PLASMA

Programming Language

Paul Bone

29th of August 2016

http://plasmalang.org

Quick facts

Paradigm:
Purely functional, effects are controlled by **Resources**.

Typing discipline:
Strong, Static, ADTs, Parametric polymorphism, **Interfaces** and probably Higher kinded types

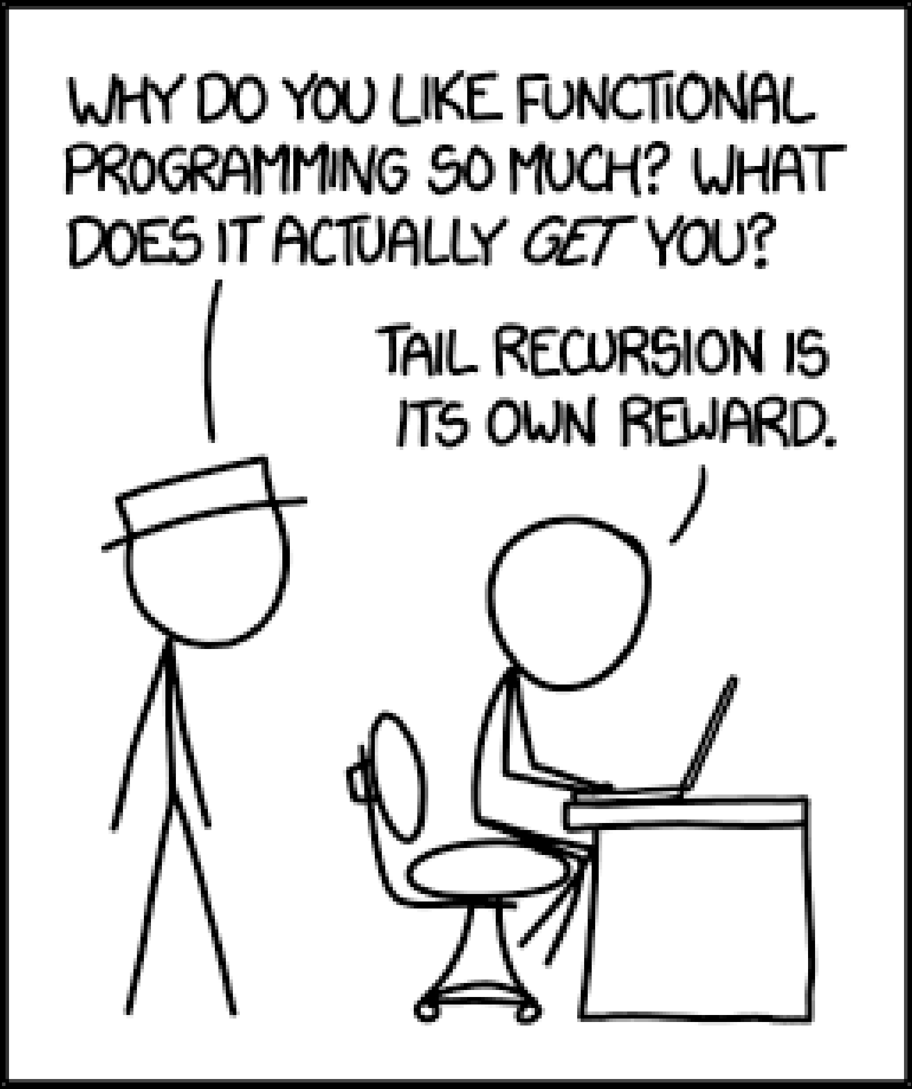
Evaluation discipline:
Strict

Runtime:
Custom virtual machine and in the future native code generation

Interoperability:
FFI to interoperate with C libraries

Functional programming is great, but...

> *Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.*
>
> — Randall Munroe (xkcd #1270)



Pure functional programming is expressive, safer and offers reasonable performance. But it is often very **weird and overly abstract**.

Goals

1. Combine **declarative** and **imperative** programming features.
    - Safety guarantees of strongly typed pure FP.
    - Pure FP is easier to reason about.
    - Imperative-like syntax is familiar for FP novices.
    - Loops, arrays and other imperative programming features benefit both experienced developers and novices.

Goals

2. Simplicity

   Keeping things simple is an excellent engineering practice. It also makes the language and tools easier to understand.

   - Reduce the emphasis and dependence on abstract concepts like monads. Allow them to be **learnt gradually**.
   - Sensible names: `Mappable` rather than ~~`Functor`~~.
   - Consistent syntax: things that **are** different will **look** different.
   - Good tooling.

Goals

3. Excellent **parallelism** and **concurrency** support.

   Channels, mvars, semaphores, streams, futures and STM provide safer abstractions than traditional threads and locks concurrency.

   Deterministic parallelism makes parallelism available without constraining the structure of your program or affecting its declarative semantics. Eg: Haskell's `par` function or *strategies*.

   Automatic parallelism introduces deterministic parallelism as a compiler optimisation.

Hello world!

```
module Hello

export main
import io

func main() -> Int using IO {
  io.print!("Hello world!")
  return 0
}
```

Resources are used to manage effects. A function call with an effect has an **annotation** (**!**) to warn anyone reading the code.
The compiler will check that the suitable resource **is available** in this function.

Resources

- Resources can be **used** or **observed**. Statements that observe the same resource may be re-ordered.
- Different resources exist. Some, like `IO`, subsume others.
- Some resources can be linked to values, like file handles. These values must be **unique** (Not designed yet).
- Higher order code must handle resources correctly. Resource usage must be polymorphic.
- Thanks to **Peter Schachte** and his **Wybe** language for this idea

Statements

Variables are single assignment, once bound they cannot be rebound or shadowed.

```
c = 25
f = c*9/5 + 32
io.print!("25c is " ++ show(f) ++ "f\n")
```

Is like writing let expressions in a language like OCaml:

```
let c = 25 in
  let f = c*9/5 + 32 in
    io.print!("25c is " ++ show(f) ++ "f\n")
```

Conditionals

Variables produced by the branching structure and used outside (**r**), must be produced on **all** branches.

```
if (cond) {
  x = ...
  r = f(x)
} else {
  r = ...
}

io.print!("Result is " ++ show(r) ++ "\n")
```

x is local to the first branch.

Conditionals

This does not apply to branches that do not *fall through*.

```
maybe_file = open!(filename, mode)
match (maybe_file) {
  case Ok(file) -> { }
  case Error(error) -> {
    return Error(error)
  }
}

result = process!(file)
close!(file)

return Ok(result)
```

Conditionals

This works easily for conditionals that produce multiple variables.

```
if (cond) {
  x = e1
  y = e2
} else {
  x = e3
  y = e4
}
```

Conditionals can also be used as expressions.

```
x, y = if (cond)
          then e1, e2
          else e3, e4
```

Loops

```
for [x <- xs] {
  y = f(x)
  output ys = list of y
}
```

Of course `map` can also be used. However loop syntax is both:

- familiar and
- very powerful for complex loops
- easier to parallelise

Loops are inspired by **SISAL**.

Loops

A loop may take any number of **inputs**, and generate any number of **outputs**.

```
for [x <- xs, y <- ys] {
    …
    output as = list of a
    output bs = array of b
}
```

Outputs can also be **reductions**. They reduce a sequence of values into a single value.

```
    output maximum = max of x
    output total = sum of y
```

Loops

Pass values between loop iterations with **accumulators**.

```
for [x <- xs] {
    accumulator warnings0 warnings initial []

    y, new_warnings = process(x)
    warnings = warnings0 ++ new_warnings

    output ys = list of y
    output warnings = value of warnings
}
```

This is just an example, it'd be better to use the `concat_list` reduction.

Loops

Valid loop inputs include lists, arrays, streams and **generators**.

Generators are implemented with coroutines, they can provide values from any source.

```
for [x0 <- xs, id <- count_from(0)] {
  x = add_id(x0, id)
  output xs_dict = dictionary of id, x
}
```

Returned items are also build using coroutines.

You can **define your own** generators and reductions.

Concurrency

- mvars
- semaphores

The basic concurrency primitives (mvars & semaphores) can be difficult to use, (but are better than locks).

However they are needed to build more advanced abstractions.

- readers / writers mvars
- read copy update mvars
- other multi-version abstractions

Concurrency

Several easier to use abstractions will also be available. These are not without their own drawbacks.

- channels
- futures
- green threads
- software transactional memory
- streams
- concurrent I/O

All of these have been proven to work for other languages. None of them are novel or risky.

We also have plans for **thread-aware garbage collection** in the future.

Software transactional memory

A transaction either completes, or is rolled back.

```
atomic {
  x = read!(stm_x)
  y = read!(stm_y)

  new_x = compute(x, y)
  update!(stm_x, new_x)

  z = ...
  update!(stm_z, z)
}
```

For example, if another thread modifies `stm_x` before this thread updates `stm_z` and completes the transaction, then this transaction will be rolled back.

Deterministic parallelism

Parallel evaluation that does not affect the declarative semantics of the program — the program **always produces the same results**.

In Haskell `par`, stratergies and `Monad.Par` all create deterministic parallelism.

C/C++ and Fortran support parallel loops with OpenMP.

```
#pragma omp parallel for
for(int x=0; x < width; x++) {
  for(int y=0; y < height; y++) {
    finalImage[x][y] = RenderPixel(x,y, &sceneData);
  }
}
```

Deterministic parallelism

SISAL supported parallel loops and stream processing. It further optimises its loops at compile time and **rivaled Fortran in performance**.

```
parallel for [x <- xs] chunk 20 {
  y = f(x)
  output ys = list of y
}
```

Plasma's loops and support for arrays and streams is inspired by **SISAL** (and also **Data Parallel Haskell**).

These code snippets are *pseduo-Plasma*, the actual syntax may be different.

Deterministic parallelism

```
parallel for [x <- xs] {
  y = f(x)
  output total = sum of y
}
```

This loop can be executed in parallel because `sum` can be split into independed sub-computations.

- addition is associtive: $A + (B + C) = (A + B) + C$
- addition has an identity element (zero)

In other words, addition is a monoid.

There are several other ways to parallelise reductions.

Deterministic parallelism

Of course, this loop could be parallelised without parallelising the reduction.

```
parallel for [x <- xs] {
  y = f(x)
  output ys = list of ys
}
for [y <- ys] {
  output total = sum of y
}
```

The best way to parallelise any code depends on the that specific code, and its typical data. Like most other optimisations, this should be **automatic** and preformed by the compiler.

Deterministic parallelism

We could create a **parallel stream** between two tasks.

```
parallel {
  task {
    parallel for [x <- xs] {
      y = f(x)
      output ys = stream of ys
    }
  }
  task {
    for [y <- ys] {
      output total = sum of y
    }
  }
}
```

Automatic parallelism

P. Bone, *Automatic Parallelisation for Mercury*, PhD Thesis, Department of Computing and Information Systems, The University of Melbourne, Australia, 2012.

Automatic parallelism

For Mercury we implemented **profiler feedback directed automatic parallelism**.

- We were able to automatically parallelise a sequence of dependent goals, and **account for their dependecies**.
- It also handled basic loops.

We will base Plasma's automatic parallelism on this work. Additionally:

- With Plasma's loops we can take this *much* further, and parallelise loops differently depending upon the **properties of their reductions and accumulators**.
- Recognize other forms of parallelism, such as stream processing.

Status

- ✓ Basic bytecode interpreter
- ✓ Basic compiler pipeline
- ✓ Hello world
- ✓ Basic expressions
- 🚧 Conditionals
- ✗ Loops
- ✗ Types
- ✗ Resources
- ✗ Parallelism and concurrency

Hard at work

Plasma is a labour of love, I work on it in my spare time.

How can I help?

Development is at an early stage and it may be unclear how to contribute.

- Feedback and support are incredibly welcome.
  Just letting us know that you want this to exist is helpful!
- Check out the reference manual, tell us if you find any problems.
- Try to build and run Plasma, including the tests (requires Mercury).
- There may items in `docs/todo.txt` that you can help with. We already have four contributors (including myself).
- Subscribe to the mailing lists and/or follow us on twitter to stay up-to-date.

About

Plasma

WWW
http://plasmalang.org

Twitter
@PlasmaLang

Mercury

WWW
http://mercurylang.org

Twitter
@MercuryLang

Paul Bone

WWW
http://paul.bone.id.au

Twitter
@Paul_Bone

Prince & YesLogic

WWW
http://princexml.com

WWW
http://yeslogic.com