What FP Can Learn From Static Introspection

Aditya Siram

October 18, 2019

Outline

What?

- Static introspection
    - Reflection at compile time
        - Types, record field names, etc.
        - Does this compile?
    - Conditionally generate code
- Compile Time Function Evaluation
    - Term level language is available at compile time
    - For/while loops, functions, assignment
- Put them together!

Why?

- Typed functional languages need this!
- Performance
    - Play the optimizer/JIT compiler
    - No silver bullet
- Type level programming boring!
    - Accessible, maintainable, possible!
- Type level querying!
    - Library level, project specific tooling!
- Customizable compiler feedback
    - Control over error messages
    - Library specific user experience

What?

- Examples in Nim and D
    - Imperative languages in the C++/Ada tradition
- Real World Examples!
- Learn not adopt!

Compile Time Evaluation In Nim

```nim
proc say_hello(s:string): string =
  when nimVM:
    "Hello to " & s & " at compile time!"
  else:
    "Hello to " & s & " at runtime!"

static:
  echo say_hello("Lambda World Cadiz")

echo say_hello("Lambda World Cadiz")
```

Compile Time Evaluation In Nim

```
$ nim c hello.nim
Hello to Lambda World Cadiz at compile time!
$ ./hello
Hello to Lambda World Cadiz at runtime!
```

Static Introspection In D

```d
import std.stdio;

struct S
{
  int anInt;
  string aString;
};

pragma(msg, __traits(allMembers,S));

void main() {}
```

```
$ dmd has_member.d
tuple("anInt", "aString")
```

Static Introspection In D

```d
import std.stdio;

struct S
{
  int anInt;
  string aString;
};

pragma(msg, typeof(__traits(getMember, S, "anInt")));

void main() {}
```

Static Introspection In D

```
$ dmd has_member.d
int
```

Static Introspection In D

```
import std.stdio;

struct S
{
  int anInt;
  string aString;
};

pragma(msg, typeof(__traits(getMember, S, "foo")));

void main() {}
```

Static Introspection In D

```
$ dmd members.d
members.d(9): Error: no property foo for type S
_error_
```

Static Introspection In D

```
import std.stdio;

class C
{
  int anInt;
  string aString;
};

pragma(msg, __traits(allMembers,C));

void main() {}
```

Static Introspection In D

```
$ dmd members.d
tuple("anInt", "aString", "toString", "toHash",
      "opCmp", "opEquals", "Monitor", "factory")
```

Performance

- Performance advantages of CTFE!

Performance

- Reading (21,000 line) CSV at compile time in Nim

```nim
proc readCsv(s:string): seq[seq[string]] =
  var p: CsvParser
  p.open(newStringStream(s),"input")
  while p.readRow():
    result.add(p.row)

const parsed = readCsv(staticRead("large.csv"))
```

Performance

- 10 second compile time
- Lookup is instant

Performance

- Not always worth it!
- 22MB binary vs. 2.4MB CSV
- Only 1.5 seconds to compile time for runtime parsing

```
# const parsed = readCsv(staticRead("large.csv"))
let parsed = readCsv(readFile("large.csv"))
```

- Initial runtime parse < 1 second
    - Compile time processing is much slower!

Performance

- D's std.regex
- Runtime regex

  ```
  auto r = r"...";
  ```

- Compile time regex

  ```
  auto r = ctRegex!(`...`);
  ```

  - Highly specialized compile time generated engine

Performance

- Test regex (primality tester)

  `^(11+?)\1+`

  - `"1111..."`
  - No match if # of '1's is prime
  - Abigail (Perl)
- Backtracks a ton
- 104729 (10,000th prime)

Performance

- Both took 2.5 minutes
- Almost no difference in performance. :(
- Performance is hard . . .
  - CTFE is not a silver bullet
  - Compile times vs. one time runtime hit
  - Increased Binary sizes
- Really need to measure

Performance

- The real MVP is common code and quickly toggle
- Measurement is possible
  - At worst you've lost a few days of work ...
  - And you have a runtime library
- Would you attempt this another typed FP language?
  - Compile time regex in Haskell?
  - Could you throw it away?

Fast Domain Specific Lookup

- Fast lookups!
- Look up fields in a domain specific way

Fast Domain Specific Lookup

- Object in Nim

```
type
  O1 = object
    o1user_id : int
    o1Ids : seq[int]
    o1age: int
    o1user_address : string
```

Fast Domain Specific Lookup

- Gather the fields and types

```
proc gatherFields(t:typedesc): seq[(string,string)] =
  var o : t
  for n,v in fieldPairs(o):
    result.add((n,$v.type))
```

Fast Domain Specific Lookup

- Run it!

```
static:
  let o1 = gatherFields(O1)
  echo o1
```

- Outputs

```
$ nim c fieldPairs
@[("o1user_id", "int"), ("o1ids", "seq[int]"),
  ("o1age", "int"), ("o1user_address", "string")]
```

Fast Domain Specific Lookup

- Big deal!
  - Language REPL is enough
  - GHCi, ':i'

Fast Domain Specific Lookup

- Object

```
type
  O1 = object
    o1user_id : int
    o1Ids : seq[int]
    o1age: int
    o1user_address : string
```

- I know there's *some* kind of "ids" like field
  - Of type 'seq[int]'

Fast Domain Specific Lookup

- Filter it!

```
static:
  let o1 = gatherFields(O1)
  echo o1.filterIt(it[0].toLower.contains("ids")
                and it[1] == $seq[int])
```

- Output

```
$ nim c fieldPairs
@[("o1Ids", "seq[int]")]
```

Fast Domain Specific Lookup

- Make domain specific tooling
  - Fits your project!
- Tiny (throwaway) tool that does one thing
  - For one specific instance
- Need a lot of work to get this in an IDE

Datatype Diffing

- Datatype diffing
    - What fields were added/removed between two versions of a datatype?
- Hugely important
- Especially when serialization becomes involved

Datatype Diffing

- Another object

```
type
  O2 = object
    id : int
    ids : seq[int]
    age: int
    address : string
    email: string
```

Datatype Diffing

- First object

```
type
  O1 = object
    o1user_id : int
    o1Ids : seq[int]
    o1age: int
    o1user_address : string
```

Datatype Diffing

- Massage the fields

```
static:
  let o1 = gatherFields(O1)
  var o1stripped : seq[(string,string)]
  for f in o1:
    var s = f[0].toLower
    s.removePrefix("o1")
    s.removePrefix("user")
    s.removePrefix("_")
    o1stripped.add((s,f[1]))
```

Datatype Diffing

- Do the diff!

```
static:
  ...
  let o2 = gatherFields(O2)
  echo o1Stripped.toHashSet - o2.toHashSet
  echo o2.toHashSet - o1Stripped.toHashSet
```

- Output

```
$ nim c fieldPairs
{}
{("email", "string")}
```

  - 'O2' added an 'email' field

Datatype Diffing

- Reliably do datatype migration
  - Same as database migration!
- Testable and human inspectable
- Crucial to {de}serializing
  - Especially when backwards compatibility is important

Compile Time Type Reflection

- Compile time JSON parsing
- Demo!

Compile Time Type Reflection

- Reflect on the structure of sample data
- Find inconsistencies!

```
{
  ...
  "grades" : [100,80,50,33.3]
}
```

- Generate API reports
- Communicate with frontend team
  - "What type does this map to?"

Compile Time Type Reflection

- We can do better!
- Compile time type reflection!
- Demo!

Compile Time Type Reflection

- Ring any bells?

Compile Time Type Reflection

- Not in F#
  - Generate a type from a composite of samples
- Quickly manage external API specs

Type Safe Printf

- Static introspection for a type safe printf!
- Demo!

Type Safe Printf

- Domain specific, user customizable holes!
- Closes the feedback loop between the computer and user
- Why Google?

Problems

"A good science fiction story should be to predict not the automobile but the traffic jam." – Frederick Pohl

Problems

- Performance
  - Please measure . . .
- Loss Of Modularity
  - Horribly coupled types
  - Refactorable only in theory
- "God" object
- Parametricity
  - This function now has infinite implementations

    ```
    id :: a -> a
    ```
  - "Sealed" types with explicit unsealing?
  - Tracked by the type system?
- IDE support
  - What should the IDE fill in for the '_'.

    ```
    proc p(a : T) =
      when T is _ :
    ```
  - Cripples any predictive ability

Conclusion

- . . . but it's still worth exploring
- Need type level reflection
- Flexible interfaces
- Type safe string formats
    - printf
    - type safe URIs
- Ability to directly query your codebase
    - Better and more flexible software
- Granular tooling support
    - At the library/module level