Generalized Newtype Compiling:
Don't let your types slow you down!

Richard A. Eisenberg
Bryn Mawr College
rae@cs.brynmawr.edu

IFIP WG2.8
May 2019
Bordeaux, France

The Problem

Types help me reason.

😀

Types can slow my code down.

☹️

The Problem

strict types only!

```
data Nat where
  Zero :: Nat
  Succ :: !Nat → Nat
```

- Simple, user-defined datatype
- Easy reasoning
- Easy algorithms
- Easy induction
- Patently ridiculous

number stored as a linked list

The Problem

```
data Fin :: Nat → Type where
  FZ :: Fin (Succ n)
  FS :: Fin n → Fin (Succ n)
```

- Simple(ish), user-defined datatype
- Easy reasoning
- Easy algorithms
- Easy induction
- Patently ridiculous

The Problem

```
data Vec :: Type → Nat → Type where
  Nil  :: Vec a Zero
  (:>) :: a → Vec a n → Vec a (Succ n)
```

- Simple(ish), user-defined datatype
- Easy reasoning
- Easy algorithms
- Easy induction
- Not always best representation

use an array!

The Solution

Separate compile-time type from runtime representation

Warning: work in progress

Suggestions welcome!

Collaborative work with undergrad My Nguyen '20

(You want her as a PhD student!)

Running example: Nat

```
data Nat = Z | S !Nat
{-# TYPECHANGE
    Nat  <-> Int
    Z    <-> 0
    S     -> (1+)
    S n  <-  (isSucc → Just n)
    plus <-> (+)
 #-}
isSucc :: Int → Maybe Int
isSucc 0 = Nothing
isSucc n = Just (n-1)
```

view pattern

Running example: Nat

```
mul :: Nat → Nat → Nat
mul Z       _ = Z
mul (S n) m = plus m (mul n m)
```

C

mul should be in TYPECHANGE

```
mul :: Int → Int → Int
mul 0 _ = 0
mul (isSucc → Just n) m
   = (+) m (mul n m)
```

Compilation is well-typed

c lifts TYPECHANGE into the AST

Theorem:
If $\varnothing \vdash e : \tau$, then $\varnothing \vdash c(e) : c(\tau)$

Generalizes to non-empty contexts, too.

TYPECHANGE must be well-formed

Compilation respects semantics

~~Theorem:~~
~~If $e \longrightarrow^* v$, then $c(e) \longrightarrow^* c(v)$~~

Counterexample:

```
pred (S (S Z)) ⟶* S Z

(λx → x - 1) ((1+) ((1+) 0))
                          ⟶̸* (1+) 0
```

Compilation respects semantics

Theorem:

If $e \longrightarrow^* v$, then

$\exists\, v'$ s.t. $c(e) \longrightarrow^* v'$ and $c(v) \longrightarrow^* v'$

Unhelpful:

True if $c(e) = \top$

Compilation respects semantics

Theorem:

~~If $e \longrightarrow^* v$, then $c(e) \longrightarrow^* v'$ and $d(v') \longrightarrow^* v$~~

~~d undoes a~~ TYPECHANGE

Counterexample:

$$c(\texttt{S}) \longrightarrow^* \texttt{(1+)}$$

$$d(\texttt{(1+)}) \longrightarrow^* \texttt{plus (S Z)}$$

Compilation respects semantics

Theorem:
If $e \longrightarrow^* v$, then
$c(e) \longrightarrow^* v'$ and $d(v') \longrightarrow^* v''$

where $v \cong v''$.

observational equivalence

User is responsible for this for elements in TYPECHANGE.

Compilation respects semantics

Theorem:
If $e \longrightarrow^* v$, then
$c(e) \longrightarrow^* v'$ and $d(v') \longrightarrow^* v''$

where $v \cong v''$.

User is responsible for this for elements in TYPECHANGE.

In Haskell, just trust. (or QuickCheck)

In Coq/Agda/Idris/F*, prove.

Observation: generalized newtypes

A Haskell newtype is just a datatype with a TYPECHANGE.

NB: Haskell's newtypes are already strict.

Observation: patterns

```
data Nat = Z | S !Nat
{-# TYPECHANGE
    Nat  <-> Int
    Z    <-> 0
    S      -> (1+)
    S n  <-  (isSucc → Just n)
    plus <-> (+)
 #-}
```

Translating a pattern is like detranslating an expression.

Design consideration: strictness

- Lazy `Nat` includes infinity. This is inconvenient.
- Translation will not preserve laziness properties.
- Need induction to prove logical equivalence.

Design consideration: modularity

TYPECHANGE *must* be in defining module for a type.

- Avoids doing translation at runtime
- Avoids lifting translations through `Functor`, `Contravariant`, `Bifunctor`, etc.
- Restriction could be lifted, if necessary

Design consideration: dependent types

- Need compile-time type to be different from runtime type
- Need compile-time type to be in "lock-step" with runtime type

Answer: singletons!

Design consideration: dependent types

Answer: singletons!

- Compile-time performance still slow
- But nice reasoning principles of compile-time types are retained
- Worst practical aspect of singletons is runtime conversions: these are gone here.

Design consideration: indexed types

```
data Fin :: Nat → Type where
  FZ :: Fin (Succ n)
  FS :: Fin n → Fin (Succ n)
```

- How can we have informative pattern-matches?
- Possible solution: new form of runtime token, similar to coercions of equality
- Possible solution: `unsafeCoerce`

Target language is less richly-typed.

NB: GHC already drops newtype distinctions.

Design consideration: roles

```
type family F a where
  F Nat = Bool
  F Int = Char
```

- Disaster if we confuse `Nat` and `Int`.
- Solution: roles.
- Well-studied:
  - Weirich et al., POPL'11
  - Breitner et al., JFP'16
  - Weirich et al., ICFP'19

Design consideration: when to optimize

- Both linked lists and arrays are sensible representations for `Vec a n`.
- Use Haskell's newtype feature to select the representation in a type-directed way.

Existing approaches

- Module swapping: Have a `X.Y.Z.Safe` export and a `X.Y.Z.Unsafe` export with the same semantics (hopefully!).
    - Done in, e.g., Galois' *parameterized-utils*
    - Trouble when using data in types (promotion)
- Pattern synonyms
    - No clear notion of logical equivalence
    - Hard to do in practice

Related work

- Refinements for Free, Denes et al
- Brady's PhD student Matusz

BRYN MAWR COLLEGE

Generalized Newtype Compiling: Don't let your types slow you down!

Richard A. Eisenberg
Bryn Mawr College
rae@cs.brynmawr.edu

IFIP WG2.8
May 2019
Bordeaux, France