Modular implicits / Multi-core OCaml

Leo White

Modular implicits

Leo White, Frédéric Bour and Jeremy Yallop

Ad-hoc polymorphism

Ad-hoc polymorphism occurs when a function is defined over several different types, acting in a different way for each type.

```
4 + 9
4.5 + 9.5
```

```
print [true; false]
print (Some 8.4)
```

Simple Overloading

```
public static String show(int x)
{
  return Integer.toString(x);
}

public static String show(float x)
{
  return Float.toString(x);
}
```

Simple Overloading

```
show(7);

show(4.2);
```

```
show("foo");
```

```
error: no suitable method found for show(String)
         show("foo")
         ^
```

Simple Overloading

```java
public static <T> String show_twice(T x)
{
  return show(x) + " " + show(x);
}
```

```
Main.java:23: error: no suitable method found for show(T)
                return show(x) + " " + show(x);
                       ^
    method Show.show(int) is not applicable
      (argument mismatch; T cannot be converted to int)
    method Show.show(float) is not applicable
      (argument mismatch; T cannot be converted to float)
  where T is a type-variable:
    T extends Object declared in method <T>show_twice(T)
```

Type classes

```
class Show a where
  show :: a -> string

instance Show Int where
  show = showInt

instance Show Float where
  show = showFloat
```

Type classes

```
> show 7
"7"

> show 4.5
"4.5"

> show (\ x -> x)

  No instance for (Show (t1 -> t1)) arising
    from a use of ‘show’
```

Type classes

```
show_twice x = show x ++ ” ” ++ show x
```

Type classes

```haskell
show_twice :: Show a => a -> String
show_twice x = show x ++ " " ++ show x
```

Type classes

```
> show_twice 5
"5 5"

> show_twice (\ x -> x)
```

```
No instance for (Show (t0 -> t0))
  arising from a use of ‘show_twice’
```

Type classes

```
instance Show a => Show [a] where
  show = showList show

> show [7, 8, 9]
"[7,8,9]"

> show [[1, 2, 3], [4, 5, 6]]
"[[1,2,3],[4,5,6]]"
```

Coherence

Every different valid typing derivation for a program leads to a resulting program that has the same dynamic semantics.

Canonicity

```
> instance Show Int where
    show x = "An Int"
```

```
<interactive>:2:10:
    Duplicate instance declarations:
      instance Show Int -- Defined at <interactive>:2:10
      instance Show Int -- Defined in ‘GHC.Show’
```

Abstract type equalities

```
module M : sig
  type t
end = struct
  type t = int
end
```

Abstract type equalities

```
module F (X : sig type t val show : t -> string end) =
struct
  instance Show X.t where
    show = X.show
end

instance Show int where
    show = string_of_int

F(struct
  type t = int
  let show _ = "An int"
end)
```

Scala implicits

```
trait Showable[T] { def show(x: T): String }

def show[T](x: T)(implicit s: Showable[T]) =
  s.show(x)

implicit object IntShowable extends Showable[Int] {
  def show(x: Int) = x.toString
}

show(7)
```

Scala implicits

```
def show_twice[T](x: T)(implicit s: Showable[T]) =
  show(x) + " " + show(x)

show_twice(7)
```

Scala implicits

```
implicit class ListShowable[T]
    extends Showable[List[T]]
      (implicit s: Showable[T]) {

  def show(x: List[T]) = x.toString(s.show, x)

}

show(List(1,2,3))
```

Scala implicits

```
implicit object IntShowable2 extends Showable[Int] {
  def show(x: Int) = x.toString
}

show(7)
```

```
error: ambiguous implicit values:
 both object IntShowable2 in object $iw of type
   object IntShowable2
 and object IntShowable in object $iw of type
   object IntShowable
 match expected type Showable[Int]
       show(7)
```

Modular implicits

Implicit *module* parameters to functions chosen by their *module type*.

Demo

Status

Working prototype based on OCaml 4.02.0 (by Frédéric Bour)

- Install it using the OCaml Package Manager (OPAM):

```
$ opam switch 4.02.0+modular-implicits
```

- Try it online (all compiled to JavaScript and running in the browser):
  http://andrewray.github.io/iocamljs/modimp.html
- When you (inevitably) find bugs, report them to
  http://github.com/ocamllabs/ocaml-modular-implicits

Multi-core OCaml

Stephen Dolan, Leo White, KC Sivaramakrishnan and Anil Madhavapeddy

Concurrency and Parallelism

Concurrency

Parallelism

Concurrency and Parallelism

Concurrency

- Concurrency is for writing programs
  “My program handles 1000s connections at once”

Parallelism

Concurrency and Parallelism

Concurrency

- Concurrency is for writing programs
  “My program handles 1000s connections at once”

Parallelism

- Parallelism is for improving performance
  “My program uses all 8 cores”

Concurrency and Parallelism

Concurrency

- Concurrency is for writing programs
  "My program handles 1000s connections at once"
- Monadic: Lwt, Async

Parallelism

- Parallelism is for improving performance
  "My program uses all 8 cores"

Concurrency and Parallelism

Concurrency

- Concurrency is for writing programs
  “My program handles 1000s connections at once”
- Monadic: Lwt, Async
- Direct: Systhreads, Vmthreads

Parallelism

- Parallelism is for improving performance
  “My program uses all 8 cores”

Concurrency and Parallelism

Concurrency

- Concurrency is for writing programs
  "My program handles 1000s connections at once"
- Monadic: Lwt, Async
- Direct: Systhreads, Vmthreads

Parallelism

- Parallelism is for improving performance
  "My program uses all 8 cores"
- Multi-process: Parmap, Async_parallel

Concurrency and Parallelism

Concurrency

- Concurrency is for writing programs
  "My program handles 1000s connections at once"
- Monadic: Lwt, Async
- Direct: Systhreads, Vmthreads

Parallelism

- Parallelism is for improving performance
  "My program uses all 8 cores"
- Multi-process: Parmap, Async_parallel
- Shared memory: ?

Multi-core OCaml

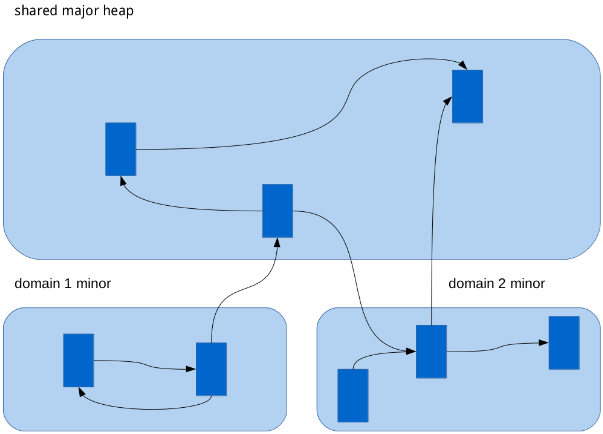
- Provide support for shared-memory parallelism
- Improve support for concurrency – avoid people abusing the parallelism primitives for concurrency (see Java).

Parallelism: Domains

The unit of parallelism is the *domain*.

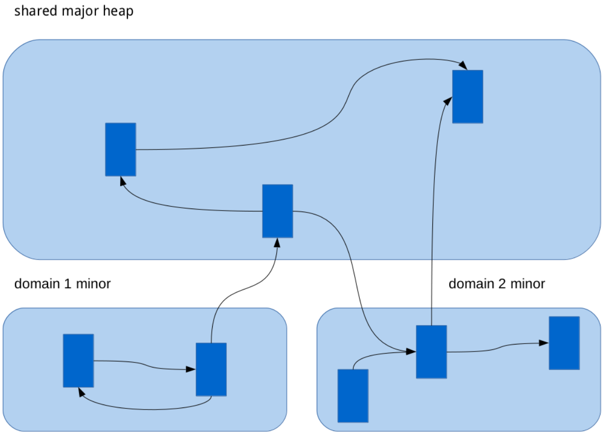
- Expensive to create
- Intention is to have roughly one per-core

Minor heaps



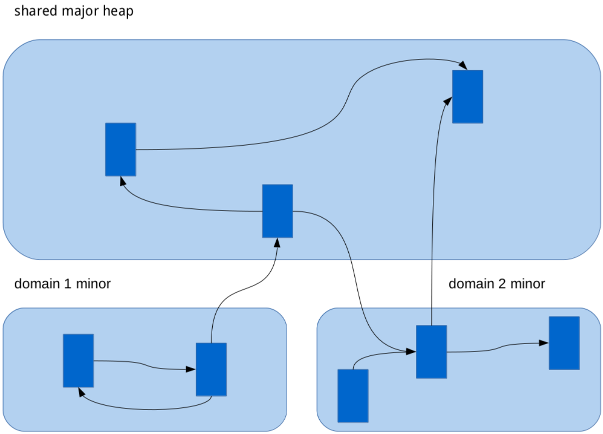
Each domain has its own minor heap

Minor heaps



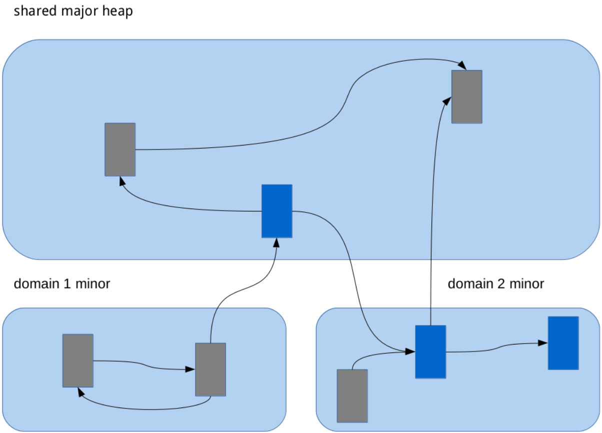
These minor heaps can be collected independently without synchronisation

Minor heaps



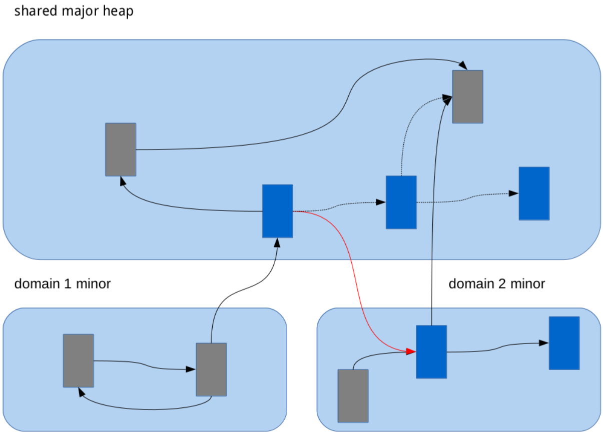
GC invariant: no pointers between minor heaps

Minor heaps



GC invariant: no pointers between minor heaps

Minor heaps

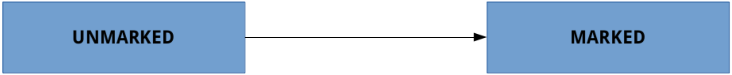


GC invariant: no pointers between minor heaps

Major heap

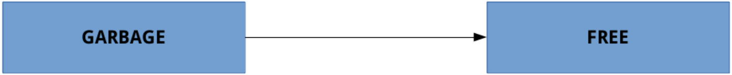
Mostly-concurrent parallel collector (VCGC)

- Domains independently mark reachable objects



UNMARKED → MARKED

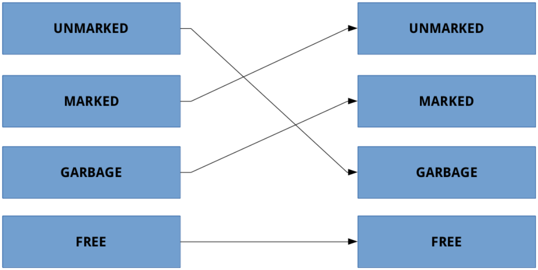
- Domains sweep separate parts of the heap



GARBAGE → FREE

Major heap

Complete a GC cycle by changing the interpretation of the mark bits



- Requires all domains to synchronise
- Most marking and sweeping should have been completed before synchronisation

Concurrency: Fibers

The unit of concurrency is the *fiber*.

- Very cheap to create
- A fiber is essentially just a stack
- Stacks start very small and are automatically resized as needed

Concurrency: Fibers

There are many interesting programming models for concurrency.

- We don't want to mandate a particular model
- Instead provide powerful primitives for implementing concurrency

Concurrency: Fibers

There are many interesting programming models for concurrency.

- We don't want to mandate a particular model
- Instead provide powerful primitives for implementing concurrency
- *Algebraic effects*

Demo

Status

Prototype based on OCaml 4.02.1 (by Stephen Dolan)

- Install it using the OCaml Package Manager (OPAM):

```
$ opam remote add ocamllabs git://github.com/ocamllabs/opam-
$ opam switch 4.02.1+multicore
```

- Byte-code only at the moment
- GC needs testing, tuning and benchmarking
- Some features broken (weak references, finalizers, lazy values)
- When you (inevitably) find bugs, report them to
  http://github.com/ocamllabs/ocaml-multicore