Zero-overhead abstractions in Haskell using Staging

Haskell Love Conference

Andres Löh

2020-07-31

Well-Typed
The Haskell Consultants

A simple program

Binary search trees:

```
data BST a =
    Node Int a (BST a) (BST a)
  | Leaf
```

A simple program

Binary search trees:

```
data BST a =
     Node Int a (BST a) (BST a)
   | Leaf
```

“Standard” lookup:

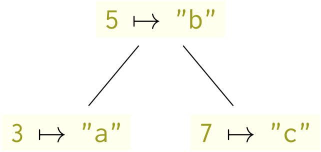
```
lookup :: Int -> BST a -> Maybe a
lookup _ Leaf             = Nothing
lookup i (Node j a l r) =
  case compare i j of
    LT -> lookup i l
    EQ -> Just a
    GT -> lookup i r
```

Well-Typed

A simple program (continued)

A statically known table (tree):

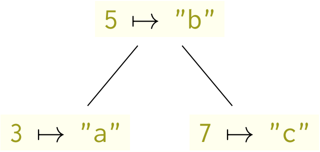
```
table :: BST String
table =
  Node 5 "b"
    (Node 3 "a" Leaf Leaf)
    (Node 7 "c" Leaf Leaf)
```



A simple program (continued)

A statically known table (tree):

```
table :: BST String
table =
  Node 5 "b"
    (Node 3 "a" Leaf Leaf)
    (Node 7 "c" Leaf Leaf)
```



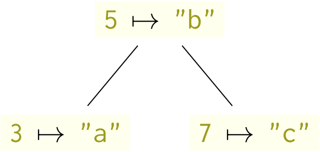
A specialised version of `lookup` :

```
lookupTable :: Int -> Maybe String
lookupTable i = lookup i table
```

A simple program (continued)

A statically known table (tree):

```
table :: BST String
table =
  Node 5 "b"
    (Node 3 "a" Leaf Leaf)
    (Node 7 "c" Leaf Leaf)
```



A specialised version of `lookup` :

```
lookupTable :: Int -> Maybe String
lookupTable i = lookup i table
```

Will the tree be optimised away?

No

Core (simplified)

```
lookupTable = \ i_a1fA -> lookup i_a1fA table

lookup
  = \ @a_a1gw ds_d1ml ds1_d1mm -> case ds1_d1mm of {
      Node j_auA a1_auB l_auC r_auD ->
        case ds_d1ml of wild1_a1mS {I# x#_a1mT ->
        case j_auA of {I# y#_a1mW ->
        case <# x#_a1mT y#_a1mW of {
          __DEFAULT ->
            case ==# x#_a1mT y#_a1mW of {
              __DEFAULT -> lookup wild1_a1mS r_auD;
              1# -> Just a1_auB
            };
          1# -> lookup wild1_a1mS l_auC
        }}};
      Leaf -> Nothing
    }
```

Why not?

- Recursive functions are never inlined.
- There is fusion for lists (and a handful of other types) . . .
- . . . but not for a tree type we just defined.

What if we want to exploit the static table?

Option 1: hand-unroll the code

```haskell
lookupTable :: Int -> Maybe String
lookupTable i =
  case compare i 5 of
    LT -> case compare i 3 of
            LT -> Nothing
            EQ -> Just "a"
            GT -> Nothing
    EQ -> Just "b"
    GT -> case compare i 7 of
            LT -> Nothing
            EQ -> Just "c"
            GT -> Nothing
```

Option 1: hand-unroll the code

```
lookupTable :: Int -> Maybe String
lookupTable i =
  case compare i 5 of
    LT -> case compare i 3 of
            LT -> Nothing
            EQ -> Just "a"
            GT -> Nothing
    EQ -> Just "b"
    GT -> case compare i 7 of
            LT -> Nothing
            EQ -> Just "c"
            GT -> Nothing
```

This is getting boring quickly …

Option 1: hand-unroll the code

```
lookupTable :: Int -> Maybe String
lookupTable i =
  case compare i 5 of
    LT -> case compare i 3 of
            LT -> Nothing
            EQ -> Just "a"
            GT -> Nothing
    EQ -> Just "b"
    GT -> case compare i 7 of
            LT -> Nothing
            EQ -> Just "c"
            GT -> Nothing
```

This is getting boring quickly …

Option 2: Type-level programming

In Haskell, we often move things that should be done statically into the types . . .

- Promote `BST` .
- Define `Lookup` as a type family?
- But we don't know the number statically . . .
- . . . thus we have to convert it using `someNatVal` or similar . . .
- . . . and everything gets more complicated . . .
- . . . and will this actually even end up being more efficient??

Option 2: Type-level programming

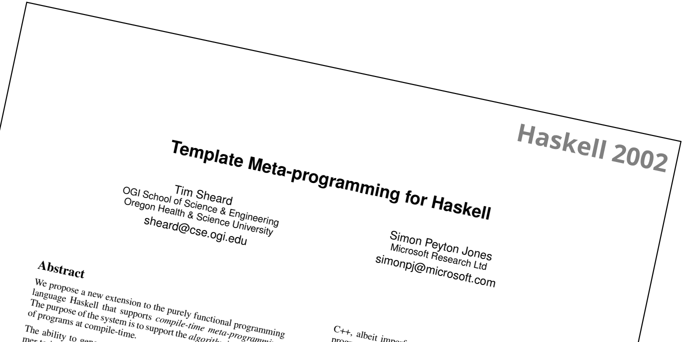
In Haskell, we often move things that should be done statically into the types . . .

- Promote `BST` .
- Define `Lookup` as a type family?
- But we don’t know the number statically . . .
- . . . thus we have to convert it using `someNatVal` or similar . . .
- . . . and everything gets more complicated . . .
- . . . and will this actually even end up being more efficient??

Option 3: Template Haskell

- Has a reputation for being low-level, dangerous, and difficult to maintain.
- Is untyped, and therefore **difficult to use**.

Primary use case: eliminating boilerplate.



Haskell 2002

Template Meta-programming for Haskell

Tim Sheard
OGI School of Science & Engineering
Oregon Health & Science University
sheard@cse.ogi.edu

Simon Peyton Jones
Microsoft Research Ltd
simonpj@microsoft.com

Abstract

We propose a new extension to the purely functional programming language Haskell that supports *compile-time meta-programming*. The purpose of the system is to support the *algorith*

of programs at compile-time.

The ability to gen

C++, albeit imperf

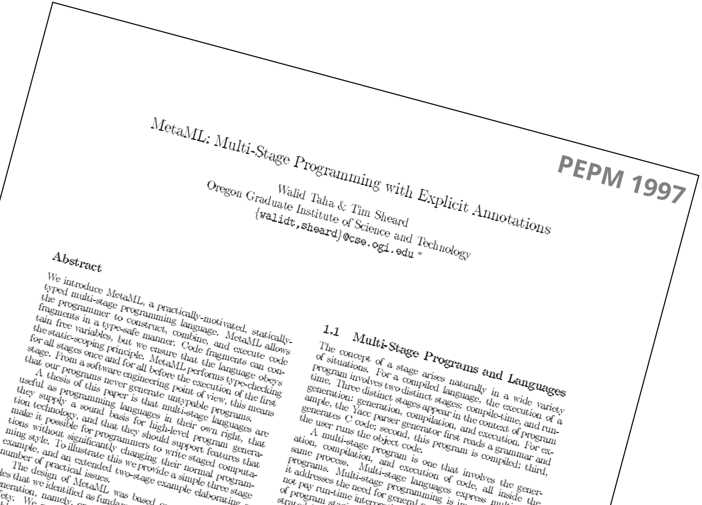
Option 3: Template Haskell

- Has a reputation for being low-level, dangerous, and difficult to maintain.
- Is untyped, and therefore **difficult to use**.

Primary use case: eliminating boilerplate.

Option 4: Typed Template Haskell

A **much more limited** variant of Template Haskell.



PEPM 1997

MetaML: Multi-Stage Programming with Explicit Annotations

Walid Taha & Tim Sheard
Oregon Graduate Institute of Science and Technology
{walidt,sheard}@cse.ogi.edu *

Abstract

We introduce MetaML, a practically-motivated, statically-typed multi-stage programming language. MetaML allows the programmer to construct, combine, and execute code fragments in a type-safe manner. Code fragments can contain free variables, but we ensure that the language obeys the static-scoping principle. MetaML performs type-checking for all stages once and for all before the execution of the first stage. From a software engineering point of view, this means that our programs never generate untypable programs.

A thesis of this paper is that multi-stage languages are useful as programming languages in their own right, that they supply a sound basis for high-level program generation technology, and that they should support features that make it possible for programmers to write staged computations without significantly changing their normal programming style. To illustrate this we provide a simple three stage example, and an extended two-stage example elaborating a number of practical issues.

The design of MetaML was based

1.1 Multi-Stage Programs and Languages

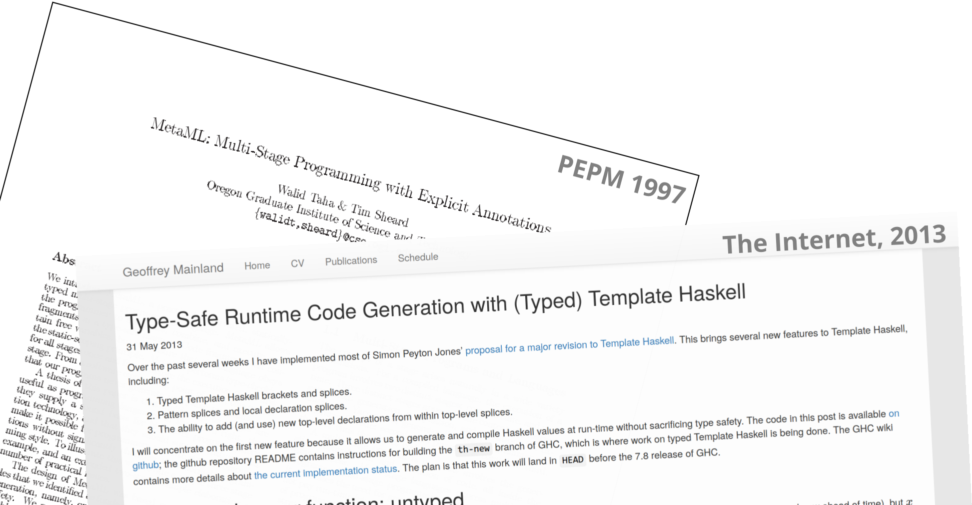
The concept of a stage arises naturally in a wide variety of situations. For a compiled language, the execution of a program involves two distinct stages: compile-time, and run-time. Three distinct stages appear in the context of program generation: generation, compilation, and execution. For example, the Yacc parser generator first reads a grammar and generates C code; second, this program is compiled; third, the user runs the object code.

A multi-stage program is one that involves the generation, compilation, and execution of code, all inside the same process.

Well-Typed

Option 4: Typed Template Haskell

A **much more limited** variant of Template Haskell.



PEPM 1997

MetaML: Multi-Stage Programming with Explicit Annotations

Walid Taha & Tim Sheard
Oregon Graduate Institute of Science and

The Internet, 2013

Geoffrey Mainland Home CV Publications Schedule

Type-Safe Runtime Code Generation with (Typed) Template Haskell

31 May 2013

Over the past several weeks I have implemented most of Simon Peyton Jones' proposal for a major revision to Template Haskell. This brings several new features to Template Haskell, including:

1. Typed Template Haskell brackets and splices.
2. Pattern splices and local declaration splices.
3. The ability to add (and use) new top-level declarations from within top-level splices.

I will concentrate on the first new feature because it allows us to generate and compile Haskell values at run-time without sacrificing type safety. The code in this post is available on github; the github repository README contains instructions for building the `th-new` branch of GHC, which is where work on typed Template Haskell is being done. The GHC wiki contains more details about the current implementation status. The plan is that this work will land in `HEAD` before the 7.8 release of GHC.

Option 4: Typed Template Haskell

A **much more limited** variant of Template Haskell.

How can this possibly be good?

- Typed.
- High-level interface.
- No `IO` at compile time.
- Generates only expressions, never top-level declarations.
- Can still access the power of normal TH underneath when really needed (akin to `unsafePerformIO`).

Option 4: Typed Template Haskell

A **much more limited** variant of Template Haskell.

How can this possibly be good?

- Typed.
- High-level interface.
- No `IO` at compile time.
- Generates only expressions, never top-level declarations.
- Can still access the power of normal TH underneath when really needed (akin to `unsafePerformIO`).

Primary use case: reliable performance!

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[|| e ||] :: Code t}}$$

Prevent reduction, build an AST.

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[|| e ||] :: Code t}}$$

Prevent reduction, build an AST.

```
type Code a = Q (TExp a)
```

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[|| e ||] :: Code t}}$$

Prevent reduction, build an AST.

Splices

$$\frac{\texttt{e :: Code t}}{\texttt{\$\$e :: t}}$$

Re-enable reduction, insert into an AST.

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[|| e ||] :: Code t}}$$

Prevent reduction, build an AST.

Splices

$$\frac{\texttt{e :: Code t}}{\texttt{\$\$e :: t}}$$

Re-enable reduction, insert into an AST.

Top-level splices insert into the current module.

Example: Staging `lookup`

```
lookup :: Int -> BST a -> Maybe a
lookup _ Leaf           = Nothing
lookup i (Node j a l r) =
  case compare i j of
    LT -> lookup i l
    EQ -> Just a
    GT -> lookup i r
```

The staged function is supposed to be invoked in a top-level splice, and thus to run at **compilation time** . . .

Example: Staging `lookup`

```
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf           = Nothing
lookup i (Node j a l r) =
  case compare i j of
    LT -> lookup i l
    EQ -> Just a
    GT -> lookup i r
```

Therefore, **dynamic** arguments are of `Code` type.

Example: Staging `lookup`

```
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf           = [|| Nothing ||]
lookup i (Node j a l r) =
  case compare i j of
    LT -> lookup i l
    EQ -> Just a
    GT -> lookup i r
```

Example: Staging `lookup`

```
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf             = [|| Nothing ||]
lookup i (Node j a l r) =
  [||
    case compare i j of
      LT -> lookup i l
      EQ -> Just a
      GT -> lookup i r
   ||]
```

Example: Staging `lookup`

```
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf           = [|| Nothing ||]
lookup i (Node j a l r) =
  [||
    case compare $$i j of
      LT -> lookup i l
      EQ -> Just a
      GT -> lookup i r
   ||]
```

Example: Staging `lookup`

```
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf           = [|| Nothing ||]
lookup i (Node j a l r) =
  [||
    case compare $$i $$(liftTyped j) of
      LT -> lookup i l
      EQ -> Just a
      GT -> lookup i r
   ||]
```

```
class Lift a where
  liftTyped :: a -> Code a

instance Lift Int
```

Well-Typed

Example: Staging `lookup`

```
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf           = [|| Nothing ||]
lookup i (Node j a l r) =
  [||
    case compare $$i $$(liftTyped j) of
      LT -> $$(lookup i l)
      EQ -> Just a
      GT -> $$(lookup i r)
   ||]
```

Example: Staging `lookup`

```haskell
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup _ Leaf           = [|| Nothing ||]
lookup i (Node j a l r) =
  [||
    case compare $$i $$(liftTyped j) of
      LT -> $$(lookup i l)
      EQ -> Just $$a
      GT -> $$(lookup i r)
   ||]
```

Note that stripping the staging constructs yields the original code.

Using staged lookup

```
table :: BST (Code String)
table =
  Node 5 [|| "b" ||]
    (Node 3 [|| "a" ||] Leaf Leaf)
    (Node 7 [|| "c" ||] Leaf Leaf)
```

```
lookupTable :: Int -> Maybe String
lookupTable i =
  $$(lookup [|| i ||] table)
```

Core again (simplified)

```
lookupTable = \ w_s4U0 ->
  case w_s4U0 of {I# ww1_s4U3 -> $wlookupTable ww1_s4U3}

$wlookupTable = \ ww_s4U3 ->
  case <# ww_s4U3 5# of {
    __DEFAULT ->
      case ww_s4U3 of wild_Xf {
        __DEFAULT ->
          case <# wild_Xf 7# of {
            __DEFAULT ->
              case wild_Xf of {
                __DEFAULT -> Nothing;
                7# -> lookupTable7
              };
            1# -> Nothing
          };
        5# -> lookupTable4
      };
    1# ->
      case <# ww_s4U3 3# of {
        __DEFAULT ->
          case ww_s4U3 of {
            __DEFAULT -> Nothing;
            3# -> lookupTable1
          };
        1# -> Nothing
      }
  }
```

```
lookupTable1 = Just lookupTable2
lookupTable2 = unpackCString# lookupTable3
lookupTable3 = "a"#

lookupTable4 = Just lookupTable5
lookupTable5 = unpackCString# lookupTable5
lookupTable6 = "b"#

lookupTable7 = Just lookupTable8
lookupTable8 = unpackCString# lookupTable9
lookupTable9 = "c"#
```

Core again (simplified)

```
lookupTable = \ w_s4U0 ->
  case w_s4U0 of {I# ww1_s4U3 -> $wlookupTable ww1_s4U3}

$wlookupTable = \ ww_s4U3 ->
  case <# ww_s4U3 5# of {
    __DEFAULT ->
      case ww_s4U3 of wild_Xf {
        __DEFAULT ->
          case <# wild_Xf 7# of {
            __DEFAULT ->
              case wild_Xf of {
                __DEFAULT -> Nothing;
                7# -> lookupTable7
              };
            1# -> Nothing
          };
        5# -> lookupTable4
      };
    1# ->
      case <# ww_s4U3 3# of {
        __DEFAULT ->
          case ww_s4U3 of {
            __DEFAULT -> Nothing;
            3# -> lookupTable1
          };
        1# -> Nothing
      }
  }
```

```
lookupTable1 = Just lookupTable2
lookupTable2 = unpackCString# lookupTable3
lookupTable3 = "a"#

lookupTable4 = Just lookupTable5
lookupTable5 = unpackCString# lookupTable5
lookupTable6 = "b"#

lookupTable7 = Just lookupTable8
lookupTable8 = unpackCString# lookupTable9
lookupTable9 = "c"#
```

- Equivalent to hand-unrolled code.
- Not relying substantially on GHC’s optimiser.
- (But still subject to optimisation.)

Another example: Routing in a web server

Example routes

```
/login
/language
/language/:lid
/language/:lid/new
/language/:lid/feature
/language/:lid/feature/:fid
/language/:lid/feature/:fid/since
```

...

Example routes

```
/login
/language
/language/:lid
/language/:lid/new
/language/:lid/feature
/language/:lid/feature/:fid
/language/:lid/feature/:fid/since
```

...

We are interested in efficient dispatch of a request to a handler.

Simplified scenario

```haskell
data Route =
    Static Text Route
  | Capture Route
  | End

data Router

at :: Route -> Handler -> Router
instance Semigroup Router
```

Simplified scenario

```
data Route =
     Static Text Route
   | Capture Route
   | End

data Router

at :: Route -> Handler -> Router
instance Semigroup Router
```

```
type Request = [Text]
type Handler = [Text] -> Response
type Response = Text

route :: Router -> Request -> Handler
```

Simplified scenario

```haskell
data Route =
    Static Text Route
  | Capture Route
  | End

data Router

at :: Route -> Handler -> Router
instance Semigroup Router

type Request = [Text]
type Handler = [Text] -> Response
type Response = Text

route :: Router -> Request -> Handler
```

We assume the `Router` to be statically known.

Staging routers

```haskell
data Router = MkRouter [(Route, Code Handler)]
```

Staging routers

```haskell
data Router = MkRouter [(Route, Code Handler)]
```

Build a suitable data structure:

```haskell
data RouteTree =
  RouteTreeNode
    (Map Text RouteTree)      -- dispatch on topmost path component
    (Maybe RouteTree)         -- routes that capture this component
    (Code Handler)            -- possibly failing handler for current path

buildRouteTree :: Router -> RouteTree
```

Staging routers

Use the tree to generate code:

```haskell
routeViaTree :: RouteTree -> Code Request -> Code [Text] -> Code Response
routeViaTree (RouteTreeNode statics captures handler) req args =
  [||
   case $$req of
     []     -> $$handler $$args
     x : xs -> $$(go (toList statics) captures [|| x ||] [|| xs ||])
   ||]
  where
```

Staging routers

```
  where
    go :: [(Text, RouteTree)] -> Maybe RouteTree -> Code Text ->
           Code Request -> Code Response
    go ((y, tree) : statics) _ x xs =
      [||
        if $$x == $$(liftTyped y)
          then $$(routeViaTree tree xs args)
          else $$(go statics captures x xs)
       ||]
    go [] Nothing x xs         = [|| "404" ||]
    go [] (Just captures) x xs =
      routeViaTree captures xs [|| $$x : $$args ||]
```

More staging

Applications of staging

Examples:

- optimising pipelines (fusion, streaming),
- parsing in all forms (pre-analyse grammar),
- printing / templating (constant folding),
- generic programming (specialising, removing intermediate representations),
- . . .

Applications of staging

Examples:

- optimising pipelines (fusion, streaming),
- parsing in all forms (pre-analyse grammar),
- printing / templating (constant folding),
- generic programming (specialising, removing intermediate representations),
- ...

Conjecture: **nearly any (E)DSL can be staged**.

Applications of staging

Examples:

- optimising pipelines (fusion, streaming),
- parsing in all forms (pre-analyse grammar),
- printing / templating (constant folding),
- generic programming (specialising, removing intermediate representations),
- ...

Conjecture: **nearly any (E)DSL can be staged**.

Promises much better and more reliable results than relying on **inlining**, **specialisation** and **rewrite rules**, all of which are brittle.

Staging techniques

1. Remove immediate overhead.
2. Exploit deeper knowledge by performing additional static analysis.
3. Make more fine-grained distinctions between static and dynamic data.

Interesting applications in Haskell

Generic programming

Matthew Pickering, Andres Löh, Nicolas Wu. **Staged Sums of Products**. Haskell 2020.

Parsing

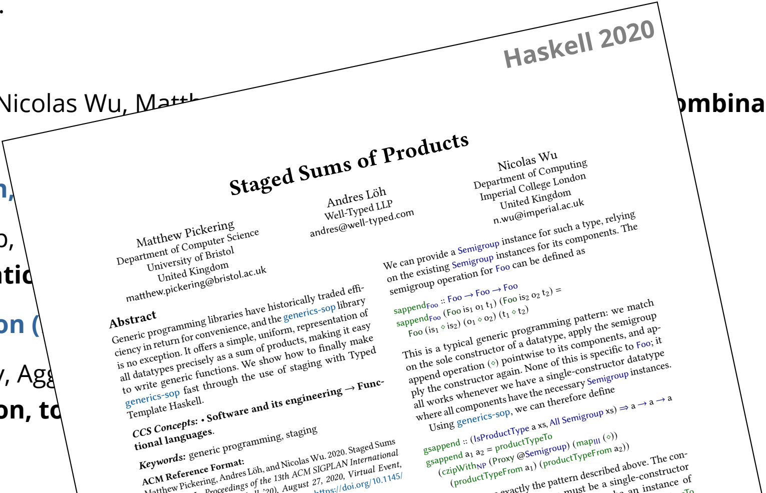
Jamie Willis, Nicolas Wu, Matth **ombinators**. ICFP 2020.

Composition,

Jeremy Yallop,
Partially-Static

Stream fusion (

Oleg Kiselyov, Agg
Stream fusion, to



Haskell 2020

Staged Sums of Products

Matthew Pickering
Department of Computer Science
University of Bristol
United Kingdom
matthew.pickering@bristol.ac.uk

Andres Löh
Well-Typed LLP
andres@well-typed.com

Nicolas Wu
Department of Computing
Imperial College London
United Kingdom
n.wu@imperial.ac.uk

Abstract

Generic programming libraries have historically traded efficiency in return for convenience, and the generics-sop library is no exception. It offers a simple, uniform, representation of all datatypes precisely as a sum of products, making it easy to write generic functions. We show how to finally make generics-sop fast through the use of staging with Typed Template Haskell.

CCS Concepts: • **Software and its engineering → Functional languages**.

Keywords: generic programming, staging

ACM Reference Format:
Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged Sums
Proceedings of the 13th ACM SIGPLAN International
ll '20), August 27, 2020, Virtual Event,
https://doi.org/10.1145/

We can provide a Semigroup instance for such a type, relying on the existing Semigroup instances for its components. The semigroup operation for Foo can be defined as

```
sappendFoo :: Foo → Foo → Foo
sappendFoo (Foo is₁ o₁ t₁) (Foo is₂ o₂ t₂) =
  Foo (is₁ ⋄ is₂) (o₁ ⋄ o₂) (t₁ ⋄ t₂)
```

This is a typical generic programming pattern: we match on the sole constructor of a datatype, apply the semigroup append operation (⋄) pointwise to its components, and apply the constructor again. None of this is specific to Foo; it all works whenever we have a single-constructor datatype where all components have the necessary Semigroup instances.

Using generics-sop, we can therefore define

```
gsappend :: (IsProductType a xs, All Semigroup xs) ⇒ a → a → a
gsappend a₁ a₂ = productTypeTo
  (czipWithNP (Proxy @Semigroup) (mapIII (⋄))
    (productTypeFrom a₁) (productTypeFrom a₂))
```

exactly the pattern described above. The con- must be a single-constructor an instance of

Interesting applications in Haskell

Generic programming

Matthew Pickering, Andres Löh, Nicolas Wu. **Staged Sums of Products**.
Haskell 2020.

Parsing

Jamie Willis, Nicolas Wu, Matthew Pickering. **Staged Selective Parser Combinators**.
ICFP 2020.

Composition, algebraic structures

Jeremy Yallop, Tamara von Glehn, Ohad Kammar.
Partially-Static Data as Free Extension of Algebras. ICFP 2018.

Stream fusion (MetaOCaml; stay tuned for a Haskell version)

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, Yannis Smaragdakis.
Stream fusion, to completeness. POPL 2017.

Conclusions

Staging can ensure code that is **obviously performing well**.

You can avoid the unpredictable (or unavailable) features of Haskell’s optimiser.

When will it be available?

In principle, now – and it has been since 2013(!).

When will it be available?

In principle, now – and it has been since 2013(!).

In practice, there are many things that Matthew Pickering has been working on to improve:

- Improving library interface.
- Improving soundness guarantees.
- Avoiding re-typechecking of generated code.
- Handling of type annotations.
- Disciplined handling of effects in code generation.
- Proper handling of class constraints.

Such issues can only be found and eliminated if staging is being used – **use staging**!