The pleasures and pain of advanced type systems

Stephanie Weirich
University of Pennsylvania

Or: how I learned to stop worrying and love a good type error
Static types *work*

Static typing is *by far* the most widely used program verification technology in use today.

- Lightweight (so programmers use them)
- Machine-checked (with every compilation)
- Ubiquitous (so programmers can’t avoid them)
Why do types work?

• Type errors identify bugs!
  – True + ‘c’
  – Memory & control-flow safety

• Types *specify* code. They say (to people) what functions do
  foozele :: Gizmo -> Gadget -> Contraption

• Types support interactive program development (Intellisense, Eclipse)

• Types support software maintenance (the **most important** benefit, seldom mentioned)
Haskell’s advanced type system

*Types work better for pure code*

\[ f :: [a] \rightarrow [a] \]
Type system Pain Pleasure

All programs (that do something)

Programs that do what you want

“Equivalent” to a well-typed program, but much easier to write

Programs that type check

Similar to a correct program but the type system can’t rule it out

“Dependent types” and EDSLs

Generic Programming
Haskell Metaprogramming

data Expr =
  CB Bool
  CI Int
  If Expr Expr Expr
  BinOp Op Expr Expr
  ...
  ...

deriving (Eq, Ord, Show, Read)

Automatic definition of equality, ordering, serialization functions
Generic Programming

deriving (Eq, …, Generic)

• Enables user-defined generic traversals
  – Operations defined over representations of the type structure, in a type-preserving way
  – Eliminates boilerplate code. Aids development & refactoring

• Examples:
  children (BinOp Plus e1 e2) == [e1; e2]
  freevars (BinOp Plus (Var “x”) (Var “y”)) == [“x”; “y”]
  freshen (If (Var “x”) (Var “y”) (Var “z”)) ==
    (If (Var “x0”) (Var “y0”) (Var “z0”))
arbitrary / shrink for random test generation
Dependent types, aka GADTs

data Expr a where
  CB :: Bool -> Expr Bool
  CI :: Int -> Expr Int
  If :: Expr Bool -> Expr a
      -> Expr a -> Expr a
  BinOp :: Op (a -> b -> c)
      -> Expr a -> Expr b -> Expr c

\[ t = \text{If} (\text{CI 3}) (\text{CI 4}) (\text{CI 5}) \]

Doesn’t type check now
Embedded Domain Specific Language

• Why define a DSL?
  – Specialize your development environment for your application
  – Reduced language, so fewer “wrong” program typecheck

• Why Embedded in Haskell?
  – Building a programming language is hard!
  – Dependent types can constrain embedded language, application-specific type checking
Ivory EDSL

• Low-level safe C-like language for safe systems programming
• DARPA research program for vehicle security
• Deeply embedded in Haskell, generates C, linked with RTOS and loaded onto quadcopter
-- | Convert an array of four 8-bit integers into a 32-bit integer.

test2 :: Def ('[Ref s (Array 4 (Stored Uint8))] :-> Uint32)

test2 = proc "test2" $ \arr -> body $ do
  a <- deref (arr ! 0)
  b <- deref (arr ! 1)
  c <- deref (arr ! 2)
  d <- deref (arr ! 3)
  ret $ ((safeCast a) `iShiftL` 24) .|
           ((safeCast b) `iShiftL` 16) .|
           ((safeCast c) `iShiftL` 8) .|
           ((safeCast d) `iShiftL` 0)
Quipper EDSL

• Embedded, scalable functional language for quantum computing
  – circuit description language
  – automatic synthesis of reversible quantum circuits

• Joint project between Dalhousie, Penn, IAS

http://www.mathstat.dal.ca/~selinger/quipper/
Unlimited possibilities

```haskell
import BASIC
main = runBASIC $ do
  10  LET X =: 1
  20  PRINT "Hello BASIC world!"
  30  LET X =: X + 1
  40  IF X <> 11 THEN 20
  50  END
```

The pain of types

- Programs that do what you want
- Programs that type check
- Sometimes the types still get in the way
Current research

Programs that do what you want
Programs that type check

Type-level computation, expresses a relationship between types as a program

“Real” dependent types, which allow program values in types

Type inference in the presence of these advanced features
Type-level computation

-- Diatonic fifths, and their class (comments with the CMaj scale)
-- See http://en.wikipedia.org/wiki/Circle_progression

type family DiatV deg :: *
type instance DiatV I   = Imp -- V   -- G7 should be Dom
type instance DiatV V   = Imp -- II  -- Dm7 should be SDom
type instance DiatV II  = VI   -- Am7
type instance DiatV VI  = III  -- Em7
type instance DiatV III = VII
    -- Bhdim7 can be explained by Dim rule
type instance DiatV VII = Imp -- IV
    -- FMaj7 should be SDom
type instance DiatV IV  = Imp -- I   -- CMaj7

http://hackage.haskell.org/package/HarmTrace-2.2.0
Not pain! Refactoring
Pain? Refactoring

• “Once it type checked...” heh, heh
• What about running tests *while* refactoring?
  ... even if the program doesn’t type check?
  ... even if parts of the program haven’t been written?

```javascript
newVersionOfMyFunction :: Widget -> Sprocket -> Assemblage
newVersionOfMyFunction = undefined
```
spaceman:- sweirich$ ghci -fdefer-type-errors
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Prelude> let x = (True, 'a' && False)
<interactive>:2:16:  Warning:
   Couldn't match expected type `Bool' with actual type `Char'
   In the first argument of `(&&)', namely 'a'
   In the expression: 'a' && False
   In the expression: (True, 'a' && False)
Prelude> :type x
x :: (Bool, Bool)
Prelude> fst x
True
Prelude> snd x
*** Exception: <interactive>:2:16:  WARNING:
   Couldn't match expected type `Bool' with actual type `Char'
   In the first argument of `(&&)', namely 'a'
   In the expression: 'a' && False
   In the expression: (True, 'a' && False)
(deferred type error)
Real Pain!

• Haskell is a research language, not supported by a major corporation
  – MSR will not invest more resources into it
• Open source (Yay!), fun for research (Yay!), but “infrastructure” things don’t get done
Questions?

thanks!