Zero-overhead abstractions in Haskell using Staging
Haskell Love Conference
Andres Löh
2020-07-31
Binary search trees:

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

"Standard" lookup:

```haskell
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
```
A simple program

Binary search trees:

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

“Standard” lookup:

```haskell
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
```
A statically known table (tree):

```
table :: BST String

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

```
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
lookupTable = \ i_a1fA -> lookup i_a1fA table
lookup = \ @a_a1gw ds_d1ml ds1_d1mm -> case ds1_d1mm of {
  Node j_a1uA a1_aub l_a1uC r_a1uD ->
  case ds_d1ml of wild1_a1mS {I# x#_a1mT ->
    case j_a1uA of {I# y#_a1mW ->
      case <# x#_a1mT y#_a1mW of {
        __DEFAULT ->
          case ==# x#_a1mT y#_a1mW of {
            __DEFAULT -> lookup wild1_a1mS r_a1uD;
            1# -> Just a1_a1uB
          });
        1# -> lookup wild1_a1mS l_a1uC
      }});
    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

`lookupTable :: Int -> Maybe String`

`lookupTable i =`

```haskell```
  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 …
```
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.
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.
A much more limited variant of Template Haskell.

Well-Typed
Option 4: Typed Template Haskell

A much more limited variant of Template Haskell.

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 splines
2. Pattern splines and total declaration splines
3. The ability to add (and use) new top-level declarations from within top-level splines

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 GHCi 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.

A thesis of this paper is that multi-stage languages are much more limited than their single-stage counterparts. A multi-stage program is one that involves the generation, compilation, and execution of programs across multiple stages.

MetaML: Multi-Stage Programming with Explicit Annotations

Well-Typed
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**).
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

\[
\begin{align*}
\text{e :: t} \\
[\text{|| e ||}] :: \text{Code t}
\end{align*}
\]

Prevent reduction, build an AST.
Prevent reduction, build an AST.

```
Quotes

\[
\begin{align*}
e &:: t \\
[|| \ e \ ||] &:: \text{Code} \ t
\end{align*}
\]

```

\text{type} \ \text{Code} \ a = Q \ (TExp \ a)
Staging constructs

**Quotes**

\[
e :: t
\]

\[
[| | e | |] :: \text{Code} \; t
\]

Prevent reduction, build an AST.

**Splices**

\[
e :: \text{Code} \; t
\]

\[
$$e :: t
\]

Re-enable reduction, insert into an AST.
Staging constructs

Quotes

\[
\begin{align*}
\text{e} &::\text{ t} \\
[|| \text{e} ||] &:: \text{Code t}
\end{align*}
\]

Prevent reduction, build an AST.

Splices

\[
\begin{align*}
\text{e} &::\text{ Code t} \\
\$\$\text{e} &::\text{ t}
\end{align*}
\]

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 \textbf{lookup}

\begin{verbatim}
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
\end{verbatim}

Therefore, \textbf{dynamic} arguments are of \textbf{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
```
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 :: 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 \textbf{lookup}

\begin{verbatim}
lookup :: Code Int -> BST (Code a) -> Code (Maybe a)
lookup Leaf = [|| Nothing ||]
lookup i (Node j a l r) =
  [||
    \textbf{case} compare $$i$$ $$($(liftTyped j)$)$$ \textbf{of}
    \text{LT} -> lookup i l
    \text{EQ} -> Just a
    \text{GT} -> lookup i r
  ||]
\end{verbatim}

\begin{verbatim}
class Lift a where
  liftTyped :: a -> Code a
\end{verbatim}

\begin{verbatim}
instance Lift Int
\end{verbatim}
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

```haskell
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 -> $lookupTable ww1_s4U3}

$lookupTable = \ 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)

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

$lookupTable = \ 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# -> lookupTable1
        };
        1# -> Nothing
    }
```

- 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

data Route =
  Static Text Route
  | Capture Route
  | End

data Router

at :: Route -> Handler -> Router

instance Semigroup Router
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
```
Simplified scenario

```hs
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)]
```

Build a suitable data structure:

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

Build a suitable data structure:

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
Use the tree to generate code:

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

Well-Typed
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.
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

Parsing

Composition, algebraic structures

Partially-Static Data as Free Extension of Algebras

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 Template Haskell.


Keywords: generic programming, staging

ACM Reference Format:

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:

\[
\text{append}_{\text{Foo}} : \text{Foo} \rightarrow \text{Foo} \rightarrow \text{Foo}
\]

This is a typical generic programming pattern: we match on the sole constructor of a datatype, apply the semigroup append operation \(\cdot\) 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 \text{generics-sop}, we can therefore define:

\[
\text{append} : (\text{toproductType } a x) \Rightarrow a \rightarrow a \rightarrow a
\]

The generic function \text{append} satisfies exactly the pattern described above. The constructor of the result must be a single-constructor constructor as well, and the components of the result will be of type \text{Semigroup}, just as in the example above.
Interesting applications in Haskell

**Generic programming**

**Parsing**

**Composition, algebraic structures**

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