What FP Can Learn From Static Introspection

Aditya Siram

July 29, 2020

Outline

What?

- Static introspection
  - What is the type of `1 + 1`
  - Does `1 + "hello world"` compile?
  - What are the fields of an object?
- Conditionally generate run time code

What?

- Compile Time Function Evaluation
  - Compile time and runtime language are the same!
  - For/while loops, functions, assignment
- Put them together!
- Typed functional languages need this!

Why?

- Performance
  - Play the optimizer/JIT compiler
  - No silver bullet

Why?

- Type level programming boring!
    - Accessible, maintainable, possible!
- Type level querying!
    - Library level, project specific tooling!

Why?

- Customizable compiler feedback
  - Control over error messages
  - Library specific user experience

Why?

- Changes how you think about system design!
  - Structural looseness

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("BuzzConf")

echo say_hello("BuzzConf")
```

Compile Time Evaluation In Nim

- At compile time . . .

```
$ nim c hello.nim
Hello to BuzzConf at compile time!
```

Compile Time Evaluation In Nim

- And when you run it . . .

```
$ ./hello
Hello to BuzzConf at runtime!
```

Compile Time Evaluation In Nim

- No compile time information in the binary!

```
> grep "at compile time" ./hello
> grep "at runtime" ./hello
Binary file hello matches
```

Static Introspection In D

- Static introspection in front+center in D
    - Core to D program design as sum types
    - First language?

Static Introspection In D

```
import std.stdio;

struct S
{
  int anInt;
  string aString;
};

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

void main() {}
```

Static Introspection In D

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

Static Introspection In 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

```d
import std.stdio;

struct S
{
  int anInt;
  string aString;
};

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

void main() {}
```

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

Static Introspection In D

```d
import std.stdio;

class C
{
  int anInt;
  string aString;
};

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

void main() {}
```

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

Performance

- Performance advantages of CTFE!
    - Benefits can be real but perf is fickle
- The best optimizations efforts can be:
    - Measured
    - Ditched

Performance

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

- Lookup is instant
- Not always worth it!
    - 10 second compile time
    - 22MB binary vs. 2.4MB CSV

Performance

- Switch to runtime parsing

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

- Only $<$ 1 second compile time
  - Compile time processing is much slower!
- 170 Kb binary
- Initial runtime parse $<$ 1 second
- Backing out was the right call!

Performance

- Regexs in D
  - D's std.regex
  - Highly specialized compile time generated engine
- Runtime regex

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

- Compile time regex

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

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. :(
- PCRE is still faster!

Performance

- The real benefit is ability to walk away
- Measurement is possible
    - At worst you've lost a few days of work ...
    - And you have a runtime library
- Selectively enable

Fast Domain Specific Lookup

- Improving developer productivity
- 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 <u>and</u> types
  - Just like D's `allMembers`

```
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

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

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

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 <u>lot</u> of work to get this in an IDE
  - Check out libclang

Datatype Diffing

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

Datatype Diffing

- Two versions of a type evolved over time ...

```
type                                    | type
  O1 = object                           |   O2 = object
    o1user_id : int                     |     id : int
    o1Ids : seq[int]                    |     email: string
    o1age: int                          |     ids : seq[int]
    o1user_address : string             |     age: int
                                        |     address : string
```

- What are the differences?
  - Semantically new information
  - Don't care about 'o1'

Datatype Diffing

- Massage the fields at compile time!

```
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

- Cleaned up fields

```
o1user_id : int            | id : int
o1Ids : seq[int]           | ids : seq[int]
o1age: int                 | age: int
o1user_address : string | address : string
```

Datatype Diffing

- Do the diff!

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

- Output
  - 'O2' added an 'email' field

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

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 in Nim

Compile Time Type Reflection

- Parse this JSON, sample data:

```json
{
  "id": "a12345",
  "age": 30,
  "address": {
    "street": "10 Main St.",
    "zip": 54321
  },
  "grades": [100, 80, 50]
}
```

Compile Time Type Reflection

- Parse it into Nim:
  - Reuse the JSON std lib at compile time!

```
let sample {.compileTime.} = %*
  {
    "id": "a12345",
    "age": 30,
    "address": {
      "street": "10 Main St.",
      "zip": 54321
    },
    "grades": [100, 80, 50]
  }
```

Compile Time Type Reflection

- Convert it to a Nim tuple with a macro:

```
let sampleTuple =
  ( id: "a12345",
    age: 30,
    address:
      ( street: "10 Main St.",
        zip: 54321
      ),
    grades: @[100, 80, 50]
  )
```

Compile Time Type Reflection

- Ask the type of that tuple and print it (at compile time!)

```
static:
  echo sampleTuple.type
```

Compile Time Type Reflection

- Prints the inferred type

```
tuple [
  id: string,
  age: int,
  address: tuple[
    street: string,
    zip: int
  ],
  grades: seq[int]
]
```

Compile Time Type Reflection

- Sample data to type using a standard macro + one-liner:

```
{                       |tuple [
  "id": "..."           |   id: string,
  "age": ..             |   age: int,
  "address": {          |   address: tuple[
    "street": "..."     |     street: string,
    "zip": ...          |     zip: int
  },                    |   ],
  "grades": [...]       |   grades: seq[int]
}                       | ]
```

Compile Time Type Reflection

- Real Data!

```
let real = %* # <-- no {.compileTime.}
  {
    "id": "11111-xxxx-1111",
    "age": 25,
    "address":
      {
        "street": "10 Some Real Street",
        "zip": 12345
      },
    "grades" : [10,10,10]
  }
```

Compile Time Type Reflection

- Parse it with that type

```
echo to(real,sampleTuple.type)
Output:
-------
( id: "11111-xxxx-1111",
  age: 25,
  address:
    ( street: "10 Some Real Street",
      zip: 12345
    ),
  grades: @[10, 10, 10]
)
```

Compile Time Type Reflection

- It catches things that are easy to miss!
  - If 'grades' had a float, the type is `seq[float]`

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

Compile Time Type Reflection

- Approximates F# JSON type provider in a few lines!
  - Generate a type from a composite of samples
- Generate API reports
- Communicate with frontend team
  - “What type does this map to?”
- Eliminates time consuming error-prone manual labor
  - But doesn’t take away control
  - Parse `id` as a Social Security #

Type Safe Printf

- Static introspection for a type safe printf in Nim
- A wrong 'printf':

```
printf("some string %s, some number %d", 1, "astring")
```

- Generates the error

  ```
  Error: Argument of type int can't be used with
  format specifier %s.
  ```

- All compile time eval + macros + static introspection

Type Safe Printf

- The heart of 'printf' is a standard switch statement:

```
case c
of "%d" :
  if argIs(int): # <-- magic!
    ... # convert int to string
  else:
    err("%d")
of "%s" :
  ...
```

Type Safe Printf

- 'getType' is the magic!

```
template argIs(t:typedesc): bool =
  typeKind(getType(args[curr])) == typeKind(getType(t))
           ^^^^^^^^^^^^^^^^^^^              ^^^^^^^^^^
```

Type Safe Printf

- 'getType' is the magic!

```
argIs(           ): bool =
  getType(args[curr])  ==           getType(t)
  ^^^^^^^^^^^^^^^^^^^               ^^^^^^^^^^
```

Type Safe Printf

- I wrote the error!

```
template err(s:string) =
  error ("Argument of type "
        & $getTypeImpl(args[curr])
        & " can't be used with format specifier "
        & s
        & ".", args[curr]
       )
```

Type Safe Printf

- I wrote the error!

```
       err(        ) =
error ("Argument of type "
       & ...
       & " can't be used with format specifier "
       & ...

      )
```

Type Safe Printf

- Can even provide help instead of just errors!
  - Add a '%_' wildcards to 'printf'.

```
printf("some string %_, some number %d", "astring", 1)
> Error: Try %s
```

Type Safe Printf

- Just one more switch statement:

```
case c
of "%d" : ...
of "%s" : ...
of "%_":
  if argIs(string):
    help("%s")
  elif argIs(int):
    help("%d")
```

Type Safe Printf

- Closes the feedback loop between the computer and user
- Domain specific DSLs with domain specific user experience
- Why Google?
- Why read docs?

Conclusion

- Need to query a codebase like a database
- Type safe string formats
    - printf (or anything you like)
- Granular tooling support
    - At the library/module level
- Treat types as heterogenous data
    - Not as canonical perfect things
    - Better and more flexible software