Capture Checking

Capture checking is a research project that modifies the Scala type system to track references to capabilities in values. It is currently implemented in an experimental branch cc-experiment in the dotty repo and can be enabled on this branch with a `-Ycc` compiler option.

To get an idea what capture checking can do, let's start with a small example:

```
def usingLogFile[T](op: FileOutputStream => T): T =
  val logFile = FileOutputStream("log")
  val result = op(logFile)
  logFile.close()
  result
```

The `usingLogFile` method invokes a given operation with a fresh log file as parameter. Once the operation has ended, the log file is closed and the operation's result is returned. This is a typical *try-with-resources* pattern, similar to many other such patterns which are often supported by special language constructs in other languages.

The problem is that `usingLogFile`'s implementation is not entirely safe. One can undermine it by passing an operation that performs the logging at some later point after it has terminated. For instance:

```
val later = usingLogFile { file => () => file.write(0) }
later() // crash
```

When `later` is executed it tries to write to a file that is already closed, which results in an uncaught `IOException`.

Capture checking gives us the mechanism to prevent such errors *statically*. To prevent unsafe usages of `usingLogFile`, we can declare it like this:

```
def usingLogFile[T](op: ({*} FileOutputStream) => T): T =
  // same body as before
```

The only thing that's changed is that the `FileOutputStream` parameter of `op` is now tagged with `{*}`. We'll see that this turns the parameter into a *capability* whose lifetime is tracked.

If we now try to define the problematic value `later`, we get a static error:

```
   |  val later = usingLogFile { f => () => f.write(0) }
   |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |The expression's type {*} () -> Unit is not allowed to capture the root cap
   |This usually means that a capability persists longer than its allowed lifet
```

In this case, it was easy to see that the `logFile` capability escapes in the closure passed to `usingLogFile`. But capture checking also works for more complex cases. For instance, capture checking is able to distinguish between the following safe code:

```
val xs = usingLogFile { f =>
  List(1, 2, 3).map { x => f.write(x); x * x }
}
```

and the following unsafe one:

```
val xs = usingLogFile { f =>
  LazyList(1, 2, 3).map { x => f.write(x); x * x }
}
```

An error would be issued in the second case, but not the first one (this assumes a capture-aware formulation of `LazyList` which we will present later in this page).

It turns out that capture checking has very broad applications. Besides the various try-with-resources patterns, it can also be a key part to the solutions of many other long standing problems in programming languages. Among them:

- How to have a simple and flexible system for checked exceptions. We show later how capture checking enables a clean and fully safe system for checked exceptions in Scala.
- How to address the problem of effect polymorphism in general.

- How to solve the "what color is your function?" problem of mixing synchronous and asynchronous computations.
- How to do region-based allocation, safely,
- How to reason about capabilities associated with memory locations.

The following sections explain in detail how capture checking works in Scala 3.

Overview

The capture checker extension introduces a new kind of types and it enforces some rules for working with these types.

Capture checking is enabled by the compiler option `-Ycc`. If the option is not given, the new type forms can still be written but they are not checked for consistency, because they are treated simply as certain uninterpreted annotated types.

Capabilities and Capturing Types

Capture checking is done in terms of *capturing types* of the form $\{c_1, \ldots, c_i\}\ T$. Here T is a type, and $\{c_1, \ldots, c_i\}$ is a *capture set* consisting of references to capabilities $c_1, \ldots, c_i$.

A *capability* is syntactically a method- or class-parameter, a local variable, or the `this` of an enclosing class. The type of a capability must be a capturing type with a non-empty capture set. We also say that variables that are capabilities are *tracked*.

In a sense, every capability gets its authority from some other, more sweeping capability which it captures. The most sweeping capability, from which ultimately all others are derived is written `*`. We call it the *universal capability*.

Here is an example:

```
class FileSystem

class Logger(fs: {*} FileSystem):
  def log(s: String): Unit = ... // Write to a log file, using `fs`

def test(fs: {*} FileSystem) =
  val l: {fs} Logger = Logger(fs)
  l.log("hello world!")
  val xs: {l} LazyList[Int] =
    LazyList.from(1)
      .map { i =>
        l.log(s"computing elem # $i")
        i * i
```

```
    }
  xs
```

Here, the `test` method takes a `FileSystem` as a parameter. `fs` is a capability since its type has a non-empty capture set. The capability is passed to the `Logger` constructor and retained as a field in class `Logger`. Hence, the local variable `l` has type `{fs} Logger`: it is a `Logger` which retains the `fs` capability.

The second variable defined in `test` is `xs`, a lazy list that is obtained from `LazyList.from(1)` by logging and mapping consecutive numbers. Since the list is lazy, it needs to retain the reference to the logger `l` for its computations. Hence, the type of the list is `{l} LazyList[Int]`. On the other hand, since `xs` only logs but does not do other file operations, it retains the `fs` capability only indirectly. That's why `fs` does not show up in the capture set of `xs`.

Capturing types come with a subtype relation where types with "smaller" capture sets are subtypes of types with larger sets (the *subcapturing* relation is defined in more detail below). If a type `T` does not have a capture set, it is called *pure*, and is a subtype of any capturing type that adds a capture set to `T`.

Function Types

The usual function type `A ⇒ B` now stands for a function that can capture arbitrary capabilities. We call such functions *impure*. By contrast, the new single arrow function type `A → B` stands for a function that cannot capture any capabilities, or otherwise said, is *pure*. One can add a capture set in front of an otherwise pure function. For instance, `{c, d} A → B` would be a function that can capture capabilities `c` and `d`, but no others.

The impure function type `A ⇒ B` is treated as an alias for `{*} A → B`. That is, impure functions are functions that can capture anything.

Function types and captures both associate to the right, so

```
{c} A -> {d} B -> C
```

is the same as

```
{c} (A -> {d} (B -> C))
```

Contrast with

```
({c} A) -> ({d} B) -> C
```

which is a curried pure function over argument types that can capture `c` and `d`, respectively.

Analogous conventions apply to context function types. `A ?=> B` is an impure context function, with `A ?→ B` as its pure complement.

Note 1: The identifiers `→` and `?→` are now treated as soft keywords when used as infix type operators. They are still available as regular identifiers for terms. For instance, the mapping syntax `Map("x" → 1, "y" → 2)` is still supported since it only applies to terms.

Note 2: The distinctions between pure vs impure function types do not apply to methods. In fact, since methods are not values they never capture anything directly. References to capabilities in a method are instead counted in the capture set of the enclosing object.

By-Name Parameter Types

A convention analogous to function types also extends to by-name parameters. In

```
def f(x: => Int): Int
```

the actual argument can refer to arbitrary capabilities. So the following would be OK:

```
f(if p(y) then throw Ex() else 1)
```

On the other hand, if `f` was defined like this

```
def f(x: -> Int): Int
```

the actual argument to `f` could not refer to any capabilities, so the call above would be rejected. One can also allow specific capabilities like this:

```
def f(x: {c}-> Int): Int
```

Here, the actual argument to `f` is allowed to use the `c` capability but no others.

Note: It is strongly recommended to write the capability set and the arrow `→` without intervening spaces, as otherwise the notation would look confusingly like a function type.

Subtyping and Subcapturing

Capturing influences subtyping. As usual we write `T₁ <: T₂` to express that the type `T₁` is a subtype of the type `T₂`, or equivalently, that `T₁` conforms to `T₂`. An analogous *subcapturing* relation applies to capture sets. If `C₁` and `C₂` are capture sets, we write `C₁ <: C₂` to express that `C₁` *is covered by* `C₂`, or, swapping the operands, that `C₂` *covers* `C₁`.

Subtyping extends as follows to capturing types:

- Pure types are subtypes of capturing types. That is, `T <: C T`, for any type `T`, capturing set `C`.
- For capturing types, smaller capturing sets produce subtypes: `C₁ T₁ <: C₂ T₂` if `C₁ <: C₂` and `T₁ <: T₂`.

A subcapturing relation `C₁ <: C₂` holds if `C₂` *accounts for* every element `c` in `C₁`. This means one of the following three conditions must be true:

- `c ∈ C₂`,
- `c` refers to a parameter of some class `Cls` and `C₂` contains `Cls.this`,
- `c`'s type has capturing set `C` and `C₂` accounts for every element of `C` (that is, `C <: C₂`).

Example 1. Given

```
fs: {*} FileSystem
ct: {*} CanThrow[Exception]
l : {fs} Logger
```

we have

```
{l}  <: {fs}     <: {*}
{fs} <: {fs, ct} <: {*}
{ct} <: {fs, ct} <: {*}
```

The set consisting of the root capability `{*}` covers every other capture set. This is a consequence of the fact that, ultimately, every capability is created from `*`.

Example 2. Consider again the FileSystem/Logger example from before. `LazyList[Int]` is a proper subtype of `{l} LazyList[Int]`. So if the `test` method in that example was declared with a result type `LazyList[Int]`, we'd get a type error. Here is the error message:

```
11 |def test(using fs: {*} FileSystem): LazyList[Int] = {
   |                                                   ^
   |                                          Found:    {fs} LazyList[Int]
   |                                          Required: LazyList[Int]
```

Why does it say `{fs} LazyList[Int]` and not `{l} LazyList[Int]`, which is, after all, the type of the returned value `xs`? The reason is that `l` is a local variable in the body of `test`, so it cannot be referred to in a type outside that body. What happens instead is that the type is *widened* to the smallest supertype that does not mention `l`. Since `l` has capture set `fs`, we have that `{fs}` covers `{l}`, and `{fs}` is acceptable in a result type of `test`, so `{fs}` is the result of that widening. This widening is called *avoidance*; it is not specific to capture checking but applies to all variable references in Scala types.

Capability Classes

Classes like `CanThrow` or `FileSystem` have the property that their values are always intended to be capabilities. We can make this intention explicit and save boilerplate by declaring these classes with a `@capability` annotation.

The capture set of a capability class type is always `{*}`. This means we could equivalently express the `FileSystem` and `Logger` classes as follows:

```
import annotation.capability

@capability class FileSystem

class Logger(using FileSystem):
  def log(s: String): Unit = ???

def test(using fs: FileSystem) =
  val l: {fs} Logger = Logger()
  ...
```

In this version, `FileSystem` is a capability class, which means that the `{*}` capture set is implied on the parameters of `Logger` and `test`. Writing the capture set explicitly produces a warning:

```
class Logger(using {*} FileSystem):
                   ^^^^^^^^^^^^^^

             redundant capture: FileSystem already accounts for *
```

Another, unrelated change in the version of the last example here is that the `FileSystem` capability is now passed as an implicit parameter. It is quite natural to model capabilities with implicit parameters since it greatly reduces the wiring overhead once multiple capabilities are in play.

Capture Checking of Closures

If a closure refers to capabilities in its body, it captures these capabilities in its type. For instance, consider:

```
def test(fs: FileSystem): {fs} String -> Unit =
  (x: String) => Logger(fs).log(x)
```

Here, the body of `test` is a lambda that refers to the capability `fs`, which means that `fs` is retained in the lambda. Consequently, the type of the lambda is `{fs} String → Unit`.

Note: Function values are always written with `⇒` (or `?=>` for context functions). There is no syntactic distinction for pure *vs* impure function values. The distinction is only made in their types.

A closure also captures all capabilities that are captured by the functions it calls. For instance, in

```
def test(fs: FileSystem) =
  def f() = g()
  def g() = (x: String) => Logger(fs).log(x)
  f
```

the result of `test` has type `{fs} String → Unit` even though function `f` itself does not refer to `fs`.

Capture Checking of Classes

The principles for capture checking closures also apply to classes. For instance, consider:

```
class Logger(using fs: FileSystem):
  def log(s: String): Unit = ... summon[FileSystem] ...

def test(xfs: FileSystem): {xfs} Logger =
  Logger(xfs)
```

Here, class `Logger` retains the capability `fs` as a (private) field. Hence, the result of `test` is of type `{xfs} Logger`

Sometimes, a tracked capability is meant to be used only in the constructor of a class, but is not intended to be retained as a field. This fact can be communicated to the capture checker by declaring the parameter as `@constructorOnly`. Example:

```
import annotation.constructorOnly

class NullLogger(using @constructorOnly fs: FileSystem):
  ...
def test2(using fs: FileSystem): NullLogger = NullLogger() // OK
```

The captured references of a class include *local capabilities* and *argument capabilities*. Local capabilities are capabilities defined outside the class and referenced from its body. Argument capabilities are passed as parameters to the primary constructor of the class. Local capabilities are inherited: the local capabilities of a superclass are also local capabilities of its subclasses. Example:

```
@capability class Cap

def test(a: Cap, b: Cap, c: Cap) =
  class Super(y: Cap):
    def f = a
  class Sub(x: Cap) extends Super(x)
    def g = b
  Sub(c)
```

Here class `Super` has local capability `a`, which gets inherited by class `Sub` and is combined with `Sub`'s own local capability `b`. Class `Sub` also has an argument capability corresponding to its parameter `x`. This capability gets instantiated to `c` in the final constructor call `Sub(c)`. Hence, the capture set of that call is `{a, b, c}`.

The capture set of the type of `this` of a class is inferred by the capture checker, unless the type is explicitly declared with a self type annotation like this one:

```
class C:
  self: {a, b} D => ...
```

The inference observes the following constraints:

- The type of `this` of a class `C` includes all captured references of `C`.

- The type of `this` of a class `C` is a subtype of the type of `this` of each parent class of `C`.
- The type of `this` must observe all constraints where `this` is used.

For instance, in

```
@capability class Cap
def test(c: Cap) =
  class A:
    val x: A = this
    def f = println(c)  // error
```

we know that the type of `this` must be pure, since `this` is the right hand side of a `val` with type `A`. However, in the last line we find that the capture set of the class, and with it the capture set of `this`, would include `c`. This leads to a contradiction, and hence to a checking error:

```
16 |    def f = println(c)  // error
   |                    ^
   |(c : Cap) cannot be referenced here; it is not included in the allowed capt
```

Capture Tunnelling

Consider the following simple definition of a `Pair` class:

```
class Pair[+A, +B](x: A, y: B):
  def fst: A = x
  def snd: B = y
```

What happens if `Pair` is instantiated like this (assuming `ct` and `fs` are two capabilities in scope)?

```
def x: {ct} Int -> String
def y: {fs} Logger
def p = Pair(x, y)
```

The last line will be typed as follows:

```
def p: Pair[{ct} Int -> String, {fs} Logger] = Pair(x, y)
```

This might seem surprising. The `Pair(x, y)` value does capture capabilities `ct` and `fs`. Why don't they show up in its type at the outside?

The answer is capture tunnelling. Once a type variable is instantiated to a capturing type, the capture is not propagated beyond this point. On the other hand, if the type variable is instantiated again on access, the capture information "pops out" again. For instance, even though `p` is technically pure because its capture set is empty, writing `p.fst` would record a reference to the captured capability `ct`. So if this access was put in a closure, the capability would again form part of the outer capture set. E.g.

```
() => p.fst : {ct} () -> {ct} Int -> String
```

In other words, references to capabilities "tunnel through" in generic instantiations from creation to access; they do not affect the capture set of the enclosing generic data constructor applications. This principle may seem surprising at first, but it is the key to make capture checking concise and practical.

Escape Checking

The universal capability `*` should be conceptually available only as a parameter to the main program. Indeed, if it was available everywhere, capability checking would be undermined since one could mint new capabilities at will. In line with this reasoning, some capture sets are restricted so that they are not allowed to contain the universal capability.

Specifically, if a capturing type is an instance of a type variable, that capturing type is not allowed to carry the universal capability `{*}`. There's a connection to tunnelling here. The capture set of a type has to be present in the environment when a type is instantiated from a type variable. But `*` is not itself available as a global entity in the environment. Hence, an error should result.

We can now reconstruct how this principle produced the error in the introductory example, where `usingLogFile` was declared like this:

```
def usingLogFile[T](op: ({*} FileOutputStream) => T): T = ...
```

The error message was:

```
   |  val later = usingLogFile { f => () => f.write(0) }
   |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |The expression's type {*} () -> Unit is not allowed to capture the root cap
   |This usually means that a capability persists longer than its allowed lifet
```

This error message was produced by the following logic:

- The `f` parameter has type `{*} FileOutputStream`, which makes it a capability.
- Therefore, the type of the expression `() ⇒ f.write(0)` is `{f} () → Unit`.
- This makes the whole type of the closure passed to `usingLogFile` the dependent function type `(f: {*} FileOutputStream) → {f} () → Unit`.
- The expected type of the closure is a simple, parametric, impure function type `({*} FileOutputStream) ⇒ T`, for some instantiation of the type variable `T`.
- The smallest supertype of the closure's dependent function type that is a parametric function type is `({*} FileOutputStream) ⇒ {*} () → Unit`
- Hence, the type variable `T` is instantiated to `* () → Unit`, which causes the error.

An analogous restriction applies to the type of a mutable variable. Another way one could try to undermine capture checking would be to assign a closure with a local capability to a global variable. Maybe like this:

```
var loophole: {*} () -> Unit = () => ()
usingLogFile { f =>
  loophole = () => f.write(0)
}
loophole()
```

But this will not compile either, since mutable variables cannot have universal capture sets.

One also needs to prevent returning or assigning a closure with a local capability in an argument of a parametric type. For instance, here is a slightly more refined attack:

```
class Cell[+A](x: A)
val sneaky = usingLogFile { f => Cell(() => f.write(0)) }
sneaky.x()
```

At the point where the `Cell` is created, the capture set of the argument is `f`, which is OK. But at the point of use, it is `*` (because `f` is no longer in scope), which causes again an error:

```
   |  sneaky.x()
   |  ^^^^^^^^
   |The expression's type {*} () -> Unit is not allowed to capture the root cap
   |This usually means that a capability persists longer than its allowed lifet
```

Looking at object graphs, we observe a monotonicity property: The capture set of an object `x` covers the capture sets of all objects reachable through `x`. This property is

reflected in the type system by the following *monotonicity rule*:

- In a class `C` with a field `f`, the capture set `{this}` covers the capture set `{this.f}` as well as the capture set of any application of `this.f` to pure arguments.

Checked Exceptions

Scala enables checked exceptions through a language import. Here is an example, taken from the safer exceptions page, and also described in a paper presented at the 2021 Scala Symposium.

```
import language.experimental.saferExceptions

class LimitExceeded extends Exception

val limit = 10e+10
def f(x: Double): Double throws LimitExceeded =
  if x < limit then x * x else throw LimitExceeded()
```

The new `throws` clause expands into an implicit parameter that provides a `CanThrow` capability. Hence, function `f` could equivalently be written like this:

```
def f(x: Double)(using CanThrow[LimitExceeded]): Double = ...
```

If the implicit parameter is missing, an error is reported. For instance, the function definition

```
def g(x: Double): Double =
  if x < limit then x * x else throw LimitExceeded()
```

is rejected with this error message:

```
  |  if x < limit then x * x else throw LimitExceeded()
  |                                ^^^^^^^^^^^^^^^^^^^^
  |The capability to throw exception LimitExceeded is missing.
  |The capability can be provided by one of the following:
  | - Adding a using clause `(using CanThrow[LimitExceeded])` to the definitio
  | - Adding `throws LimitExceeded` clause after the result type of the enclos
  | - Wrapping this piece of code with a `try` block that catches LimitExceede
```

`CanThrow` capabilities are required by `throw` expressions and are created by `try` expressions. For instance, the expression

```
try xs.map(f).sum
catch case ex: LimitExceeded => -1
```

would be expanded by the compiler to something like the following:

```
try
  erased given ctl: CanThrow[LimitExceeded] = compiletime.erasedValue
  xs.map(f).sum
catch case ex: LimitExceeded => -1
```

(The `ctl` capability is only used for type checking but need not show up in the generated code, so it can be declared as erased.)

As with other capability based schemes, one needs to guard against capabilities that are captured in results. For instance, here is a problematic use case:

```
def escaped(xs: Double*): (() => Double) throws LimitExceeded =
  try () => xs.map(f).sum
  catch case ex: LimitExceeded => () => -1
val crasher = escaped(1, 2, 10e+11)
crasher()
```

This code needs to be rejected since otherwise the call to `crasher()` would cause an unhandled `LimitExceeded` exception to be thrown.

Under `-Ycc`, the code is indeed rejected

```
14 |  try () => xs.map(f).sum
   |  ^
   |The expression's type {*} () -> Double is not allowed to capture the root c
   |This usually means that a capability persists longer than its allowed lifet
15 |  catch case ex: LimitExceeded => () => -1
```

To integrate exception and capture checking, only two changes are needed:

- `CanThrow` is declared as a `@capability` class, so all references to `CanThrow` instances are tracked.
- Escape checking is extended to `try` expressions. The result type of a `try` is not allowed to capture the universal capability.

A Larger Example

As a larger example, we present an implementation of lazy lists and some use cases. For simplicity, our lists are lazy only in their tail part. This corresponds to what the

Scala-2 type `Stream` did, whereas Scala 3's `LazyList` type computes strictly less since it is also lazy in the first argument.

Here is the base trait `LzyList` for our version of lazy lists:

```
trait LzyList[+A]:
  def isEmpty: Boolean
  def head: A
  def tail: {this} LzyList[A]
```

Note that `tail` carries a capture annotation. It says that the tail of a lazy list can potentially capture the same references as the lazy list as a whole.

The empty case of a `LzyList` is written as usual:

```
object LzyNil extends LzyList[Nothing]:
  def isEmpty = true
  def head = ???
  def tail = ???
```

Here is a formulation of the class for lazy cons nodes:

```
import scala.compiletime.uninitialized

final class LzyCons[+A](hd: A, tl: () => {*} LzyList[A]) extends LzyList[A]:
  private var forced = false
  private var cache: {this} LzyList[A] = uninitialized
  private def force =
    if !forced then { cache = tl(); forced = true }
    cache

  def isEmpty = false
  def head = hd
  def tail: {this} LzyList[A] = force
end LzyCons
```

The `LzyCons` class takes two parameters: A head `hd` and a tail `tl`, which is a function returning a `LzyList`. Both the function and its result can capture arbitrary capabilities. The result of applying the function is memoized after the first dereference of `tail` in the private mutable field `cache`. Note that the typing of the assignment `cache = tl()` relies on the monotonicity rule for `{this}` capture sets.

Here is an extension method to define an infix cons operator `#:` for lazy lists. It is analogous to `::` but instead of a strict list it produces a lazy list without evaluating

its right operand.

```
extension [A](x: A)
  def #:(xs1: => {*} LzyList[A]): {xs1} LzyList[A] =
    LzyCons(x, () => xs1)
```

Note that `#:` takes an impure call-by-name parameter `xs1` as its right argument. The result of `#:` is a lazy list that captures that argument.

As an example usage of `#:`, here is a method `tabulate` that creates a lazy list of given length with a generator function `gen`. The generator function is allowed to have side effects.

```
def tabulate[A](n: Int)(gen: Int => A) =
  def recur(i: Int): {gen} LzyList[A] =
    if i == n then LzyNil
    else gen(i) #: recur(i + 1)
  recur(0)
```

Here is a use of `tabulate`:

```
class LimitExceeded extends Exception
def squares(n: Int)(using ct: CanThrow[LimitExceeded]) =
  tabulate(10) { i =>
    if i > 9 then throw LimitExceeded()
    i * i
  }
```

The inferred result type of `squares` is `{ct} LzyList[Int]`, i.e it is a lazy list of `Int`s that can throw the `LimitExceeded` exception when it is elaborated by calling `tail` one or more times.

Here are some further extension methods for mapping, filtering, and concatenating lazy lists:

```
extension [A](xs: {*} LzyList[A])
  def map[B](f: A => B): {xs, f} LzyList[B] =
    if xs.isEmpty then LzyNil
    else f(xs.head) #: xs.tail.map(f)

  def filter(p: A => Boolean): {xs, p} LzyList[A] =
    if xs.isEmpty then LzyNil
    else if p(xs.head) then xs.head #: xs.tail.filter(p)
    else xs.tail.filter(p)
```

```
  def concat(ys: {*} LzyList[A]): {xs, ys} LzyList[A] =
    if xs.isEmpty then ys
    else xs.head #: xs.tail.concat(ys)

  def drop(n: Int): {xs} LzyList[A] =
    if n == 0 then xs else xs.tail.drop(n - 1)
```

Their capture annotations are all as one would expect:

- Mapping a lazy list produces a lazy list that captures the original list as well as the (possibly impure) mapping function.
- Filtering a lazy list produces a lazy list that captures the original list as well as the (possibly impure) filtering predicate.
- Concatenating two lazy lists produces a lazy list that captures both arguments.
- Dropping elements from a lazy list gives a safe approximation where the original list is captured in the result. In fact, it's only some suffix of the list that is retained at run time, but our modelling identifies lazy lists and their suffixes, so this additional knowledge would not be useful.

Of course the function passed to `map` or `filter` could also be pure. After all, `A → B` is a subtype of `{*} A → B` which is the same as `A ⇒ B`. In that case, the pure function argument will *not* show up in the result type of `map` or `filter`. For instance:

```
val xs = squares(10)
val ys: {xs} LzyList[Int] = xs.map(_ + 1)
```

The type of the mapped list `ys` has only `xs` in its capture set. The actual function argument does not show up since it is pure. Likewise, if the lazy list `xs` was pure, it would not show up in any of the method results. This demonstrates that capability-based effect systems with capture checking are naturally *effect polymorphic*.

This concludes our example. It's worth mentioning that an equivalent program defining and using standard, strict lists would require no capture annotations whatsoever. It would compile exactly as written now in standard Scala 3, yet one gets the capture checking for free. Essentially, `⇒` already means "can capture anything" and since in a strict list side effecting operations are not retained in the result, there are no additional captures to record. A strict list could of course capture side-effecting closures in its elements but then tunnelling applies, since these elements are represented by a type variable. This means we don't need to annotate anything there either.

Another possibility would be a variant of lazy lists that requires all functions passed to `map`, `filter` and other operations like it to be pure. E.g. `map` on such a list would be defined like this:

```
extension [A](xs: LzyList[A])
  def map[B](f: A -> B): LzyList[B] = ...
```

That variant would not require any capture annotations either.

To summarize, there are two "sweet spots" of data structure design: strict lists in side-effecting or resource-aware code and lazy lists in purely functional code. Both are already correctly capture-typed without requiring any explicit annotations. Capture annotations only come into play where the semantics gets more complicated because we deal with delayed effects such as in impure lazy lists or side-effecting iterators over strict lists. This property is probably one of the greatest plus points of our approach to capture checking compared to previous techniques which tend to be more noisy.

Function Type Shorthands

TBD

Compilation Options

The following options are relevant for capture checking.

- -Ycc Enables capture checking.
- -Xprint:cc Prints the program with capturing types as inferred by capture checking.
- -Ycc-debug Gives more detailed, implementation-oriented information about capture checking, as described in the next section.

The implementation supporting capture checking with these options is currently in branch `cc-experiment` on dotty.epfl.ch.

Capture Checking Internals

The capture checker is architected as a propagation constraint solver, which runs as a separate phase after type-checking and some initial transformations.

Constraint variables stand for unknown capture sets. A constraint variable is introduced

- for every part of a previously inferred type,
- for the accessed references of every method, class, anonymous function, or by-name argument,
- for the parameters passed in a class constructor call.

Capture sets in explicitly written types are treated as constants (before capture checking, such sets are simply ignored).

The capture checker essentially rechecks the program with the usual typing rules. Every time a subtype requirement between capturing types is checked, this translates to a subcapturing test on capture sets. If the two sets are constant, this is simply a yes/no question, where a no will produce an error message.

If the lower set `C₁` of a comparison `C₁ <: C₂` is a variable, the set `C₂` is recorded as a *superset* of `C₁`. If the upper set `C₂` is a variable, the elements of `C₁` are *propagated* to `C₂`. Propagation of an element `x` to a set `C` means that `x` is included as an element in `C`, and it is also propagated to all known supersets of `C`. If such a superset is a constant, it is checked that `x` is included in it. If that's not the case, the original comparison `C₁ <: C₂` has no solution and an error is reported.

The type checker also performs various maps on types, for instance when substituting actual argument types for formal parameter types in dependent functions, or mapping member types with "as-seen-from" in a selection. Maps keep track of the variance of positions in a type. The variance is initially covariant, it flips to contravariant in function parameter positions, and can be either covariant, contravariant, or nonvariant in type arguments, depending on the variance of the type parameter.

When capture checking, the same maps are also performed on capture sets. If a capture set is a constant, its elements (which are capabilities) are mapped as regular types. If the result of such a map is not a capability, the result is approximated according to the variance of the type. A covariant approximation replaces a type by its capture set. A contravariant approximation replaces it with the empty capture set. A nonvariant approximation replaces the enclosing capturing type with a range of possible types that gets propagated and resolved further out.

When a mapping `m` is performed on a capture set variable `C`, a new variable `Cm` is created that contains the mapped elements and that is linked with `C`. If `C` subsequently acquires further elements through propagation, these are also propagated to `Cm` after being transformed by the `m` mapping. `Cm` also gets the same supersets as `C`, mapped again using `m`.

One interesting aspect of the capture checker concerns the implementation of capture tunnelling. The foundational theory on which capture checking is based makes tunnelling explicit through so-called *box* and *unbox* operations. Boxing hides a capture set and unboxing recovers it. The capture checker inserts virtual box and unbox operations based on actual and expected types similar to the way the type checker inserts implicit conversions. When capture set variables are first introduced, any capture set in a capturing type that is an instance of a type parameter instance is marked as "boxed". A boxing operation is inserted if the expected type of an expression is a capturing type with a boxed capture set variable. The effect of the insertion is that any references to capabilities in the boxed expression are forgotten, which means that capture propagation is stopped. Dually, if the actual type of an expression has a boxed variable as capture set, an unbox operation is inserted, which adds all elements of the capture set to the environment.

Boxing and unboxing has no runtime effect, so the insertion of these operations is only simulated; the only visible effect is the retraction and insertion of variables in the capture sets representing the environment of the currently checked expression.

The `-Ycc-debug` option provides some insight into the workings of the capture checker. When it is turned on, boxed sets are marked explicitly and capture set variables are printed with an ID and some information about their provenance. For instance, the string `{f, xs}33M5V` indicates a capture set variable that is known to hold elements `f` and `xs`. The variable's ID is `33`. The `M` indicates that the variable was created through a mapping from a variable with ID `5`. The latter is a regular variable, as indicated by `V`.

Generally, the string following the capture set consists of alternating numbers and letters where each number gives a variable ID and each letter gives the provenance of the variable. Possible letters are

- `V`: a regular variable,
- `M`: a variable resulting from a *mapping* of the variable indicated by the string to the right,
- `B`: similar to `M` but where the mapping is a *bijection*,
- `F`: a variable resulting from *filtering* the elements of the variable indicated by the string to the right,
- `I`: a variable resulting from an *intersection* of two capture sets,
- `D`: a variable resulting from the set *difference* of two capture sets.

At the end of a compilation run, `-Ycc-debug` will print all variable dependencies of variables referred to in previous output. Here is an example:

```
Capture set dependencies:
  {}2V                  ::
  {}3V                  ::
  {}4V                  ::
  {f, xs}5V             :: {f, xs}31M5V, {f, xs}32M5V
  {f, xs}31M5V          :: {xs, f}
  {f, xs}32M5V          ::
```

This section lists all variables that appeared in previous diagnostics and their dependencies, recursively. For instance, we learn that

- variables 2, 3, 4 are empty and have no dependencies,
- variable `5` has two dependencies: variables `31` and `32` which both result from mapping variable `5`,
- variable `31` has a constant fixed superset `{xs, f}`
- variable `32` has no dependencies.

‹ MainA... Tupled ... ›