From the Tagless-Final Cookbook

Embedding and Optimizing (Hardware) Domain-Specific Languages in the Typed Final Style

http://okmij.org/ftp/tagless-final/course/index.html

Metaprogramming Summer School 2019
Dagstuhl, August 13-15, 2019

Tagless-Final

Tagless-Final

What's new

A variant, but not repetition

of an earlier course (CUFP 2015, Metaprogramming Summer School 2016)

The same DSL of combinational circuits

but developed at a faster pace and to a larger extent

Semantic rather than syntactic

Focus on denotations rather than term rewriting

Goals

- Introduce the Tagless-final style on a familiar example: combinational circuits
- Show off the features of the approach and design choice
- Introduce various optimizations

Optimizing EDSL in the typed final style is not only possible: it is modular and systematic

You can do it!

Which language?

OCaml, Haskell, Scala, ..., Coq, ...

Overview

Interactivity

- Please do ask questions
- I will ask questions
- Interactive writing of code (me vs. OCaml)
- Several exercises to do in class (and homework)
- Work alone or in group
- Installed OCaml?
  `http://try.ocamlpro.com/`

Problems

- A DSL for basic logical circuits (AND/OR/NOT)
- Various interpreters
- Compiler to NAND circuit
- Simplification and other transformers
- Conversion to CNF in one easy step
- From gates to circuits
- Circuit optimization, composionally
- Adding Gates (higher-order), reusing previous transformation rules

Real-life application: efficient language-integrated query

The web page of the approach (Tutorials, applications, etc.)
`http://okmij.org/ftp/tagless-final/course/index.html`

Similar, in spirit tutorial `http://okmij.org/ftp/tagless-final/course/optimizations.html`

Main ideas

- Multiple interpretations:
  write once, interpret many times
- Extensibility
- *Types*
  - typed implementation language
  - typed object language
  - typed optimization rules
  - connections with logic
- 'Final'
  - everything is in lower-case
  - prefer elimination over introduction
  - connections to denotational semantics

Main ideas

- Multiple interpretations:
  write once, interpret many times
- Extensibility
- *Types*
  - typed implementation language
  - typed object language
  - typed optimization rules
  - connections with logic

- Denotational
  - seek meaning
  - algebras
  - evaluation rather than rewriting

Compositionality

The meaning of a complex expression is determined by its structure and the meanings of its constituents.

`http://plato.stanford.edu/entries/compositionality/`

```
eval (Add e1 e2) = eval e1 + eval e2
```

- Evaluators and other interpreters are compositional
- Denotational semantics must be compositional
- Compositionality is modularity
- Compositionality is context-insensitivity
- Bottom-up reconstruction of meaning
- Compositional processing is `fold`

More problems, homework

- Assemblies, with multiple inputs and outputs
- Adder
- Sharing
- AND X X $\rightsquigarrow$ X
- Implement various simplifications

Outline

- **Conclusions**

Why Tagless Final style?

Thinking about meaning helps

- Algebraic perspective: focus on operations, on what we need to do (and how to add more: extensibility)
- Denotational perspective: focus on what we eventually want: eye on the prize rather than on word shuffling