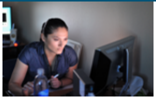
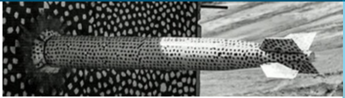
-7037C

National Laboratories

Functional First





Engineering a Research code in the OCaml Language

Presented by:

Kirk Landin

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell

dia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly ow subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

I Have a Dream...

Programming C++, I dreamed of a language that would make my life easier:

- Sound typing & better type inference
- No nasty template errors
- Better support for functional programming style
- Simple
- Very fast
- Doesn't crash

Dream Come True?

- Spoiler: nobody's dream language actually exists :-P
- A few years ago, we launched a research code using the OCaml language [1]
- Of all languages I have used, OCaml comes **closest to this dream language**.
- OCaml was great for developing this research code
- This talk relays some of my experience

The OCaml Language

- *Functional-First* language: FP concepts (immutability, function composition, algebraic data types, etc. [2]) are the core programming model
- ML family, cousin of the Haskell language [3]
- High-performance, statically-typed, machine-compiled
- Automatic memory management, garbage-collected
- Compact, UNIX-y
- State-of-the-art functional features **AND** sits close to the metal

This Research Code

- For performing automated software analysis
- Focused on two exemplar problems to build a framework that is easy to extend
- Implements *Abstract Interpretation* [4]
- 10-15k LOC

Why We Chose OCaml

- Long track record of success for compiler-style analyses
- Availability of software analysis libraries
- Functional-first Language
- Smaller learning curve than Haskell
- Very transparent compilation model
- "Sweet spot" in language design space:
  - Very expressive
  - Strongly typed with automatic type-inference
  - Compact and close-to-metal

Reasons not to Choose OCaml

- Unfamiliar to most people
- Small community & library ecosystem
    - Jane Street Capital [5] trying to fix this
- Sub-optimal Windows support
- Syntax could be better
- Lack of multiprocessor support (remedied in OCaml 5.0 [6])
- Other idiosyncrasies (true of any language)

Project Characteristics

- Code would constantly change direction
  - Good abstractions one week, next week research headed in a direction that broke them
  - Messy Code!
- Despite the churn, we always made steady progress.

Research Software Engineering

- Build a software system with little to no clue about what the finished product will do
- Flexibility is king!

Why was this Successful?

My Hypothesis: OCaml provides two important things:

1. Immutability
2. Type-system that serves as a high-level modeling language

Immutability

- “Mutable state is the GOTO statement of the 21st Century”
- Immutability makes referentially-transparent abstractions
- Behavioral complexity of and immutable component $\rightarrow$ at least an **order of magnitude smaller** than a comparable mutable component
- *Encapsulation* and *synchronization* $\rightarrow$ make small amounts of mutable state behave as if immutable
- Restricts us to a **reasonably well-behaved software regime**, even when programming blind

Modeling Domain Concepts as Algebraic Data Types

Express types exactly

```
type part = ...
type screw = ...
type nail = ...

type build_step =
  | Do_weld of part * part
  | Do_screw of screw * part
  | Do_rivet of Part * Part
  | Do_nail of nail * part
```

Partial knowledge or polymorphic types → free type variables

```
type ('part, 'screw, 'nail) build_step =
  | Do_weld of 'part * 'part
  | Do_screw of 'screw * 'part
  | Do_rivet of 'part * 'part
  | Do_nail of 'nail * 'part
```

With Types, Some Computations Write Themselves

```
type system_state = ...
run_step : build_step -> system_state -> system_state
type batch = build_step list
```

How do we run batch of steps?

```
let run_batch batch state = ???
```

- Type should be: `batch -> system_state -> system_state`
- Can program at the type level to figure things out
- Hint: use a standard function

```
foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

With Types, Some Computations Write Themselves

Type info

```
type system_state = ...
run_step : build_step -> system_state -> system_state
type batch = build_step list
foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Just assemble the types to get the desired operation

```
let run_batch batch state = foldl
  (fun st step -> run_step step st)
  state batch
```

Type-Checker Automatically Generates Models for Functions

```
let run_block block state = foldl
  (fun st instr -> run_instr instr st)
  state block
```

The compiler automatically deduces that its type is

```
batch -> system_state -> system_state
```

These Models Compose

- Pure functions are free of side-effects
  - Can soundly compose `A -> B` with `B -> C` to get `A -> C` without worrying about "extra behavior"
- Model composability is **extremely important**

Types Let us Index our Knowledge

Current domain knowledge is all labeled with appropriate types

With my current knowledge, can I derive `B` from `A`?

- Try to compose existing functions to build type `A -> B`
- If impossible:
  - Which sub-steps, `C -> D`, were missing?
  - Work to develop these sub-steps

- Reason about the problem at high level of abstraction
- Refactoring $\rightarrow$ Reason at type level. Compact representation of the system

Large-Scale Code Design with Immutable "Objects"

- OCaml language has *Modules* [7], which serve a similar role to classes
- Modules are essentially immutable classes/objects:
  - Constructor sets instance data, objects immutable after construction
  - Methods can reference an object's immutable instance data
  - Everything is pure/immutable
- "Immutable objects" approach works in almost any language

On-boarding/Mentoring People

- Biggest challenge when programming non-mainstream languages
- Success with one-on-one pair-programming
  - Work as two-person team: “Student” and “Teacher”
  - After awhile “Student” can work independently
  - Some of my “Students” went on to have their own “Students”
- Pair-programming approach also applicable to codes in mainstream languages

On-boarding/Mentoring People

- Two experienced functional programmers on team. Everyone else inexperienced
- *Foreign language syndrome*: affects some people more than others
- Would the project keep thriving if the experts left?

Pitch: Use C & OCaml in the Scientific Stack

- Clear separation of concerns Low-level: (C) vs High-level (OCaml)
- Do in C: optimized inner-loops & CUDA kernels
  - small percentage of total code
  - Very explicit about low-level tasks $\rightarrow$ understandable code
  - Doesn't seduce developer into building unsound abstractions
- Do in OCaml: Full system design, interfaces, refactoring, evolution
  - large percentage of total code
  - Automatic memory management
  - Automatic type inference & sound abstractions $\rightarrow$ understandable, productive, and agile
  - Still a very fast language
- Linking OCaml & C/C++ objects straightforward

Pitch: Use C & OCaml in the Scientific Stack

- My experience: **way more productive** in OCaml than in C++
- Major downside of C++: tries to mix **BOTH** low-level and high-level
  - Objects tend to hide "magic code", hard to understand/modify a system
  - Duck-typing because sound typing is impractical for C++ semantics
  - Unsound abstractions: component interfaces say they **should** work together but cause template errors/program crashes
  - Seduces developer into building unsound abstractions
- No single language to rule them all

References

[1] "Welcome to a world of OCaml." Available: https://ocaml.org/

[2] "Functional programming." Available: https://en.wikipedia.org/wiki/Functional_programming

[3] "Haskell." Available: https://www.haskell.org/

[4] "Abstract interpretation." Available: https://en.wikipedia.org/wiki/Abstract_interpretation

[5] "Jane street capital." Available: https://www.janestreet.com/

[6] "The road to OCaml 5.0." Available: https://discuss.ocaml.org/t/the-road-to-ocaml-5-0/8584

[7] "Chapter 2 the module system." Available: https://v2.ocaml.org/manual/moduleexamples.html