族的构造器来为第 n 行的坑 f 查找一个值。该命令也可以接受一个可选的 **提示（Hint）** 列表，也就是可用于尝试解决此坑的函数列表。例如，若从第 94 行开始的代码为：

```
vzipWith : (a -> b -> c) ->
           Vect n a -> Vect n b -> Vect n c
vzipWith f [] [] = ?vzipWith_rhs_1
vzipWith f (x :: xs) (y :: ys) = ?vzipWith_rhs_2
```

那么 :ps 96 vzipWith_rhs_1 会给出：

```
[]
```

它能工作是因为它在对长度为 0 的 Vect 进行搜索，而空向量是唯一的可能。同样，在试图解决 :ps 97 vzipWith_rhs_2 时也出乎意料地只有一种可能：

```
f x y :: (vzipWith f xs ys)
```

它能工作是因为 vzipWith 拥有足够精确的类型：其结果向量一定非空（即至少有一个 ::）；第一个元素的类型必须为 c，而得到它的唯一方法就是将 f 应用于 x 和 y；最后，该向量的尾部只能递归地构造。

:makewith

:makewith n f 命令，缩写为 :mw n f，它为模式添加一个 with 从句。以之前的 parity 为例。若第 10 行为：

```
parity (S k) = ?parity_rhs
```

那么 :mw 10 parity 会给出：

```
parity (S k) with (_)
  parity (S k) | with_pat = ?parity_rhs
```

若我们在占位符 _ 处填上 parity k，并用 :cs 11 with_pat 拆分 with_pat 的情况，就会得到以下模式:

```
parity (S (plus n n)) | even = ?parity_rhs_1
parity (S (S (plus n n))) | odd = ?parity_rhs_2
```

注意情况拆分规范化了该模式（即给的是 plus 而非 +）。我们会看到在任何情况下， 使用交互式编辑向程序员展示有效的模式都能显著简化依赖模式匹配的实现。

1.11.3 Vim 交互式编辑

Vim 的编辑器模式提供语法高亮和缩进，通过前文所述的命令提供交互式编辑的支持。 交互式编辑使用以下编辑器命令来进行，每一条都会直接更新缓冲区:

- \d 使用 :addclause 为当前行声明的名字添加模版定义。
- \c 使用 :casesplit 为光标处的变量执行情况拆分。
- \m 使用 :addmissing 为光标处的名字添加缺少的情况。
- \w 使用 :makewith 添加 with 从句。
- \o 使用 :proofsearch 调用证明搜索来解决光标处的坑。
- \p 使用 :proofsearch 根据附加的提示调用证明搜索以解决光标处的坑。

还有一些用来调用类型检查器与求值器的命令:

- \t 显示光标下（全局可见的）名称的类型。对坑而言，它会显示其上下文和预期的类型。
- \e 提醒要求值的表达式。
- \r 重新加载缓冲区并执行类型检查。

对应的命令在 Emacs 模式中也可用。其它编辑器的支持可通过使用 idris -client 以相对直接的方式来添加。

1.12 语法扩展

Idris 支持以多种方式实现 **嵌入式领域特定语言（Embedded Domain Specific Language, EDSL）**[1] 。 我们见过的一种方式是扩展 do 记法。另一种重要的方式就是对核心语法进行扩展。在本节中，我们描述了两种扩展语法的方式: syntax 规则与 dsl 记法。

1.12.1 syntax 规则

我们已经见过 if...then...else 表达式了，然而它并不是内建的。同样， 我们可以定义一个 Prelude 中的函数（它在 惰性 (éąţ 13) 一节中出现过了）:

[1] Edwin Brady and Kevin Hammond. 2012. Resource-Safe systems programming with embedded domain specific languages. In Proceedings of the 14th international conference on Practical Aspects of Declarative Languages (PADL '12), Claudio Russo and Neng-Fa Zhou (Eds.). Springer-Verlag, Berlin, Heidelberg, 242-257. DOI=10.1007/978-3-642-27694-1_18 http://dx.doi.org/10.1007/978-3-642-27694-1_18

```
ifThenElse : (x:Bool) -> Lazy a -> Lazy a -> a;
ifThenElse True  t e = t;
ifThenElse False t e = e;
```

接着用 syntax 声明来扩展核心语法:

```
syntax if [test] then [t] else [e] = ifThenElse test t e;
```

syntax 声明的左式描述了语法规则，而右式描述了其展开式。语法规则的构成为:

- **关键字** — 在这里为 if、then 和 else，它必须是有效的标识符。
- **非终止符（Non-terminal）** — 位于方括号内，此处为 [test]、[t] 和 [e]，它们表示任意表达式。为避免解析歧义，这些表达式不能在顶层使用语法扩展 （也就是说你可以在括号中使用）。
- **名称** — 位于大括号内，它表示可在右侧被绑定的名字。
- **符号** — 位于引号内，例如 ":="。它也可在语法规则中包含保留字，例如 "let" 或 "in"。

语法规则形式的限制在于它必须包含至少一个符号或关键字，且表示非终止符的变量不能重复。 任何表达式都可以使用，不过如果在一个规则的同一行内有两个非终止符，那么只有简单的表达式会被使用（即，变量、常量或方括号括起的表达式）。规则可在其定义之前使用，但无法递归地使用。 因此以下语法扩展是有效的:

```
syntax [var] ":=" [val]                = Assign var val;
syntax [test] "?" [t] ":" [e]          = if test then t else e;
syntax select [x] from [t] "where" [w] = SelectWhere x t w;
syntax select [x] from [t]             = Select x t;
```

语法宏可被进一步限制为只能在模式（即，只能在模式匹配从句的左侧）中或只能在被标为 pattern 或 term 语法规则的项（即除了模式匹配从句左侧的任何地方）中应用。 例如，假设我们定义了一个区间 Interval，用 So 静态检查以保证下界小于上界:

```
data Interval : Type where
   MkInterval : (lower : Double) -> (upper : Double) ->
                So (lower < upper) -> Interval
```

我们可以用 pattern 定义一个语法，它总是匹配 Oh 作为证明论据，用 term 请求提供一个证明项:

```
pattern syntax "[" [x] "..." [y] "]" = MkInterval x y Oh
term    syntax "[" [x] "..." [y] "]" = MkInterval x y ?bounds_lemma
```

在 term 中，语法 [x...y] 会生成一个证明义务 bounds_lemma （可能被重命名）。

最后，语法规则可引入另一种绑定形式。例如，for 循环在每次迭代中绑定一个参数:

```
syntax for {x} "in" [xs] ":" [body] = forLoop xs (\x => body)

main : IO ()
main = do for x in [1..10]:
              putStrLn ("Number " ++ show x)
          putStrLn "Done!"
```

注意，我们用形式 {x} 指明 x 表示一个已绑定的变量，它在右侧会被替换。 我们还把 in 放在了引号中，因为它是保留字。

1.12.2 dsl 记法

示例：良类型的解释器 (éąţ 35) 一节中的良类型解释器是个依赖类型编程模式的简单例子。也就是说：先用依赖类型描述一个 **目标语言** 及其类型系统，保证只有良类型的程序可被表示， 然后再通过这种方式表示程序。通过这种方式，我们可以编写序列化二进制数据[2] 或安全运行并发过程[3] 的程序。

然而，目标语言的形式使其难以在实践中编程。回想一下用 Expr 编写的阶乘程序：

```
fact : Expr G (TyFun TyInt TyInt)
fact = Lam (If (Op (==) (Var Stop) (Val 0))
               (Val 1) (Op (*) (App fact (Op (-) (Var Stop) (Val 1)))
                               (Var Stop)))
```

由于这是一种特别有用的模式，因此 Idris 提供了语法重载[1] 使其在这种目标语言中更易于编程：

```
mkLam : TTName -> Expr (t::g) t' -> Expr g (TyFun t t')
mkLam _ body = Lam body

dsl expr
    variable    = Var
    index_first = Stop
    index_next  = Pop
    lambda      = mkLam
```

dsl 块描述了每个语法构造是如何在目标语言中表示的。在这里的 expr 语言中， 任何变量都会被翻译为 Var 构造器，使用 Pop 和 Stop 来构造 de Bruijn 索引（即，由于变量本身被绑定，所以要统计有多少个绑定）；而任何 λ-表达式都会被翻译为 Lam 构造器。mkLam 函数会简单地忽略其第一个参数，它是用户为变量选择的名字。 我们也可以通过这种方式来重载 let 与依赖函数的语法。现在可以将 fact 写成下面这样了：

```
fact : Expr G (TyFun TyInt TyInt)
fact = expr (\x => If (Op (==) x (Val 0))
                      (Val 1) (Op (*) (app fact (Op (-) x (Val 1))) x))
```

在这个新的版本中，expr 声明了下一个要被重载的表达式。我们可以利用习语括号， 通过以下声明再进一步：

```
(<*>) : (f : Lazy (Expr G (TyFun a t))) -> Expr G a -> Expr G t
(<*>) f a = App f a

pure : Expr G a -> Expr G a
pure = id
```

注意，它无需成为 Applicative 实现的一部分，因为习语括号记法会直接被翻译为 名字 <*> 和 pure，针对特设（ad-hoc）类型的重载也被允许。现在我们可以写成：

```
fact : Expr G (TyFun TyInt TyInt)
fact = expr (\x => If (Op (==) x (Val 0))
                      (Val 1) (Op (*) [| fact (Op (-) x (Val 1)) |] x))
```

使用更加特设的重载、接口，以及新的语法规则，我们甚至可以再进一步：

[2] Edwin C. Brady. 2011. IDRIS —: systems programming meets full dependent types. In Proceedings of the 5th ACM workshop on Programming languages meets program verification (PLPV '11). ACM, New York, NY, USA, 43-54. DOI=10.1145/1929529.1929536 http://doi.acm.org/10.1145/1929529.1929536

[3] Edwin Brady and Kevin Hammond. 2010. Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols. Fundam. Inf. 102, 2 (April 2010), 145-176. http://dl.acm.org/citation.cfm?id=1883636

```
syntax "IF" [x] "THEN" [t] "ELSE" [e] = If x t e

fact : Expr G (TyFun TyInt TyInt)
fact = expr (\x => IF x == 0 THEN 1 ELSE [| fact (x - 1) |] * x)
```

1.13 杂项

在本节中，我们讨论了多种附加特性：

- 自动、隐式与默认参数
- 文学编程（Literate Programming）
- 通过外部函数与外部库交互
- 接口
- 类型提供器（Type Provider）
- 代码生成，以及
- 全域层级（Universe Hierarchy）

1.13.1 隐式参数

我们已经见过隐式参数了，它允许忽略能够被类型检查器推断出来的参数，例如：

```
index : {a:Type} -> {n:Nat} -> Fin n -> Vect n a -> a
```

自动隐式参数

在其它情况下，可以不通过类型检查来推断参数，而是通过在上下文中搜索恰当的值， 或通过构造证明来推断参数。例如，以下 `head` 的定义需要证明列表非空：

```
isCons : List a -> Bool
isCons [] = False
isCons (x :: xs) = True

head : (xs : List a) -> (isCons xs = True) -> a
head (x :: xs) _ = x
```

如果可以静态地获知列表非空，那是由于它的值是已知的，或上下文中存在对它的证明。 证明可以自动地构造，自动隐式参数允许它静默地发生。我们将 `head` 定义为：

```
head : (xs : List a) -> {auto p : isCons xs = True} -> a
head (x :: xs) = x
```

将隐式参数注解为 `auto`，表示 Idris 会试图搜索与其类型对应的值来填充它。 它会按照以下顺序尝试：

- 局部变量，即类型正确的，在模式匹配中绑定的名字或 `let` 绑定。
- 所需类型的构造器。若它们有参数，就会递归地搜索到最深 100 层的深度。

- 带函数类型的局部变量，对参数进行递归地搜索。
- 任何标出 %hint 注解的，带有相应返回类型的函数。

在找不到证明的情况下，它会像往常一样被显式地给出：

```
head xs {p = ?headProof}
```

定义隐式参数

除了让 Idris 自动查找给定类型的值外，有时我们还想要带有具体默认值的隐式参数。在 Idris 中，我们可以用 default 注解来做这件事。尽管它主要是为了在 auto 自动构造证明失败，或找到的值没有帮助时起辅助作用。然而，首先考虑一个更简单的， 不涉及证明的情况可能会更容易。

如果我们想要计算第 n 个斐波那契数（第 0 个斐波那契数定义为 0），我们可以写：

```
fibonacci : {default 0 lag : Nat} -> {default 1 lead : Nat} -> (n : Nat) -> Nat
fibonacci {lag} Z = lag
fibonacci {lag} {lead} (S n) = fibonacci {lag=lead} {lead=lag+lead} n
```

定义完之后，fibonacci 5 等价于 fibonacci {lag=0} {lead=1} 5，它会返回第 5 个斐波那契数。注意虽然它可以工作，但这并不是 default 注解的用途，在这里它只用作展示。 通常，default 用于提供定制证明搜索脚本的东西。

1.13.2 隐式转换

Idris 支持创建 **隐式转换** ，在需要某项的类型能够匹配时， 它允许将一个类型的值自动转换成另一个类型。这是为了增强便利性并减少冗余。 下面是个故意构造的简单例子：

```
implicit intString : Int -> String
intString = show

test : Int -> String
test x = "Number " ++ x
```

通常，我们无法将一个 Int 附加到一个 String 之后，不过隐式转换函数 intString 可以将 x 转换为 String，因此 test 定义的类型是正确的。 隐式转换的实现和其它函数一样，只不过加上了 implicit 修饰符， 且被限制为只能接受一个显式参数。

一次只有一个隐式转换会被应用。也就是说，隐式转换无法被链式调用。如前文所见， 简单类型的隐式转换是不被鼓励的！更常见的做法是，使用隐式转换来减少 EDSL 的冗长度，或者隐藏证明的细节。这些示例超出了本教程的范围。

1.13.3 文学编程

和 Haskell 一样，Idris 支持 **文学编程**。如果某个文件的扩展名外 .lidr， 那么它会被当做文学编程文件。在文学编程中，除了以大于号 > 开头的行外， 所有的内容都会被视为注释。例如：

```
> module literate

这是一行注释，主程序在下面

> main : IO ()
> main = putStrLn "Hello literate world!\n"
```

附加的限制为，程序行（以 > 开头）与注释行（以任何其它字符开头）之间必须有一空行。

1.13.4 外部函数调用

在编程实践中，我们经常需要使用外部库，特别在与操作系统、文件系统、网络 **等等** 进行交互时。作为 Prelude 的一部分，Idris 提供了轻量的外部函数接口。 我们假定读者有一定的 C 和 gcc 编译器的知识。首先，我们来定一个数据类型， 它描述了我们能够处理的外部类型：

```
data FTy = FInt | FFloat | FChar | FString | FPtr | FUnit
```

它们每一个都与 C 的类型直接对应。分别为：int、double、char、 char*、void* 和 void。以下函数还描述了它们到对应的 Idris 类型的翻译：

```
interpFTy : FTy -> Type
interpFTy FInt    = Int
interpFTy FFloat  = Double
interpFTy FChar   = Char
interpFTy FString = String
interpFTy FPtr    = Ptr
interpFTy FUnit   = ()
```

外部函数由一组输入类型和返回类型描述，它们可以被转换为 Idris 类型：

```
ForeignTy : (xs:List FTy) -> (t:FTy) -> Type
```

外部函数被视作不纯粹的，因此 ForeignTy 构建了一个 IO 类型，例如：

```
Idris> ForeignTy [FInt, FString] FString
Int -> String -> IO String : Type

Idris> ForeignTy [FInt, FString] FUnit
Int -> String -> IO () : Type
```

我们通过为函数赋予名字、一系列参数的类型和返回值构建了一个外部函数调用。 内建的构造 mkForeign 将该函数的描述转换为一个可由 Idris 调用的函数：

```
data Foreign : Type -> Type where
    FFun : String -> (xs:List FTy) -> (t:FTy) ->
           Foreign (ForeignTy xs t)

mkForeign : Foreign x -> x
```

注意编译器期望 mkForeign 能够被完全地应用以构建完整的外部函数调用。例如， putStr 作为一个在运行时系统中定义的外部函数 putStr 的调用，其实现如下：

```
putStr : String -> IO ()
putStr x = mkForeign (FFun "putStr" [FString] FUnit) x
```

include 与链接器指令

外部函数调用会按照 Idris 和 C 的值的表示之间对应的转换，被直接翻译为 C 函数的调用。 通常这会需要将额外的库、头文件或目标文件链接进来。我们可以通过以下指令来完成：

- %lib target x — 将 libx 库包含进来。如果目标为 C，它等价于向 gcc 传递 -lx 选项。如果目标为 Java，该库会被解释为 maven 的依赖关系定位符 groupId:artifactId:packaging:version。

- `%include target x` — 使用头文件或为给定的后端目标导入 `x`。
- `%link target x.o` — 在使用给定的后端目标时链接目标文件 `x.o`。
- `%dynamic x.so` — 动态链接共享的解释器目标文件 `x.so`。

测试外部函数调用

一般来说，Idris 解释器（用于类型检查和 REPL）不会处理 IO 活动。除此之外，它既不会生成 C 代码，也不会将它编译成机器码，因此 `%lib`、`%include` 与 `%link` 是没有效果的。 IO 活动与 FFI 调用可使用特殊的 REPL 命令 `:x EXPR` 来测试，而 C 库可通过 `:dynamic` 命令或 `%dynamic` 指令来动态地加载到解释器中。例如：

```
Idris> :dynamic libm.so
Idris> :x unsafePerformIO ((mkForeign (FFun "sin" [FFloat] FFloat)) 1.6)
0.9995736030415051 : Double
```

1.13.5 类型提供器

Idris 类型提供器，灵感来自 F# 的类型提供器，它能让我们的类型与 Idris 之外的世界建立「联系」。例如，给定一个表示数据库模式（Schema）的类型， 和一个针对它检查过的查询，类型提供器可以在进行类型检查时读取真实数据库的模式。

Idris 类型提供器使用普通的 Idris 可执行语义来运行 IO 活动并提取出结果。 该结果会作为编译代码时的常量被保存。它可以是个类型，此时它能像其它类型一样使用； 它也可以是个值，此时它也可以像其它值一样使用，并作为一个索引被包含在类型中。

类型提供器尚且是个实验性的扩展。要启用它，请使用 `%language` 指令：

```
%language TypeProviders
```

某个类型 `t` 的提供器 `p` 不过就是个类型为 `IO (Provider t)` 的表达式。 `%provide` 指令会导致类型检查器去执行该活动，并将其结果绑定到一个名字上。 我们最好用一个简单的例子来展示它。类型提供器 `fromFile` 用于读取文本文件。 如果该文件由字符串 `Int` 构成，那么它就会提供 `Int` 类型。否则，它就会提供 `Nat` 类型。

```
strToType : String -> Type
strToType "Int" = Int
strToType _ = Nat

fromFile : String -> IO (Provider Type)
fromFile fname = do Right str <- readFile fname
                      | Left err => pure (Provide Void)
                    pure (Provide (strToType (trim str)))
```

接着我们使用 `%provide` 指令：

```
%provide (T1 : Type) with fromFile "theType"

foo : T1
foo = 2
```

如果名为 `theType` 的文件由单词 `Int` 构成，那么 `foo` 的类型会是 `Int`。 否则，它会是 `Nat`。当 Idris 遇到该指令时，它首先会检查确认提供器表达式 `fromFile theType` 的类型为 `IO (Provider Type)`。接着它会执行该提供器。 如果其结果为 `Provide t`，那么 `T1` 就会被定义为 `t`。否则，就会产生一个错误。

我们的数据类型 `Provider t` 定义如下：

```
data Provider a = Error String
                | Provide a
```

我们已经见过 `Provide` 构造器了。`Error` 构造器允许类型提供器返回有用的错误信息， 本节中的示例为了简单并未提供。更加复杂的类型提供器实现，包括一个静态检查的 SQLite 绑定， 可从外部链接[1]获取。

1.13.6 以 C 为编译目标

Idris 的默认编译目标为 C。它通过以下命令编译：

```
$ idris hello.idr -o hello
```

此命令等价于：

```
$ idris --codegen C hello.idr -o hello
```

当使用以上命令编译时，它会生成一个临时的 C 源码，接着再编译成名为 `hello` 的可执行文件。

要查看生成的 C 代码，请通过以下命令编译：

```
$ idris hello.idr -S -o hello.c
```

要开启优化，请在代码中使用 `%flag C` 编译指令：

```
module Main
%flag C "-O3"

factorial : Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))

main : IO ()
main = do
     putStrLn $ show $ factorial 3
```

要在编译生成的 C 代码时加上调试信息，例如使用 `gdb` 在 Idris 程序中调试段错误时， 请使用 `%flag C` 编译指令来包括调试符号，如下所示：

```
%flag C "-g"
```

1.13.7 以 JavaScript 为编译目标

Idris 可生成能够运行在浏览器以及 *NodeJS* 等类似环境中的 *JavaScript* 代码。 它可以通过 FFI 与 *JavaScript* 生态进行交互。

代码生成

代码生成分为两种独立的目标。要生成适合在浏览器中运行的代码，请使用以下命令：

[1] https://github.com/david-christiansen/idris-type-providers

```
$ idris --codegen javascript hello.idr -o hello.js
```

产生的文件可以像其它 *JavaScript* 代码那样嵌入到 HTML 中。

生成 *NodeJS* 代码的方式有点不同。Idris 会输出一个可直接调用 `node` 运行的 *JavaScript* 文件。

```
$ idris --codegen node hello.idr -o hello
$ ./hello
Hello world
```

考虑到 *JavaScript* 代码生成器使用 `console.log` 将文本写入到 `stdout`，因此它会自动在每个字符串之后加上换行。而此行为不会在 *NodeJS* 代码生成器中出现。

使用 FFI

要编写一个有用的应用，我们需要与外部世界进行交流。可能我们想要操作 DOM 或发送 Ajax 请求。为此可以使用 FFI。由于大部分 *JavaScript* API 需要回调，因此我们需要扩展 FFI 以将函数作为参数来传入。

JavaScript FFI 的工作方式与一般的 FFI 有点不同。它使用位置参数将我们的参数直接插入一段 *JavaScript* 代码中。

我们可以使用 *JavaScript* 的原语加法:

```
module Main

primPlus : Int -> Int -> IO Int
primPlus a b = mkForeign (FFun "%0 + %1" [FInt, FInt] FInt) a b

main : IO ()
main = do
  a <- primPlus 1 1
  b <- primPlus 1 2
  print (a, b)
```

注意 `%n` 记法确定了从 0 开始的第 `n` 个给定的外部函数的参数。当你需要一个百分号而非位置时，请使用 `%%` 来代替。

将函数传入外部函数中是非常相似的。假设我们想要从 *JavaScript* 世界中调用以下函数:

```
function twice(f, x) {
  return f(f(x));
}
```

显然我们需要在这里传入一个函数 `f` （我们可以从 `twice` 中使用 `f` 的方式推断出来，如果 *JavaScript* 有类型的话会更加明显）。

当你给 *JavaScript* FFI 一个类型为 `FFunction` 的东西时，它能够理解要将函数作为参数。以下示例代码在 *JavaScript* 中调用了 `twice` 并将结果返回到了 Idris 程序中:

```
module Main

twice : (Int -> Int) -> Int -> IO Int
twice f x = mkForeign (
  FFun "twice(%0,%1)" [FFunction FInt FInt, FInt] FInt
) f x

main : IO ()
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
main = do
  a <- twice (+1) 1
  print a
```

该程序输出 3，正如我们所料。

包含外部的 *JavaScript* 文件

只要某人使用 *JavaScript*，他就有可能想要包含外部库，或者通过 FFI 调用存储在外部文件中的函数。*JavaScript* 和 *NodeJS* 代码生成器能够理解 `%include` 指令。请注意 *JavaScript* 和 *NodeJS* 是由不同的代码生成器处理的，因此你需要指明所需的目标。这也就表示你可以在同一个 Idris 源文件中分别为 *JavaScript* 和 *NodeJS* 包含不同的文件。

因此如果你想要添加外部的 *JavaScript* 文件，可以这样做：

对于 *NodeJS*：

```
%include Node "path/to/external.js"
```

要在浏览器中使用：

```
%include JavaScript "path/to/external.js"
```

给定的文件会被添加的生成代码的顶部。对于库包，你也可以使用 ipkg 文件中的 objs 选项来将 js 文件包含在安装中，并使用：

```
%include Node "package/external.js"
```

Idris 的 *JavaScript* 和 *NodeJS* 后端也会在此位置查找文件。

包含 *NodeJS* 模块

NodeJS 代码生成器也可以通过 `%lib` 指令来包含模块。

```
%lib Node "fs"
```

该指令会编译成以下 *JavaScript*：

```
var fs = require("fs");
```

缩减生成的 *JavaScript*

Idris 会产生非常大的 *JavaScript* 代码块。然而，生成的代码可通过 Google 的 `closure-compiler` 缩小。其它的缩减器也可用，不过 `closure-compiler` 提供了更高级的编译，它会做一些侵入性的内敛和代码消除。Idris 可以充分利用这种编译模式，强烈建议在传输用 Idris 编写的 *JavaScript* 应用时使用它。

1.13.8 累积性

由于值可以出现在类型中，**反之亦然**，因此类型自然也有类型。例如：

```
*universe> :t Nat
Nat : Type
*universe> :t Vect
Vect : Nat -> Type -> Type
```

但是 Type 的类型呢？如果我们询问 Idris，它会报告：

```
*universe> :t Type
Type : Type 1
```

如果 Type 是它自己的类型，那么它会因为 Girard 悖论 而导致不一致性，因此在内部存在类型的 **层级（Hierarchy）**（或 **全域**，Universe）：

```
Type : Type 1 : Type 2 : Type 3 : ...
```

全域类型是 **积累（Cumulative）** 的，也就是说，如果有 `x : Type n`，我们也可以有 `x : Type m` 使得 `n < m`。如果类型检查器发现了任何不一致性，它就会生成这种全域约束并报告一个错误。一般来说，程序员无须担心它，但它确实可以防止（构造出）如下的程序：

```
myid : (a : Type) -> a -> a
myid _ x = x

idid :  (a : Type) -> a -> a
idid = myid _ myid
```

将 myid 应用到其自身会导致在全域层级中出现一个循环，即 myid 第一个参数的类型为 Type，如果要将其应用到自身，那么其级别不能低于所要求的级别。

1.14 扩展阅读

有关 Idris 编程的更多信息，以及依赖类型编程的一般问题解答，可从多种来源获取：

- Idris 网站（http://www.idris-lang.org/）以及在邮件列表中提问。
- webchat.freenode.net 上的 `#idris` IRC 频道。
- 维基（https://github.com/idris-lang/Idris-dev/wiki/）上有更多用户提供的信息，特别是：
  - https://github.com/idris-lang/Idris-dev/wiki/Manual
  - https://github.com/idris-lang/Idris-dev/wiki/Language-Features
- **查看发行版中的 Prelude 并浏览 `samples` 目录。Idris 的源码可在线获取：** https://github.com/idris-lang/Idris-dev 。
- `Idris Hackers` 网站上的既有项目：http://idris-hackers.github.io 。
- 多篇论文（例如[1]、[2] 和[3]）。虽然它们大多描述的是旧版的 Idris。

[1] Edwin Brady and Kevin Hammond. 2012. Resource-Safe systems programming with embedded domain specific languages. In Proceedings of the 14th international conference on Practical Aspects of Declarative Languages (PADL '12), Claudio Russo and Neng-Fa Zhou (Eds.). Springer-Verlag, Berlin, Heidelberg, 242-257. DOI=10.1007/978-3-642-27694-1_18 http://dx.doi.org/10.1007/978-3-642-27694-1_18

[2] Edwin C. Brady. 2011. IDRIS —: systems programming meets full dependent types. In Proceedings of the 5th ACM workshop on Programming languages meets program verification (PLPV '11). ACM, New York, NY, USA, 43-54. DOI=10.1145/1929529.1929536 http://doi.acm.org/10.1145/1929529.1929536

[3] Edwin C. Brady and Kevin Hammond. 2010. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10). ACM, New York, NY, USA, 297-308. DOI=10.1145/1863543.1863587

http://doi.acm.org/10.1145/1863543.1863587

CHAPTER 2

常见问题解答（FAQ）

2.1 Agda 跟 Idris 有啥不一样啊?

和 Idris 一样，Agda 也是个带有依赖类型的函数式语言，并且支持依赖模式匹配。 它们都能编写程序和证明。不过，Idris 一开始就侧重于为通用编程而设计，而非 Agda 侧重的定理证明。因此，它支持与系统库和 C 程序的互操作性，以及用于实现 EDSL （领域特定语言）的语言构造。它还包括更高级的编程构造，例如接口（类似于类型类）和 `do` 记法。

Idris 支持多种后端（默认为 C 和 JavaScript，可以通过插件添加更多），拥有一个 C 编写的，带有垃圾收集器和内建并发消息传递的参考运行时系统。

2.2 有标准库文档没? 有函数列表没?

预置包的 API 文档已在文档页中列出。

不幸的是，*Idris* 的默认前导和预置包在文档方面未必完整。其它查找函数的方式包括:

- REPL 命令:
  - 使用 *:apropos* 来搜索文档和函数名中的文本。
  - 使用 *:search* 来根据类型搜索函数。
  - 使用 *:browse* 来列出给定名字空间中的内容。
- 使用 REPL 的自动补全功能。
- 使用 *grep* 在 *libs/* 中搜索源码

如果你发现预置包缺少文档，请随心为它补充一些。当然也可以通知别人来做这件事。 *Idris* 提供了显示富文档的语法，富文档可使用 *:doc* 命令浏览，并在生成的 HTML API 文档中列出。

2.3 Idris 能拿来干活不?

Idris 主要是个研究工具，用来探索用依赖类型开发软件的可能性，也就是说它的主要目的（还） 不是开发可用于生产的系统。因此，某些方面上它还有些粗糙，也缺少很多库。现在还没人全职搞 Idris，我们目前也没有资源来独立完善该系统。因此，我们不推荐你围绕它来开展业务！

话虽如此，我们非常欢迎帮助推进 Idris 适应生产的贡献，包括（但不限于）额外的库支持、 完善运行时系统（并确保其健壮性）、提供并维护 JVM 后端等等。

2.4 为啥 Idris 用了及早求值，不用惰性求值啊?

Idris 采用及早求值主要是为了能更好地预测性能，特别是其长期目标之一就是能够编写高效 且经过验证的底层代码，例如设备驱动以及网络设施。此外，Idris 的类型系统能让我们精确地 指明每个值的类型，因此也就能确定每个值在运行时的形式。在惰性求值的语言中，考虑一个 `Int` 类型的值：

```
thing : Int
```

`thing` 在运行时是如何表示的? 它是一个按照位模式来表示的整数， 还一个是指向了能计算出整数的代码的指针? 在 Idris 中，我们决定按照类型来精确地区分它们：

```
thing_val : Int
thing_comp : Lazy Int
```

这样，`thing_val` 就能确保为具体的 `Int`，而 `thing_comp` 则是会产生 `Int` 的计算。

2.5 俺咋弄个惰性控制结构呐?

你可以用特殊的 `Lazy` 类型来创建这种控制结构。例如，Idris 中的 `if...then...else...` 会被扩展为 `ifThenElse` 函数的应用。 它在库中对布尔值的默认实现如下：

```
ifThenElse : Bool -> (t : Lazy a) -> (e : Lazy a) -> a
ifThenElse True  t e = t
ifThenElse False t e = e
```

`t` 和 `e` 的类型 `Lazy a` 指出它们只会在需要时被求值，也就是说，它们是惰性求值的。

2.6 REPL 里头的求值行为跟咱想的不一样啊，这咋回事儿= =?

作为一个完全依赖类型的语言，Idris 的求值分为两个阶段，编译时与运行时。在编译时， 它只会对已知完全（即保证会终止且覆盖了所有可能输入）的东西求值， 以此来保持类型检查的可判定性。编译时的求值器属于 Idris 核心的一部分，它用 Haskell 编写，按照值的 HOAS（Higher Order Abstract Syntax 高阶抽象语法）风格的表示来实现。 由于已知所有东西都有规范的形式，选用哪种求值策略实际上也就无关紧要了， 因为它们都会得到相同的答案，而在实践中，它会执行 Haskell 运行时系统选择的任何事情。

按照约定，REPL 使用编译时求值的概念。除了更容易实现外（因为我们可以用求值器）， 它在展示项（Term）如何在类型检查器中求值时也非常有用。

```
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
```

2.7 为啥咱不能在类型里用没参数的函数呐?

如果你在类型中使用小写字母开头的名字，且它们没用被应用于任何参数，那么 Idris 会把它视为隐式绑定的参数。例如：

```
append : Vect n ty -> Vect m ty -> Vect (n + m) ty
```

这里的 `n`、`m` 和 `ty` 是隐式绑定的。此规则同样适用于在别处定义的， 带这类名字的函数。例如，你或许还有：

```
ty : Type
ty = String
```

即便如此，`ty` 还是会被视作 `append` 定义中隐式绑定的参数，而不会让 `append` 的类型等价于你预想的：

```
append : Vect n String -> Vect m String -> Vect (n + m) String
```

采用了此规则后，你无需其它上下文，只要查看 `append` 的类型就能明白哪些是隐式绑定的名字。

如果你想在类型中使用不被应用的名字，那么有两种选择：你可以显式地限定它，例如， 若 `ty` 在 `Main` 的命名空间中定义，你可以这样做：

```
append : Vect n Main.ty -> Vect m Main.ty -> Vect (n + m) Main.ty
```

此外，你还可以使用不以小写字母开头的名字，它决不会被隐式绑定：

```
Ty : Type
Ty = String

append : Vect n Ty -> Vect m Ty -> Vect (n + m) Ty
```

按照约定，如果你打算将一个名字用作类型同义，那么最好以大写字母开头来避免此限制。

2.8 这破程序明显能停，为啥 Idris 还说它有可能不完全?

由于停机问题的不可判定性， Idris 通常无法判定一个程序是否会停止。然而，我们可以找出某些确定可以终止的程序。 Idris 使用「大小改变终止（size change termination）」来寻找从函数返回到自身的递归路径。 在此路径上，至少必有一个参数会收敛到基本情况。

- Idris 支持相互递归的函数
- 不过，递归路径上的所有函数必须被完整地应用。此外，Idris 不支持高阶应用。
- 在递归调用的过程中，Idris 会查找语法上更小的输入参数，以此来识别能收敛到基本情况的参数。 例如，`k` 在语法上小于 `S (S k)`，因为 `k` 是 `S (S k)` 的子项（subterm）， 然而 `(k, k)` 在语法上却不小于 `(S k, S k)`。

如果你有个确信会终止的函数，但 Idris 不信，那么你可以调整程序的结构，或者使用 `assert_total` 函数。

<<<<<<< HEAD Idris 啥时候能自举啊？ =======

这事不急，虽说从长远来看这主意不错。就目前来说，实现支持自举的库是一项很有价值的工作, 比如说参数解析器以及符合 POSIX 标准的库用于系统交互。

2.9 Idris 有全域多态不？Type 是啥类型的?

Idris 并没有全域多态（Universe Polymorphism），而是拥有全域的积累层级（Cumulative Hierarchy），如 `Type : Type 1`、`Type 1 : Type 2` 等等。积累性的意思是，若 `x : Type n` 且 `n <= m`，则 `x : Type m`。全域的级别总是由 Idris 推导，且无法被显式地指定。执行 REPL 命令 `:type Type 1` 以及试图为任何类型指定全域级别时，都会产生一个错误。

2.10 为啥 Idris 用 Double 不用 Float64?

历史上 C 和很多语言都用分别用 `Float` 和 `Double` 来表示 32 位和 64 位的浮点数。较新的语言，如 Rust 和 Julia 都开始遵循 IEEE 浮点运算标准 (IEEE 754) 的命名规范了。它将单精度和双精度的数描述为 `Float32` 和 `Float64`，其大小在类型名中描述。

由于开发者更熟悉旧有的命名约定，而 Idris 的开发者也选择了它，因此 Idris 采用了 C 风格的约定。也就是说名称 `Double` 用于描述双精度浮点数，而 Idris 现在还不支持 32 位浮点数。

2.11 -ffreestanding 是啥?

在相对路径中拥有自己的库和编译器时，可使用 `freestanding` 命令行参数来构建 Idris 二进制文件。当构建过程中的安装目录未知时，它对于构建二进制文件来说非常有用。当传入此参数时，`IDRIS_LIB_DIR` 环境变量需要设置为相对与 `idris` 可执行文件所在的 Idris 库的路径。`IDRIS_TOOLCHAIN_DIR` 环境变量是可选的，如果设置了它，Idris 就会在该路径下寻找 C 编译器。例如:

```
IDRIS_LIB_DIR="./libs" \
IDRIS_TOOLCHAIN_DIR="./mingw/bin" \
CABALFLAGS="-fffi -ffreestanding -frelease" \
make
```

2.12 话说「Idris」是个啥名儿 O_O?

有一定年龄的英国人可能很熟悉唱歌贼好听的小火龙妹砸。你要是不满意，自己想个好词儿啊（手动微笑 :-)

2.13 还能有 Unicode 操作符不?

下面是为什么我们不应该支持 Unicode 操作符的原因:

- 它难以输入（如果你在使用别人的代码，这点就很重要）。很多编辑器都有它自己的输入法， 不过你必须知道怎么输入。
- 并不是任何软件都能轻松支持它。在一些移动 Email 客户端、基于终端的 IRC 客户端、 以及 Web 浏览器等软件中都会出现渲染问题。
- 即便我们不在标准库中使用它（绝对不会！），然而只要有人在他们的库代码中用了它， 别人就得去处理它。
- 有太多字符看起来太像了。单是分不清 0 和 O 就会造成很多麻烦，更不说各式各样的冒号和括号了。

如果使用得当，Unicode 操作符能让代码看起来更漂亮，然而 `lhs2TeX` 也能。 也许几年后情况有变，软件能更好地应对它，到时候重新审视它才有意义。然而目前， Idris 不会为操作符提供任何 Unicode 符号。

这似乎是个 Wadler 定律 在工作中的实例。

本答案基于 Edwin Brady 对此 推送请求 的回应。

2.14 Idris 有社区准则不?

这里 是Idris 社区规范的声明。

2.15 还有哪儿能找到解答啊?

Github 的维基上还有个非官方 FAQ， 其中解答了更多技术问题，而且经常更新。

CHAPTER 3

用 Idris 实现带有状态的系统：ST 教程

本教程叙述了如何用 Idris 的 `Control.ST` 库实现带有状态的系统。

3.1 概览

像 Idris 这种可以直接用类型系统对程序进行推理的函数式语言， 仅凭程序能够编译这一点，我们就能**确定**该程序会正确运行（即按照类型描述的规范运行）。

然而在现实中，软件依赖于状态，很多组件则依赖于状态机。它们可以描述像 TCP 这样的网络传输协议， 实现事件驱动的系统以及正则表达式的匹配。此外，像网络 Socket 和文件这类的很多基础资源，都由状态机隐式地管理；特定操作只有在资源处于特定状态时才可行，而这些操作也可以改变底层资源的状态。 例如，只有向已建立连接的网络 Socket 发送消息才有意义，关闭 Socket 会将其状态从“打开”切换为“关闭”。 状态机同样可以编码重要的安全性质。比如，在一个 ATM（自动取款机）的软件实现中， 有一点性质非常重要：ATM 只有在卡片插入且通过密码验证的状态下才能吐出钞票。

本教程将介绍`Control.ST` 库，它支持对带有状态和副作用的程序进行编程和推理。 该库已包含在 Idris 发行版当中（目前属于 `contrib` 包）。本教程假设读者已经熟悉 *Idris* 教程 (éąţ 2) 中所述的纯函数式的编程方法。`ST` 库基于《Idris 类型驱动开发》一书中第 13 和 14 章所讨论的内容，如需更多背景信息可以参考此书。

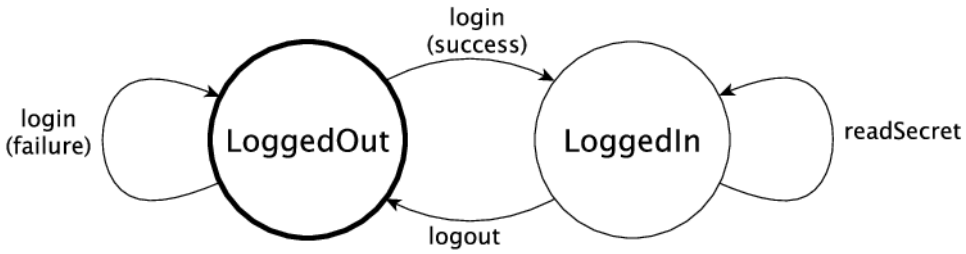
我们可使用 `ST` 库编写由多个状态转移系统复合而成的程序。它支持两种复合方式：第一，我们能够同时使用数个独立实现的状态转移系统；第二，我们能够基于其它状态转移系统实现新的状态转移系统。

3.1.1 示例：需要登录的数据存储系统

许多软件的组件依赖于状态，有些操作仅在特定状态下才有效。 例如，考虑一个安全的数据存储系统，它只有在用户登录的情况下才能访问某些机密数据。 该系统可以处于以下两种状态之一：

- 已登录，在此状态下，用户可以访问机密数据
- 未登录，在此状态下，用户无法访问机密数据

我们可以提供登录、登出和读取数据命令，如下图所示：



如果 登录 命令被成功执行，就会使系统状态从 未登录 转移到 已登录 状态。登出 命令会使系统状态从 已登录 转移到 未登录 状态。最重要的是，读取数据 命令仅在系统处于 已登录 状态时才有效。

我们通常使用类型检查器来确保变量和参数的使用一致性。然而，主流的类型系统并不能很好地支持像 “某些操作只有在资源处于合适的状态时才被执行” 这类性质的静态检查。 例如，在数据存储系统的示例中，检查 “用户在执行 读取数据 前是否成功登录” 这一性质非常重要。 `ST` 库能让我们在类型系统中表达这类**协议**，并且在**编译阶段**确保机密数据只有在用户处于已登录状态时才能被读取。

3.1.2 大纲

本教程从描述如何操作独立的状态开始（*ST 介绍：用状态来工作* (ęąţ 70)），引入了 `STrans` 数据类型来描述带有状态的函数，以及 `ST` 用来描述顶层的状态转移。 接下来的章节（用类型表示状态机 (ęąţ 78)）描述了如何用类型表示状态机，以及如何定义**接口**以描述带有状态的系统。之后（复合状态机 (ęąţ 85)）描述了如何复合带有多个状态机的系统。 它解释了如何实现同时使用多个状态机的系统，以及如何基于低级系统来实现高级的带有状态的系统。 最后（*示例：网络 Socket 编程* (ęąţ 95)）我们将看到一个带有状态的 API 应用于真实场景的例子，它实现了 POSIX 网络 Socket API。

`Control.ST` 库在 Edwin Brady 的一篇文稿 “State Machines All The Way Down” 中亦有提及，你可以从这里获取它。 这篇文章展示了本教程中的很多例子，更加深入地描述了它们的动机、设计、以及实现。

3.2 ST 介绍：用状态来工作

`Control.ST` 库提供了在函数中创建、读取、写入和销毁状态，以及在函数类型中跟踪状态变化的功能。它基于**资源（Resources）**的概念，本质上就是可变变量和依赖类型。 `STrans` 会在函数运行时跟踪资源的变化：

```
STrans : (m : Type -> Type) ->
         (resultType : Type) ->
         (in_res : Resources) ->
         (out_res : resultType -> Resources) ->
         Type
```

类型为 `STrans m resultType in_res out_res_fn` 的值表示一系列可以操作状态的动作。 其参数分

别为：

- `m` 表示一个底层的**计算上下文（Computation Context）**，各种动作会在其中执行。通常，它是一个实现了 `Monad` 的泛型，但并非必须如此。具体来说，我们无需理解单子（Monad）就能高效地使用 `ST`！
- `resultType` 表示这一系列动作会产生的值的类型。
- `in_res` 表示一个在执行动作**之前**可用的**资源**列表。
- `out_res` 表示一个在执行动作**之后**可用的**资源**列表，它随动作的结果而不同。

我们可以用 `STrans` 在函数的类型中描述**状态转移系统（State Transition Systems）**。我们会在之后定义资源（`Resources`），现在你可以先把它看做“现实世界的状态”的抽象表示。通过给定输入资源（`in_res`）和输出资源（`out_res`），我们可以描述允许函数执行的**前提条件（Precondition）**和描述了函数如何影响世界的所有状态的**后置条件（Postcondition）**。

本节以一些 `STrans` 函数的小例子开始，看看如何执行它们。我们还介绍了 `ST`，一个类型级的函数，它能让我们简明地描述带有状态的函数的状态转移。

对示例进行类型检查

对于本节和整个教程，你需要 `import Control.ST` 并通过向 `idris` 传递 `-p contrib` 参数来添加 `contrib` 包。

3.2.1 初试：操作 State

`STrans` 函数的类型解释了它如何影响一组 `Resources`。资源拥有一个类型为 `Var` 的**标签（label）**，我们会用它在函数中引用该资源，并通过 `label ::: type` 的形式将该资源的状态写入“Resources“ 列表。

例如，以下函数的输入资源为 `x`，类型为 `State Integer`；输出资源的类型仍为 `State Integer`：

```
increment : (x : Var) -> STrans m () [x ::: State Integer]
                                     (const [x ::: State Integer])
increment x = do num <- read x
                 write x (num + 1)
```

increment 类型中的冗余

`increment` 的类型看起来有点冗余，其中**输入**和**输出**资源的类型虽然是相同的，但还是重复了一遍。我们在 *ST*：直接表示状态转移 (ęąţ 77)一节中描述 `ST` 类型时，会介绍一种更加简洁的类型写法。

本函数通过 `read` 读取资源 `x` 中存储的值，将其自增后的结果用 `write` 写回资源 `x` 中。稍后我们会看到 `read` 和 `write` 的类型（见 *STrans* 的原语操作 (ęąţ 76)）。我们还可以创建和删除资源：

```
makeAndIncrement : Integer -> STrans m Integer [] (const [])
makeAndIncrement init = do var <- new init
                           increment var
                           x <- read var
                           delete var
                           pure x
```

`makeAndIncrement` 的类型指明了它的入口（`[]`）和出口（`const []`）中**没有**可用的资源。它会用 `new`

创建一个新的 State 资源（接受一个资源的初始值），对其值自增，将该值读出来，然后用 delete 删除它，返回该资源的最终值。我们后面同样会看到 new 和 delete 的类型。

STrans`` （类型为 ``Type -> Type）的参数 m 叫做**计算上下文（Computation Context）**，函数会在其中运行。在这里，该类型级变量指明了我们可以在**任何**上下文中运行它。我们可以在同一个上下文中用 runPure 运行它。例如，将以上定义保存在 Intro.idr 文件中，然后在 REPL 中运行以下语句：

```
*Intro> runPure (makeAndIncrement 93)
94 : Integer
```

通过交互式的，类型驱动的方式来实现 STrans 是个不错的主意。例如，在通过 new init 创建了资源后，你可以为程序剩余的部分留一个**坑（Hole）**来看到资源的创建是如何影响类型的：

```
makeAndIncrement : Integer -> STrans m Integer [] (const [])
makeAndIncrement init = do var <- new init
                           ?whatNext
```

如果你检查一下 ?whatNext 的类型，就会发现有一个可用的资源 var，而在函数调用完成后应当不会再有可用的资源：

```
  init : Integer
  m : Type -> Type
  var : Var
--------------------------------------
whatNext : STrans m Integer [var ::: State Integer] (\value => [])
```

这个小例子可以在任何计算上下文 m 中工作。然而通常，我们会在一个更加严格的上下文中工作。例如，我们可能想要编写一个只能在支持交互式程序的上下文中工作的程序。为此，我们需要学习如何从底层上下文中**提升（Lift）**操作。

3.2.2 提升：使用计算上下文

比如说，我们现在并不想直接把初始整数传入 makeAndIncrement，而是想要把它从控制台读进来。 那么我们就要把一般的工作上下文 m 换成特定的上下文 IO：

```
ioMakeAndIncrement : STrans IO () [] (const [])
```

lift 函数给了我们访问 IO 操作的方式。我们可以将 ioMakeAndIncrement 定义如下：

```
ioMakeAndIncrement : STrans IO () [] (const [])
ioMakeAndIncrement
   = do lift $ putStr "Enter a number: "
        init <- lift $ getLine
        var <- new (cast init)
        lift $ putStrLn ("var = " ++ show !(read var))
        increment var
        lift $ putStrLn ("var = " ++ show !(read var))
        delete var
```

lift 函数能让我们直接使用底层计算上下文（此处为 IO）中的函数。同样，我们很快就会看到 lift 具体的类型。

!-记法

在 ioMakeAndIncrement 中，我们使用了 !(read var) 从资源中读取信息。 你可以在 Idris 教程

（见单子与 *do*-记法 (éąţ 24)）中找到关于 !-记法的详情。 简单来说，它允许我们直接就地使用 STrans 类型的函数，而不必先将其结果绑定到一个变量。

至少从概念上来说，你可以将它当做拥有以下类型的函数：

```
(!) : STrans m a state_in state_out -> a
```

这个语法糖会在执行 do-语句块中的当前动作之前立即绑定一个变量， 然后在 !-表达式的位置就地使用该变量。

然而在实践中，使用像 IO 这样**特定**的上下文通常是种糟糕的做法。首先，它需要我们在代码中到处泼洒 lift，这会影响可读性。再者，也是更重要的一点，它会降低函数的安全性，我们会在下一节（用类型表示状态机 (éąţ 78)）中看到这一点。

所以我们改用定义**接口**的方式来限制计算上下文。例如，Control.ST 中定义了 ConsoleIO 接口，它为控制台的基本交互提供了必要的方法：

```
interface ConsoleIO (m : Type -> Type) where
  putStr : String -> STrans m () res (const res)
  getStr : STrans m String res (const res)
```

也就是说，我们能够以任何可用的资源 res 读写控制台，这两个方法均不会对可用的资源产生影响。IO 对它的实现如下：

```
ConsoleIO IO where
  putStr str = lift (Interactive.putStr str)
  getStr = lift Interactive.getLine
```

现在，我们可以将 ioMakeAndIncrement 定义为：

```
ioMakeAndIncrement : ConsoleIO io => STrans io () [] (const [])
ioMakeAndIncrement
   = do putStr "Enter a number: "
        init <- getStr
        var <- new (cast init)
        putStrLn ("var = " ++ show !(read var))
        increment var
        putStrLn ("var = " ++ show !(read var))
        delete var
```

它不仅可以在特定的 IO 中工作，还可以在一般的 io 上下文中工作，我们只需在该上下文中提供一个 ConsoleIO 的实现即可。相较于初版而言，它有以下优点：

- 所有对 lift 的调用都在接口的实现中，而非在 ioMakeAndIncrement 中
- 我们可以提供另一种 ConsoleIO 的实现，比如在基本的 I/O 中支持异常或日志。
- 在下一节（用类型表示状态机 (éąţ 78)）中我们将会看到，它可以让我们定义安全的 API，以便更加精确地操作具体的资源。

我们之前在同一个上下文中使用 runPure 来运行 makeAndIncrement。而在这里，我们则使用 run，它能够让我们在任何上下文中执行 STrans 程序（只要该上下文实现了 Applicative 即可）。我们可以像下面这样在 REPL 中执行 ioMakeAndIncrement：

```
*Intro> :exec run ioMakeAndIncrement
Enter a number: 93
var = 93
var = 94
```

3.2.3 用依赖类型操作 State

在第一个 State 的例子中，当我们将值自增后，其**类型**并未改变。然而，当我们使用依赖类型时，状态的更新同样也会涉及到其类型的更新。例如，当我们向存储在状态中的向量添加一个元素时，其长度会改变：

```
addElement : (vec : Var) -> (item : a) ->
             STrans m () [vec ::: State (Vect n a)]
                  (const [vec ::: State (Vect (S n) a)])
addElement vec item = do xs <- read vec
                         write vec (item :: xs)
```

注意你需要 import Data.Vect 来执行此示例。

直接用 update 更新状态

除了分别使用 read 和 write 以外，你还可以使用 update，它从一个 State 中读取内容，对它应用一个函数，然后写入其结果。通过 update 你可以将 addElement 写为如下形式：

```
addElement : (vec : Var) -> (item : a) ->
             STrans m () [vec ::: State (Vect n a)]
                  (const [vec ::: State (Vect (S n) a)])
addElement vec item = update vec (item ::)
```

然而，我们并不总是能够知道在一系列动作中类型具体是**如何**变化的。例如，如果我们有一个包含整数向量的状态，那么可以从控制台读取一个输入，只有当该输入为有效的整数时才将它添加到该向量中。根据该整数是否读取成功，我们的输出状态会有不同的类型，简直令人无语。因此，下面两个函数的类型都不太正确：

```
readAndAdd_OK : ConsoleIO io => (vec : Var) ->
                STrans m ()  -- 返回空元组
                           [vec ::: State (Vect n Integer)]
                    (const [vec ::: State (Vect (S n) Integer)])
readAndAdd_Fail : ConsoleIO io => (vec : Var) ->
                  STrans m ()  -- 返回空元组
                             [vec ::: State (Vect n Integer)]
                      (const [vec ::: State (Vect n Integer)])
```

不过请记住，**输出**资源的类型可以从函数的结果中**计算**出来。 目前，我们用 const 表示输出资源总是保持不变。不过在这里，我们可以用一个类型级函数来**计算**出输出资源。我们首先将返回空元组改为 Bool，当读取输入成功时它返回 True；然后为输出资源挖一个**坑**：

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             STrans m Bool [vec ::: State (Vect n Integer)]
                           ?output_res
```

如果你检查 ?output_res 的类型，就会看到 Idris 期望一个类型为 Bool -> Resources 的函数，它表示输出资源的类型可以随 readAndAdd 的结果而不同：

```
  n : Nat
  m : Type -> Type
  io : Type -> Type
  constraint : ConsoleIO io
  vec : Var
--------------------------------------
output_res : Bool -> Resources
```

所以，当输入无效时，输出资源为 Vect n Integer（例如 readAndAdd 返回 False）；当输入有效时，输出资源为 Vect (S n) Integer。我们可以用类型将它表示出来：

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             STrans io Bool [vec ::: State (Vect n Integer)]
                   (\res => [vec ::: State (if res then Vect (S n) Integer
                                                   else Vect n Integer)])
```

接着，我们在实现 readAndAdd 时需要为输出的状态返回适当的值。如果为向量添加了一个元素，就返回 True，否则就要返回 False：

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             STrans io Bool [vec ::: State (Vect n Integer)]
                   (\res => [vec ::: State (if res then Vect (S n) Integer
                                                   else Vect n Integer)])
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then do
                         update vec ((cast num) ::)
                         pure True     -- 添加一个元素，因此返回 True
                       else pure False -- 没有添加元素，因此返回 False
```

如果进行交互式开发的话则稍微有点不同。如果我们挖一个坑，那么在知道要返回的值以前，所需的输出状态并不显而易见。例如，在以下未完成的 readAndAdd 定义中，我们为成功的情况留了个坑：

```
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then ?whatNow
                       else pure False
```

我们可以查看 ?whatNow 的类型，很遗憾信息不足：

```
  vec : Var
  n : Nat
  io : Type -> Type
  constraint : ConsoleIO io
  num : String
--------------------------------------
whatNow : STrans io Bool [vec ::: State (Vect (S n) Integer)]
                 (\res =>
                    [vec :::
                     State (ifThenElse res
                                       (Delay (Vect (S n) Integer))
                                       (Delay (Vect n Integer)))])
```

问题是我们只有在知道值会被返回时才能知道需要的输出状态。为了帮助交互式开发， Control.ST 提供了一个 returning 函数，我们可以用它来提前指定返回值，然后更新相应的状态。例如，我们可以将未完成的 readAndAdd 编写为：

```
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then returning True ?whatNow
                       else pure False
```

它表示在成功的分支中，我们会返回 True，?whatNow 应该解释如何相应地更新状态，使其对于返回值 True 来说是正确的。我们只需检查 ?whatNow，就会发现现在的信息多了一点：

```
  vec : Var
  n : Nat
  io : Type -> Type
  constraint : ConsoleIO io
  num : String
--------------------------------------
whatnow : STrans io () [vec ::: State (Vect n Integer)]
                 (\value => [vec ::: State (Vect (S n) Integer)])
```

现在这个类型表示，在 STrans 的输出资源列表中，我们可以通过向 vec 添加一个元素来完成其定义：

```
readAndAdd vec = do putStr "Enter a number: "
                    num <- getStr
                    if all isDigit (unpack num)
                       then returning True (update vec ((cast num) ::))
                       else returning False (pure ()) -- 返回 False，因此无需更新状态
```

3.2.4 STrans 的原语操作

我们已经写过几个关于 STrans 函数的小例子了，是时候详细地了解这些状态操作函数了。 首先，为了读写状态，我们使用了 read 和 write 函数：

```
read : (lbl : Var) -> {auto prf : InState lbl (State ty) res} ->
       STrans m ty res (const res)
write : (lbl : Var) -> {auto prf : InState lbl ty res} ->
        (val : ty') ->
        STrans m () res (const (updateRes res prf (State ty')))
```

它们的类型看起来有点吓人，特别是隐式的 prf 参数，其类型为：

```
prf : InState lbl (State ty) res
```

它依赖于一个断言 InState。一个类型为 InState x ty res 的值表示在资源列表 res 中，引用 x 的类型必须为 ty。实际上，所有这种类型都表示， 如果一个对某资源的引用存在于资源列表中，那么我们我们只能读取或写入该资源。

给定一个资源标签 res 和一个 res 存在于资源列表中的证明，那么 updateRes 会更新该资源的类型。因此，write 的类型表示该资源的类型会被更新为给定值的类型。

update 的类型与 read 和 write 类型类似，它也需要资源的类型为给定函数的输入类型，并将它更新为该函数的输出类型：

```
update : (lbl : Var) -> {auto prf : InState lbl (State ty) res} ->
         (ty -> ty') ->
         STrans m () res (const (updateRes res prf (State ty')))
```

new 的类型表示它返回一个 Var，给定一个类型为 state 的初始值， 输出资源包含一个新的类型为 State state 的资源：

```
new : (val : state) ->
      STrans m Var res (\lbl => (lbl ::: State state) :: res)
```

新资源的类型为 State state 而非只是 state 这一点很重要，因为这能让我们隐藏 API 的实现细节。在下一节用类型表示状态机 (ệąţ 78)中，我们会看到更多关于其意义的内容。

delete 的类型表示，给定一个标签存在于输入资源内的隐式证明，该标签会从资源列表中移除：

```
delete : (lbl : Var) -> {auto prf : InState lbl (State st) res} ->
         STrans m () res (const (drop res prf))
```

这里的 drop 是一个类型级函数，它用于更新资源列表，从该列表中移除给定的资源 lbl。

我们之前已经用 lift 在底层上下文中运行过函数了。它的类型如下：

```
lift : Monad m => m t -> STrans m t res (const res)
```

给定一个 result 值，pure 会返回产生该值的 STrans 程序， 当产生该值时，它会假设当前资源列表是正确：

```
pure : (result : ty) -> STrans m ty (out_fn result) out_fn
```

我们可以用 returning 将从 STrans 函数中返回值的过程分为两部分：提供值本身， 以及更新资源列表使其对应于该返回值：

```
returning : (result : ty) ->
            STrans m () res (const (out_fn result)) ->
            STrans m ty res out_fn
```

最后，我们已经用 run 和 runPure 在特定上下文中执行过 STrans 函数了。run 会在任何上下文中执行函数，若该上下文实现了 Applicative， 那么 runPure 会在同一上下文中执行函数：

```
run : Applicative m => STrans m a [] (const []) -> m a
runPure : STrans Basics.id a [] (const []) -> a
```

注意在任何情况下，输入和输出资源列表都必须为空。没有一种方法能够提供初始资源列表， 或提取最终的资源。这是有意设计的：它确保了**所有的**资源管理都在受控的 STrans 环境下进行，并且我们将会看到，这能够让我们实现安全的 API， 以精确的类型来解释在程序的执行过程中，资源是如何被跟踪的。

这些函数构成了 ST 库的核心。在遇到更加复杂的情况时，我们还会用到一些其它的函数， 不过目前所见的函数足以让我们用 Idris 进行细致的状态跟踪和推理了。

3.2.5 ST：直接表示状态转移

我们已经见过一些简单的 STrans 函数的例子了，由于需要提供显式的输入输出资源列表， 它们的类型会变得非常冗长。在需要为原语操作提供类型时这很方便，不过对于更一般的使用来说， 能为独立的资源表示**状态转移**，而无需完整地给出输入和输出资源列表的话会更加方便。 我们可以用 ST 来做到这一点：

```
ST : (m : Type -> Type) ->
     (resultType : Type) ->
     List (Action resultType) -> Type
```

ST 是一个类型级函数，它会为给定的**活动（Action）**列表计算出对应的 STrans 类型，该类型描述了资源的状态转移。函数类型中的 Action 可接受以下形式（我们之后还会见到其它形式）：

- lbl ::: ty 表示资源 lbl 的开始和结束状态均为 ty
- lbl ::: ty_in :-> ty_out 表示资源 lbl 以状态 ty_in 开始，以状态 ty_out 结束
- lbl ::: ty_in :-> (\res -> ty_out) 表示资源 lbl 以状态 ty_in 开始，以状态 ty_out 结束，其中 ty_out 从函数 res 的结果中计算而来。

现在，我们可以将前面的一些函数的类型写成如下形式：

```
increment : (x : Var) -> ST m () [x ::: State Integer]
```

即，increment 的开始和结束状态均为 State Integer 状态的 x。

```
makeAndIncrement : Integer -> ST m Integer []
```

即，makeAndIncrement 的开始和结束均没有资源。

```
addElement : (vec : Var) -> (item : a) ->
             ST m () [vec ::: State (Vect n a) :-> State (Vect (S n) a)]
```

即，addElement 将 vec 从 State (Vect n a) 改变为 State (Vect (S n) a)。

```
readAndAdd : ConsoleIO io => (vec : Var) ->
             ST io Bool
                   [vec ::: State (Vect n Integer) :->
                    \res => State (if res then Vect (S n) Integer
                                          else Vect n Integer)]
```

我们通过这种编写类型的方式，表达出了每个函数如何影响整体资源状态的最小必要条件。 如果某个资源的更新依赖于某个结果（如 readAndAdd），那么我们需要完整地描述它。 否则（如 increment 和 makeAndIncrement），我们可以写出输入输出资源列表以避免重复。

Action 也可以描述**添加**和**移除**状态：

- add ty，如果该操作返回一个 Var，那么它会添加一个 ty 类型的新资源。
- remove lbl ty 表示该操作会从资源列表中的状态 ty 开始，移除名为 lbl 的资源。

例如，我们可以写出：

```
newState : ST m Var [add (State Int)]
removeState : (lbl : Var) -> ST m () [remove lbl (State Int)]
```

第一个函数 newState 返回一个新的资源标签并将该资源添加到 State Int 类型的资源列表中。第二个函数 removeState 根据给定的标签 lbl 从列表中移除该资源。二者的类型与以下形式等价：

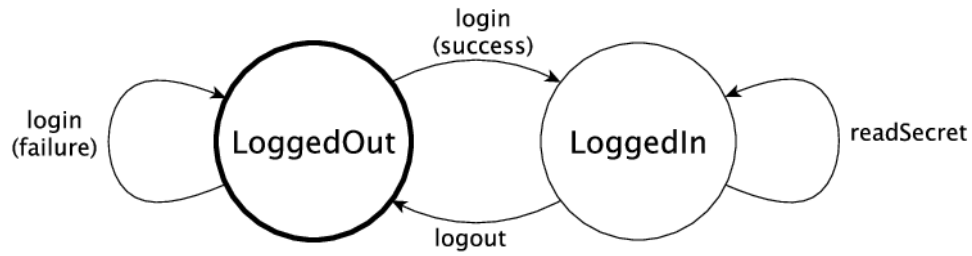
```
newState : STrans m Var [] (\lbl => [lbl ::: State Int])
removeState : (lbl : Var) -> STrans m () [lbl ::: State Int] (const [])
```

它们是构造 Action 的原语方法。我们后面还会遇到一些用类型级函数来提高可读性的方式。

除了极少数需要准确完整的 STrans 的情况外，在本教程剩余的部分中，我们通常会使用 ST。在下一节中，我们会看到如何用 ST 提供的设施来为需要安全性的系统编写准确的 API：一个需要登录的数据存储系统。

3.3 用类型表示状态机

在概览 (ęąţ 69)一节中，我们用状态转移图展示了数据存储系统的抽象状态， 以及可以对它执行的动作：



我们之所以把它称作该存储系统的**抽象**状态，是因为具体的状态会包含更多信息。例如，它可能包含用户名、散列化的密码和存储内容等等。然而，就目前我们我们所关心的动作 `login`、`logout` 和 `readSecret` 而言，登录状态决定了哪些动作是有效的。

我们已经见过如何用 `ST` 操作状态，用依赖类型表示状态了。在本节中，我们会看到如何用 `ST` 为数据存储系统提供安全的 API。在 API 中，我们会用类型编码上面的状态转移图。 通过这种方式，我们可以只在状态有效时才能执行 `login`、`logout` 和 `readSecret` 操作。

我们已经用过 `State` 及其原语操作 `new`、`read`、`write` 和 `delete` 来操作状态了。而对于数据存储的 API，我们则以定义**接口**开始（见 Idris 教程中的接口 (éąţ 21)一节）。接口描述了对存储系统的操作，其类型则准确解释了每种操作何时才有效，以及它是如何影响存储系统的状态的。通过接口，我们可以确保这是访问存储系统的**唯一的**方式。

3.3.1 为数据存储系统定义接口

我们首先在 `Login.idr` 文件中定义数据类型，它表示存储系统的两种抽象状态，即 `LoggedOut` 和 `LoggedIn`:

```
data Access = LoggedOut | LoggedIn
```

我们可以定义一个数据类型来表示存储系统的当前状态，保存所有必要的信息（如用户名、散列化的密码、存储内容等等），并根据登录状态来参数化该数据类型:

```
Store : Access -> Type
```

不过我们现在先不定义具体的类型，而是将以下代码包含在数据存储的**接口**中，之后再定义具体的类型:

```
interface DataStore (m : Type -> Type) where
  Store : Access -> Type
```

我们可以继续为此接口补充其它存储系统的操作。这样做优点众多。通过将**接口**与其**实现**相分离，我们可以为不同的上下文提供相应的具体实现。此外，我们还可以编写与存储系统协作的程序而无需知道任何实现的细节。

我们需要用 `connect` 连接到该存储系统，在结束后用 `disconnect` 断开连接。 我们为 `DataStore` 接口添加以下方法:

```
connect : ST m Var [add (Store LoggedOut)]
disconnect : (store : Var) -> ST m () [remove store (Store LoggedOut)]
```

`connect` 的类型表明它会返回一个初始类型为 `Store LoggedOut` 的新资源。 反之，`disconnect` 则会给出一个状态为 `Store LoggedOut` 的资源并移除该资源。 我们可以通过以下（未完成的）定义更加清楚地看到 `connect` 做了什么:

```
doConnect : DataStore m => ST m () []
doConnect = do st <- connect
               ?whatNow
```

注意我们正在一个 **一般**的上下文 m 中工作，为了能够执行 doConnect，我们必须为 m 实现 DataStore 接口来限制它。如果我们检查 ?whatNow 的类型，就会看到剩下的操作以一个状态为 Store LoggedOut 的资源 st 开始， 以没有可用的资源结束：

```
  m : Type -> Type
  constraint : DataStore m
  st : Var
--------------------------------------
whatNow : STrans m () [st ::: Store LoggedOut] (\result => [])
```

接着，我们可以用 disconnect 来移除该资源：

```
doConnect : DataStore m => ST m () []
doConnect = do st <- connect
               disconnect st
               ?whatNow
```

现在检查 ?whatNow 的类型会显示我们没有可用的资源：

```
  m : Type -> Type
  constraint : DataStore m
  st : Var
--------------------------------------
whatNow : STrans m () [] (\result => [])
```

为了继续完善 DataStore 接口的实现，我们接下来添加一个读取机密数据的方法。 这需要 store 的状态为 Store LoggedIn：

```
readSecret : (store : Var) -> ST m String [store ::: Store LoggedIn]
```

此时我们可以试着编写一个函数，它先连接到存储系统，然后读取机密数据，之后断开连接。 然而它并不会成功，因为执行 readSecret 需要我们处于已登录状态。

```
badGet : DataStore m => ST m () []
badGet = do st <- connect
            secret <- readSecret st
            disconnect st
```

它会产生以下错误，因为 connect 创建了状态为 LoggedOut 的新存储，而 readSecret 需要该存储的状态为 LoggedIn：

```
When checking an application of function Control.ST.>>=:
    Error in state transition:
            Operation has preconditions: [st ::: Store LoggedOut]
            States here are: [st ::: Store LoggedIn]
            Operation has postconditions: \result => []
            Required result states here are: \result => []
```

该错误信息解释了所需的输入状态（前提条件）和输出状态（后置条件）与该操作中的状态有何不同。为了使用 readSecret，我们需要一种方式将 Store LoggedOut 转换为 Store LoggedIn 状态。我们可以先尝试将 login（登录）指定为以下类型：

```
login : (store : Var) -> ST m () [store ::: Store LoggedOut :-> Store LoggedIn] -- 类型不正确！
```

注意，**接口**中并没有说明 login 是**如何**工作的，只是表达了它如何影响所有的状态。即便如此，login

的类型还是有点问题，因为它假设了登录总会成功。如果登录失败（比如在该实现提示输入密码时用户输入了错误的密码），那么它一定不会产生 LoggedIn 状态的存储。

因此，login 需要通过以下类型返回登录是否成功：

```
data LoginResult = OK | BadPassword
```

接着，我们可以从结果中**计算**出结果状态（见用依赖类型操作 *State* (éąţ 74)）。我们为 DataStore 接口添加以下方法：

```
login : (store : Var) ->
        ST m LoginResult [store ::: Store LoggedOut :->
                          (\res => Store (case res of
                                               OK => LoggedIn
                                               BadPassword => LoggedOut))]
```

如果 login 成功，那么 login 之后的状态会变成 Store LoggedIn。否则，状态仍然为 Store LoggedOut。

为完成此接口，我们还需要添加一个退出该存储系统的方法。我们假设退出总是成功，并将存储系统的状态从 Store LoggedIn 转换为 Store LoggedOut。

```
logout : (store : Var) -> ST m () [store ::: Store LoggedIn :-> Store LoggedOut]
```

这样就完成了此接口。完整代码如下：

```
interface DataStore (m : Type -> Type) where
  Store : Access -> Type

  connect : ST m Var [add (Store LoggedOut)]
  disconnect : (store : Var) -> ST m () [remove store (Store LoggedOut)]

  readSecret : (store : Var) -> ST m String [store ::: Store LoggedIn]
  login : (store : Var) ->
          ST m LoginResult [store ::: Store LoggedOut :->
                            (\res => Store (case res of
                                                 OK => LoggedIn
                                                 BadPassword => LoggedOut))]
  logout : (store : Var) -> ST m () [store ::: Store LoggedIn :-> Store LoggedOut]
```

在尝试创建此接口的实现之前，我们来看看如何用它来编写函数，以此来登录数据存储系统、在登录成功后读取机密数据，然后再退出。

3.3.2 用数据存储接口编写函数

我们以编写 getData 函数为例，展示如何使用 DataStore 接口。 该函数用于连接到存储系统并从中读取数据。我们使用该操作的类型来逐步指导开发， 交互式地编写此函数。它的类型如下：

```
getData : (ConsoleIO m, DataStore m) => ST m () []
```

该类型表示在进入或退出时没有资源可用。也就是说，整个动作列表为 []，这表示至少从外部来说，该函数完全没有对资源产生作用。换句话说，对于每一个在调用 getData 时创建的资源，我们都需要在退出前删除它。

由于我们要使用 DataStore 接口的方法，因此必须约束计算上下文 m 使其实现 DataStore 接口。我们还有一个 ConsoleIO m 约束，这样就能将我们从存储系统中读取的任何数据或者错误信息显示出来。

我们从连接到存储系统开始，创建一个新的资源 st，然后尝试用 login 登录：

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             ?whatNow
```

登录可能成功也可能失败，两种状态可从 ok 值反映出来。如果我们检查 ?whatNow 的类型，就会看到当前存储系统的状态:

```
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : DataStore m
  st : Var
  ok : LoginResult
--------------------------------------
whatNow : STrans m () [st ::: Store (case ok of
                                       OK => LoggedIn
                                       BadPassword => LoggedOut)]
                      (\result => [])
```

由于 st 的当前状态依赖于 ok 的值，因此我们可以对 ok 分情况讨论来继续推进:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             case ok of
                  OK => ?whatNow_1
                  BadPassword => ?whatNow_2
```

两个分支上的坑 ?whatNow_1 和 ?whatNow_2 的类型展现了状态是如何随着登录成功与否而改变的。如果登录成功，那么该存储系统的状态为 LoggedIn:

```
--------------------------------------
whatNow_1 : STrans m () [st ::: Store LoggedIn] (\result => [])
```

如果失败，那么它的状态为 LoggedOut:

```
--------------------------------------
whatNow_2 : STrans m () [st ::: Store LoggedOut] (\result => [])
```

在 ?whatNow_1 中，由于登录成功，因此可以读取机密数据并将它显示在终端上:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             case ok of
                  OK => do secret <- readSecret st
                           putStrLn ("Secret is: " ++ show secret)
                           ?whatNow_1
                  BadPassword => ?whatNow_2
```

我们要以「无资源可用」的状态来结束 OK 分支，因此需要退出存储系统并断开连接:

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             ok <- login st
             case ok of
                  OK => do secret <- readSecret st
                           putStrLn ("Secret is: " ++ show secret)
                           logout st
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
                    disconnect st
                BadPassword => ?whatNow_2
```

注意我们在调用 disconnect 断开连接前，**必须**用 logout 退出 st，因为 disconnect 需要存储系统处于 LoggedOut 状态。

此外，我们不能像上一节中 State 的示例那样，简单地用 delete 来删除该资源，因为对于某个类型 ty 来说，delete 只能在资源的类型为 State ty 时起效。如果我们试图用 delete 来代替 disconnect，就会看到以下错误：

```
When checking argument prf to function Control.ST.delete:
        Can't find a value of type
                InState st (State st) [st ::: Store LoggedOut]
```

换句话说，类型检查器找不到一个「资源 st 拥有 State st 形式的类型」的证明，因为其类型为 Store LoggedOut。由于 Store 是 DataStore 接口的一部分，而我们**尚未知道** Store 的具体表示，因此我们需要通过此接口的 disconnect 而非直接用 delete 来删除资源。

我们可以将 getData 完成如下，使用模式匹配来绑定候选（见 Idris 教程的单子与 *do*-记法 (éąţ 24)），而非使用 case 语句来捕获 login 可能产生的错误：

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

然而它现在还跑不起来，因为我们还没有任何 DataStore 的实现！如果我们试着在一个 IO 上下文中执行它，就会产生一个没有 DataStore IO 的实现的错误：

```
*Login> :exec run {m = IO} getData
When checking an application of function Control.ST.run:
        Can't find implementation for DataStore IO
```

因此，要实现遵循其状态转移图的数据存储系统，最后一步就是提供一个 DataStore 的实现。

3.3.3 实现接口

要在 IO 中执行 getData，我们需要提供一个能够在 IO 上下文中工作的 DataStore 的实现。我们可以这样开始：

```
implementation DataStore IO where
```

接着，我们可以让 Idris 根据必要方法的基本定义来填充该接口（在 Atom 中按下 Ctrl-Alt-A，或者在你喜欢的编辑器中按下对应的快捷键来「添加定义」）：

```
implementation DataStore IO where
  Store x = ?DataStore_rhs_1
  connect = ?DataStore_rhs_2
  disconnect store = ?DataStore_rhs_3
  readSecret store = ?DataStore_rhs_4
  login store = ?DataStore_rhs_5
  logout store = ?DataStore_rhs_6
```

我们首先要确定表示该数据存储系统的方式。为简单起见，我们将数据存储为单个的 String，并使用硬编码的密码来获取访问权限。我们可以将 Store 定义如下，无论在 LoggedOut 还是在 LoggedIn 状态下均使用 String 来表示数据。

```
Store x = State String
```

现在我们给出了 Store 的一个具体类型，我们可以实现建立连接、断开连接和访问数据的操作。而由于我们使用了 State，因此也就可以使用 new、delete、read 和 write 来操作该存储系统。

坑的类型会告诉我们如何操作状态。例如，?DataStore_rhs_2 坑告诉我们要实现 connect 需要做些什么。我们需要返回一个新的 Var，表示一个类型为 State String 的资源：

```
--------------------------------------
DataStore_rhs_2 : STrans IO Var [] (\result => [result ::: State String])
```

我们可以通过创建一个带有某些数据作为存储内容的新变量来实现它（我们可以使用任何 String），然后返回该变量：

```
connect = do store <- new "Secret Data"
             pure store
```

对于 disconnect 而言，我们只需删除该资源即可：

```
disconnect store = delete store
```

对于 readSecret，我们需要读取机密数据并返回 String。由于我们并不知道该数据的具体表示为 State String，因此可以直接用 read 来访问数据：

```
readSecret store = read store
```

我们先来完成 logout，之后回到 login 上来。检查坑的类型会显示以下信息：

```
  store : Var
--------------------------------------
DataStore_rhs_6 : STrans IO () [store ::: State String] (\result => [store ::: State String])
```

因此在此小型实现中，我们实际上不用做任何事情！

```
logout store = pure ()
```

对于 login，我们需要返回登录是否成功。为此，我们需要提示用户输入密码，并在匹配到硬编码的密码时返回 OK，否则返回 BadPassword：

```
login store = do putStr "Enter password: "
                 p <- getStr
                 if p == "Mornington Crescent"
                    then pure OK
                    else pure BadPassword
```

下面给出完整的实现以供参考，它能让我们在 REPL 中执行 DataStore 程序：

```
implementation DataStore IO where
  Store x = State String
  connect = do store <- new "Secret Data"
               pure store
  disconnect store = delete store
  readSecret store = read store
  login store = do putStr "Enter password: "
                   p <- getStr
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
                  if p == "Mornington Crescent"
                     then pure OK
                     else pure BadPassword
  logout store = pure ()
```

最后，我们可以像下面这样在 REPL 中尝试它（如果有可用的 IO 实现，那么在 Idris 的 REPL 中，上下文会默认为 IO，因此这里无需显式给出 m 参数）：

```
*Login> :exec run getData
Enter password: Mornington Crescent
Secret is: "Secret Data"

*Login> :exec run getData
Enter password: Dollis Hill
Failure
```

对于 State 类型的资源，我们只能使用 read、write、new 和 delete。因此，在 DataStore 的实现或任何已知上下文为 IO 的环境**内部**，我们可以随意访问数据存储系统，因为这里是实现 DataStore 内部细节的地方。然而，如果我们只有 DataStore m 的约束，那么就无法知道该存储系统是否已实现，因此我们只能通过 DataStore 提供的 API 来访问它。

因此比较好的做法是在 getData 这类函数中使用**泛型（Generic）**上下文 m，并只根据我们需要的接口进行约束，而非使用具体的上下文 IO。

现在我们已经学过如何处理状态，以及如何用接口来封装数据存储这类具体系统的状态转移了。 然而，真正的系统需要能够**复合**多种状态机。我们一次需要使用多个状态机，或者基于多个状态机来实现一个状态机。

3.4 复合状态机

在上一节中，我们定义了 DataStore 接口并用它实现了以下小程序，它能让用户登录该存储系统然后打印存储的内容：

```
getData : (ConsoleIO m, DataStore m) => ST m () []
getData = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

该函数只用了一个状态，即存储本身。然而，更大的程序通常会有更多的状态，在执行过程中可能还需要添加、删除或更新状态。就本例而言，无限循环，在状态中记录登录失败的次数都是有用的扩展。

此外，状态机还可以有**层级**，即一个状态机可以通过复合其它状态机来实现。例如，我们可以有一个表示图形系统的状态机，并用它来实现「海龟绘图」这种**高级**的图形 API，「海龟绘图」会使用图形系统加上一些额外的状态。

在本节中，我们会看到如何使用多个状态工作，以及如何将状态机复合成更高级别的状态机。我们先来看看如何为 getData 添加登录失败的计数器。

3.4.1 使用多个资源来工作

为了了解如何使用多个资源来工作，我们需要修改 getData 使其循环，然后记录用户登录失败的总次数。例如，如果我们编写的 main 程序初始记录次数为 0， 那么会话过程看起来可能会是这样的：

```
*LoginCount> :exec main
Enter password: Mornington Crescent
Secret is: "Secret Data"
Enter password: Dollis Hill
Failure
Number of failures: 1
Enter password: Mornington Crescent
Secret is: "Secret Data"
Enter password: Codfanglers
Failure
Number of failures: 2
...
```

我们首先为 getData 添加一个跟踪失败次数的状态资源：

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
```

getData 的类型检查

如果你跟着本教程写代码，就会发现更新了 getData 的类型后就无法通过编译了。这是意料之中的！我们暂且先将 getData 的定义注释掉，回头再说。

接着，我们创建一个 main 程序，它将状态初始化为 0，然后调用 getData：

```
main : IO ()
main = run (do fc <- new 0
               getData fc
               delete fc)
```

现在着手实现 getData。我们先来添加一个参数表示允许的失败次数：

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
        = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

然而，它无法通过类型检查，因为我们用来调用 connect 的资源不正确。错误信息描述了资源为何不必配：

```
When checking an application of function Control.ST.>>=:
    Error in state transition:
            Operation has preconditions: []
            States here are: [failcount ::: State Integer]
            Operation has postconditions: \result => [result ::: Store LoggedOut] ++ []
            Required result states here are: st2_fn
```

换句话说，connect 需要在进入时**没有**资源，但我们却**有一个**资源，即失败次数！这理应没什么问题：毕竟我们需要的资源是拥有的资源的**子集**，而额外的资源（这里是失败次数）与 connect 无关。因此我们需要一种临时**隐藏**附加资源的方法。

我们可以用 call 函数来达此目的：

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
   = do st <- call connect
        ?whatNow
```

我们在这里为 getData 剩下的部分挖了个坑，这样你可以看到 call 的作用。它移除了调用 connect 时资源列表中不必要的部分，然后在返回时恢复了它们。因此 whatNow 的类型表明我们添加了一个新的资源 st，而 failcount依然可用：

```
  failcount : Var
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : DataStore m
  st : Var
--------------------------------------
whatNow : STrans m () [failcount ::: State Integer, st ::: Store LoggedOut]
                      (\result => [failcount ::: State Integer])
```

在此函数函数的最后，whatNow 表明我们需要以状态 st 结束，然而此时尚有 failcount 可用。我们可以在调用数据存储系统中的操作时添加 call 来消除资源列表，这样完成的 getData 就能带着附加的状态资源工作：

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
        = do st <- call connect
             OK <- call $ login st
                | BadPassword => do putStrLn "Failure"
                                    call $ disconnect st
             secret <- call $ readSecret st
             putStrLn ("Secret is: " ++ show secret)
             call $ logout st
             call $ disconnect st
```

这样有点嗦，实际上我们可以将 call 变成隐式的从而去掉它。默认情况下，你需要显式地添加 call，但如果你导入了 Control.ST.ImplicitCall，那么 Idris 就会在需要的地方插入它。

```
import Control.ST.ImplicitCall
```

现在的 getData 就和之前一样了：

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
        = do st <- connect
             OK <- login st
                | BadPassword => do putStrLn "Failure"
                                    disconnect st
             secret <- readSecret st
             putStrLn ("Secret is: " ++ show secret)
             logout st
             disconnect st
```

这里需要权衡：如果你导入了 Control.ST.ImplicitCall，那么使用多个资源的函数会更加易读，因

为没有嗦的 call 了。另一方面，Idris 对你函数的类型检查会变得有点困难，这会导致它花费更多时间，错误信息也会少一点帮助。

看一下 call 的类型，你会有所启发：

```
call : STrans m t sub new_f -> {auto res_prf : SubRes sub old} ->
       STrans m t old (\res => updateWith (new_f res) old res_prf)
```

被调用的函数有一个资源列表 sub，还有一个隐式证明 SubRes sub old，它证明了被调用函数的资源列表是整个资源列表的子集。尽管资源的顺序可以改变，然而在 old 中出现的资源无法在 sub 列表中出现超过一次（如果你尝试它就会得到一个类型错误）。

函数 updateWith 接受被调用函数的**输出**资源，然后在当前资源列表中更新它们。此函数会尽可能保持顺序不变，尽管被调用的函数在进行复杂的资源操作时并不总是可以保持顺序。

在被调用的函数中新创建的资源

如果被调用的函数创建了新的资源，那么基于 updateWith 的工作方式，它们通常会出现在资源列表的**末尾**。你可以在前面未完成的 getData 的定义中看到这一点。

最后我们可以更新 getData 使其可以循环，并在需要时保持更新 failCount：

```
getData : (ConsoleIO m, DataStore m) =>
          (failcount : Var) -> ST m () [failcount ::: State Integer]
getData failcount
   = do st <- call connect
        OK <- login st
           | BadPassword => do putStrLn "Failure"
                               fc <- read failcount
                               write failcount (fc + 1)
                               putStrLn ("Number of failures: " ++ show (fc + 1))
                               disconnect st
                               getData failcount
        secret <- readSecret st
        putStrLn ("Secret is: " ++ show secret)
        logout st
        disconnect st
        getData failcount
```

注意在这里我们会在每次迭代中建立并断开连接。另一种实现方式是首先用 connect 建立连接，然后调用 getData，其实现如下：

```
getData : (ConsoleIO m, DataStore m) =>
          (st, failcount : Var) -> ST m () [st ::: Store {m} LoggedOut, failcount ::: State␣
↪Integer]
getData st failcount
   = do OK <- login st
           | BadPassword => do putStrLn "Failure"
                               fc <- read failcount
                               write failcount (fc + 1)
                               putStrLn ("Number of failures: " ++ show (fc + 1))
                               getData st failcount
        secret <- readSecret st
        putStrLn ("Secret is: " ++ show secret)
        logout st
        getData st failcount
```

在 st 的类型 Store {m} LoggedOut 中添加显式的 {m} 是十分重要的，因为这给了 Idris 足够的信息来判断哪一个 DataStore 的实现是用于查找其对应的 Store 的实现的。 否则，如果我们只写 Store

LoggedOut，那么将无法获知 Store 所关联到的计算上下文 m。

接着我们可以在 main 中只 connect 并 disconnect 一次：

```
main : IO ()
main = run (do fc <- new 0
               st <- connect
               getData st fc
               disconnect st
               delete fc)
```

通过使用 call 或导入 Control.ST.ImplicitCall，我们可以编写使用多个资源的程序， 然后在调用一个只用到全部资源的子集的函数时，将资源列表按需删减。

3.4.2 复合资源：状态机的层级

我们现在已经见过如何在一个函数中使用多个资源了，这对于任何能够操作状态的实际的程序来说都是必须的。我们可以把它看做是「横向的」复合：一次使用多个资源。我们通常还需要「纵向」的复合：基于一个或多个资源来实现单个资源。

在本节中，我们会看到一个这种的例子。首先，我们在一个接口 Draw 中实现一个小型的图形 API，它支持：

- 打开一个窗口，创建一个双缓冲（double-buffered）的平面（surface）来绘图
- 在平面上绘制线条和矩形
- 缓冲区「翻页（flipping）」，在窗口中显示我们刚绘制完毕的图像
- 关闭一个窗口

提示： 绘图并不是即时呈现的，需要一个缓冲区（buffer）来暂存即将呈现的图像。 缓冲区一般有三个基本操作：clear（清空），flip（翻页）和 rewind（重显）。 clear 会将当前缓冲区清空以便重新绘图；flip 会将当前缓冲区中的图像显示在屏幕上，然后清空缓冲区等待下一次绘图；而 rewind 则保持缓冲区内容不变，重新显示到屏幕上。

术语 flip 和 rewind 来源于磁带的播放，rewind 即倒带重放，而 flip 则表示把磁带翻过来播放另一面。flip 一词尚无标准译法，此处译作「翻页」表示本画面完成，开始下一画面。

接着，我们在一个 interface 中用此 API 来为「海龟绘图」实现一个更高级的 API。这不仅需要 Draw 接口，还需要表示海龟的状态（位置，方向和画笔颜色）。

SDL 绑定

为了测试本节中的示例，你需要为 Idris 安装一个非常基本的 SDL 绑定，它可从 https://github.com/edwinb/SDL-idris 获取。这些绑定实现了 SDL API 的一个很小的子集，只用作演示的目的。尽管如此，它们已经足以实现一个小型的绘图程序来展示本节的概念了。

一旦你安装完这个包，就可以通过参数启动 Idris 了，-p sdl 用于 SDL 绑定，-p contrib 用于 Control.ST。

Draw 接口

我们要在此 API 中使用 Idris 的 SDL 绑定，因此你需要在安装完该绑定库后导入 Graphics.SDL。我们先来定义 Draw 接口，它包含一个表示平面的数据类型，我们会在其之上绘制线条和矩形：

```
interface Draw (m : Type -> Type) where
    Surface : Type
```

我们需要能够通过打开新窗口来创建新的 Surface：

```
initWindow : Int -> Int -> ST m Var [add Surface]
```

然而这样不太正确。如果我们的程序运行在没有可用的窗口系统的终端环境中，那么打开窗口可能会失败。因此， initWindow 需要以某种方式来应对可能的失败。我们可以通过返回 Maybe Var 而非 Var，以及只在成功时添加 Surface 来做到这一点：

```
initWindow : Int -> Int -> ST m (Maybe Var) [addIfJust Surface]
```

它使用了 Control.ST 中定义的类型级函数 addIfJust，该函数返回一个 Action，仅在操作成功时添加资源（也就是说，它返回一个形如 Just val 的结果）。

addIfJust 与 addIfRight

Control.ST 中定义了能够在操作成功时构造新资源的函数，其中的 addIfJust 会在操作返回 Just ty 时添加资源。此外还有 addIfRight：

```
addIfJust : Type -> Action (Maybe Var)
addIfRight : Type -> Action (Either a Var)
```

二者均基于下面的原语动作 Add 开发。此动作接受一个函数，该函数从操作的结果中构造出一个资源列表：

```
Add : (ty -> Resources) -> Action ty
```

如有需要，你可以用它来创建自己的动作，以此来基于某个操作的结果添加资源。例如，addIfJust 的实现如下：

```
addIfJust : Type -> Action (Maybe Var)
addIfJust ty = Add (maybe [] (\var => [var ::: ty]))
```

如果我们能创建窗口，那么也需要能删除它：

```
closeWindow : (win : Var) -> ST m () [remove win Surface]
```

我们还需要响应按下键盘或点击鼠标这类事件。Graphics.SDL 库为此提供了 Event 类型，而我们可以用 poll 轮询事件，如果存在的话，它会返回最后一个发生的事件：

```
poll : ST m (Maybe Event) []
```

Draw 中剩下的方法包括：flip，它会将从上次 flip 以来绘制的所有图像都显示出来；还有两个绘图的方法 filledRectangle 和 drawLine。

```
flip : (win : Var) -> ST m () [win ::: Surface]
filledRectangle : (win : Var) -> (Int, Int) -> (Int, Int) -> Col -> ST m () [win ::: Surface]
drawLine : (win : Var) -> (Int, Int) -> (Int, Int) -> Col -> ST m () [win ::: Surface]
```

我们按如下方式定义色彩，四个整数表示色彩通道（红、绿、蓝、不透明度）：

```
data Col = MkCol Int Int Int Int

black : Col
black = MkCol 0 0 0 255

red : Col
red = MkCol 255 0 0 255

green : Col
green = MkCol 0 255 0 255

-- 蓝、黄、品红、青、白类似……
```

在导入 Graphics.SDL 之后，你就可以像下面这样用 SDL 的绑定实现 Draw 接口了：

```
implementation Draw IO where
  Surface = State SDLSurface

  initWindow x y = do Just srf <- lift (startSDL x y)
                            | pure Nothing
                      var <- new srf
                      pure (Just var)

  closeWindow win = do lift endSDL
                       delete win

  flip win = do srf <- read win
                lift (flipBuffers srf)
  poll = lift pollEvent

  filledRectangle win (x, y) (ex, ey) (MkCol r g b a)
       = do srf <- read win
            lift $ filledRect srf x y ex ey r g b a
  drawLine win (x, y) (ex, ey) (MkCol r g b a)
       = do srf <- read win
            lift $ drawLine srf x y ex ey r g b a
```

在本实现中，我们使用 startSDL 来初始化窗口，它在失败时返回 Nothing。由于 initWindow 的类型说明它会在返回形如 Just val 的值时添加一个资源，因此我们在成功时添加 startSDL 返回的平面，在失败时什么也不做。我们只能在 startDSL 成功时初始化成功。

现在我们有了 Draw 的实现，可以试着写一些函数了，他们在窗口中绘图并通过 SDL 绑定执行它们。例如，假设我们有一个可以绘图的平面 win，那么可以编写 render 函数在黑色背景上绘图：

```
render : Draw m => (win : Var) -> ST m () [win ::: Surface {m}]
render win = do filledRectangle win (0,0) (640,480) black
                drawLine win (100,100) (200,200) red
                flip win
```

最后的 flip win 是必须的，因为绘图原语使用了双缓冲区来避免图形闪烁。我们在屏幕之外的缓冲区上绘图，同时显示另一个缓冲区。在调用 flip 时，它会将当前屏幕之外的缓冲区显示出来，并创建一个新的屏幕外缓冲区绘制下一帧。

要将它包含在程序里，我们需要编写一个主循环来渲染图像，同时等待用户关闭应用的事件：

```
loop : Draw m => (win : Var) -> ST m () [win ::: Surface {m}]
loop win = do render win
              Just AppQuit <- poll
                   | _ => loop win
              pure ()
```

最后，我们创建一个主程序。如果可能，它会创建窗口，然后运行主循环：

```
drawMain : (ConsoleIO m, Draw m) => ST m () []
drawMain = do Just win <- initWindow 640 480
                 | Nothing => putStrLn "Can't open window"
              loop win
              closeWindow win
```

我们可以在 REPL 中用 `run` 运行它：

```
*Draw> :exec run drawMain
```

更高级的接口：TurtleGraphics

「海龟绘图」会操纵一只「海龟」在屏幕上移动，在它移动时用「画笔」来画线。一只海龟拥有描述它位置的属性，它面对的方向，以及当前画笔的颜色。此外，还有一些命令来让海龟向前移动，转一个角度，以及更改画笔的颜色。下面是一种可行的接口：

```
interface TurtleGraphics (m : Type -> Type) where
  Turtle : Type

  start : Int -> Int -> ST m (Maybe Var) [addIfJust Turtle]
  end : (t : Var) -> ST m () [Remove t Turtle]

  fd : (t : Var) -> Int -> ST m () [t ::: Turtle]
  rt : (t : Var) -> Int -> ST m () [t ::: Turtle]

  penup : (t : Var) -> ST m () [t ::: Turtle]
  pendown : (t : Var) -> ST m () [t ::: Turtle]
  col : (t : Var) -> Col -> ST m () [t ::: Turtle]

  render : (t : Var) -> ST m () [t ::: Turtle]
```

和 `Draw` 一样，我们也需要一个初始化海龟的命令（这里叫做 `start`），如果它无法创建用来绘图的平面就会失败。此外还有一个 `render` 方法，它用来渲染窗口中目前已绘制的图像。使用此接口的一个可用的程序如下所示，它画了一个彩色的正方形：

```
turtle : (ConsoleIO m, TurtleGraphics m) => ST m () []
turtle = with ST do
             Just t <- start 640 480
                  | Nothing => putStr "Can't make turtle\n"
             col t yellow
             fd t 100; rt t 90
             col t green
             fd t 100; rt t 90
             col t red
             fd t 100; rt t 90
             col t blue
             fd t 100; rt t 90
             render t
             end t
```

with ST do

在 `turtle` 中使用 `with ST do` 是为了区分 `(>>=)` 的版本，它可能来自于 `Monad` 或者 `ST`。虽然 Idris 自己能够解决此问题，不过它会尝试所有可能性，因此使用 `with` 从句可以减少耗时，加速类型检查。

要实现此接口，我们可以尝试用 Surface 来表示海龟用来作画的平面：

```
implementation Draw m => TurtleGraphics m where
  Turtle = Surface {m}
```

知道了 Turtle 被表示为 Surface 之后，我们就能使用 Draw 提供的方法来实现海龟了。然而这还不够，我们还需要存储更多信息：具体来说，海龟有一些属性需要存储在某处。因此我们不仅需要将海龟表示为一个 Surface，还需要存储一些附加的状态。我们可以通过**复合**资源来做到这一点。

复合资源简介

复合资源由一个资源列表构造而来，它使用以下 Control.ST 中定义的类型实现：

```
data Composite : List Type -> Type
```

如果我们有一个复合资源，那么可以用 split 将其分解为构成它的资源，并为每个资源创建新的变量。例如：

```
splitComp : (comp : Var) -> ST m () [comp ::: Composite [State Int, State String]]
splitComp comp = do [int, str] <- split comp
                    ?whatNow
```

调用 split comp 会从复合资源 comp 中提取出 State Int 和 State String，并分别将它们存储在变量 int 和 str 中。如果我们检查 whatNow 的类型，就会看到它影响了资源列表：

```
  int : Var
  str : Var
  comp : Var
  m : Type -> Type
--------------------------------------
whatNow : STrans m () [int ::: State Int, str ::: State String, comp ::: State ()]
                      (\result => [comp ::: Composite [State Int, State String]])
```

这样，我们就有了两个新的资源 int 和 str，而 comp 的类型则被更新为单元类型，即当前没有保存数据。这点符合预期：我们只是要将数据提取为独立的资源而已。

现在我们提取出了独立的资源，可以直接操作它们（比如，增加 Int 或为 String 添加换行）并使用 combine 重新构造复合资源：

```
splitComp : (comp : Var) ->
            ST m () [comp ::: Composite [State Int, State String]]
splitComp comp = do [int, str] <- split comp
                    update int (+ 1)
                    update str (++ "\n")
                    combine comp [int, str]
                    ?whatNow
```

同样，我们可以检查 whatNow 的类型来查看 combine 的作用：

```
  comp : Var
  int : Var
  str : Var
  m : Type -> Type
--------------------------------------
whatNow : STrans m () [comp ::: Composite [State Int, State String]]
                  (\result => [comp ::: Composite [State Int, State String]])
```

所以 combine 的作用就是接受既有的资源并将它们合并成一个复合资源。在执行 combine 之前，目标资源必须存在（这里是 comp）且类型为 State ()。

查看 split 和 combine 的类型，了解它们使用的资源列表的要求是很有启发的。split 的类型如下：

```
split : (lbl : Var) -> {auto prf : InState lbl (Composite vars) res} ->
        STrans m (VarList vars) res (\vs => mkRes vs ++ updateRes res prf (State ()))
```

隐式的 prf 参数说明要被分解的 lbl 必须为复合资源。它返回一个由复合资源分解而来的变量列表，而 mkRes 函数则创建一个对应类型的资源列表。最后，updateRes 会将复合资源的类型更新为 State ()。

而 combine 函数则反过来：

```
combine : (comp : Var) -> (vs : List Var) ->
          {auto prf : InState comp (State ()) res} ->
          {auto var_prf : VarsIn (comp :: vs) res} ->
          STrans m () res (const (combineVarsIn res var_prf))
```

隐式的 prf 参数确保了目标资源 comp 的类型为 State ()。也就是说，我们不会覆盖任何其它的信息。隐式的 var_prf 参数类似于 call 中的 SubRes，它确保了我们用来构造复合资源的每个变量都确实存在于当前的资源列表中。

我们可以基于 Draw 以及任何需要的附加资源，通过复合资源的方式来实现高级的 TurtleGraphics API。

实现 Turtle

现在我们已经知道如何用一组既有的资源来构造出新的资源了。我们可以用复合资源实现 Turtle，它包括一个用于绘图的 Surface，以及一些表示画笔颜色、位置和方向的独立的状态。我们还有一个线条的列表，它描述了我们在调用 render 时要绘制到 Surface 上的图像：

```
Turtle = Composite [Surface {m}, -- 用来绘图的平面
                    State Col,  -- 画笔颜色
                    State (Int, Int, Int, Bool), -- 画笔的位置/方向/落笔标记
                    State (List Line)] -- 渲染时要画的线条
```

一条 Line 由它的起始位置，终止位置和颜色定义：

```
Line : Type
Line = ((Int, Int), (Int, Int), Col)
```

首先来实现 start，它会创建一个新的 Turtle`` （如果不可能则返回 ``Nothing） 。我们从初始化绘图平面开始，然后是所有状态构成的组件。最后，我们将所有的组件组合成一个复合资源来表示海龟：

```
start x y = do Just srf <- initWindow x y
                    | Nothing => pure Nothing
               col <- new white
               pos <- new (320, 200, 0, True)
               lines <- new []
               turtle <- new ()
               combine turtle [srf, col, pos, lines]
               pure (Just turtle)
```

然后来实现 end，它会把海龟处理掉。我们先析构复合资源，然后关闭窗口，删除所有独立的资源。记住我们只能用 delete 删除一个 State，因此需要用 split 将复合资源分解掉，用 closeWindow 干净地关闭绘图平面，然后用 delete 删除状态：

```
end t = do [srf, col, pos, lines] <- split t
           closeWindow srf; delete col; delete pos; delete lines; delete t
```

对于其它的方法，我们需要用 split 分解资源以获取每一个组件，然后在结束时用 combine 将它们组合成一个复合资源。例如，以下为 penup 的实现：

```
penup t = do [srf, col, pos, lines] <- split t -- 分解复合资源
             (x, y, d, _) <- read pos           -- 析构画笔位置
             write pos (x, y, d, False)         -- 将落笔标记置为 False
             combine t [srf, col, pos, lines]   -- 重新组合组件
```

海龟剩下的操作遵循相同的模式。完整的细节见 Idris 发行版中的 samples/ST/Graphics/Turtle.idr 文件。之后就是是渲染海龟创建的图像：

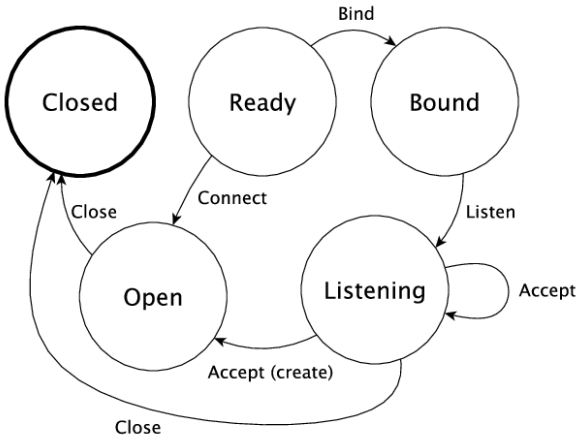
```
render t = do [srf, col, pos, lines] <- split t -- 分解复合资源
              filledRectangle srf (0, 0) (640, 480) black -- 绘制背景
              drawAll srf !(read lines)          -- 渲染海龟绘制的线条
              flip srf                           -- 缓冲区翻页以显示图像
              combine t [srf, col, pos, lines]
              Just ev <- poll
                | Nothing => render t            -- 继续直到按下按键
              case ev of
                   KeyUp _ => pure ()            -- 按键按下，于是退出
                   _ => render t
 where drawAll : (srf : Var) -> List Line -> ST m () [srf ::: Surface {m}]
       drawAll srf [] = pure ()
       drawAll srf ((start, end, col) :: xs)
          = do drawLine srf start end col       -- 按相应的颜色绘制线条
               drawAll srf xs
```

3.5 示例：网络 Socket 编程

POSIX 的插口（Socket，另译作「套接字」）API 支持跨网络的进程间通信。**插口**表示网络通信的端点，它可以是以下几种状态之一：

- Ready 为初始状态
- Bound 表示已绑定至某个地址，准备接受传入的连接
- Listening 表示正在监听传入的连接
- Open 表示准备发送或接受数据
- Closed 表示不再活动

下图展示了该 API 提供的操作是如何改变其状态的，其中 Ready 为初始状态：



如果某个连接为 Open 状态，那么我们可以用 send 发送消息到该连接的另一端，也可以用 recv 从另一端接收消息。

contrib 包提供了一个 Network.Socket 模块，该模块提供了创建插口以及发送和接收消息的原语。其中包含以下函数：

```
bind : (sock : Socket) -> (addr : Maybe SocketAddress) -> (port : Port) -> IO Int
connect : (sock : Socket) -> (addr : SocketAddress) -> (port : Port) -> IO ResultCode
listen : (sock : Socket) -> IO Int
accept : (sock : Socket) -> IO (Either SocketError (Socket, SocketAddress))
send : (sock : Socket) -> (msg  : String) -> IO (Either SocketError ResultCode)
recv : (sock : Socket) -> (len : ByteLength) -> IO (Either SocketError (String, ResultCode))
close : Socket -> IO ()
```

这些函数虽然涵盖了上图中所有的状态转移，然而却均未解释这些操作是如何影响其状态的！例如，我们完全有可能在一个尚未就绪的插口上发送消息，或者在插口关闭后从中接收消息。

我们可以用 ST 提供更好的 API，它精确地解释了每个操作是如何影响连接的状态的。在本节中，我们会定义一个插口 API，然后用它来实现一个「回显（echo）」服务器，通过原样返回客户端发送的消息来响应客户端的请求。

3.5.1 定义 Sockets 接口

我们不直接用 IO 进行底层插口编程，而是用 ST 实现一个接口来精确地描述每个操作如何影响插口的状态，以及何时创建或删除插口。我们首先来创建描述插口抽象状态的类型：

```
data SocketState = Ready | Bound | Listening | Open | Closed
```

接着定义一个接口，用 Sock 类型表示插口，以其当前状态作为该类型构造子的参数：

```
interface Sockets (m : Type -> Type) where
  Sock : SocketState -> Type
```

我们用 socket 方法创建插口。插口库中定义了 SocketType，它描述了插口为 TCP、UDP 还是其它形式。我们后面会一直使用 Stream 来表示 TCP 插口。

```
socket : SocketType -> ST m (Either () Var) [addIfRight (Sock Ready)]
```

记住 addIfRight 会在操作结果的形式为 Right val 时添加资源。按照此接口的约定，我们用 Either 来表示可能失败的操作，它可以保存任何关于错误的附加信息。如此一来，我们就能一致地使用 addIfRight 和其它类型级函数了。

现在来定义一个服务器。一旦我们创建了插口，就需要用 bind 方法将它绑定到一个端口上：

```
bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
       ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Bound)]
```

绑定插口可能会失败，例如如果已经有一个插口被绑定到给定的端口上，因此它同样要返回一个类型为 Either 的值。这里的动作使用了类型级函数 or，它的意思是：

- 若 bind 失败，插口就会转移到 Sock Closed 状态
- 若 bind 成功，插口就会转移到 Sock Bound 状态，如前图所示

or 的实现如下：

```
or : a -> a -> Either b c -> a
or x y = either (const x) (const y)
```

这样一来，bind 的类型就能写成如下等价形式：

```
bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
       STrans m (Either () ()) [sock ::: Sock Ready]
                    (either [sock ::: Sock Closed] [sock ::: Sock Bound])
```

然而，使用 or 会更加简洁，它也能尽可能直接地反映出状态转移图，同时仍然刻画了它可能失败的性质。

一旦我们将插口绑定到了端口，就可以开始监听来自客户端的连接了：

```
listen : (sock : Var) ->
         ST m (Either () ()) [sock ::: Sock Bound :-> (Sock Closed `or` Sock Listening)]
```

Listening 状态的插口表示准备接受来自独立客户端的连接：

```
accept : (sock : Var) ->
         ST m (Either () Var)
              [sock ::: Sock Listening, addIfRight (Sock Open)]
```

如果有客户端传入的连接，accept 会在资源列表的最后添加一个**新的**资源（按照约定，在列表末尾添加资源可以更好地配合 updateWith 工作，如上一节所述）。现在，我们有了**两个**插口：一个继续监听传入的连接，另一个准备与客户端通信。

我们还需要能够在插口上发送和接受数据的方法：

```
send : (sock : Var) -> String ->
       ST m (Either () ()) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
recv : (sock : Var) ->
       ST m (Either () String) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
```

一旦我们与另一台机器通过插口进行的通信结束， 就需要用 close 关闭连接并用 remove 移除该插口：

```
close : (sock : Var) ->
        {auto prf : CloseOK st} -> ST m () [sock ::: Sock st :-> Sock Closed]
remove : (sock : Var) ->
         ST m () [Remove sock (Sock Closed)]
```

close 使用了断言 CloseOK 作为隐式证明参数，它描述了何时可以关闭插口：

```
data CloseOK : SocketState -> Type where
     CloseOpen : CloseOK Open
     CloseListening : CloseOK Listening
```

也就是说，我们可以关闭 Open 状态的插口，告诉另一台机器终止通信。我们也可以关闭 Listening 状态下等待传入连接的插口，这会让服务器停止接受请求。

在本节中，我们实现了一个服务器，不过为了完整性，我们还需要在另一台机器上实现客户端来连接到服务器。这可以通过 connect 来完成：

```
connect : (sock : Var) -> SocketAddress -> Port ->
          ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Open)]
```

以下完整的接口作为参考：

```
interface Sockets (m : Type -> Type) where
  Sock : SocketState -> Type
  socket : SocketType -> ST m (Either () Var) [addIfRight (Sock Ready)]
  bind : (sock : Var) -> (addr : Maybe SocketAddress) -> (port : Port) ->
         ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Bound)]
  listen : (sock : Var) ->
           ST m (Either () ()) [sock ::: Sock Bound :-> (Sock Closed `or` Sock Listening)]
  accept : (sock : Var) ->
           ST m (Either () Var) [sock ::: Sock Listening, addIfRight (Sock Open)]
  connect : (sock : Var) -> SocketAddress -> Port ->
            ST m (Either () ()) [sock ::: Sock Ready :-> (Sock Closed `or` Sock Open)]
  close : (sock : Var) -> {auto prf : CloseOK st} ->
          ST m () [sock ::: Sock st :-> Sock Closed]
  remove : (sock : Var) -> ST m () [Remove sock (Sock Closed)]
  send : (sock : Var) -> String ->
         ST m (Either () ()) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
  recv : (sock : Var) ->
         ST m (Either () String) [sock ::: Sock Open :-> (Sock Closed `or` Sock Open)]
```

我们稍后会看到如何实现它。这些方法大部分都可以直接通过原始的插口 API 在 IO 中实现。不过首先，我们会看到如何用这些 API 实现一个「回显」服务器。

3.5.2 用 Sockets 实现「回显」服务器

从顶层来说，我们的回显（echo）服务器在开始和结束时均没有资源可用，它使用了 ConsoleIO 和 Sockets 接口：

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
```

首先我们需要用 socket 创建一个插口，绑定到一个端口并监听传入的连接。它可能会失败，因此我们需要处理它返回 Right sock 的情况，其中 sock 是新的插口变量，不过也可能返回 Left err：

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
startServer =
  do Right sock <- socket Stream
           | Left err => pure ()
     ?whatNow
```

交互式地实现这类函数是个不错的想法，我们可以通过挖坑来逐步观察整个系统的状态是如何变化的。在成功调用了 socket 之后，我们就有了 Ready 状态的插口：

```
  sock : Var
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : Sockets m
--------------------------------------
whatNow : STrans m () [sock ::: Sock Ready] (\result1 => [])
```

接着，我们需要将插口绑定到端口，然后开始监听连接。同样，每一步都可能会失败，此时我们会移除该插口。失败总会导致插口转为 Closed 状态，此时我们能做的就是用 remove 移除它：

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
startServer =
  do Right sock <- socket Stream        | Left err => pure ()
     Right ok <- bind sock Nothing 9442 | Left err => remove sock
     Right ok <- listen sock            | Left err => remove sock
     ?runServer
```

最后，我们就有了一个监听传入连接的插口：

```
  ok : ()
  sock : Var
  ok1 : ()
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : Sockets m
--------------------------------------
runServer : STrans m () [sock ::: Sock Listening]
                  (\result1 => [])
```

我们会在一个独立的函数中实现它。runServer 的类型告诉我们 echoServer 的类型必须是什么（我们无需显式地为 Sock 给出参数 m）：

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
             ST m () [remove sock (Sock {m} Listening)]
```

我们可以完成 startServer 的定义：

```
startServer : (ConsoleIO m, Sockets m) => ST m () []
startServer =
  do Right sock <- socket Stream        | Left err => pure ()
     Right ok <- bind sock Nothing 9442 | Left err => remove sock
     Right ok <- listen sock            | Left err => remove sock
     echoServer sock
```

在 echoServer 中，我们会继续接受并相应请求直到出现失败，此时我们会关闭插口并返回。我们从尝试接受传入的连接开始：

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
             ST m () [remove sock (Sock {m} Listening)]
echoServer sock =
  do Right new <- accept sock | Left err => do close sock; remove sock
     ?whatNow
```

若 accept 失败，我们就需要关闭 Listening 状态的插口并在返回前移除它，因为 echoServer 的类型要求如此。

通常，逐步实现 echoServer 意味着我们可以在开发过程中检查当前的状态。如果 accept 成功，那么我们既有的 sock 会继续监听连接，此外一个新的 new 插口会被打开用于通信：

```
  new : Var
  sock : Var
  m : Type -> Type
  constraint : ConsoleIO m
  constraint1 : Sockets m
--------------------------------------
whatNow : STrans m () [sock ::: Sock Listening, new ::: Sock Open]
                    (\result1 => [])
```

要完成 echoServer，我们需要从 new 插口上接收一条消息，然后原样返回它。在完成后，我们就关闭 new 插口，然后回到 echoServer 的开始处，准备响应下一次连接：

```
echoServer : (ConsoleIO m, Sockets m) => (sock : Var) ->
             ST m () [remove sock (Sock {m} Listening)]
echoServer sock =
  do Right new <- accept sock | Left err => do close sock; remove sock
     Right msg <- recv new | Left err => do close sock; remove sock; remove new
     Right ok <- send new ("You said " ++ msg)
           | Left err => do remove new; close sock; remove sock
     close new; remove new; echoServer sock
```

3.5.3 实现 Sockets

为了在 IO 中实现 Sockets，我们需要从给出具体的 Sock 类型开始。我们可以用原始的插口 API（在 Network.Socket 中实现），将 Socket 存储在 State 中来得到具体的类型，而不必关心该插口所处的抽象状态：

```
implementation Sockets IO where
  Sock _ = State Socket
```

大部分方法都可以直接用原始的插口 API 来实现，返回相应的 Left 或 Right。例如，我们可以实现 socket、bind 和 listen：

```
socket ty = do Right sock <- lift $ Socket.socket AF_INET ty 0
                  | Left err => pure (Left ())
               lbl <- new sock
               pure (Right lbl)
bind sock addr port = do ok <- lift $ bind !(read sock) addr port
                         if ok /= 0
                            then pure (Left ())
                            else pure (Right ())
listen sock = do ok <- lift $ listen !(read sock)
                 if ok /= 0
                    then pure (Left ())
                    else pure (Right ())
```

然而，这里的 accept 有点不同，因为我们在用 new 为打开连接创建新资源时，它出现在了资源列表的**起始**处而非末尾。我们可以通过写出不完整的定义来看到这一点，使用 returning 来查看返回 Right lbl 需要什么资源：

```
accept sock = do Right (conn, addr) <- lift $ accept !(read sock)
                       | Left err => pure (Left ())
                 lbl <- new conn
                 returning (Right lbl) ?fixResources
```

使用 new 将资源添加到列表起始处是很方便的，因为通常来说，这会让 Idris 使用隐式 auto 自动构造证明更加容易。另一方面，当我们用 call 来构造更小的资源集合时，updateWith 会将新创建的资源放到列表的**末尾**处，因为通常这样会减少需要重新排序的资源的数量。

如果我们查看 fixResources 的类型，就会知道结束 accept 需要做的事情：

```
  _bindApp0 : Socket
  conn : Socket
  addr : SocketAddress
  sock : Var
  lbl : Var
-------------------------------------
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
fixResources : STrans IO () [lbl ::: State Socket, sock ::: State Socket]
                            (\value => [sock ::: State Socket, lbl ::: State Socket])
```

当前资源列表的顺序为 lbl、sock，而我们需要它们的顺序变成 sock、lbl。为此，Control.ST 提供了原语 toEnd，它会将一个资源移到列表的末尾。这样我们就能完成 accept 了：

```
accept sock = do Right (conn, addr) <- lift $ accept !(read sock)
                       | Left err => pure (Left ())
                 lbl <- new conn
                 returning (Right lbl) (toEnd lbl)
```

Sockets 的完整实现见 Idris 发行版中的 samples/ST/Net/Network.idr 文件。你也可以在同目录下的 EchoServer.idr 文件中找到回显服务器。此外还有一个高级网络协议 RandServer.idr，它基于底层的插口 API，通过状态机的层级实现了一个高级的网络通信协议。它还使用线程来异步地处理传入的请求。你可以在 Edwin Brady 的文稿 State Machines All The Way Down （深入理解状态机）中找到关于线程和随机数服务器的更多详情。

CHAPTER 4

The Effects Tutorial

A tutorial on the *Effects* package in *Idris*.

Effects and the `Control.ST` module

There is a new module in the `contrib` package, `Control.ST`, which provides the resource tracking facilities of *Effects* but with better support for creating and deleting resources, and implementing resources in terms of other resources.

Unless you have a particular reason to use *Effects* you are strongly recommended to use `Control.ST` instead. There is a tutorial available on this site for `Control.ST` with several examples (用 *Idris* 实现带有状态的系统：*ST* 教程 (éąţ 69)).

4.1 Introduction

Pure functional languages with dependent types such as Idris support reasoning about programs directly in the type system, promising that we can *know* a program will run correctly (i.e. according to the specification in its type) simply because it compiles. Realistically, though, things are not so simple: programs have to interact with the outside world, with user input, input from a network, mutable state, and so on. In this tutorial I will introduce the library, which is included with the distribution and supports programming and reasoning with side-effecting programs, supporting mutable state, interaction with the outside world, exceptions, and verified resource management.

This tutorial assumes familiarity with pure programming in Idris, as described in Sections 1–6 of the main tutorial[1]. The examples presented are tested with Idris and can be found in the examples directory of the Idris repository.

Consider, for example, the following introductory function which illustrates the kind of properties which can be expressed in the type system:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd []        []        = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

This function adds corresponding elements in a pair of vectors. The type guarantees that the vectors will contain only elements of type `Int`, and that the input lengths and the output length all correspond. A natural question to ask here, which is typically neglected by introductory tutorials, is "How do I turn this into a program?" That is, given some lists entered by a user, how do we get into a position to be able to apply the `vadd` function? Before doing so, we will have to:

- Read user input, either from the keyboard, a file, or some other input device.
- Check that the user inputs are valid, i.e. contain only `Int` and are the same length, and report an error if not.
- Write output

The complete program will include side-effects for I/O and error handling, before we can get to the pure core functionality. In this tutorial, we will see how Idris supports side-effects. Furthermore, we will see how we can use the dependent type system to *reason* about stateful and side-effecting programs. We will return to this specific example later.

4.1.1 Hello world

To give an idea of how programs with effects look, here is the ubiquitous "Hello world" program, written using the `Effects` library:

```
module Main

import Effects
import Effect.StdIO

hello : Eff () [STDIO]
hello = putStrLn "Hello world!"

main : IO ()
main = run hello
```

As usual, the entry point is `main`. All `main` has to do is invoke the `hello` function which supports the `STDIO` effect for console I/O, and returns the unit value. All programs using the `Effects` library must `import Effects`. The details of the `Eff` type will be presented in the remainder of this tutorial.

To compile and run this program, Idris needs to be told to include the `Effects` package, using the `-p effects` flag (this flag is required for all examples in this tutorial):

```
idris hello.idr -o hello -p effects
./hello Hello world!
```

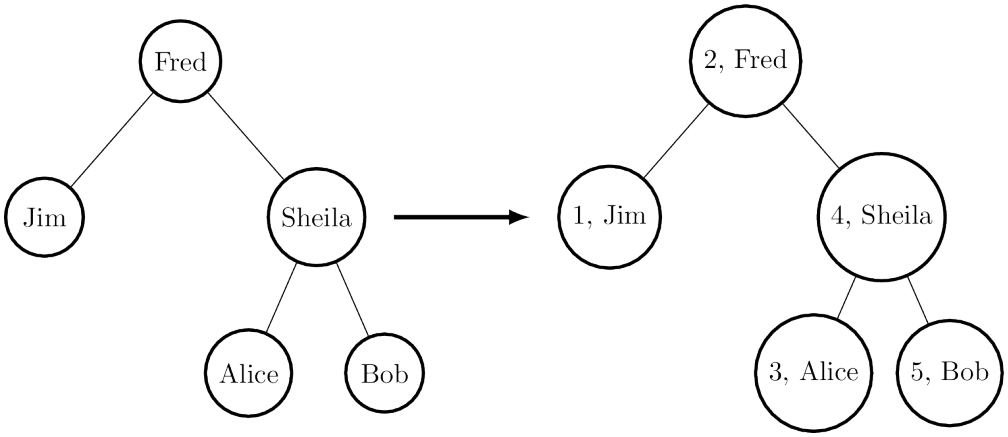
[1] You do not, however, need to know what a monad is!

4.1.2 Outline

The tutorial is structured as follows: first, in Section *State* (éąţ 104), we will discuss state management, describing why it is important and introducing the `effects` library to show how it can be used to manage state. This section also gives an overview of the syntax of effectful programs. Section *Simple Effects* (éąţ 111) then introduces a number of other effects a program may have: I/O; Exceptions; Random Numbers; and Non-determinism, giving examples for each, and an extended example combining several effects in one complete program. Section *Dependent Effects* (éąţ 118) introduces *dependent* effects, showing how states and resources can be managed in types. Section *Creating New Effects* (éąţ 123) shows how new effects can be implemented. Section *Example: A "Mystery Word" Guessing Game* (éąţ 127) gives an extended example showing how to implement a "mystery word" guessing game, using effects to describe the rules of the game and ensure they are implemented accurately. References to further reading are given in Section *Further Reading* (éąţ 132).

4.2 State

Many programs, even pure programs, can benefit from locally mutable state. For example, consider a program which tags binary tree nodes with a counter, by an inorder traversal (i.e. counting depth first, left to right). This would perform something like the following:



We can describe binary trees with the following data type `BTree` and `testTree` to represent the example input above:

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

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
                "Fred"
                (Node (Node Leaf "Alice" Leaf)
                      "Sheila"
                      (Node Leaf "Bob" Leaf))
```

Then our function to implement tagging, beginning to tag with a specific value `i`, has the following type:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
```

4.2.1 First attempt

Navely, we can implement `treeTag` by implementing a helper function which propagates a counter, returning the result of the count for each subtree:

```
treeTagAux : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTagAux i Leaf = (i, Leaf)
treeTagAux i (Node l x r)
       = let (i', l') = treeTagAux i l in
         let x' = (i', x) in
         let (i'', r') = treeTagAux (i' + 1) r in
             (i'', Node l' x' r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = snd (treeTagAux i x)
```

This gives the expected result when run at the REPL prompt:

```
*TreeTag> treeTag 1 testTree
Node (Node Leaf (1, "Jim") Leaf)
     (2, "Fred")
     (Node (Node Leaf (3, "Alice") Leaf)
           (4, "Sheila")
           (Node Leaf (5, "Bob") Leaf)) : BTree (Int, String)
```

This works as required, but there are several problems when we try to scale this to larger programs. It is error prone, because we need to ensure that state is propagated correctly to the recursive calls (i.e. passing the appropriate `i` or `i'`). It is hard to read, because the functional details are obscured by the state propagation. Perhaps most importantly, there is a common programming pattern here which should be abstracted but instead has been implemented by hand. There is local mutable state (the counter) which we have had to make explicit.

4.2.2 Introducing `Effects`

Idris provides a library, `Effects`[3], which captures this pattern and many others involving effectful computation[1]. An effectful program `f` has a type of the following form:

```
f : (x1 : a1) -> (x2 : a2) -> ... -> Eff t effs
```

That is, the return type gives the effects that `f` supports (`effs`, of type `List EFFECT`) and the type the computation returns `t`. So, our `treeTagAux` helper could be written with the following type:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
```

That is, `treeTagAux` has access to an integer state, because the list of available effects includes `STATE Int`. `STATE` is declared as follows in the module `Effect.State` (that is, we must `import Effect.State` to be able to use it):

```
STATE : Type -> EFFECT
```

It is an effect parameterised by a type (by convention, we write effects in all capitals). The `treeTagAux` function is an effectful program which builds a new tree tagged with `Int`s, and is implemented as follows:

[3] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. SIGPLAN Not. 48, 9 (September 2013), 133-144. DOI=10.1145/2544174.2500581 http://dl.acm.org/citation.cfm?doid=2544174.2500581

[1] The earlier paper[3] describes the essential implementation details, although the library presented there is an earlier version which is less powerful than that presented in this tutorial.

```
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r)
    = do l' <- treeTagAux l
         i <- get
         put (i + 1)
         r' <- treeTagAux r
         pure (Node l' (i, x) r')
```

There are several remarks to be made about this implementation. Essentially, it hides the state, which can be accessed using `get` and updated using `put`, but it introduces several new features. Specifically, it uses `do`-notation, binding variables with `<-`, and a `pure` function. There is much to be said about these features, but for our purposes, it suffices to know the following:

- `do` blocks allow effectful operations to be sequenced.
- **`x <- e` binds the result of an effectful operation `e` to a** variable `x`. For example, in the above code, `treeTagAux l` is an effectful operation returning `BTree (Int, a)`, so `l'` has type `BTree (Int, a)`.
- **`pure e` turns a pure value `e` into the result of an effectful** operation.

The `get` and `put` functions read and write a state `t`, assuming that the `STATE t` effect is available. They have the following types, polymorphic in the state `t` they manage:

```
get :      Eff t [STATE t]
put : t -> Eff () [STATE t]
```

A program in `Eff` can call any other function in `Eff` provided that the calling function supports at least the effects required by the called function. In this case, it is valid for `treeTagAux` to call both `get` and `put` because all three functions support the `STATE Int` effect.

Programs in `Eff` are run in some underlying *computation context*, using the `run` or `runPure` function. Using `runPure`, which runs an effectful program in the identity context, we can write the `treeTag` function as follows, using `put` to initialise the state:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i
                          treeTagAux x)
```

We could also run the program in an impure context such as `IO`, without changing the definition of `treeTagAux`, by using `run` instead of `runPure`:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
...

treeTag : (i : Int) -> BTree a -> IO (BTree (Int, a))
treeTag i x = run (do put i
                      treeTagAux x)
```

Note that the definition of `treeTagAux` is exactly as before. For reference, this complete program (including a `main` to run it) is shown in Listing [introprog].

```
module Main

import Effects
import Effect.State

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

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
Show a => Show (BTree a) where
    show Leaf = "[]"
    show (Node l x r) = "[" ++ show l ++ " "
                            ++ show x ++ " "
                            ++ show r ++ "]"

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
                "Fred"
                (Node (Node Leaf "Alice" Leaf)
                      "Sheila"
                      (Node Leaf "Bob" Leaf))

treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r) = do l' <- treeTagAux l
                             i <- get
                             put (i + 1)
                             r' <- treeTagAux r
                             pure (Node l' (i, x) r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i; treeTagAux x)

main : IO ()
main = print (treeTag 1 testTree)
```

4.2.3 Effects and Resources

Each effect is associated with a *resource*, which is initialised before an effectful program can be run. For example, in the case of `STATE Int` the corresponding resource is the integer state itself. The types of `runPure` and `run` show this (slightly simplified here for illustrative purposes):

```
runPure : {env : Env id xs} -> Eff a xs -> a
run : Applicative m => {env : Env m xs} -> Eff a xs -> m a
```

The `env` argument is implicit, and initialised automatically where possible using default values given by implementations of the following interface:

```
interface Default a where
    default : a
```

Implementations of `Default` are defined for all primitive types, and many library types such as `List`, `Vect`, `Maybe`, pairs, etc. However, where no default value exists for a resource type (for example, you may want a `STATE` type for which there is no `Default` implementation) the resource environment can be given explicitly using one of the following functions:

```
runPureInit : Env id xs -> Eff a xs -> a
runInit : Applicative m => Env m xs -> Eff a xs -> m a
```

To be well-typed, the environment must contain resources corresponding exactly to the effects in `xs`. For example, we could also have implemented `treeTag` by initialising the state as follows:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPureInit [i] (treeTagAux x)
```

4.2.4 Labelled Effects

What if we have more than one state, especially more than one state of the same type? How would `get` and `put` know which state they should be referring to? For example, how could we extend the tree tagging example such that it additionally counts the number of leaves in the tree? One possibility would be to change the state so that it captured both of these values, e.g.:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE (Int, Int)]
```

Doing this, however, ties the two states together throughout (as well as not indicating which integer is which). It would be nice to be able to call effectful programs which guaranteed only to access one of the states, for example. In a larger application, this becomes particularly important.

The library therefore allows effects in general to be *labelled* so that they can be referred to explicitly by a particular name. This allows multiple effects of the same type to be included. We can count leaves and update the tag separately, by labelling them as follows:

```
treeTagAux : BTree a ->  Eff (BTree (Int, a))
                             ['Tag ::: STATE Int,
                              'Leaves ::: STATE Int]
```

The `:::` operator allows an arbitrary label to be given to an effect. This label can be any type—it is simply used to identify an effect uniquely. Here, we have used a symbol type. In general `' name` introduces a new symbol, the only purpose of which is to disambiguate values[2].

When an effect is labelled, its operations are also labelled using the `:-` operator. In this way, we can say explicitly which state we mean when using `get` and `put`. The tree tagging program which also counts leaves can be written as follows:

```
treeTagAux Leaf = do
    'Leaves :- update (+1)
    pure Leaf
treeTagAux (Node l x r) = do
    l' <- treeTagAux l
    i <- 'Tag :- get
    'Tag :- put (i + 1)
    r' <- treeTagAux r
    pure (Node l' (i, x) r')
```

The `update` function here is a combination of `get` and `put`, applying a function to the current state.

```
update : (x -> x) -> Eff () [STATE x]
```

Finally, our top level `treeTag` function now returns a pair of the number of leaves, and the new tree. Resources for labelled effects are initialised using the `:=` operator (reminiscent of assignment in an imperative language):

```
treeTag : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTag i x = runPureInit ['Tag := i, 'Leaves := 0]
                    (do x' <- treeTagAux x
                        leaves <- 'Leaves :- get
                        pure (leaves, x'))
```

To summarise, we have:

- `:::` to convert an effect to a labelled effect.
- `:-` to convert an effectful operation to a labelled effectful operation.

[2] In practice, `' name` simply introduces a new empty type

- `:=` to initialise a resource for a labelled effect.

Or, more formally with their types (slightly simplified to account only for the situation where available effects are not updated):

```
(:::) : lbl -> EFFECT -> EFFECT
(:-)  : (l : lbl) -> Eff a [x] -> Eff a [l ::: x]
(:=)  : (l : lbl) -> res -> LRes l res
```

Here, `LRes` is simply the resource type associated with a labelled effect. Note that labels are polymorphic in the label type `lbl`. Hence, a label can be anything—a string, an integer, a type, etc.

4.2.5 `!`-notation

In many cases, using `do`-notation can make programs unnecessarily verbose, particularly in cases where the value bound is used once, immediately. The following program returns the length of the `String` stored in the state, for example:

```
stateLength : Eff Nat [STATE String]
stateLength = do x <- get
                 pure (length x)
```

This seems unnecessarily verbose, and it would be nice to program in a more direct style in these cases. provides `!`-notation to help with this. The above program can be written instead as:

```
stateLength : Eff Nat [STATE String]
stateLength = pure (length !get)
```

The notation `!expr` means that the expression `expr` should be evaluated and then implicitly bound. Conceptually, we can think of `!` as being a prefix function with the following type:

```
(!) : Eff a xs -> a
```

Note, however, that it is not really a function, merely syntax! In practice, a subexpression `!expr` will lift `expr` as high as possible within its current scope, bind it to a fresh name `x`, and replace `!expr` with `x`. Expressions are lifted depth first, left to right. In practice, `!`-notation allows us to program in a more direct style, while still giving a notational clue as to which expressions are effectful.

For example, the expression:

```
let y = 42 in f !(g !(print y) !x)
```

is lifted to:

```
let y = 42 in do y' <- print y
                 x' <- x
                 g' <- g y' x'
                 f g'
```

4.2.6 The Type `Eff`

Underneath, `Eff` is an overloaded function which translates to an underlying type `EffM`:

```
EffM : (m : Type -> Type) -> (t : Type)
       -> (List EFFECT)
       -> (t -> List EFFECT) -> Type
```

This is more general than the types we have been writing so far. It is parameterised over an underlying computation context `m`, a result type `t` as we have already seen, as well as a `List EFFECT` and a function type `t -> List EFFECT`.

These additional parameters are the list of *input* effects, and a list of *output* effects, computed from the result of an effectful operation. That is: running an effectful program can change the set of effects available! This is a particularly powerful idea, and we will see its consequences in more detail later. Some examples of operations which can change the set of available effects are:

- Updating a state containing a dependent type (for example adding an element to a vector).
- Opening a file for reading is an effect, but whether the file really *is* open afterwards depends on whether the file was successfully opened.
- Closing a file means that reading from the file should no longer be possible.

While powerful, this can make uses of the `EffM` type hard to read. Therefore the library provides an overloaded function `Eff` There are the following three versions:

```
SimpleEff.Eff : (t : Type) -> (input_effs : List EFFECT) -> Type
TransEff.Eff  : (t : Type) -> (input_effs : List EFFECT) ->
                              (output_effs : List EFFECT) -> Type
DepEff.Eff    : (t : Type) -> (input_effs : List EFFECT) ->
                              (output_effs_fn : t -> List EFFECT) -> Type
```

So far, we have used only the first version, `SimpleEff.Eff`, which is defined as follows:

```
Eff : (x : Type) -> (es : List EFFECT) -> Type
Eff x es = {m : Type -> Type} -> EffM m x es (\v => es)
```

i.e. the set of effects remains the same on output. This suffices for the `STATE` example we have seen so far, and for many useful side-effecting programs. We could also have written `treeTagAux` with the expanded type:

```
treeTagAux : BTree a ->
             EffM m (BTree (Int, a)) [STATE Int] (\x => [STATE Int])
```

Later, we will see programs which update effects:

```
Eff a xs xs'
```

which is expanded to

```
EffM m a xs (\_ => xs')
```

i.e. the set of effects is updated to `xs'` (think of a transition in a state machine). There is, for example, a version of `put` which updates the type of the state:

```
putM : y -> Eff () [STATE x] [STATE y]
```

Also, we have:

```
Eff t xs (\res => xs')
```

which is expanded to

```
EffM m t xs (\res => xs')
```

i.e. the set of effects is updated according to the result of the operation `res`, of type `t`.

Parameterising `EffM` over an underlying computation context allows us to write effectful programs which are specific to one context, and in some cases to write programs which *extend* the list of effects available using the `new` function, though this is beyond the scope of this tutorial.

4.3 Simple Effects

So far we have seen how to write programs with locally mutable state using the `STATE` effect. To recap, we have the definitions below in a module `Effect.State`

```
module Effect.State

STATE : Type -> EFFECT

get    :             Eff x  [STATE x]
put    : x ->        Eff () [STATE x]
putM   : y ->        Eff () [STATE x] [STATE y]
update : (x -> x) -> Eff () [STATE x]

Handler State m where { ... }
```

The last line, `Handler State m where { ... }`, means that the `STATE` effect is usable in any computation context `m`. That is, a program which uses this effect and returns something of type `a` can be evaluated to something of type `m a` using `run`, for any `m`. The lower case `State` is a data type describing the operations which make up the `STATE` effect itself—we will go into more detail about this in Section *Creating New Effects* (ęąţ 123).

In this section, we will introduce some other supported effects, allowing console I/O, exceptions, random number generation and non-deterministic programming. For each effect we introduce, we will begin with a summary of the effect, its supported operations, and the contexts in which it may be used, like that above for `STATE`, and go on to present some simple examples. At the end, we will see some examples of programs which combine multiple effects.

All of the effects in the library, including those described in this section, are summarised in Appendix *Effects Summary* (ęąţ 133).

4.3.1 Console I/O

Console I/O is supported with the `STDIO` effect, which allows reading and writing characters and strings to and from standard input and standard output. Notice that there is a constraint here on the computation context `m`, because it only makes sense to support console I/O operations in a context where we can perform (or at the very least simulate) console I/O:

```
module Effect.StdIO

STDIO : EFFECT

putChar  : Char ->   Eff () [STDIO]
putStr   : String -> Eff () [STDIO]
putStrLn : String -> Eff () [STDIO]

getStr   :           Eff String [STDIO]
getChar  :           Eff Char [STDIO]

Handler StdIO IO where { ... }
Handler StdIO (IOExcept a) where { ... }
```

Examples

A program which reads the user' s name, then says hello, can be written as follows:

```
hello : Eff () [STDIO]
hello = do putStr "Name? "
           x <- getStr
           putStrLn ("Hello " ++ trim x ++ "!")
```

We use `trim` here to remove the trailing newline from the input. The resource associated with `STDIO` is simply the empty tuple, which has a default value `()`, so we can run this as follows:

```
main : IO ()
main = run hello
```

In `hello` we could also use `!`-notation instead of `x <- getStr`, since we only use the string that is read once:

```
hello : Eff () [STDIO]
hello = do putStr "Name? "
           putStrLn ("Hello " ++ trim !getStr ++ "!")
```

More interestingly, we can combine multiple effects in one program. For example, we can loop, counting the number of people we' ve said hello to:

```
hello : Eff () [STATE Int, STDIO]
hello = do putStr "Name? "
           putStrLn ("Hello " ++ trim !getStr ++ "!")
           update (+1)
           putStrLn ("I've said hello to: " ++ show !get ++ " people")
           hello
```

The list of effects given in `hello` means that the function can call `get` and `put` on an integer state, and any functions which read and write from the console. To run this, `main` does not need to be changed.

Aside: Resource Types

To find out the resource type of an effect, if necessary (for example if we want to initialise a resource explicitly with `runInit` rather than using a default value with `run`) we can run the `resourceType` function at the REPL:

```
*ConsoleIO> resourceType STDIO
() : Type
*ConsoleIO> resourceType (STATE Int)
Int : Type
```

4.3.2 Exceptions

The `EXCEPTION` effect is declared in module `Effect.Exception`. This allows programs to exit immediately with an error, or errors to be handled more generally:

```
module Effect.Exception

EXCEPTION : Type -> EFFECT
```

(äÿŃéąţżğçżŋ)

(çżŋäÿŁéąţ)

```
raise : a -> Eff b [EXCEPTION a]

Handler (Exception a) Maybe where { ... }
Handler (Exception a) List where { ... }
Handler (Exception a) (Either a) where { ... }
Handler (Exception a) (IOExcept a) where { ... }
Show a => Handler (Exception a) IO where { ... }
```

Example

Suppose we have a `String` which is expected to represent an integer in the range `0` to `n`. We can write a function `parseNumber` which returns an `Int` if parsing the string returns a number in the appropriate range, or throws an exception otherwise. Exceptions are parameterised by an error type:

```
data Error = NotANumber | OutOfRange

parseNumber : Int -> String -> Eff Int [EXCEPTION Error]
parseNumber num str
   = if all isDigit (unpack str)
        then let x = cast str in
             if (x >=0 && x <= num)
                then pure x
                else raise OutOfRange
        else raise NotANumber
```

Programs which support the `EXCEPTION` effect can be run in any context which has some way of throwing errors, for example, we can run `parseNumber` in the `Either Error` context. It returns a value of the form `Right x` if successful:

```
*Exception> the (Either Error Int) $ run (parseNumber 42 "20")
Right 20 : Either Error Int
```

Or `Left e` on failure, carrying the appropriate exception:

```
*Exception> the (Either Error Int) $ run (parseNumber 42 "50")
Left OutOfRange : Either Error Int

*Exception> the (Either Error Int) $ run (parseNumber 42 "twenty")
Left NotANumber : Either Error Int
```

In fact, we can do a little bit better with `parseNumber`, and have it return a *proof* that the integer is in the required range along with the integer itself. One way to do this is define a type of bounded integers, `Bounded`:

```
Bounded : Int -> Type
Bounded x = (n : Int ** So (n >= 0 && n <= x))
```

Recall that `So` is parameterised by a `Bool`, and only `So True` is inhabited. We can use `choose` to construct such a value from the result of a dynamic check:

```
data So : Bool -> Type = Oh : So True

choose : (b : Bool) -> Either (So b) (So (not b))
```

We then write `parseNumber` using `choose` rather than an `if/then/else` construct, passing the proof it returns on success as the boundedness proof:

```
parseNumber : (x : Int) -> String -> Eff (Bounded x) [EXCEPTION Error]
parseNumber x str
   = if all isDigit (unpack str)
        then let num = cast str in
             case choose (num >=0 && num <= x) of
                  Left p => pure (num ** p)
                  Right p => raise OutOfRange
        else raise NotANumber
```

4.3.3 Random Numbers

Random number generation is also implemented by the library, in module `Effect.Random`:

```
module Effect.Random

RND : EFFECT

srand  : Integer ->            Eff () [RND]
rndInt : Integer -> Integer -> Eff Integer [RND]
rndFin : (k : Nat) ->          Eff (Fin (S k)) [RND]

Handler Random m where { ... }
```

Random number generation is considered side-effecting because its implementation generally relies on some external source of randomness. The default implementation here relies on an integer *seed*, which can be set with `srand`. A specific seed will lead to a predictable, repeatable sequence of random numbers. There are two functions which produce a random number:

- **`rndInt`, which returns a random integer between the given lower** and upper bounds.
- **`rndFin`, which returns a random element of a finite set** (essentially a number with an upper bound given in its type).

Example

We can use the `RND` effect to implement a simple guessing game. The `guess` function, given a target number, will repeatedly ask the user for a guess, and state whether the guess is too high, too low, or correct:

```
guess : Int -> Eff () [STDIO]
```

For reference, the code for `guess` is given below:

```
guess : Int -> Eff () [STDIO]
guess target
    = do putStr "Guess: "
         case run {m=Maybe} (parseNumber 100 (trim !getStr)) of
              Nothing => do putStrLn "Invalid input"
                            guess target
              Just (v ** _) =>
                        case compare v target of
                            LT => do putStrLn "Too low"
                                     guess target
                            EQ => putStrLn "You win!"
                            GT => do putStrLn "Too high"
                                     guess target
```

Note that we use `parseNumber` as defined previously to read user input, but we don't need to list the `EXCEPTION` effect because we use a nested `run` to invoke `parseNumber`, independently of the calling effectful program.

To invoke this, we pick a random number within the range 0–100, having set up the random number generator with a seed, then run `guess`:

```
game : Eff () [RND, STDIO]
game = do srand 123456789
          guess (fromInteger !(rndInt 0 100))

main : IO ()
main = run game
```

If no seed is given, it is set to the `default` value. For a less predictable game, some better source of randomness would be required, for example taking an initial seed from the system time. To see how to do this, see the `SYSTEM` effect described in *Effects Summary* (ęąţ 133).

4.3.4 Non-determinism

The listing below gives the definition of the non-determinism effect, which allows a program to choose a value non-deterministically from a list of possibilities in such a way that the entire computation succeeds:

```
import Effects
import Effect.Select

SELECT : EFFECT

select : List a -> Eff a [SELECT]

Handler Selection Maybe where { ... }
Handler Selection List where { ... }
```

Example

The `SELECT` effect can be used to solve constraint problems, such as finding Pythagorean triples. The idea is to use `select` to give a set of candidate values, then throw an exception for any combination of values which does not satisfy the constraint:

```
triple : Int -> Eff (Int, Int, Int) [SELECT, EXCEPTION String]
triple max = do z <- select [1..max]
                y <- select [1..z]
                x <- select [1..y]
                if (x * x + y * y == z * z)
                   then pure (x, y, z)
                   else raise "No triple"
```

This program chooses a value for `z` between `1` and `max`, then values for `y` and `x`. In operation, after a `select`, the program executes the rest of the `do`-block for every possible assignment, effectively searching depth-first. If the list is empty (or an exception is thrown) execution fails.

There are handlers defined for `Maybe` and `List` contexts, i.e. contexts which can capture failure. Depending on the context `m`, `triple` will either return the first triple it finds (if in `Maybe` context) or all triples in the range (if in `List` context). We can try this as follows:

```
main : IO ()
main = do print $ the (Maybe _) $ run (triple 100)
          print $ the (List _) $ run (triple 100)
```

4.3.5 vadd revisited

We now return to the vadd program from the introduction. Recall the definition:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd []        []        = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

Using , we can set up a program so that it reads input from a user, checks that the input is valid (i.e both vectors contain integers, and are the same length) and if so, pass it on to vadd. First, we write a wrapper for vadd which checks the lengths and throw an exception if they are not equal. We can do this for input vectors of length n and m by matching on the implicit arguments n and m and using decEq to produce a proof of their equality, if they are equal:

```
vadd_check : Vect n Int -> Vect m Int ->
             Eff (Vect m Int) [EXCEPTION String]
vadd_check {n} {m} xs ys with (decEq n m)
  vadd_check {n} {m=n} xs ys | (Yes Refl) = pure (vadd xs ys)
  vadd_check {n} {m}   xs ys | (No contra) = raise "Length mismatch"
```

To read a vector from the console, we implement a function of the following type:

```
read_vec : Eff (p ** Vect p Int) [STDIO]
```

This returns a dependent pair of a length, and a vector of that length, because we cannot know in advance how many integers the user is going to input. We can use -1 to indicate the end of input:

```
read_vec : Eff (p ** Vect p Int) [STDIO]
read_vec = do putStr "Number (-1 when done): "
              case run (parseNumber (trim !getStr)) of
                   Nothing => do putStrLn "Input error"
                                 read_vec
                   Just v => if (v /= -1)
                                then do (_ ** xs) <- read_vec
                                        pure (_ ** v :: xs)
                                else pure (_ ** [])
  where
    parseNumber : String -> Eff Int [EXCEPTION String]
    parseNumber str
      = if all (\x => isDigit x || x == '-') (unpack str)
           then pure (cast str)
           else raise "Not a number"
```

This uses a variation on parseNumber which does not require a number to be within range.

Finally, we write a program which reads two vectors and prints the result of pairwise addition of them, throwing an exception if the inputs are of differing lengths:

```
do_vadd : Eff () [STDIO, EXCEPTION String]
do_vadd = do putStrLn "Vector 1"
             (_ ** xs) <- read_vec
             putStrLn "Vector 2"
             (_ ** ys) <- read_vec
             putStrLn (show !(vadd_check xs ys))
```

By having explicit lengths in the type, we can be sure that `vadd` is only being used where the lengths of inputs are guaranteed to be equal. This does not stop us reading vectors from user input, but it does require that the lengths are checked and any discrepancy is dealt with gracefully.

4.3.6 Example: An Expression Calculator

To show how these effects can fit together, let us consider an evaluator for a simple expression language, with addition and integer values.

```
data Expr = Val Integer
          | Add Expr Expr
```

An evaluator for this language always returns an `Integer`, and there are no situations in which it can fail!

```
eval : Expr -> Integer
eval (Val x) = x
eval (Add l r) = eval l + eval r
```

If we add variables, however, things get more interesting. The evaluator will need to be able to access the values stored in variables, and variables may be undefined.

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr
```

To start, we will change the type of `eval` so that it is effectful, and supports an exception effect for throwing errors, and a state containing a mapping from variable names (as `String`) to their values:

```
Env : Type
Env = List (String, Integer)

eval : Expr -> Eff Integer [EXCEPTION String, STATE Env]
eval (Val x)   = pure x
eval (Add l r) = pure $ !(eval l) + !(eval r)
```

Note that we are using `!`-notation to avoid having to bind subexpressions in a `do` block. Next, we add a case for evaluating `Var`:

```
eval (Var x) = case lookup x !get of
                    Nothing => raise $ "No such variable " ++ x
                    Just val => pure val
```

This retrieves the state (with `get`, supported by the `STATE Env` effect) and raises an exception if the variable is not in the environment (with `raise`, supported by the `EXCEPTION String` effect).

To run the evaluator on a particular expression in a particular environment of names and their values, we can write a function which sets the state then invokes `eval`:

```
runEval : List (String, Integer) -> Expr -> Maybe Integer
runEval args expr = run (eval' expr)
  where eval' : Expr -> Eff Integer [EXCEPTION String, STATE Env]
        eval' e = do put args
                     eval e
```

We have picked `Maybe` as a computation context here; it needs to be a context which is available for every effect supported by `eval`. In particular, because we have exceptions, it needs to be a context which supports exceptions. Alternatively, `Either String` or `IO` would be fine, for example.

What if we want to extend the evaluator further, with random number generation? To achieve this, we add a new constructor to `Expr`, which gives a random number up to a maximum value:

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr
          | Random Integer
```

Then, we need to deal with the new case, making sure that we extend the list of events to include `RND`. It doesn't matter where `RND` appears in the list, as long as it is present:

```
eval : Expr -> Eff Integer [EXCEPTION String, RND, STATE Env]

eval (Random upper) = rndInt 0 upper
```

For test purposes, we might also want to print the random number which has been generated:

```
eval (Random upper) = do val <- rndInt 0 upper
                         putStrLn (show val)
                         pure val
```

If we try this without extending the effects list, we would see an error something like the following:

```
Expr.idr:28:6:When elaborating right hand side of eval:
Can't solve goal
   SubList [STDIO]
           [(EXCEPTION String), RND, (STATE (List (String, Integer)))]
```

In other words, the `STDIO` effect is not available. We can correct this simply by updating the type of `eval` to include `STDIO`.

```
eval : Expr -> Eff Integer [STDIO, EXCEPTION String, RND, STATE Env]
```

注解: Using `STDIO` will restrict the number of contexts in which `eval` can be `run` to those which support `STDIO`, such as `IO`. Once effect lists get longer, it can be a good idea instead to encapsulate sets of effects in a type synonym. This is achieved as follows, simply by defining a function which computes a type, since types are first class in Idris:

```
EvalEff : Type -> Type
EvalEff t = Eff t [STDIO, EXCEPTION String, RND, STATE Env]

eval : Expr -> EvalEff Integer
```

4.4 Dependent Effects

In the programs we have seen so far, the available effects have remained constant. Sometimes, however, an operation can *change* the available effects. The simplest example occurs when we have a state with a dependent type—adding an element to a vector also changes its type, for example, since its length is explicit in the type. In this section, we will see how the library supports this. Firstly, we will see how states with dependent types can be implemented. Secondly, we will see how the effects can depend on the *result* of an effectful operation. Finally, we will see how this can be used to implement a type-safe and resource-safe protocol for file management.

4.4.1 Dependent States

Suppose we have a function which reads input from the console, converts it to an integer, and adds it to a list which is stored in a `STATE`. It might look something like the following:

```
readInt : Eff () [STATE (List Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

But what if, instead of a list of integers, we would like to store a `Vect`, maintaining the length in the type?

```
readInt : Eff () [STATE (Vect n Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

This will not type check! Although the vector has length `n` on entry to `readInt`, it has length `S n` on exit. The library allows us to express this as follows:

```
readInt : Eff ()[STATE (Vect n Int), STDIO]
                [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
             putM (cast x :: !get)
```

The type `Eff a xs xs'` means that the operation begins with effects `xs` available, and ends with effects `xs'` available. We have used `putM` to update the state, where the `M` suffix indicates that the *type* is being updated as well as the value. It has the following type:

```
putM : y -> Eff () [STATE x] [STATE y]
```

4.4.2 Result-dependent Effects

Often, whether a state is updated could depend on the success or otherwise of an operation. In our `readInt` example, we might wish to update the vector only if the input is a valid integer (i.e. all digits). As a first attempt, we could try the following, returning a `Bool` which indicates success:

```
readInt : Eff Bool [STATE (Vect n Int), STDIO]
                   [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pure False
                  True => do putM (cast x :: !get)
                             pure True
```

Unfortunately, this will not type check because the vector does not get extended in both branches of the `case`!

```
MutState.idr:18:19:When elaborating right hand side of Main.case
block in readInt:
Unifying n and S n would lead to infinite value
```

Clearly, the size of the resulting vector depends on whether or not the value read from the user was valid. We can express this in the type:

```
readInt : Eff Bool [STATE (Vect n Int), STDIO]
            (\ok => if ok then [STATE (Vect (S n) Int), STDIO]
                          else [STATE (Vect n Int), STDIO])
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pureM False
                  True => do putM (cast x :: !get)
                             pureM True
```

Using `pureM` rather than `pure` allows the output effects to be calculated from the value given. Its type is:

```
pureM : (val : a) -> EffM m a (xs val) xs
```

When using `readInt`, we will have to check its return value in order to know what the new set of effects is. For example, to read a set number of values into a vector, we could write the following:

```
readN : (n : Nat) ->
        Eff () [STATE (Vect m Int), STDIO]
               [STATE (Vect (n + m) Int), STDIO]
readN Z = pure ()
readN {m} (S k) = case !readInt of
                      True => rewrite plusSuccRightSucc k m in readN k
                      False => readN (S k)
```

The `case` analysis on the result of `readInt` means that we know in each branch whether reading the integer succeeded, and therefore how many values still need to be read into the vector. What this means in practice is that the type system has verified that a necessary dynamic check (i.e. whether reading a value succeeded) has indeed been done.

注解: Only `case` will work here. We cannot use `if/then/else` because the `then` and `else` branches must have the same type. The `case` construct, however, abstracts over the value being inspected in the type of each branch.

4.4.3 File Management

A practical use for dependent effects is in specifying resource usage protocols and verifying that they are executed correctly. For example, file management follows a resource usage protocol with the following (informally specified) requirements:

- It is necessary to open a file for reading before reading it
- Opening may fail, so the programmer should check whether opening was successful
- A file which is open for reading must not be written to, and vice versa
- When finished, an open file handle should be closed
- When a file is closed, its handle should no longer be used

These requirements can be expressed formally in , by creating a `FILE_IO` effect parameterised over a file handle state, which is either empty, open for reading, or open for writing. The `FILE_IO` effect' s definition is given below. Note that this effect is mainly for illustrative purposes—typically we would also like to support random access files and better reporting of error conditions.

```
module Effect.File
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
import Effects
import Control.IOExcept

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                           True => OpenFile m
                                           False => ())])
close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine  : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]

Handler FileIO IO where { ... }
```

In particular, consider the type of `open`:

```
open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                           True => OpenFile m
                                           False => ())])
```

This returns a `Bool` which indicates whether opening the file was successful. The resulting state depends on whether the operation was successful; if so, we have a file handle open for the stated purpose, and if not, we have no file handle. By `case` analysis on the result, we continue the protocol accordingly.

```
readFile : Eff (List String) [FILE_IO (OpenFile Read)]
readFile = readAcc [] where
    readAcc : List String -> Eff (List String) [FILE_IO (OpenFile Read)]
    readAcc acc = if (not !eof)
                     then readAcc (!readLine :: acc)
                     else pure (reverse acc)
```

Given a function `readFile`, above, which reads from an open file until reaching the end, we can write a program which opens a file, reads it, then displays the contents and closes it, as follows, correctly following the protocol:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Read) of
                    True => do putStrLn (show !readFile)
                               close
                    False => putStrLn ("Error!")
```

The type of `dumpFile`, with `FILE_IO ()` in its effect list, indicates that any use of the file resource will follow the protocol correctly (i.e. it both begins and ends with an empty resource). If we fail to follow the protocol correctly (perhaps by forgetting to close the file, failing to check that `open` succeeded, or opening the file for writing) then we will get a compile-time error. For example, changing `open name Read` to `open name Write` yields a compile-time error of the following form:

```
FileTest.idr:16:18:When elaborating right hand side of Main.case
block in testFile:
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
Can't solve goal
        SubList [(FILE_IO (OpenFile Read))]
                [(FILE_IO (OpenFile Write)), STDIO]
```

In other words: when reading a file, we need a file which is open for reading, but the effect list contains a `FILE_IO` effect carrying a file open for writing.

4.4.4 Pattern-matching bind

It might seem that having to test each potentially failing operation with a `case` clause could lead to ugly code, with lots of nested case blocks. Many languages support exceptions to improve this, but unfortunately exceptions may not allow completely clean resource management—for example, guaranteeing that any `open` which did succeed has a corresponding close.

Idris supports *pattern-matching* bindings, such as the following:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = do True <- open name Read
                   putStrLn (show !readFile)
                   close
```

This also has a problem: we are no longer dealing with the case where opening a file failed! The solution is to extend the pattern-matching binding syntax to give brief clauses for failing matches. Here, for example, we could write:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name  = do True <- open name Read | False => putStrLn "Error"
                    putStrLn (show !readFile)
                    close
```

This is exactly equivalent to the definition with the explicit `case`. In general, in a `do`-block, the syntax:

```
do pat <- val | <alternatives>
   p
```

is desugared to

```
do x <- val
   case x of
        pat => p
        <alternatives>
```

There can be several `alternatives`, separated by a vertical bar `|`. For example, there is a `SYSTEM` effect which supports reading command line arguments, among other things (see Appendix *Effects Summary* (éąţ 133)). To read command line arguments, we can use `getArgs`:

```
getArgs : Eff (List String) [SYSTEM]
```

A main program can read command line arguments as follows, where in the list which is returned, the first element `prog` is the executable name and the second is an expected argument:

```
emain : Eff () [SYSTEM, STDIO]
emain = do [prog, arg] <- getArgs
           putStrLn $ "Argument is " ++ arg
           {- ... rest of function ... -}
```

Unfortunately, this will not fail gracefully if no argument is given, or if too many arguments are given. We can use pattern matching bind alternatives to give a better (more informative) error:

```
emain : Eff () [SYSTEM, STDIO]
emain = do [prog, arg] <- getArgs | [] => putStrLn "Can't happen!"
                                  | [prog] => putStrLn "No arguments!"
                                  | _ => putStrLn "Too many arguments!"
           putStrLn $ "Argument is " ++ arg
           {- ... rest of function ... -}
```

If `getArgs` does not return something of the form `[prog, arg]` the alternative which does match is executed instead, and that value returned.

4.5 Creating New Effects

We have now seen several side-effecting operations provided by the `Effects` library, and examples of their use in Section *Simple Effects* (ęąţ 111). We have also seen how operations may *modify* the available effects by changing state in Section *Dependent Effects* (ęąţ 118). We have not, however, yet seen how these operations are implemented. In this section, we describe how a selection of the available effects are implemented, and show how new effectful operations may be provided.

4.5.1 State

Effects are described by *algebraic data types*, where the constructors describe the operations provided when the effect is available. Stateful operations are described as follows:

```
data State : Effect where
     Get :      State a  a (\x => a)
     Put : b -> State () a (\x => b)
```

`Effect` itself is a type synonym, giving the required type for an effect signature:

```
Effect : Type
Effect = (result : Type) ->
         (input_resource : Type) ->
         (output_resource : result -> Type) -> Type
```

Each effect is associated with a *resource*. The second argument to an effect signature is the resource type on *input* to an operation, and the third is a function which computes the resource type on *output*. Here, it means:

- `Get` takes no arguments. It has a resource of type `a`, which is not updated, and running the `Get` operation returns something of type `a`.
- `Put` takes a `b` as an argument. It has a resource of type `a` on input, which is updated to a resource of type `b`. Running the `Put` operation returns the element of the unit type.

The effects library provides an overloaded function `sig` which can make effect signatures more concise, particularly when the result has no effect on the resource type. For `State`, we can write:

```
data State : Effect where
     Get :      sig State a  a
     Put : b -> sig State () a b
```

There are four versions of `sig`, depending on whether we are interested in the resource type, and whether we are updating the resource. Idris will infer the appropriate version from usage.

```
NoResourceEffect.sig : Effect -> Type -> Type
NoUpdateEffect.sig   : Effect -> (ret : Type) ->
                                 (resource : Type) -> Type
UpdateEffect.sig     : Effect -> (ret : Type) ->
                                 (resource_in : Type) ->
                                 (resource_out : Type) -> Type
DepUpdateEffect.sig  : Effect -> (ret : Type) ->
                                 (resource_in : Type) ->
                                 (resource_out : ret -> Type) -> Type
```

In order to convert `State` (of type `Effect`) into something usable in an effects list, of type `EFFECT`, we write the following:

```
STATE : Type -> EFFECT
STATE t = MkEff t State
```

`MkEff` constructs an `EFFECT` by taking the resource type (here, the `t` which parameterises `STATE`) and the effect signature (here, `State`). For reference, `EFFECT` is declared as follows:

```
data EFFECT : Type where
     MkEff : Type -> Effect -> EFFECT
```

Recall that to run an effectful program in `Eff`, we use one of the `run` family of functions to run the program in a particular computation context `m`. For each effect, therefore, we must explain how it is executed in a particular computation context for `run` to work in that context. This is achieved with the following interface:

```
interface Handler (e : Effect) (m : Type -> Type) where
      handle : resource -> (eff : e t resource resource') ->
               ((x : t) -> resource' x -> m a) -> m a
```

We have already seen some implementation declarations in the effect summaries in Section *Simple Effects* (éąţ 111). An implementation of `Handler e m` means that the effect declared with signature `e` can be run in computation context `m`. The `handle` function takes:

- The `resource` on input (so, the current value of the state for `State`)
- The effectful operation (either `Get` or `Put x` for `State`)
- A *continuation*, which we conventionally call `k`, and should be passed the result value of the operation, and an updated resource.

There are two reasons for taking a continuation here: firstly, this is neater because there are multiple return values (a new resource and the result of the operation); secondly, and more importantly, the continuation can be called zero or more times.

A `Handler` for `State` simply passes on the value of the state, in the case of `Get`, or passes on a new state, in the case of `Put`. It is defined the same way for all computation contexts:

```
Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k () n
```

This gives enough information for `Get` and `Put` to be used directly in `Eff` programs. It is tidy, however, to define top level functions in `Eff`, as follows:

```
get : Eff x [STATE x]
get = call Get
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
put : x -> Eff () [STATE x]
put val = call (Put val)

putM : y -> Eff () [STATE x] [STATE y]
putM val = call (Put val)
```

An implementation detail (aside): The `call` function converts an `Effect` to a function in `Eff`, given a proof that the effect is available. This proof can be constructed automatically, since it is essentially an index into a statically known list of effects:

```
call : {e : Effect} ->
       (eff : e t a b) -> {auto prf : EffElem e a xs} ->
       Eff t xs (\v => updateResTy v xs prf eff)
```

This is the reason for the `Can' t solve goal` error when an effect is not available: the implicit proof `prf` has not been solved automatically because the required effect is not in the list of effects `xs`.

Such details are not important for using the library, or even writing new effects, however.

Summary

The following listing summarises what is required to define the `STATE` effect:

```
data State : Effect where
     Get :      sig State a  a
     Put : b -> sig State () a b

STATE : Type -> EFFECT
STATE t = MkEff t State

Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k () n

get : Eff x [STATE x]
get = call Get

put : x -> Eff () [STATE x]
put val = call (Put val)

putM : y -> Eff () [STATE x] [STATE y]
putM val = call (Put val)
```

4.5.2 Console I/O

Then listing below gives the definition of the `STDIO` effect, including handlers for `IO` and `IOExcept`. We omit the definition of the top level `Eff` functions, as this merely invoke the effects `PutStr`, `GetStr`, `PutCh` and `GetCh` directly.

Note that in this case, the resource is the unit type in every case, since the handlers merely apply the `IO` equivalents of the effects directly.

```
data StdIO : Effect where
     PutStr : String -> sig StdIO ()
     GetStr : sig StdIO String
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
    PutCh : Char -> sig StdIO ()
    GetCh : sig StdIO Char

Handler StdIO IO where
    handle () (PutStr s) k = do putStr s; k () ()
    handle () GetStr     k = do x <- getLine; k x ()
    handle () (PutCh c)  k = do putChar c; k () ()
    handle () GetCh      k = do x <- getChar; k x ()

Handler StdIO (IOExcept a) where
    handle () (PutStr s) k = do ioe_lift $ putStr s; k () ()
    handle () GetStr     k = do x <- ioe_lift $ getLine; k x ()
    handle () (PutCh c)  k = do ioe_lift $ putChar c; k () ()
    handle () GetCh      k = do x <- ioe_lift $ getChar; k x ()

STDIO : EFFECT
STDIO = MkEff () StdIO
```

4.5.3 Exceptions

The listing below gives the definition of the `Exception` effect, including two of its handlers for `Maybe` and `List`. The only operation provided is `Raise`. The key point to note in the definitions of these handlers is that the continuation `k` is not used. Running `Raise` therefore means that computation stops with an error.

```
data Exception : Type -> Effect where
     Raise : a -> sig (Exception a) b

Handler (Exception a) Maybe where
     handle _ (Raise e) k = Nothing

Handler (Exception a) List where
     handle _ (Raise e) k = []

EXCEPTION : Type -> EFFECT
EXCEPTION t = MkEff () (Exception t)
```

4.5.4 Non-determinism

The following listing gives the definition of the `Select` effect for writing non-deterministic programs, including a handler for `List` context which returns all possible successful values, and a handler for `Maybe` context which returns the first successful value.

```
data Selection : Effect where
     Select : List a -> sig Selection a

Handler Selection Maybe where
     handle _ (Select xs) k = tryAll xs where
         tryAll [] = Nothing
         tryAll (x :: xs) = case k x () of
                                 Nothing => tryAll xs
                                 Just v => Just v

Handler Selection List where
     handle r (Select xs) k = concatMap (\x => k x r) xs
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
SELECT : EFFECT
SELECT = MkEff () Selection
```

Here, the continuation is called multiple times in each handler, for each value in the list of possible values. In the `List` handler, we accumulate all successful results, and in the `Maybe` handler we try the first value in the list, and try later values only if that fails.

4.5.5 File Management

Result-dependent effects are no different from non-dependent effects in the way they are implemented. The listing below illustrates this for the `FILE_IO` effect. The syntax for state transitions `{ x ==> {res} x' }`, where the result state `x'` is computed from the result of the operation `res`, follows that for the equivalent `Eff` programs.

```
data FileIO : Effect where
     Open : (fname: String)
            -> (m : Mode)
            -> sig FileIO Bool () (\res => case res of
                                               True => OpenFile m
                                               False => ())
     Close : sig FileIO () (OpenFile m)

     ReadLine  :           sig FileIO String (OpenFile Read)
     WriteLine : String -> sig FileIO ()     (OpenFile Write)
     EOF       :           sig FileIO Bool   (OpenFile Read)

Handler FileIO IO where
    handle () (Open fname m) k = do h <- openFile fname m
                                    if !(validFile h)
                                             then k True (FH h)
                                             else k False ()
    handle (FH h) Close      k = do closeFile h
                                    k () ()

    handle (FH h) ReadLine        k = do str <- fread h
                                         k str (FH h)
    handle (FH h) (WriteLine str) k = do fwrite h str
                                         k () (FH h)
    handle (FH h) EOF             k = do e <- feof h
                                         k e (FH h)

FILE_IO : Type -> EFFECT
FILE_IO t = MkEff t FileIO
```

Note that in the handler for `Open`, the types passed to the continuation `k` are different depending on whether the result is `True` (opening succeeded) or `False` (opening failed). This uses `validFile`, defined in the `Prelude`, to test whether a file handler refers to an open file or not.

4.6 Example: A "Mystery Word" Guessing Game

In this section, we will use the techniques and specific effects discussed in the tutorial so far to implement a larger example, a simple text-based word-guessing game. In the game, the computer chooses a word, which the player must guess letter by letter. After a limited number of wrong guesses, the player loses[1].

[1] Readers may recognise this game by the name "Hangman".

We will implement the game by following these steps:

1. Define the game state, in enough detail to express the rules
2. Define the rules of the game (i.e. what actions the player may take, and how these actions affect the game state)
3. Implement the rules of the game (i.e. implement state updates for each action)
4. Implement a user interface which allows a player to direct actions

Step 2 may be achieved by defining an effect which depends on the state defined in step 1. Then step 3 involves implementing a `Handler` for this effect. Finally, step 4 involves implementing a program in `Eff` using the newly defined effect (and any others required to implement the interface).

4.6.1 Step 1: Game State

First, we categorise the game states as running games (where there are a number of guesses available, and a number of letters still to guess), or non-running games (i.e. games which have not been started, or games which have been won or lost).

```
data GState = Running Nat Nat | NotRunning
```

Notice that at this stage, we say nothing about what it means to make a guess, what the word to be guessed is, how to guess letters, or any other implementation detail. We are only interested in what is necessary to describe the game rules.

We will, however, parameterise a concrete game state `Mystery` over this data:

```
data Mystery : GState -> Type
```

4.6.2 Step 2: Game Rules

We describe the game rules as a dependent effect, where each action has a *precondition* (i.e. what the game state must be before carrying out the action) and a *postcondition* (i.e. how the action affects the game state). Informally, these actions with the pre- and postconditions are:

Guess Guess a letter in the word.

- Precondition: The game must be running, and there must be both guesses still available, and letters still to be guessed.
- Postcondition: If the guessed letter is in the word and not yet guessed, reduce the number of letters, otherwise reduce the number of guesses.

Won Declare victory

- Precondition: The game must be running, and there must be no letters still to be guessed.
- Postcondition: The game is no longer running.

Lost Accept defeat

- Precondition: The game must be running, and there must be no guesses left.
- Postcondition: The game is no longer running.

NewWord Set a new word to be guessed

- Precondition: The game must not be running.
- Postcondition: The game is running, with 6 guesses available (the choice of 6 is somewhat arbitrary here) and the number of unique letters in the word still to be guessed.

Get Get a string representation of the game state. This is for display purposes; there are no pre- or postconditions.

We can make these rules precise by declaring them more formally in an effect signature:

```
data MysteryRules : Effect where
     Guess : (x : Char) ->
             sig MysteryRules Bool
                 (Mystery (Running (S g) (S w)))
                 (\inword => if inword
                             then Mystery (Running (S g) w)
                             else Mystery (Running g (S w)))
     Won  : sig MysteryRules () (Mystery (Running g 0))
                                (Mystery NotRunning)
     Lost : sig MysteryRules () (Mystery (Running 0 g))
                                (Mystery NotRunning)
     NewWord : (w : String) ->
               sig MysteryRules () (Mystery NotRunning) (Mystery (Running 6 (length (letters␣
↪w))))
     Get : sig MysteryRules String (Mystery h)
```

This description says nothing about how the rules are implemented. In particular, it does not specify *how* to tell whether a guessed letter was in a word, just that the result of `Guess` depends on it.

Nevertheless, we can still create an `EFFECT` from this, and use it in an `Eff` program. Implementing a `Handler` for `MysteryRules` will then allow us to play the game.

```
MYSTERY : GState -> EFFECT
MYSTERY h = MkEff (Mystery h) MysteryRules
```

4.6.3 Step 3: Implement Rules

To *implement* the rules, we begin by giving a concrete definition of game state:

```
data Mystery : GState -> Type where
     Init     : Mystery NotRunning
     GameWon  : (word : String) -> Mystery NotRunning
     GameLost : (word : String) -> Mystery NotRunning
     MkG      : (word : String) ->
                (guesses : Nat) ->
                (got : List Char) ->
                (missing : Vect m Char) ->
                Mystery (Running guesses m)
```

If a game is `NotRunning`, that is either because it has not yet started (`Init`) or because it is won or lost (`GameWon` and `GameLost`, each of which carry the word so that showing the game state will reveal the word to the player). Finally, `MkG` captures a running game' s state, including the target word, the letters successfully guessed, and the missing letters. Using a `Vect` for the missing letters is convenient since its length is used in the type.

To initialise the state, we implement the following functions: `letters`, which returns a list of unique letters in a `String` (ignoring spaces) and `initState` which sets up an initial state considered valid as a postcondition for `NewWord`.

```
letters : String -> List Char
initState : (x : String) -> Mystery (Running 6 (length (letters x)))
```

When checking if a guess is in the vector of missing letters, it is convenient to return a *proof* that the guess is in the vector, using `isElem` below, rather than merely a `Bool`:

```
data IsElem : a -> Vect n a -> Type where
     First : IsElem x (x :: xs)
     Later : IsElem x xs -> IsElem x (y :: xs)

isElem : DecEq a => (x : a) -> (xs : Vect n a) -> Maybe (IsElem x xs)
```

The reason for returning a proof is that we can use it to remove an element from the correct position in a vector:

```
shrink : (xs : Vect (S n) a) -> IsElem x xs -> Vect n a
```

We leave the definitions of `letters`, `init`, `isElem` and `shrink` as exercises. Having implemented these, the `Handler` implementation for `MysteryRules` is surprisingly straightforward:

```
Handler MysteryRules m where
    handle (MkG w g got []) Won k = k () (GameWon w)
    handle (MkG w Z got m) Lost k = k () (GameLost w)

    handle st Get k = k (show st) st
    handle st (NewWord w) k = k () (initState w)

    handle (MkG w (S g) got m) (Guess x) k =
        case isElem x m of
             Nothing => k False (MkG w _ got m)
             (Just p) => k True (MkG w _ (x :: got) (shrink m p))
```

Each case simply involves directly updating the game state in a way which is consistent with the declared rules. In particular, in `Guess`, if the handler claims that the guessed letter is in the word (by passing `True` to `k`), there is no way to update the state in such a way that the number of missing letters or number of guesses does not follow the rules.

4.6.4 Step 4: Implement Interface

Having described the rules, and implemented state transitions which follow those rules as an effect handler, we can now write an interface for the game which uses the `MYSTERY` effect:

```
game : Eff () [MYSTERY (Running (S g) w), STDIO]
              [MYSTERY NotRunning, STDIO]
```

The type indicates that the game must start in a running state, with some guesses available, and eventually reach a not-running state (i.e. won or lost). The only way to achieve this is by correctly following the stated rules.

Note that the type of `game` makes no assumption that there are letters to be guessed in the given word (i.e. it is `w` rather than `S w`). This is because we will be choosing a word at random from a vector of `String`, and at no point have we made it explicit that those `String` are non-empty.

Finally, we need to initialise the game by picking a word at random from a list of candidates, setting it as the target using `NewWord`, then running `game`:

```
runGame : Eff () [MYSTERY NotRunning, RND, SYSTEM, STDIO]
runGame = do srand !time
             let w = index !(rndFin _) words
             call $ NewWord w
             game
             putStrLn !(call Get)
```

We use the system time (`time` from the `SYSTEM` effect; see Appendix *Effects Summary* (éąţ 133)) to initialise the random number generator, then pick a random `Fin` to index into a list of `words`. For example, we could initialise a word list as follows:

```
words : ?wtype
words = with Vect ["idris","agda","haskell","miranda",
         "java","javascript","fortran","basic",
         "coffeescript","rust"]

wtype = proof search
```

注解: Rather than have to explicitly declare a type with the vector' s length, it is convenient to give a hole `?wtype` and let Idris' s proof search mechanism find the type. This is a limited form of type inference, but very useful in practice.

A possible complete implementation of `game` is presented below:

```
game : Eff () [MYSTERY (Running (S g) w), STDIO]
              [MYSTERY NotRunning, STDIO]
game {w=Z} = Won
game {w=S _}
     = do putStrLn !Get
          putStr "Enter guess: "
          let guess = trim !getStr
          case choose (not (guess == "")) of
               (Left p) => processGuess (strHead' guess p)
               (Right p) => do putStrLn "Invalid input!"
                               game
  where
    processGuess : Char -> Eff () [MYSTERY (Running (S g) (S w)), STDIO]
                                  [MYSTERY NotRunning, STDIO]
    processGuess {g} {w} c
      = case !(Main.Guess c) of
             True => do putStrLn "Good guess!"
                        case w of
                             Z => Won
                             (S k) => game
             False => do putStrLn "No, sorry"
                         case g of
                              Z => Lost
                              (S k) => game
```

4.6.5 Discussion

Writing the rules separately as an effect, then an implementation which uses that effect, ensures that the implementation must follow the rules. This has practical applications in more serious contexts; `MysteryRules` for example can be though of as describing a *protocol* that a game player most follow, or alternative a *precisely-typed API*.

In practice, we wouldn' t really expect to write rules first then implement the game once the rules

were complete. Indeed, I didn't do so when constructing this example! Rather, I wrote down a set of likely rules making any assumptions *explicit* in the state transitions for `MysteryRules`. Then, when implementing `game` at first, any incorrect assumption was caught as a type error. The following errors were caught during development:

- Not realising that allowing `NewWord` to be an arbitrary string would mean that `game` would have to deal with a zero-length word as a starting state.
- Forgetting to check whether a game was won before recursively calling `processGuess`, thus accidentally continuing a finished game.
- Accidentally checking the number of missing letters, rather than the number of remaining guesses, when checking if a game was lost.

These are, of course, simple errors, but were caught by the type checker before any testing of the game.

4.7 Further Reading

This tutorial has given an introduction to writing and reasoning about side-effecting programs in Idris, using the `Effects` library. More details about the *implementation* of the library, such as how `run` works, how handlers are invoked, etc, are given in a separate paper[1].

Some libraries and programs which use `Effects` can be found in the following places:

- https://github.com/edwinb/SDL-idris — some bindings for the SDL media library, supporting graphics in particular.
- https://github.com/edwinb/idris-demos — various demonstration programs, including several examples from this tutorial, and a "Space Invaders" game.
- https://github.com/SimonJF/IdrisNet2 — networking and socket libraries.
- https://github.com/edwinb/Protocols — a high level communication protocol description language.

The inspiration for the `Effects` library was Bauer and Pretnar's Eff language[2], which describes a language based on algebraic effects and handlers. Other recent languages and libraries have also been built on this ideas, for example[3] and[4]. The theoretical foundations are also well-studied see[5],[6],[7],[8].

[1] Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. SIGPLAN Not. 48, 9 (September 2013), 133-144. DOI=10.1145/2544174.2500581 https://dl.acm.org/citation.cfm?doid=2544174.2500581

[2] Andrej Bauer, Matija Pretnar, Programming with algebraic effects and handlers, Journal of Logical and Algebraic Methods in Programming, Volume 84, Issue 1, January 2015, Pages 108-123, ISSN 2352-2208, http://math.andrej.com/wp-content/uploads/2012/03/eff.pdf

[3] Ben Lippmeier. 2009. Witnessing Purity, Constancy and Mutability. In Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09), Zhenjiang Hu (Ed.). Springer-Verlag, Berlin, Heidelberg, 95-110. DOI=10.1007/978-3-642-10672-9_9 http://link.springer.com/chapter/10.1007%2F978-3-642-10672-9_9

[4] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. SIGPLAN Not. 48, 9 (September 2013), 145-158. DOI=10.1145/2544174.2500590 https://dl.acm.org/citation.cfm?doid=2544174.2500590

[5] Martin Hyland, Gordon Plotkin, John Power, Combining effects: Sum and tensor, Theoretical Computer Science, Volume 357, Issues 1–3, 25 July 2006, Pages 70-99, ISSN 0304-3975, (https://www.sciencedirect.com/science/article/pii/S0304397506002659)

[6] Paul Blain Levy. 2004. Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2). Kluwer Academic Publishers, Norwell, MA, USA.

[7] Plotkin, Gordon, and Matija Pretnar. "Handlers of algebraic effects." Programming Languages and Systems. Springer Berlin Heidelberg, 2009. 80-94.

[8] Pretnar, Matija. "Logic and handling of algebraic effects." (2010).

4.8 Effects Summary

This appendix gives interfaces for the core effects provided by the library.

4.8.1 EXCEPTION

```
module Effect.Exception

import Effects
import System
import Control.IOExcept

EXCEPTION : Type -> EFFECT

raise : a -> Eff b [EXCEPTION a]

Handler (Exception a) Maybe where { ... }
Handler (Exception a) List where { ... }
Handler (Exception a) (Either a) where { ... }
Handler (Exception a) (IOExcept a) where { ... }
Show a => Handler (Exception a) IO where { ... }
```

4.8.2 FILE_IO

```
module Effect.File

import Effects
import Control.IOExcept

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                           True => OpenFile m
                                           False => ())])
close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine  : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]

Handler FileIO IO where { ... }
```

4.8.3 RND

```
module Effect.Random

import Effects
import Data.Vect
import Data.Fin
```

(äÿŃéąţźžğçżŋ)

(çżŋäÿŁéąţ)

```
RND : EFFECT

srand  : Integer ->            Eff m () [RND]
rndInt : Integer -> Integer -> Eff m Integer [RND]
rndFin : (k : Nat) ->          Eff m (Fin (S k)) [RND]

Handler Random m where { ... }
```

4.8.4 SELECT

```
module Effect.Select

import Effects

SELECT : EFFECT

select : List a -> Eff m a [SELECT]

Handler Selection Maybe where { ... }
Handler Selection List where { ... }
```

4.8.5 STATE

```
module Effect.State

import Effects

STATE : Type -> EFFECT

get    :             Eff m x [STATE x]
put    : x ->        Eff m () [STATE x]
putM   : y ->        Eff m () [STATE x] [STATE y]
update : (x -> x) -> Eff m () [STATE x]

Handler State m where { ... }
```

4.8.6 STDIO

```
module Effect.StdIO

import Effects
import Control.IOExcept

STDIO : EFFECT

putChar  : Handler StdIO m => Char ->   Eff m () [STDIO]
putStr   : Handler StdIO m => String -> Eff m () [STDIO]
putStrLn : Handler StdIO m => String -> Eff m () [STDIO]

getStr   : Handler StdIO m =>           Eff m String [STDIO]
getChar  : Handler StdIO m =>           Eff m Char [STDIO]
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
Handler StdIO IO where { ... }
Handler StdIO (IOExcept a) where { ... }
```

4.8.7 SYSTEM

```
module Effect.System

import Effects
import System
import Control.IOExcept

SYSTEM : EFFECT

getArgs : Handler System e =>           Eff e (List String) [SYSTEM]
time    : Handler System e =>           Eff e Int [SYSTEM]
getEnv  : Handler System e => String -> Eff e (Maybe String) [SYSTEM]

Handler System IO where { ... }
Handler System (IOExcept a) where { ... }
```

CHAPTER 5

Theorem Proving

A tutorial on theorem proving in Idris.

5.1 Running example: Addition of Natural Numbers

Throughout this tutorial, we will be working with the following function, defined in the Idris prelude, which defines addition on natural numbers:

```
plus : Nat -> Nat -> Nat
plus Z     m = m
plus (S k) m = S (plus k m)
```

It is defined by the above equations, meaning that we have for free the properties that adding `m` to zero always results in `m`, and that adding `m` to any non-zero number `S k` always results in `S (plus k m)`. We can see this by evaluation at the Idris REPL (i.e. the prompt, the read-eval-print loop):

```
Idris> \m => plus Z m
\m => m : Nat -> Nat

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

Note that unlike many other language REPLs, the Idris REPL performs evaluation on *open* terms, meaning that it can reduce terms which appear inside lambda bindings, like those above. Therefore, we can introduce unknowns `k` and `m` as lambda bindings and see how `plus` reduces.

The `plus` function has a number of other useful properties, for example:

- It is *commutative*, that is for all `Nat` inputs `n` and `m`, we know that `plus n m = plus m n`.
- It is *associative*, that is for all `Nat` inputs `n`, `m` and `p`, we know that `plus n (plus m p) = plus (plus m n) p`.

We can use these properties in an Idris program, but in order to do so we must *prove* them.

5.1.1 Equality Proofs

Idris has a built-in propositional equality type, conceptually defined as follows:

```
data (=) : a -> b -> Type where
   Refl : x = x
```

Note that this must be built-in, rather than defined in the library, because `=` is a reserved operator — you cannot define this directly in your own code.

It is *propositional* equality, where the type states that any two values in different types `a` and `b` may be proposed to be equal. There is only one way to *prove* equality, however, which is by reflexivity (`Refl`).

We have a *type* for propositional equality here, and correspondingly a *program* inhabiting an instance of this type can be seen as a proof of the corresponding proposition[1]. So, trivially, we can prove that `4` equals `4`:

```
four_eq : 4 = 4
four_eq = Refl
```

However, trying to prove that `4 = 5` results in failure:

```
four_eq_five : 4 = 5
four_eq_five = Refl
```

The type `4 = 5` is a perfectly valid type, but is uninhabited, so when trying to type check this definition, Idris gives the following error:

```
When elaborating right hand side of four_eq_five:
Type mismatch between
        x = x (Type of Refl)
and
        4 = 5 (Expected type)
```

Type checking equality proofs

An important step in type checking Idris programs is *unification*, which attempts to resolve implicit arguments such as the implicit argument `x` in `Refl`. As far as our understanding of type checking proofs is concerned, it suffices to know that unifying two terms involves reducing both to normal form then trying to find an assignment to implicit arguments which will make those normal forms equal.

When type checking `Refl`, Idris requires that the type is of the form `x = x`, as we see from the type of `Refl`. In the case of `four_eq_five`, Idris will try to unify the expected type `4 = 5` with the type of `Refl`, `x = x`, notice that a solution requires that `x` be both `4` and `5`, and therefore fail.

Since type checking involves reduction to normal form, we can write the following equalities directly:

[1] This is known as the Curry-Howard correspondence.

```
twoplustwo_eq_four : 2 + 2 = 4
twoplustwo_eq_four = Refl

plus_reduces_Z : (m : Nat) -> plus Z m = m
plus_reduces_Z m = Refl

plus_reduces_Sk : (k, m : Nat) -> plus (S k) m = S (plus k m)
plus_reduces_Sk k m = Refl
```

5.1.2 Heterogeneous Equality

Equality in Idris is *heterogeneous*, meaning that we can even propose equalities between values in different types:

```
idris_not_php : 2 = "2"
```

Obviously, in Idris the type `2 = "2"` is uninhabited, and one might wonder why it is useful to be able to propose equalities between values in different types. However, with dependent types, such equalities can arise naturally. For example, if two vectors are equal, their lengths must be equal:

```
vect_eq_length : (xs : Vect n a) -> (ys : Vect m a) ->
                 (xs = ys) -> n = m
```

In the above declaration, `xs` and `ys` have different types because their lengths are different, but we would still like to draw a conclusion about the lengths if they happen to be equal. We can define `vect_eq_length` as follows:

```
vect_eq_length xs xs Refl = Refl
```

By matching on `Refl` for the third argument, we know that the only valid value for `ys` is `xs`, because they must be equal, and therefore their types must be equal, so the lengths must be equal.

Alternatively, we can put an underscore for the second `xs`, since there is only one value which will type check:

```
vect_eq_length xs _ Refl = Refl
```

5.1.3 Properties of `plus`

Using the `(=)` type, we can now state the properties of `plus` given above as Idris type declarations:

```
plus_commutes : (n, m : Nat) -> plus n m = plus m n
plus_assoc : (n, m, p : Nat) -> plus n (plus m p) = plus (plus n m) p
```

Both of these properties (and many others) are proved for natural number addition in the Idris standard library, using `(+)` from the `Num` interface rather than using `plus` directly. They have the names `plusCommutative` and `plusAssociative` respectively.

In the remainder of this tutorial, we will explore several different ways of proving `plus_commutes` (or, to put it another way, writing the function.) We will also discuss how to use such equality proofs, and see where the need for them arises in practice.

5.2 Inductive Proofs

Before embarking on proving `plus_commutes` in Idris itself, let us consider the overall structure of a proof of some property of natural numbers. Recall that they are defined recursively, as follows:

```
data Nat : Type where
     Z : Nat
     S : Nat -> Nat
```

A *total* function over natural numbers must both terminate, and cover all possible inputs. Idris checks functions for totality by checking that all inputs are covered, and that all recursive calls are on *structurally smaller* values (so recursion will always reach a base case). Recalling `plus`:

```
plus : Nat -> Nat -> Nat
plus Z     m = m
plus (S k) m = S (plus k m)
```

This is total because it covers all possible inputs (the first argument can only be `Z` or `S k` for some `k`, and the second argument `m` covers all possible `Nat`) and in the recursive call, `k` is structurally smaller than `S k` so the first argument will always reach the base case `Z` in any sequence of recursive calls.

In some sense, this resembles a mathematical proof by induction (and this is no coincidence!). For some property `P` of a natural number `x`, we can show that `P` holds for all `x` if:

- `P` holds for zero (the base case).
- Assuming that `P` holds for `k`, we can show `P` also holds for `S k` (the inductive step).

In `plus`, the property we are trying to show is somewhat trivial (for all natural numbers `x`, there is a `Nat` which need not have any relation to `x`). However, it still takes the form of a base case and an inductive step. In the base case, we show that there is a `Nat` arising from `plus n m` when `n = Z`, and in the inductive step we show that there is a `Nat` arising when `n = S k` and we know we can get a `Nat` inductively from `plus k m`. We could even write a function capturing all such inductive definitions:

```
nat_induction : (P : Nat -> Type) ->              -- Property to show
                (P Z) ->                          -- Base case
                ((k : Nat) -> P k -> P (S k)) -> -- Inductive step
                (x : Nat) ->                      -- Show for all x
                P x
nat_induction P p_Z p_S Z = p_Z
nat_induction P p_Z p_S (S k) = p_S k (nat_induction P p_Z p_S k)
```

Using `nat_induction`, we can implement an equivalent inductive version of `plus`:

```
plus_ind : Nat -> Nat -> Nat
plus_ind n m
   = nat_induction (\x => Nat)
                   m                        -- Base case, plus_ind Z m
                   (\k, k_rec => S k_rec)   -- Inductive step plus_ind (S k) m
                                            -- where k_rec = plus_ind k m
                   n
```

To prove that `plus n m = plus m n` for all natural numbers `n` and `m`, we can also use induction. Either we can fix `m` and perform induction on `n`, or vice versa. We can sketch an outline of a proof; performing induction on `n`, we have:

- Property `P` is `\x => plus x m = plus m x`.
- Show that `P` holds in the base case and inductive step:

– Base case: `P Z`, i.e.
`plus Z m = plus m Z`, which reduces to
`m = plus m Z` due to the definition of `plus`.

– Inductive step: Inductively, we know that `P k` holds for a specific, fixed `k`, i.e.
`plus k m = plus m k` (the induction hypothesis). Given this, show `P (S k)`, i.e.
`plus (S k) m = plus m (S k)`, which reduces to
`S (plus k m) = plus m (S k)`. From the induction hypothesis, we can rewrite this to
`S (plus m k) = plus m (S k)`.

To complete the proof we therefore need to show that `m = plus m Z` for all natural numbers `m`, and that `S (plus m k) = plus m (S k)` for all natural numbers `m` and `k`. Each of these can also be proved by induction, this time on `m`.

We are now ready to embark on a proof of commutativity of `plus` formally in Idris.

5.3 Pattern Matching Proofs

In this section, we will provide a proof of `plus_commutes` directly, by writing a pattern matching definition. We will use interactive editing features extensively, since it is significantly easier to produce a proof when the machine can give the types of intermediate values and construct components of the proof itself. The commands we will use are summarised below. Where we refer to commands directly, we will use the Vim version, but these commands have a direct mapping to Emacs commands.

Command	Vim binding	Emacs binding	Explanation
Check type	`\t`	`C-c C-t`	Show type of identifier or hole under the cursor.
Proof search	`\o`	`C-c C-a`	Attempt to solve hole under the cursor by applying simple proof search.
Make new definition	`\d`	`C-c C-s`	Add a template definition for the type defined under the cursor.
Make lemma	`\l`	`C-c C-e`	Add a top level function with a type which solves the hole under the cursor.
Split cases	`\c`	`C-c C-c`	Create new constructor patterns for each possible case of the variable under the cursor.

5.3.1 Creating a Definition

To begin, create a file `pluscomm.idr` containing the following type declaration:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
```

To create a template definition for the proof, press `\d` (or the equivalent in your editor of choice) on the line with the type declaration. You should see:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes n m = ?plus_commutes_rhs
```

To prove this by induction on `n`, as we sketched in Section *Inductive Proofs* (ęąţ 139), we begin with a case split on `n` (press `\c` with the cursor over the `n` in the definition.) You should see:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = ?plus_commutes_rhs_1
plus_commutes (S k) m = ?plus_commutes_rhs_2
```

If we inspect the types of the newly created holes, `plus_commutes_rhs_1` and `plus_commutes_rhs_2`, we see that the type of each reflects that `n` has been refined to `Z` and `S k` in each respective case. Pressing `\t` over `plus_commutes_rhs_1` shows:

```
  m : Nat
--------------------------------------
plus_commutes_rhs_1 : m = plus m 0
```

Note that `Z` renders as `0` because the pretty printer renders natural numbers as integer literals for readability. Similarly, for `plus_commutes_rhs_2`:

```
  k : Nat
  m : Nat
--------------------------------------
plus_commutes_rhs_2 : S (plus k m) = plus m (S k)
```

It is a good idea to give these slightly more meaningful names:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = ?plus_commutes_Z
plus_commutes (S k) m = ?plus_commutes_S
```

5.3.2 Base Case

We can create a separate lemma for the base case interactively, by pressing `\l` with the cursor over `plus_commutes_Z`. This yields:

```
plus_commutes_Z : m = plus m 0

plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = plus_commutes_Z
plus_commutes (S k) m = ?plus_commutes_S
```

That is, the hole has been filled with a call to a top level function `plus_commutes_Z`. The argument `m` has been made implicit because it can be inferred from context when it is applied.

Unfortunately, we cannot prove this lemma directly, since `plus` is defined by matching on its *first* argument, and here `plus m 0` has a specific value for its *second argument* (in fact, the left hand side of the equality has been reduced from `plus 0 m`.) Again, we can prove this by induction, this time on `m`.

First, create a template definition with `\d`:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z = ?plus_commutes_Z_rhs
```

Since we are going to write this by induction on `m`, which is implicit, we will need to bring `m` into scope manually:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m} = ?plus_commutes_Z_rhs
```

Now, case split on `m` with `\c`:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = ?plus_commutes_Z_rhs_1
plus_commutes_Z {m = (S k)} = ?plus_commutes_Z_rhs_2
```

Checking the type of `plus_commutes_Z_rhs_1` shows the following, which is easily proved by reflection:

```
--------------------------------------
plus_commutes_Z_rhs_1 : 0 = 0
```

For such trivial proofs, we can let write the proof automatically by pressing `\o` with the cursor over `plus_commutes_Z_rhs_1`. This yields:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = Refl
plus_commutes_Z {m = (S k)} = ?plus_commutes_Z_rhs_2
```

For `plus_commutes_Z_rhs_2`, we are not so lucky:

```
  k : Nat
--------------------------------------
plus_commutes_Z_rhs_2 : S k = S (plus k 0)
```

Inductively, we should know that `k = plus k 0`, and we can get access to this inductive hypothesis by making a recursive call on k, as follows:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = Refl
plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  ?plus_commutes_Z_rhs_2
```

For `plus_commutes_Z_rhs_2`, we now see:

```
  k : Nat
  rec : k = plus k (fromInteger 0)
--------------------------------------
plus_commutes_Z_rhs_2 : S k = S (plus k 0)
```

Again, the `fromInteger 0` is merely due to `Nat` having an implementation of the `Num` interface. So we know that `k = plus k 0`, but how do we use this to update the goal to `S k = S k`?

To achieve this, Idris provides a `replace` function as part of the prelude:

```
*pluscomm> :t replace
replace : (x = y) -> P x -> P y
```

Given a proof that `x = y`, and a property `P` which holds for `x`, we can get a proof of the same property for `y`, because we know `x` and `y` must be the same. In practice, this function can be a little tricky to use because in general the implicit argument `P` can be hard to infer by unification, so Idris provides a high level syntax which calculates the property and applies `replace`:

```
rewrite prf in expr
```

If we have `prf : x = y`, and the required type for `expr` is some property of `x`, the `rewrite ... in` syntax will search for `x` in the required type of `expr` and replace it with `y`. Concretely, in our example, we can say:

```
plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  rewrite rec in ?plus_commutes_Z_rhs_2
```

Checking the type of `plus_commutes_Z_rhs_2` now gives:

```
  k : Nat
  rec : k = plus k (fromInteger 0)
  _rewrite_rule : plus k 0 = k
--------------------------------------
plus_commutes_Z_rhs_2 : S (plus k 0) = S (plus k 0)
```

Using the rewrite rule `rec` (which we can see in the context here as `_rewrite_rule`[1], the goal type has been updated with `k` replaced by `plus k 0`.

Alternatively, we could have applied the rewrite in the other direction using the `sym` function:

```
*pluscomm> :t sym
sym : (l = r) -> r = l

plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  rewrite sym rec in ?plus_commutes_Z_rhs_2
```

In this case, inspecting the type of the hole gives:

```
  k : Nat
  rec : k = plus k (fromInteger 0)
  _rewrite_rule : k = plus k 0
--------------------------------------
plus_commutes_Z_rhs_2 : S k = S k
```

Either way, we can use proof search (`\o`) to complete the proof, giving:

```
plus_commutes_Z : m = plus m 0
plus_commutes_Z {m = Z} = Refl
plus_commutes_Z {m = (S k)} = let rec = plus_commutes_Z {m=k} in
                                  rewrite rec in Refl
```

The base case is now complete.

5.3.3 Inductive Step

Our main theorem, `plus_commutes` should currently be in the following state:

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = plus_commutes_Z
plus_commutes (S k) m = ?plus_commutes_S
```

Looking again at the type of `plus_commutes_S`, we have:

```
  k : Nat
  m : Nat
--------------------------------------
plus_commutes_S : S (plus k m) = plus m (S k)
```

Conveniently, by induction we can immediately tell that `plus k m = plus m k`, so let us rewrite directly by making a recursive call to `plus_commutes`. We add this directly, by hand, as follows:

[1] Note that the left and right hand sides of the equality have been swapped, because `replace` takes a proof of `x=y` and the property for `x`, not `y`.

```
plus_commutes : (n : Nat) -> (m : Nat) -> n + m = m + n
plus_commutes Z m = plus_commutes_Z
plus_commutes (S k) m = rewrite plus_commutes k m in ?plus_commutes_S
```

Checking the type of `plus_commutes_S` now gives:

```
  k : Nat
  m : Nat
  _rewrite_rule : plus m k = plus k m
--------------------------------------
plus_commutes_S : S (plus m k) = plus m (S k)
```

The good news is that `m` and `k` now appear in the correct order. However, we still have to show that the successor symbol `S` can be moved to the front in the right hand side of this equality. This remaining lemma takes a similar form to the `plus_commutes_Z`; we begin by making a new top level lemma with `\l`. This gives:

```
plus_commutes_S : (k : Nat) -> (m : Nat) -> S (plus m k) = plus m (S k)
```

Unlike the previous case, `k` and `m` are not made implicit because we cannot in general infer arguments to a function from its result. Again, we make a template definition with `\d`:

```
plus_commutes_S : (k : Nat) -> (m : Nat) -> S (plus m k) = plus m (S k)
plus_commutes_S k m = ?plus_commutes_S_rhs
```

Again, this is defined by induction over `m`, since `plus` is defined by matching on its first argument. The complete definition is:

```
total
plus_commutes_S : (k : Nat) -> (m : Nat) -> S (plus m k) = plus m (S k)
plus_commutes_S k Z = Refl
plus_commutes_S k (S j) = rewrite plus_commutes_S k j in Refl
```

All holes have now been solved.

The `total` annotation means that we require the final function to pass the totality checker; i.e. it will terminate on all possible well-typed inputs. This is important for proofs, since it provides a guarantee that the proof is valid in *all* cases, not just those for which it happens to be well-defined.

Now that `plus_commutes` has a `total` annotation, we have completed the proof of commutativity of addition on natural numbers.

5.4 DEPRECATED: Interactive Theorem Proving

> **警告:** The interactive theorem-proving interface documented here has been deprecated in favor of *Elaborator Reflection* (ép. 199).

Idris also supports interactive theorem proving via tactics. This is generally not recommended to be used directly, but rather used as a mechanism for building proof automation which is beyond the scope of this tutorial. In this section, we briefly discus tactics.

One way to write proofs interactively is to write the general *structure* of the proof, and use the interactive mode to complete the details. Consider the following definition, proved in 定理证明 (ép. 41):

```
plusReduces : (n:Nat) -> plus Z n = n
```

We'll be constructing the proof by *induction*, so we write the cases for `Z` and `S`, with a recursive call in the `S` case giving the inductive hypothesis, and insert *holes* for the rest of the definition:

```
plusReducesZ' : (n:Nat) -> n = plus n Z
plusReducesZ' Z     = ?plusredZ_Z
plusReducesZ' (S k) = let ih = plusReducesZ' k in
                      ?plusredZ_S
```

On running , two global names are created, `plusredZ_Z` and `plusredZ_S`, with no definition. We can use the `:m` command at the prompt to find out which holes are still to be solved (or, more precisely, which functions exist but have no definitions), then the `:t` command to see their types:

```
*theorems> :m
Global holes:
        [plusredZ_S,plusredZ_Z]

*theorems> :t plusredZ_Z
plusredZ_Z : Z = plus Z Z

*theorems> :t plusredZ_S
plusredZ_S : (k : Nat) -> (k = plus k Z) -> S k = plus (S k) Z
```

The `:p` command enters interactive proof mode, which can be used to complete the missing definitions.

```
*theorems> :p plusredZ_Z

---------------------------------- (plusredZ_Z) --------
{hole0} : Z = plus Z Z
```

This gives us a list of premises (above the line; there are none here) and the current goal (below the line; named `{hole0}` here). At the prompt we can enter tactics to direct the construction of the proof. In this case, we can normalise the goal with the `compute` tactic:

```
-plusredZ_Z> compute

---------------------------------- (plusredZ_Z) --------
{hole0} : Z = Z
```

Now we have to prove that `Z` equals `Z`, which is easy to prove by `Refl`. To apply a function, such as `Refl`, we use `refine` which introduces subgoals for each of the function's explicit arguments (`Refl` has none):

```
-plusredZ_Z> refine Refl
plusredZ_Z: no more goals
```

Here, we could also have used the `trivial` tactic, which tries to refine by `Refl`, and if that fails, tries to refine by each name in the local context. When a proof is complete, we use the `qed` tactic to add the proof to the global context, and remove the hole from the unsolved holes list. This also outputs a trace of the proof:

```
-plusredZ_Z> qed
plusredZ_Z = proof
    compute
    refine Refl
```

```
*theorems> :m
Global holes:
        [plusredZ_S]
```

The `:addproof` command, at the interactive prompt, will add the proof to the source file (effectively in an appendix). Let us now prove the other required lemma, `plusredZ_S`:

```
*theorems> :p plusredZ_S

--------------------------------- (plusredZ_S) --------
{hole0} : (k : Nat) -> (k = plus k Z) -> S k = plus (S k) Z
```

In this case, the goal is a function type, using `k` (the argument accessible by pattern matching) and `ih` — the local variable containing the result of the recursive call. We can introduce these as premises using the `intro` tactic twice (or `intros`, which introduces all arguments as premises). This gives:

```
  k : Nat
  ih : k = plus k Z
--------------------------------- (plusredZ_S) --------
{hole2} : S k = plus (S k) Z
```

Since plus is defined by recursion on its first argument, the term `plus (S k) Z` in the goal can be simplified, so we use `compute`.

```
  k : Nat
  ih : k = plus k Z
--------------------------------- (plusredZ_S) --------
{hole2} : S k = S (plus k Z)
```

We know, from the type of `ih`, that `k = plus k Z`, so we would like to use this knowledge to replace `plus k Z` in the goal with `k`. We can achieve this with the `rewrite` tactic:

```
-plusredZ_S> rewrite ih

  k : Nat
  ih : k = plus k Z
--------------------------------- (plusredZ_S) --------
{hole3} : S k = S k

-plusredZ_S>
```

The `rewrite` tactic takes an equality proof as an argument, and tries to rewrite the goal using that proof. Here, it results in an equality which is trivially provable:

```
-plusredZ_S> trivial
plusredZ_S: no more goals
-plusredZ_S> qed
plusredZ_S = proof {
    intros;
    rewrite ih;
    trivial;
}
```

Again, we can add this proof to the end of our source file using the `:addproof` command at the interactive prompt.

CHAPTER 6

Language Reference

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.

6.1 Code Generation Targets

`Idris` has been designed such that the compiler can generate code for different backends upon request. By default `Idris` generates a `C` backend when generating an executable. Included within the standard Idris installation are backends for Javascript and Node.js.

However, there are third-party code generators out there. Below we describe some of these backends and how you can use them when compiling your `Idris` code. If you want to write your own codegen for your language there is a stub project on GitHub that can help point you in the right direction.

6.1.1 Official Backends

C Language

Javascript

To generate code that is tailored for running in the browser issue the following command:

```
$ idris --codegen javascript hello.idr -o hello.js
```

Idris can produce very big chunks of JavaScript code (hello world weighs in at 1500 lines). However, the generated code can be minified using the closure-compiler from Google.

```
java -jar compiler.jar --compilation_level ADVANCED_OPTIMIZATIONS --js hello.js
```

Node.js

Generating code for NodeJS is slightly different. Idris outputs a JavaScript file that can be directly executed via node.

```
$ idris --codegen node hello.idr -o hello
$ ./hello
Hello world
```

6.1.2 Third Party

注解: These are third-party code generations and may have bit-rotted or do not work with current versions of Idris. Please speak to the project's maintainors if there are any problems.

CIL (.NET, Mono, Unity)

```
idris --codegen cil Main.idr -o HelloWorld.exe \
  && mono HelloWorld.exe
```

The resulting assemblies can also be used with .NET or Unity.

Requires idris-cil.

Erlang

Available online

Java

Available online

```
idris hello.idr --codegen java -o hello.jar
```

Note: The resulting .jar is automatically prefixed by a header including an .sh script to allow executing it directly.

JVM

Available online

LLVM

Available online

Malfunction

Available online

Ocaml

Available online

PHP

Available online

Python

Available online

Ruby

Available online

WS

Available online

WebAssembly

Available online

6.2 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).

6.2.1 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.

6.2.2 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.
|||
||| You can even provide examples which are inlined in the documentation:
||| ```idris example
||| add 4 5
||| ```
|||
||| 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
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
||| 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
```

6.3 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.

6.3.1 Package Descriptions

A package description includes the following:

- A header, consisting of the keyword package followed by the package name. Package names can be any valid Idris identifier. The iPKG format also takes a quoted version that accepts any valid filename.
- 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.

Metadata

From Idris *v0.12* the *iPKG* format supports additional metadata associated with the package. The added fields are:

- `brief = "<text>"`, a string literal containing a brief description of the package.
- `version = <text>`, a version string to associate with the package.
- `readme = <file>`, location of the README file.
- `license = <text>`, a string description of the licensing information.
- `author = <text>`, the author information.
- `maintainer = <text>`, Maintainer information.
- `homepage = <url>`, the website associated with the package.
- `sourceloc = <url>`, the location of the DVCS where the source can be found.
- `bugtracker = <url>`, the location of the project's bug tracker.

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. Executable names can be any valid Idris identifier. the iPKG format also takes a quoted version that accepts any valid filename.
- `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.

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. When building, Idris sets the environment variables `IDRIS_INCLUDES` (with C include flags) and `IDRIS_LDFLAGS` (with C linking flags) so they can be used from inside the `Makefile`.
- `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 files to be installed (object files, headers), perhaps generated by the `Makefile`.

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.

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

Comments

Package files support comments using the standard Idris singleline `--` and multiline `{- -}` format.

6.3.2 Using Package files

Given an Idris package file `test.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 --installdoc test.ipkg` will install the packages documentation into Idris' central documentation folder located at `idris --docdir`.
- `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.

When building or install packages the commandline flag `--warnipkg` will audit the project and warn of any potentiable problems.

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
```

6.4 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

6.4.1 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 where
     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 where
     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.

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
            pure ()
```

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
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
showU' (x :: xs) = show x ++ ", " ++ showU' xs
```

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.

6.5 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.

6.5.1 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 implementation 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).

6.5.2 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
public export
data C_IntTypes : Type -> Type where
    C_IntChar   : C_IntTypes Char
    C_IntNative : C_IntTypes Int
    C_IntBits8  : C_IntTypes Bits8
    C_IntBits16 : C_IntTypes Bits16
    C_IntBits32 : C_IntTypes Bits32
    C_IntBits64 : C_IntTypes Bits64

||| Supported C function types
public export
data C_FnTypes : Type -> Type where
    C_Fn : C_Types s -> C_FnTypes t -> C_FnTypes (s -> t)
    C_FnIO : C_Types t -> C_FnTypes (IO' FFI_C t)
    C_FnBase : C_Types t -> C_FnTypes t

||| Supported C foreign types
public export
data C_Types : Type -> Type where
    C_Str   : C_Types String
    C_Float : C_Types Double
    C_Ptr   : C_Types Ptr
    C_MPtr  : C_Types ManagedPtr
    C_Unit  : C_Types ()
    C_Any   : C_Types (Raw a)
    C_FnT   : C_FnTypes t -> C_Types (CFnPtr t)
    C_IntT  : C_IntTypes i -> C_Types i

||| A descriptor for the C FFI. See the constructors of `C_Types`
||| and `C_IntTypes` for the concrete types that are available.
%error_reverse
public export
FFI_C : FFI
    FFI_C = MkFFI C_Types String String
```

6.5.3 Linking foreign code

This is the example of linking C code.

Example Makefile

6.5.4 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
```

C callbacks

It is possible to pass an Idris function to a C function taking a function pointer by using `CFnPtr` in the function type. The Idris function is passed to `MkCFnPtr` in the arguments. The example below shows declaring the C standard library function `qsort` which takes a pointer to a comparison function.

```
myComparer : Ptr -> Ptr -> Int
myComparer = ...

qsort : Ptr -> Int -> Int -> IO ()
qsort data elems elsize = foreign FFI_C "qsort"
                (Ptr -> Int -> Int -> CFnPtr (Ptr -> Ptr -> Int) -> IO ())
                data elems elsize (MkCFnPtr myComparer)
```

There are a few limitations to callbacks in the C FFI. The foreign function can' t take the function to make a callback of as an argument. This will give a compilation error:

```
-- This does not work
example : (Int -> ()) -> IO ()
example f = foreign FFI_C "callbacker" (CFnPtr (Int -> ()) -> IO ()) f
```

Note that the function that is used as a callback can' t be a closure, that is it can' t be a partially applied function. This is because the mechanism used is unable to pass the closed-over values through C.

If we want to pass Idris values to the callback we have to pass them through C explicitly. Non-primitive Idris values can be passed to C via the `Raw` type.

The other big limitation is that it doesn't support IO functions. Use `unsafePerformIO` to wrap them (i.e. to make an IO function usable as a callback, change the return type from IOr to r, and change the = do to = unsafePerformIO $ do).

There are two special function names: `%wrapper` returns the function pointer that wraps an Idris function. This is useful if the function pointer isn't taken by a C function directly but should be inserted into a data structure. A foreign declaration using `%wrapper` must return `IO Ptr`.

```
-- this returns the C function pointer to a qsort comparer
example_wrapper : IO Ptr
example_wrapper = foreign FFI_C "%wrapper" (CFnPtr (Ptr -> Ptr -> Int) -> IO Ptr)
                          (MkCFnPtr myComparer)
```

`%dynamic` calls a C function pointer with some arguments. This is useful if a C function returns or data structure contains a C function pointer, for example structs of function pointers are common in object-oriented C such as in COM or the Linux kernel. The function type contains an extra `Ptr` at the start for the function pointer. `%dynamic` can be seen as a pseudo-function that calls the function in the first argument, passing the remaining arguments to it.

```
-- we have a pointer to a function with the signature int f(int), call it
example_dynamic : Ptr -> Int -> IO Int
example_dynamic fn x = foreign FFI_C "%dynamic" (Ptr -> Int -> IO Int) fn x
```

If the foreign name is prefixed by a `&`, it is treated as a pointer to the global variable with the following name. The type must be just `IO Ptr`.

```
-- access the global variable errno
errno : IO Ptr
errno = foreign FFI_C "&errno" (IO Ptr)
```

If the foreign name is prefixed by a `#`, the name is pasted in literally. This is useful to access constants that are preprocessor definitions (like `INT_MAX`).

```
%include C "limits.h"

-- access the preprocessor definition INT_MAX
intMax : IO Int
intMax = foreign FFI_C "#INT_MAX" (IO Int)

main : IO ()
main = print !intMax
```

For more complicated interactions with C (such as reading and setting fields of a C struct), there is a module CFFI available in the contrib package.

C heap

Idris has two heaps where objects can be allocated:

FP heap	C heap
Cheney-collected	Mark-and-sweep-collected
Garbage collections touches only live objects.	Garbage collection has to traverse all registered items.
Ideal for FP-style rapid allocation of lots of small short-lived pieces of memory, such as data constructors.	Ideal for C-style allocation of a few big buffers.
Finalizers are impossible to support reasonably.	Items have finalizers that are called on deallocation.
Data is copied all the time (when collecting garbage, modifying data, registering managed pointers, etc.)	Copying does not happen.
Contains objects of various types.	Contains C heap items: `(void *)` pointers with finalizers. A finalizer is a routine that deallocates the resources associated with the item.
Fixed set of object types.	The data pointer may point to anything, as long as the finalizer cleans up correctly.
Not suitable for C resources and arbitrary pointers.	Suitable for C resources and arbitrary pointers.
Values form a compact memory block.	Items are kept in a linked list.
Any Idris value, most notably `ManagedPtr`.	Items represented by the Idris type `CData`.
Data of `ManagedPtr` allocated in C, buffer then copied into the FP heap.	Data allocated in C, pointer copied into the C heap.
Allocation and reallocation not possible from C code (without having a reference to the VM). Everything is copied instead.	Allocated and reallocate freely in C, registering the allocated items in the FFI.

The FP heap is the primary heap. It may contain values of type `CData`, which are references to items in the C heap. A C heap item contains a `(void *)` pointer and the corresponding finalizer. Once a C heap item is no longer referenced from the FP heap, it is marked as unused and the next GC sweep will call its finalizer and deallocate it.

There is no Idris interface for `CData` other than its type and FFI.

Usage from C code

- Although not enforced in code, `CData` is meant to be opaque and non-RTS code (such as libraries or C bindings) should access only its `(void *)` field called `data`.
- Feel free to mutate both the pointer `data` (eg. after calling `realloc`) and the memory it points to. However, keep in mind that this must not break Idris' s referential transparency.
- **WARNING!** If you call `cdata_allocate` or `cdata_manage`, the resulting `CData` object *must* be returned from your FFI function so that it is inserted in the C heap by the RTS. Otherwise the memory will be leaked.

```
some_allocating_fun : Int -> IO CData
some_allocating_fun i = foreign FFI_C "some_allocating_fun" (Int -> IO CData) i

other_fun : CData -> Int -> IO Int
other_fun cd i = foreign FFI_C "other_fun" (CData -> Int -> IO Int) cd i
```

```
#include "idris_rts.h"

static void finalizer(void * data)
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
{
    MyStruct * ptr = (MyStruct *) data;
    free_something(ptr->something);
    free(ptr);
}

CData some_allocating_fun(int arg)
{
    size_t size = sizeof(...);
    void * data = (void *) malloc(size);
    // ...
    return cdata_manage(data, size, finalizer);
}

int other_fun(CData cd, int arg)
{
    int result = foo(cd->data);
    return result;
}
```

The `Raw` type constructor allows you to access or return a runtime representation of the value. For instance, if you want to copy a string generated from C code into an Idris value, you may want to return a `Raw String``instead of a ``String` and use `MKSTR` or `MKSTRlen` to copy it over.

```
getString : () -> IO (Raw String)
getString () = foreign FFI_C "get_string" (IO (Raw String))

const VAL get_string ()
{
    char * c_string = get_string_allocated_with_malloc()
    const VAL idris_string = MKSTR(c_string);
    free(c_string);
    return idris_string
}
```

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))
```

6.5.5 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.

注解: 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.

6.5.6 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 Double
       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
  pure $ MkTimeout timeout
```

6.6 Syntax Guide

Examples are mostly adapted from the Idris tutorial.

6.6.1 Source File Structure

Source files consist of:

1. An optional *Module Header* (ęąţ 165).
2. Zero or more *Imports* (ęąţ 165).
3. Zero or more declarations, e.g. *Variables* (ęąţ 165), *Data types* (ęąţ 166), etc.

For example:

```
module MyModule   -- module header

import Data.Vect  -- an import

%default total    -- a directive

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

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* (éąţ 170) can be used to provide documentation for a module:

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

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* (éąţ 170). (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
```

6.6.2 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"
```

6.6.3 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.

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
```

The signature of type constructors may use dependent types

```
data DPair : (a : Type) -> (a -> Type) -> Type where
  MkDPair : {P : a -> Type} -> (x : a) -> (pf : P x) -> DPair a P
```

Records

There is a special syntax for data types with one constructor and multiple fields.

```
record A a where
  constructor MkA
  foo, bar : a
  baz : Nat
```

This defines a constructor as well as getter and setter function for each field.

```
MkA : a -> a -> Nat -> A a
foo : A a -> a
set_foo : a -> A a -> A a
```

The types of record fields may depend on the value of other fields

```
record Collection a where
  constructor MkCollection
  size : Nat
  items : Vect size a
```

Setter functions are only provided for fields that do not use dependant types. In the example above neither `set_size` nor `set_items` are defined.

Co-data

Inifinite data structures can be introduced with the `codata` keyword.

```
codata Stream : Type -> Type where
  (::) a -> Stream a -> Stream a
```

This is syntactic sugar for the following, which is usually preferred:

```
data Stream : Type -> Type where
  (::) a -> Inf (Stream a) -> Stream a
```

Every occurence of the defined type in a constructor argument will be wrapped in the `Inf` type constructor. This has the effect of delaying the evaluation of the second argument when the data constructor is applied. An `Inf` argument is constructed using `Delay` (which Idris will insert implicitly) and evaluated using `Force` (again inserted implicitly).

Furthermore, recursive calls under a `Delay` must be guarded by a constructor to pass the totality checker.

6.6.4 Operators

Arithmetic

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

Equality and Relational

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

Conditional

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

6.6.5 Conditionals

If Then Else

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

Case Expressions

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

6.6.6 Functions

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"
```

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>)
```

Modifiers

Visibility

```
public export
export
private
```

Totality

```
total
partial
covering
```

Sets explicitly to which extent pattern matching is terminating and/or exhaustive. A `partial` pattern matching makes no assumption. A `covering` pattern matching ensures that pattern matching is exhaustive on its clauses. Furthermore, a `total` pattern matching enforces both exhaustivity and termination of the evaluation of its clauses.

Implicit Coercion

```
implicit
```

Options

```
%export
%hint
%no_implicit
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
%error_handler
%error_reverse
%reflection
%specialise [<name list>]
```

6.6.7 Misc

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* (éąţ 165) 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`.

Comments

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

Multi line String literals

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

6.6.8 Directives

```
%lib <path>
%link <path>
%flag <path>
%include <path>
%hide <function>
%freeze <name>
%access <accessibility>
%default <totality>
%logging <level 0--11>
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
%dynamic <list of libs>
%name <list of names>
%error_handlers <list of names>
%language <extension>
```

6.7 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.

6.7.1 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.

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 an Idris programmer would write.

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   y2    (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.

6.7.2 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 interface implementations)

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.

6.7.3 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.

6.7.4 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.

6.7.5 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).

6.7.6 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.

6.7.7 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

Interface 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.

6.7.8 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.

6.7.9 Troubleshooting

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.

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* (éąţ 176).

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.

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
```

6.7.10 How to read and resolve erasure warnings

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.

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)
```

6.7.11 References

6.8 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.

6.8.1 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 [LINE])` Load the named file. If a `LINE` number is provided, the file is only loaded up to that line. Otherwise, the entire file is loaded.

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

`(: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 func-

tion `NAME` on line `LINE`. The new code is returned with no highlighting.

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

`(:make-lemma LINE NAME)` Create a top level function with a type which solves the hole named `NAME` on line `LINE`.

`(: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 [MODE])` Look up the documentation for `NAME`, and return it as a highlighted string. If `MODE` is `:overview`, only the first paragraph of documentation is provided for `NAME`. If `MODE` is `:full`, or omitted, the full documentation is returned for `NAME`.

`(:apropos STRING)` Search the documentation for mentions of `STRING`, and return any found as a list of highlighted strings.

`(: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 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.

`(:repl-completions NAME)` Search names, types and documentations which contain `NAME`. Return the result of tab-completing `NAME` as a REPL command.

`:version` Return the version information of the Idris compiler.

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)
```

6.8.2 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.

6.8.3 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.

6.9 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*.

6.9.1 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	

6.9.2 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>`

6.9.3 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.

6.9.4 Further Information

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

6.10 DEPRECATED: Tactics and Theorem Proving

警告: The interactive theorem-proving interface documented here has been deprecated in favor of *Elaborator Reflection* (éąţ 199).

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:

6.10.1 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)

6.10.2 Commonly Used Tactics

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>
```

Exact

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

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

Refine

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

Trivial

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

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

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
```

Intros

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

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

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

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

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>
```

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>
```

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.

6.10.3 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 sec-

ond 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` - ?

6.11 The Idris REPL

Idris comes with a `REPL`.

6.11.1 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
```

6.11.2 Customisation

Idris supports initialisation scripts.

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.

Example initialisation script

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

6.11.3 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 interpre
: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, autosolv
: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 spec
:q :quit		Exit the Idris system

èąſ 1 – 续上页

Command	Arguments	Purpose
:w :warranty		Displays warranty information
:let	(<top-level-declaration>)···	Evaluate a declaration, such as a function definition, instance imp
:unlet :undefine	(<name>)···	Remove the listed repl definitions, or all repl definitions if no nam
: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

6.11.4 Using the REPL

Getting help

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

Quitting Idris

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

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
```

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
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
Idris> :let y = 10
Idris> y
10 : Integer
Idris> :let data Foo : Type where Bar : Foo
Idris> Bar
Bar : Foo
```

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 interfaces with `:doc`:

```
Idris> :doc Monad
Interface 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.Interfaces.(+) : Num ty => ty -> ty -> ty

infixl 8

The function is Total


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

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
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
Interface Monad

Parameters:
    m

Methods:
    (>>=) : Monad m => m a -> (a -> m b) -> m b
        Also called bind.
        infixl 5

        The function is Total
    join : Monad m => m (m a) -> m a
        Also called flatten or mu

        The function is Total
Implementations:
    Monad (IO' ffi)
    Monad Stream
    Monad Provider
    Monad Elab
    Monad PrimIO
    Monad Maybe
    Monad (Either e)
    Monad List
```

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')) -> _|_
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
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 : Double -> Double -> 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 interface 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
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
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.Interfaces.(*) : Num a => a -> a -> a


> Prelude.Interfaces.(+) : 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
```

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.

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.

Editing files

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

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.

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.

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* (ęąţ **??**) and `:exec`.

6.11.5 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.

6.12 Compilation, Logging, and Reporting

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

6.12.1 Compilation Process

Idris follows the following compilation process:

1. Parsing
2. Type Checking
    1. Elaboration
    2. Coverage
    3. Unification
    4. Totality Checking
    5. Erasure
3. Code Generation
    1. Defunctionalisation
    2. Inlining
    3. Resolving variables
    4. Code Generation

6.12.2 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.

6.12.3 Reporting Compilation Process

During compilation the reporting of Idris' progress can be controlled by setting a verbosity level.

- `-V`, or alternatively `--verbose` and `--V0`, will report which file Idris is currently type checking.
- `--V1` will additionally report: Parsing, IBC Generation, and Code Generation.
- `--V2` will additionally report: Totality Checking, Universe Checking, and the individual steps prior to code generation.

By default Idris' progress reporting is set to quiet-`--q`, or `--quiet`.

6.12.4 Logging Internal Operation

For those that develop on the Idris compiler, the internal operation of Idris is captured using a category based logger. Currently, the logging infrastructure has support for the following categories:

- Parser (`parser`)
- Elaborator (`elab`)
- Code generation (`codegen`)
- Erasure (`erasure`)
- Coverage Checking (`coverage`)
- IBC generation (`ibc`)

These categories are specified using the command-line option: `--logging-categories CATS`, where `CATS` is a quoted colon separated 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: Interface, Clauses, Data, Term, and Types,
- Level 3: Provides details of compilation of the IRTS, erasure, parsing, case splitting, and further details elaboration of: Implementations, 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.

6.12.5 Environment Variables

Several paths set by default within the Idris compiler can be overridden through environment variables. The provided variables are:

- *IDRIS_CC* Change the *C* compiler used by the *C* backend.
- *IDRIS_CFLAGS* Change the *C* flags passed to the *C* compiler.
- *TARGET* Change the target directory i.e. *data dir* where Idris installs files when installing using Cabal/Stack.
- *IDRIS_LIBRARY_PATH* Change the location of where installed packages are found/installed.
- *IDRIS_DOC_PATH* Change the location of where generated idrisdoc for packages are installed.

注解: In versions of Idris prior to 0.12.3 the environment variables *IDRIS_LIBRARY_PATH* and *TARGET* were both used to affect the installation of single packages and direct where Idris installed its data. The meaning of these variables has changed, and command line options are preferred when changing where individual packages are installed.

The CLI option *–ibcsubdir* can be used to direct where generated IBC files are placed. However, this means Idris will install files in a non-standard location separate from the rest of the installed packages. The CLI option *–idrispath <dir>* allows you to add a directory to the library search path; this option can be used multiple times and can be shortened to *-i <dir>*. Similary, the *–sourcepath <dir>* option can be used to add directories to the source search path. There is no shortened version for this option as *-s* is a reserved flag.

Further, Idris also supports options to augment the paths used, and pass options to the code generator backend. The option *–cg-opt <ARG>* can be used to pass options to the code generator. The format of *<ARG>* is dependent on the selected backend.

6.13 Idris' Internals

Note: this is still a fairly raw set of notes taken by David Christiansen at Edwin's presentation at the 2013 Idris Developers Meeting. They're in the process of turning into a useful guide - feel free to contribute.

This document assumes that you are already familiar with Idris. It is intended for those who want to work on the internals.

People looking to develop new back ends may want to look at [[Idris back end IRs|Idris-back-end-IRs]]

6.13.1 Core/TT.hs

Idris is compiled to a simple, explicit core language. This core language is called TT because it looks a bit like a Π. It's a minimal language, with a locally nameless representation. That is, local variables are represented with de Bruijn indices and globally-defined constants are represented with names.

The `TT` datatype uses a trick that is common in the Idris code: it is polymorphic over the type of names stored in it, and it derives `Functor`. This allows `fmap` to be used as a general-purpose traversal.

There is a general construction for binders, used for λ, Π, and let-bindings. These are distinguished using a `BinderType`.

During compilation, some terms (especially types) will be erased. This is represented using the `Erased` constructor of `TT`. A handy trick when generating TT terms is to insert `Erased` where a term is uniquely determined, as the typechecker will fill it out.

The constructor `Proj` is a result of the optimizer. It is used to extract a specific constructor argument, in a more economical way than defining a new pattern-matching operation.

The datatype `Raw` represents terms that have not yet been typechecked. The typechecker converts a `Raw` to a `TT` if it can.

6.13.2 Core/CaseTree.hs

Case trees are used to represent top-level pattern-matching definitions in the TT language.

Just as with the `TT` datatype, the `deriving Functor` trick is used with `SC` and `CaseAlt` to get GHC to generate a function for mapping over contained terms.

Constructor cases (`ConCase` in `CaseAlt`) refer to numbered constructors. Every constructor is numbered 0,1,2,⋯. At this stage in the compiler, the tags are datatype-local. After defunctionalization, however, they are made globally unique.

The `n+1` patterns (`SucCase`) and hacky-seeming things are to make code fast – please ask before "cleaning up" the representation.

6.13.3 Core/Evaluate.hs

This module contains the main evaluator for Idris. The evaluator is used both at the REPL and during type checking, where normalised terms need to be compared for equality.

A key datatype in the evaluator is a *context*. Contexts are mappings from global names to their values, but they are organized to make type-directed disambiguation quick. In particular, the main part of a name that a user might type is used as the key, and its values are maps from namespaces to actual values.

The datatype `Def` represents a definition in the global context. All global names map to this structure.

`Type` and `Term` are both synonyms for `TT`.

Datatypes are represented by a `TyDecl` with the appropriate `NameType`. A `Function` is a global constant term with an annotated type, `Operator` represents primitives implemented in Haskell, and `CaseOp` represents ordinary pattern-matching definitions. `CaseOp` has four versions for different purposes, and all are saved because that's easiest.

`CaseInfo`: the `tc_dictionary` is because it's a type class dictionary which makes totality checking easier.

The `normalise*` functions give different behaviors - but `normalise` is the most common.

`normaliseC` - "resolved" means with names converted to de Bruijn indices as appropriate.

`normaliseAll` - reduce everything, even if it's non-total

`normaliseTrace` - special-purpose for debugging

`simplify` - reduce the things that are small - the list argument is the things to not reduce.

6.13.4 Core/Typecheck.hs

Standard stuff. Hopefully no changes are necessary.

6.13.5 Core/Elaborate.hs

Idris definitions are elaborated one by one and turned into the corresponding TT. This is done with a tactic language as an EDSL in the Elab monad (or Elab' when there's a custom state).

Lots of plumbing for errors.

All elaboration is relative to a global context.

The string in the pair returned by elaborate is log information.

See JFP paper, but the names don't necessarily map to each other. The paper is the "idealized version" without logging, additional state, etc.

All the tactics take Raws, typechecking happens there.

claim (x : t) assumes a new x : t.

PLEASE TIDY THINGS UP!

proofSearch flag to try' is whether the failure came from a human (so fail) or from a machine (so continue)

Idris-level syntax for providing alternatives explicitly: (| x, y, z |) try x, y, z in order, and take the first that succeeds.

6.13.6 Core/ProofState.hs

6.13.7 Core/Unify.hs

Deals with unification. Unification can reply with: - this works - this can never work - this will work if these other unification problems work out (eg unifying f x with 1)

match_unify: same thing as unification except it's just matching name against name, term against term. x + y matches to 0 + y with x = 0. Used for <== syntax as well as type class resolution.

6.13.8 Idris/AbsSyntaxTree.hs

PTerm is the datatype of Idris syntax. P is for Program. Each PTerm turns into a TT term by applying a series of tactics.

IState is the major interpreter state. The global context is the tt_ctxt field.

Ctxt maps possibly ambiguous names to their referents.

6.13.9 Idris/ElabDecls.hs

This is where the actual elaboration from PTerm to TT happens.

6.13.10 Idris/ElabTerm.hs

build is the function that creates a Raw. All the "junk" is to deal with things like metavars and so forth. It has to remember what names are still to be defined, and it doesn't yet know the type (filled in by unificaiton later). Also case expressions have to turn into top-level functions.

resolveTC is type class resolution.

6.14 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`

6.15 Language Extensions

6.15.1 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.

6.16 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.

6.16.1 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`.

6.16.2 Running Elaborator Scripts

On their own, tactics have no effect. The meta-operation `%runElab script` runs `script` in the current elaboration context. Before you can use `%runElab`, you will have to enable the language extension by adding `%language ElabReflection` in your file (or by passing `-X ElabReflection` to the `idris` executable from your command line). For example, the following script constructs the identity function at type `Nat`:

```
idNat : Nat -> Nat
idNat = %runElab (do intro `{{x}}
                     fill (Var `{{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 => {hole≈x}`. 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 `{{x}}
          fill (Var `{{x}})
          solve

idNat : Nat -> Nat
idNat = %runElab mkId

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

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

6.16.3 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. It also supports the `do`-syntax, meaning you can write `res <- command` to bind the result of `command` to variable `res`.

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.

6.16.4 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` implementation 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.

6.16.5 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.

6.16.6 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 interface deriving. The operations `declareType`, `defineFunction`, and `addImplementation` allow `Elab` scripts to modify the global context.

6.16.7 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 interface resolution, `search` invokes the proof search mechanism, and `sourceLocation` finds the context in the original file at which the elaboration script is invoked.

6.16.8 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.

6.16.9 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.

6.17 Type Directed Search `:search`

Idris' `:search` command searches for terms according to their approximate type signature (much like Hoogle for Haskell). For example:

```
Idris> :search e -> List e -> List e
= Prelude.List.(::) : a -> List a -> List a
Cons cell

= Prelude.List.intersperse : a -> List a -> List a
Insert some separator between the elements of a list.

> Prelude.List.delete : Eq a => a -> List a -> List a


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

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

The best results are listed first. As we can see, `(::)` and `intersperse` are exact matches; the `=` symbol to the left of those results tells us the types of `(::)` and `intersperse` are effectively the same as the type that was searched.

The next result is `delete`, whose type is more specific than the type that was searched; that's indicated by the `>` symbol. If we had a function with the signature `e -> List e -> List e`, we could have given it the type `Eq a => a -> List a -> List a`, but not necessarily the other way around.

The final two results, `assert_smaller` and `const`, have types more general than the type that was searched, and so they have `<` symbols to their left. For example, `e -> List e -> List e` would be a valid type for `assert_smaller`. The correspondence for `const` is more complicated than any of the four previous results. `:search` shows this result because we could change the order of the arguments! That is, the following definition would be legal:

```
f : e -> List e -> List e
f x xs = const xs x
```

6.17.1 About :search results

:search's functionality is based on the notion of type isomorphism. Informally, two types are isomorphic if we can identify terms of one type exactly with terms of the other. For example, we can consider the types `Nat -> a -> List a` and `a -> Nat -> List a` to be isomorphic, because if we have `f : Nat -> a -> List a`, then `flip f : a -> Nat -> List a`. Similarly, if `g : a -> Nat -> List a`, then `flip g : Nat -> a -> List a`.

With :search, we create a partial order on types; that is, given two types `A` and `B`, we may choose to say

that `A <= B`, `A >= B`, or both (in which case we say `A == B`), or neither. For :search, we say that `A >= B` if all of the terms inhabiting `A` correspond to terms of `B`, but it need not necessarily be the case that *all* the terms of `B` correspond to terms of `A`. Here' s an example:

```
a -> a            >=          Nat -> Nat
```

The left-hand type has just a single inhabitant, `id`, which corresponds to the term `id {a = Nat}`, which has the right-hand type. However, there are various terms inhabiting the right-hand type (such as `S`) which cannot correspond with terms of type `a -> a`.

We can consider the partial order for `:search` to be, in some sense, inductively generated by several classes of "edits" which are described below.

Possible edits

Here is a simple approximate list of the edits that are possible in `:search`. They are not entirely formal, and do not necessarily reflect the `:search` command' s actual behavior. For example, the *argument application* rule may be used directly on arguments that are bound after other arguments, without using several applications of the *argument transposition* rule.

- **Argument transposition**

  ```
     A : Type          B : Type           a : A, b : B  |-  M : Type
  ------------------------------------------------------------------------
  (x : A) -> (y : B) -> [x,y/a,b]M      ==    (y : B) -> (x : A) -> [x,y/a,b]M
  ```

  Score: 1 point

Example:

```
a -> Vect n a -> Vect (S n) a    ==    Vect n a -> a -> Vect (S n) a
```

Note that in order for it to make sense to change the order of arguments, neither of the arguments' types may depend on the value bound by the other argument!

- **Symmetry of equality**

  ```
  A = B : Type      t : Type  |-  M : Type
  ---------------------------------------
   [A = B/t]M        ==       [B = A/t]M
  ```

  Score: 1 point

Example:

```
(x,y,z : Nat) -> x + (y + z) = (x + y) + z
                 ==
(x,y,z : Nat) -> (x + y) + z = x + (y + z)
```

Note that this rule means that we can flip equalities anywhere they occur (i.e., not only in the return type).

- **Argument application**

  ```
    e : A  |-  M : Type        y1 : T1, ..., yn : Tn  |-  x : A
  ------------------------------------------------------------------
  (z : A) -> [z/e]M     >=   (y1 : T1) -> ... -> (yn :Tn) -> [x/e]M
  ```

  Score: <= : 3 points, >= : 9 points

Examples:

```
a -> a            >=    Nat -> Nat
a -> a            >=    List e -> List e
Vect k (Fin k)    >=    Vect 5 (Fin 5)
```

Note that the `n` shown in the scheme above may be 0; that is, there are no Pi terms to be added on the right side. For example, that' s the case for the first example shown above. This is probably the most important, and most widely used, rule of all.

- **Type class application**

  ```
  C : Type -> TypeClass
  ,   y1 : T1, ..., yn : Tn  |-  A : Type, instance : C A
  ,   t : Type  |- M : Type
  ------------------------------------------------------------------
  C a => [a/t]M     >=      (y1 : T1) -> ... -> (yn : Tn) -> [A/t]M
  ```

  Score: <= : 4 points, >= : 12 points

Examples

```
Ord a => a                          >=    Int
Show (List e) => List e -> String   >=    Show a => List a -> String
```

This rule is used by looking at the instances for a particular type class. While the scheme is shown only for single-parameter type classes, it naturally generalizes to multi-parameter type classes. This rule is particularly useful in conjunction with argument application. Again, note that the `n` in the scheme above may be 0.

- **Type class introduction**

  ```
  t : Type |- M : Type      C : Type -> TypeClass
  ------------------------------------------------
  (t : Type) -> M      >=      C t => M
  ```

  Score: <= : 2 points, >= : 6 points

Example:

```
a -> a -> Bool    >=    Eq a => a -> a -> Bool
```

Scoring and listing search results

When a type `S` is searched, the type is compared against the types of all of the terms which are currently in context. When :search compares two types `S` and `T`, it essentially tries to find a chain of inequalities

```
   R1    R2      Rn    Rn+1
S  <= A1 <= ... <= An <= T
```

using the edit rules listed above. It also tries to find chains going the other way (i.e., showing `S >= T`) as well. Each rule has an associated score which indicates how drastic of a change the rule corresponds to. These scores are listed above. Note that for the rules which are not symmetric, the score depends on the direction in which the rule is used. Finding types which are more general that the searched typed (`S <= T`) is preferred to finding types which are less general.

The score for the entire chain is, at minimum, the sum of the scores of the individual rules (some non-linear interactions may be added). The :search function tries to find the chain between `S` and `T` which

results in the lowest score, and this is the score associated to the search result for `T`.

Search results are listed in order of ascending score. The symbol which is shown along with the search result reflects the type of the chain which resulted in the minimum score.

6.17.2 Implementation of :search

Practically, naive and undirected application of the rules enumerated above is not possible; not only is this obviously inefficient, but the two application rules (particularly *argument application*) are really impossible to use without context given by other types. Therefore, we use a heuristic algorithm that is meant to be practical, though it might not find ways to relate two types which may actually be related by the rules listed above.

Suppose we wish to match two types, `S` and `T`. We think of the problem as a non-deterministic state machine. There is a `State` datatype which keeps track of how well we' ve matched `S` and `T` so far. It contains:

- Names of argument variables (Pi-bound variables) in either type which have yet to be matched
- A directed acyclic graph (DAG) of arguments (Pi-bindings) for `S` and `T` which have yet to be matched
- A list of typeclass constraints for `S` and `T` which have yet to be matched
- A record of the rules which have been used so far to get to this point

A function `nextSteps : State -> [State]` finds the next states which may follow from a given state. Some states, where everything has been matched, are considered final. The algorithm can be roughly broken down into multiple stages; if we start from having two types, `S` and `T`, which we wish to match, they are as follows:

1. For each of `S` and `T`, split the types up into their return types and directed acyclic graphs of the arguments, where there is an edge from argument A to argument B if the term bound in A appears in the type of B. The topological sorts of the DAG represent all the possible ways in which the arguments may be permuted.

2. For type `T`, recursively find (saturated) uses of the `=` type constructor and produce a list of modified versions of `T` containing all possible flips of the `=` constructor (this corresponds to the *symmetry of equality rule*).

3. For each modified type for `T`, try to unify the return type of the modified `T` with `S`, considering arguments from both `S` and `T` to be holes, so that the unifier may match pieces of the two types. For each modified version of `T` where this succeeds, an initial `State` can be made. The arguments and typeclasses are updated accordingly with the results of unification. The remainder of the algorithm involves applying `nextSteps` to these states until either no states remain (corresponding to no path from `S` to `T`) or a final state is found. `nextSteps` also has several stages:

4. Try to unify arguments of `S` with arguments of `T`, much like is done with the return types. We work "backwards" through the arguments: we try matching all remaining arguments of `S` which lack outgoing edges in the DAG of remaining arguments (that is, the bound value doesn' t appear in the type of any other remaining arguments) with the all of the corresponding remaining arguments of `T`. This is done recursively until no arguments remain for both `S` and `T`; otherwise, we give up at this point. This step corresponds to application of the *argument application rule*, as well as the *argument transposition* rule.

5. Now, we try to match the type classes. First, we take all possible subsets of type class constraints for `S` and `T`. So if `S` and `T` have a total of `n` type class constraints, this produces `2^n` states for every state, and this quickly becomes infeasible as `n` grows large. This is probably the biggest bottleneck of the algorithm at the moment. This step corresponds to applications of the *type class*

introduction rule.

6. Try to match type class constraints for `S` with those for `T`. We attempt to unify each type class constraint for `S` with each constraint for `T`. This may result in applications of the *type class application* rule. Once we are unable to match any more type class constraints between `S` and `T`, we proceed to the final step.

7. Try instantiating type classes with their instances (in either `S` or `T`). This corresponds to applications of the *type class application* rule. After instantiating a type class, we hopefully open up more opportunities to match typeclass constraints of `S` with those of `T`, so we return to the previous step.

The code for :search is located in the Idris.TypeSearch module.

Aggregating results

The search for chains of rules/edits which relate two types can be viewed as a shortest path problem where nodes correspond to types and edges correspond to rules relating two types. The weights or distances on each edge correspond to the score of each rule. We then may imagine that we have a single start node, our search type `S`, and several final nodes: all of the types for terms which are currently in context. The problem, then, is to find the shortest paths (where they exist) to all of the final nodes. In particular, we wish to find the "closest" types (those with the minimum score) first, as we'd like to display them first.

This problem nicely maps to usage of Dijkstra's algorithm. We search for all types simultaneously so we can find the closest ones with the minimum amount of work. In practice, this results in using a priority queue of priority queues. We first ask "which goal type should we work on next?", and then ask "which state should we expand upon next?" By using this strategy, the best results can be shown quickly, even if it takes a bit of time to find worse results (or at least rule them out).

6.17.3 Miscellaneous Notes

Whether arguments are explicit or implicit does not affect search results.

6.18 Static Arguments and Partial Evaluation

As of version 0.9.15, Idris has support for *partial evaluation* of statically known arguments. This involves creating specialised versions of functions with arguments annotated as `%static`.

(This is an implementation of the partial evaluator described in this ICFP 2010 paper. Please refer to this for more precise definitions of what follows.)

Partial evaluation is switched off by default since Idris 1.0. It can be enabled with the `--partial-eval` flag.

6.18.1 Introductory Example

Consider the power function over natural numbers, defined as follows (we'll call it `my_pow` since `pow` already exists in the Prelude):

```
my_pow : Nat -> Nat -> Nat
my_pow x Z = 1
my_pow x (S k) = mult x (my_pow x k)
```

This is implemented by recursion on the second argument, and we can evaluate the definition further if the second argument is known, even if the first isn't. For example, we can build a function at the REPL to cube a number as follows:

```
*pow> \x => my_pow x 3
\x => mult x (mult x (mult x 1)) : Nat -> Nat
*pow> it 3
27 : Nat
```

Note that in the resulting function the recursion has been eliminated, since `my_pow` is implemented by recursion on the known argument. We have no such luck if the first argument is known and the second isn't:

```
*pow> \x => my_pow 2 x
\x => my_pow 2 x : Nat -> Nat
```

Now, consider the following definition which calculates x^2 + 1:

```
powFn : Nat -> Nat
powFn x = plus (my_pow x (S (S Z))) (S Z)
```

Since the second argument to `my_pow` here is statically known, it seems a shame to have to make the recursive calls every time. However, Idris will not in general inline recursive definitions, in particular since they may diverge or duplicate work without some deeper analysis.

We can, however, give Idris some hints that here we really would like to create a specialised version of `my_pow`.

Automatic specialisation of `pow`

The trick is to mark the statically known arguments with the `%static` flag:

```
my_pow : Nat -> %static Nat -> Nat
my_pow k Z = 1
my_pow k (S j) = mult k (my_pow k j)
```

When an argument is annotated in this way, Idris will try to create a specialised version whenever it accounts a call with a concrete value (i.e. a constant, constructor form, or globally defined function) in a `%static` position. If `my_pow` is defined this way, and `powFn` defined as above, we can see the effect by typing `:printdef powFn` at the REPL:

```
*pow> :printdef powFn
powFn : Nat -> Nat
powFn x = plus (PE_my_pow_3f3e5ad8 x) 1
```

What is this mysterious `PE_my_pow_3f3e5ad8`? It's a specialised power function where the statically known argument has been specialised away. The name is generated from a hash of the specialised arguments, and we can see its definition with `:printdef` too:

```
*petest> :printdef PE_my_pow_3f3e5ad8
PE_my_pow_3f3e5ad8 : Nat -> Nat
PE_my_pow_3f3e5ad8 (0arg) = mult (0arg) (mult (0arg) (PE_fromInteger_7ba9767f 1))
```

The (0arg) is an internal argument name (programmers can't give variable names beginning with a digit after all). Notice also that there is a specialised version of fromInteger for Nats, since type class dictionaries are themselves a particularly common case of statically known arguments!

6.18.2 Specialising Type Classes

Type class dictionaries are very often statically known, so Idris automatically marks any type class constraint as %static and builds specialised versions of top level functions where the class is instantiated. For example, given:

```
calc : Int -> Int
calc x = (x * x) + x
```

If we print this definition, we'll see a specialised version of + is used:

```
*petest> :printdef calc
calc : Int -> Int
calc x = PE_+_954510b4 (PE_*_954510b4 x x) x
```

More interestingly, consider vadd which adds corresponding elements in a vector of anything numeric:

```
vadd : Num a => Vect n a -> Vect n a -> Vect n a
vadd [] [] = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

If we use this on something concrete as follows···

```
test : List Int -> List Int
test xs = let xs' = fromList xs in
              toList $ vadd xs' xs'
```

···then in fact, we get a specialised version of vadd in the definition of test, and indeed the specialised version of toList:

```
test : List Int -> List Int
test xs = let xs' = fromList xs
          in PE_toList_888ae67 (PE_vadd_33f98d3d xs' xs')
```

Here's the specialised version of vadd:

```
PE_vadd_33f98d3d : Vect n Int -> Vect n Int -> Vect n Int
PE_vadd_33f98d3d [] [] = []
PE_vadd_33f98d3d (x :: xs) (y :: ys) = ((PE_+_954510b4 x y) ::
                                        (PE_vadd_33f98d3d xs ys))
```

Note that the recursive structure has been preserved, and the recursive call to vadd has been replaced with a recursive call to the specialised version. We've also got the same specialised version of + that we had above in calc.

6.18.3 Specialising Higher Order Functions

Another case where partial evaluation can be useful is in automatically making specialised versions of higher order functions. Unlike type class dictionaries, this is not done automatically, but we might consider writing map as follows:

```
my_map : %static (a -> b) -> List a -> List b
my_map f [] = []
my_map f (x :: xs) = f x :: my_map f xs
```

Then using `my_map` will yield specialised versions, for example to double every value in a list of `Int`s we could write:

```
doubleAll : List Int -> List Int
doubleAll xs = my_map (*2) xs
```

This would yield a specialised version of `my_map`, used in `doubleAll` as follows:

```
doubleAll : List Int -> List Int
doubleAll xs = PE_my_map_1f8225c4 xs

PE_my_map_1f8225c4 : List Int -> List Int
PE_my_map_1f8225c4 [] = []
PE_my_map_1f8225c4 (x :: xs) = ((PE_*_954510b4 x 2) :: (PE_my_map_1f8225c4 xs))
```

6.18.4 Specialising Interpreters

A particularly useful situation where partial evaluation becomes effective is in defining an interpreter for a well-typed expression language, defined as follows (see the Idris tutorial, section 4 for more details on how this works):

```
data Expr : Vect n Ty -> Ty -> Type where
     Var : HasType i gamma t -> Expr gamma t
     Val : (x : Int) -> Expr gamma TyInt
     Lam : Expr (a :: gamma) t -> Expr gamma (TyFun a t)
     App : Lazy (Expr gamma (TyFun a t)) -> Expr gamma a -> Expr gamma t
     Op  : (interpTy a -> interpTy b -> interpTy c) -> Expr gamma a -> Expr gamma
           Expr gamma c
     If  : Expr gamma TyBool -> Expr gamma a -> Expr gamma a -> Expr gamma a

dsl expr
    lambda = Lam
    variable = Var
    index_first = stop
    index_next = pop
```

We can write a couple of test functions in this language as follows, using the `dsl` notation to overload lambdas; first a function which multiplies two inputs:

```
eMult : Expr gamma (TyFun TyInt (TyFun TyInt TyInt))
eMult = expr (\x, y => Op (*) x y)
```

Then, a function which calculates the factorial of its input:

```
eFac : Expr gamma (TyFun TyInt TyInt)
eFac = expr (\x => If (Op (==) x (Val 0))
            (Val 1)
            (App (App eMult (App eFac (Op (-) x (Val 1)))) x))
```

The interpreter's type is written as follows, marking the expression to be evaluated as `%static`:

```
interp : (env : Env gamma) -> %static (e : Expr gamma t) -> interpTy t
```

This means that if we write an Idris program to calculate a factorial by calling `interp` on `eFac`, the

resulting definition will be specialised, partially evaluating away the interpreter:

```
runFac : Int -> Int
runFac x = interp [] eFac x
```

We can see that the call to `interp` has been partially evaluated away as follows:

```
*interp> :printdef runFac
runFac : Int -> Int
runFac x = PE_interp_ed1429e [] x
```

If we look at `PE_interp_ed1429e` we' ll see that it follows exactly the structur of `eFac`, with the interpreter evaluated away:

```
*interp> :printdef PE_interp_ed1429e
PE_interp_ed1429e : Env gamma -> Int -> Int
PE_interp_ed1429e (3arg) = \x =>
                             boolElim (x == 0)
                                      (Delay 1)
                                      (Delay (PE_interp_b5c2d0ff (x :: (3arg))
                                                                 (PE_interp_ed1429e (x ::␣
↪(3arg)) (x - 1)) x))
```

For the sake of readability, I have simplified this slightly: what you will really see also includes specialised versions of `==`, `-` and `fromInteger`. Note that `PE_interp_ed1429e`, which represents `eFac` has become a recursive function following the structure of `eFac`. There is also a call to `PE_interp_b5c2d0ff` which is a specialised interpeter for `eMult`.

These definitions arise because the partial evaluator will only specialise a definition by a specific concrete argument once, then it is cached for future use. So any future applications of `interp` on `eFac` will also be translated to `PE_interp_ed1429e`.

The specialised version of `eMult`, without any simplification for readability, is:

```
PE_interp_b5c2d0ff : Env gamma -> Int -> Int -> Int
PE_interp_b5c2d0ff (3arg) = \x => \x1 => PE_*_954510b4 x x1
```

6.19 Miscellaneous

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

6.19.1 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.

6.19.2 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.::
```

6.19.3 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 |)
```

6.19.4 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))))
```

6.19.5 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.

6.19.6 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}`.

6.19.7 Existence of an implementation

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

```
foo : Num Nat
foo = %implementation
```

6.19.8 '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
```

6.19.9 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`.

6.19.10 Bash Completion

Use of `optparse-applicative` allows Idris to support Bash completion. You can obtain the completion script for Idris using the following command:

```
idris --bash-completion-script `which idris`
```

To enable completion for the lifetime of your current session, run the following command:

```
source <(idris --bash-completion-script `which idris`)
```

To enable completion permanently you must either:

- Modify your bash init script with the above command.
- Add the completion script to the appropriate `bash_completion.d/` folder on your machine.

CHAPTER 7

Tutorials on the Idris Language

Tutorials submitted by community members.

7.1 Type Providers in Idris

Type providers in Idris are simple enough, but there are a few caveats to using them that it would be worthwhile to go through the basic steps. We also go over foreign functions, because these will often be used with type providers.

7.1.1 The use case

First, let's talk about *why* we might want type providers. There are a number of reasons to use them and there are other examples available around the net, but in this tutorial we'll be using them to port C's `struct stat` to Idris.

Why do we need type providers? Well, Idris's FFI needs to know the types of the things it passes to and from C, but the fields of a `struct stat` are implementation-dependent types that cannot be relied upon. We don't just want to hard-code these types into our program··· so we'll use a type provider to find them at compile time!

7.1.2 A simple example

First, let's go over a basic usage of type providers, because foreign functions can be confusing but it's important to remember that providers themselves are simple.

A type provider is simply an IO action that returns a value of this type:

```
data Provider a = Provide a | Error String
```

Looks familiar? `Provider` is just `Either a String`, given a slightly more descriptive name.

Remember though, type providers we use in our program must be IO actions. Let's write a simple one now:

```
module Provider
-- Asks nicely for the user to supply the size of C's size_t type on this
-- machine
getSizeT : IO (Provider Int)
getSizeT = do
  putStrLn "I'm sorry, I don't know how big size_t is. Can you tell me, in bytes?"
  resp <- getLine
  case readInt resp of
     Just sizeTSize => pure (Provide sizeTSize)
     Nothing => pure (Error "I'm sorry, I don't understand.")
-- the readInt function is left as an exercise
```

We assume that whoever's compiling the library knows the size of `size_t`, so we'll just ask them! (Don't worry, we'll get it ourselves later.) Then, if their response can be converted to an integer, we present `Provide sizeTSize` as the result of our IO action; or if it can't, we signal a failure. (This will then become a compile-time error.)

Now we can use this IO action as a type provider:

```
module Main
-- to gain access to the IO action we're using as a provider
import Provider

-- TypeProviders is an extension, so we'll enable it
%language TypeProviders

-- And finally, use the provider!
-- Note that the parentheses are mandatory.
%provide (sizeTSize : Int) with getSizeT

-- From now on it's just a normal program where `sizeTSize` is available
-- as a top-level constant
main : IO ()
main = do
  putStr "Look! I figured out how big size_t is! It's "
  putStr (show sizeTSize)
  putStr " bytes!"
```

Yay! We··· asked the user something at compile time? That's not very good, actually. Our library is going to be difficult to compile! This is hardly a step up from having them edit in the size of `size_t` themselves!

Don't worry, there's a better way.

7.1.3 Foreign Functions

It's actually pretty easy to write a C function that figures out the size of `size_t`:

```
int sizeof_size_t() { return sizeof(size_t); }
```

(Why an int and not a `size_t`? The FFI needs to know how to receive the return value of this function and translate it into an Idris value. If we knew how to do this for values of C type `size_t`, we wouldn't need to write this function at all! If we really wanted to be safe from overflow, we could use an array of multiple integers, but the SIZE of `size_t` is never going to be a 65535 byte integer.)

So now we can get the size of `size_t` as long as we're in C code. We'd like to be able to use this from Idris. Can we do this? It turns out we can.

`foreign`

With foreign, we can turn a C function into an IO action. It works like this:

```
getSizeT : IO Int
getSizeT = foreign FFI_C "sizeof_size_t" (IO Int)
```

Pretty simple. `foreign` takes a specification of what function it needs to call and that function's return type.

Running foreign functions

This is all well and good for writing code that will typecheck. To actually run the code, we'll need to do just a bit more work. Exactly what we need to do depends on whether we want to interpret or compile our code.

In the interpreter

If we want to call our foreign functions from interpreted code (such as the REPL or a type provider), we need to dynamically link a library containing the symbols we need. This is pretty easy to do with the `%dynamic` directive:

```
%dynamic "./filename.so"
```

Note that the leading "./" is important: currently, the string you provide is interpreted as by `dlopen()`, which on Unix does not search in the current directory by default. If you use the "./", the library will be searched for in the directory from which you run idris (*not* the location of the file you're running!). Of course, if you're using functions from an installed library rather than something you wrote yourself, the "./" is not necessary.

In an executable

If we want to run our code from an executable, we can statically link instead. We'll use the `%include` and `%link` directives:

```
%include C "filename.h"
%link C "filename.o"
```

Note the extra argument to the directive! We specify that we're linking a C header and library. Also, unlike `%dynamic`, these directives search in the current directory by default. (That is, the directory from which we run idris.)

7.1.4 Putting it all together

So, at the beginning of this article I said we'd use type providers to port `struct stat` to Idris. The relevant part is just translating all the mysterious typedef'd C types into Idris types, and that's what we'll do here.

First, let's write a C file containing functions that we'll bind to.

```
/* stattypes.c */
#include <sys/stat.h>

int sizeof_dev_t() { return sizeof(dev_t); }
int sizeof_ino_t() { return sizeof(ino_t); }
/* lots more functions like this */
```

Next, an Idris file to define our providers:

```
-- Providers.idr
module Providers

%dynamic "./stattypes.so"
%access export

sizeOfDevT : IO Int
sizeOfDevT = foreign FFI_C "sizeof_dev_t" (IO Int)
{- lots of similar functions -}

-- Indicates how many bits are used to represent various system
-- stat types.
public export
data BitWidth = B8 | B16 | B32 | B64

Show BitWidth where
  show B8 = "8 bits"
  show B16 = "16 bits"
  show B32 = "32 bits"
  show B64 = "64 bits"

-- Now we have an integer, but we want a Provider BitWidth.
-- Since our sizeOf* functions are ordinary IO actions, we
-- can just map over them.
bytesToType : Int -> Provider BitWidth
bytesToType 1 = Provide B8  -- "8 bit value"
bytesToType 2 = Provide B16
bytesToType 4 = Provide B32
bytesToType 8 = Provide B64
bytesToType _ = Error "Unrecognised integral type."

getDevT : IO (Provider BitWidth)
getDevT = map bytesToType sizeOfDevT
{- lots of similar functions -}
```

Finally, we'll write one more idris file where we use the type providers:

```
-- Main.idr
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
module Main
import Providers
%language TypeProviders
%provide (DevTBitWidth : BitWidth) with getDevT

-- We can now use DevTBitWidth in our program!
main : IO ()
main = putStrLn $ "size of dev_t: " ++ show DevTBitWidth
```

7.2 The Interactive Theorem Prover

This short guide contributed by a community member illustrates how to prove associativity of addition on `Nat` using the interactive theorem prover.

First we define a module `Foo.idr`

```
module Foo

plusAssoc : plus n (plus m o) = plus (plus n m) o
plusAssoc = ?rhs
```

We wish to perform induction on `n`. First we load the file into the Idris `REPL` as follows:

```
$ idris Foo.idr
```

We will be given the following prompt, in future releases the version string will differ:

```
     ____    __     _
    /  _/___/ /____(_)____
    / // __  / ___/ / ___/     Version 0.9.18.1
  _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
 /___/\__,_/_/  /_/____/       Type :? for help

Idris is free software with ABSOLUTELY NO WARRANTY.
For details type :warranty.
Type checking ./Foo.idr
Metavariables: Foo.rhs
*Foo>
```

7.2.1 Explore the Context

We start the interactive session by asking Idris to prove the hole `rhs` using the command `:p rhs`. Idris by default will show us the initial context. This looks as follows:

```
*Foo> :p rhs
----------                 Goal:                 ----------
{ hole 0 }:
 (n : Nat) ->
 (m : Nat) ->
 (o : Nat) ->
 plus n (plus m o) = plus (plus n m) o
```

7.2.2 Application of Intros

We first apply the `intros` tactic:

```
-Foo.rhs> intros
----------              Other goals:              ----------
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
{ hole 3 }:
 plus n (plus m o) = plus (plus n m) o
```

7.2.3 Induction on `n`

Then apply `induction` on to `n`:

```
-Foo.rhs> induction n
----------              Other goals:              ----------
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_Z0:
 plus Z (plus m o) = plus (plus Z m) o
```

7.2.4 Compute

```
-Foo.rhs> compute
----------              Other goals:              ----------
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
----------                 Goal:                  ----------
elim_Z0:
 plus m o = plus m o
```

7.2.5 Trivial

```
-Foo.rhs> trivial
----------              Other goals:              ----------
```

(äÿŃéąţçżğçżŋ)

(çżŋäÿŁéąţ)

```
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------                Assumptions:                ----------
 n : Nat
 m : Nat
 o : Nat
----------                   Goal:                    ----------
elim_S0:
 (n__0 : Nat) ->
 (plus n__0 (plus m o) = plus (plus n__0 m) o) ->
 plus (S n__0) (plus m o) = plus (plus (S n__0) m) o
```

7.2.6 Intros

```
-Foo.rhs> intros
----------                Other goals:                ----------
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------                Assumptions:                ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                   Goal:                    ----------
{ hole 5 }:
 plus (S n__0) (plus m o) = plus (plus (S n__0) m) o
```

7.2.7 Compute

```
-Foo.rhs> compute
----------                Other goals:                ----------
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------                Assumptions:                ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                   Goal:                    ----------
{ hole 5 }:
 S (plus n__0 (plus m o)) = S (plus (plus n__0 m) o)
```

7.2.8 Rewrite

```
-Foo.rhs> rewrite ihn__0
----------              Other goals:              ----------
{ hole 5 }
{ hole 4 }
elim_S0
{ hole 2 }
{ hole 1 }
{ hole 0 }
----------              Assumptions:              ----------
 n : Nat
 m : Nat
 o : Nat
 n__0 : Nat
 ihn__0 : plus n__0 (plus m o) = plus (plus n__0 m) o
----------                  Goal:                 ----------
{ hole 6 }:
 S (plus n__0 (plus m o)) = S (plus n__0 (plus m o))
```

7.2.9 Trivial

```
-Foo.rhs> trivial
rhs: No more goals.
-Foo.rhs> qed
Proof completed!
Foo.rhs = proof
  intros
  induction n
  compute
  trivial
  intros
  compute
  rewrite ihn__0
  trivial
```

Two goals were created: one for `Z` and one for `S`. Here we have proven associativity, and assembled a tactic based proof script. This proof script can be added to `Foo.idr`.

Bibliography

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