FUN WITH TYPE FUNCTIONS

Simon Peyton Jones (Microsoft Research)
Chung-Chieh Shan (Rutgers University)
Oleg Kiselyov (Fleet Numerical Meteorology and Oceanography Center)

Tony Hoare's 75th birthday celebration, April 2009

Getting it right

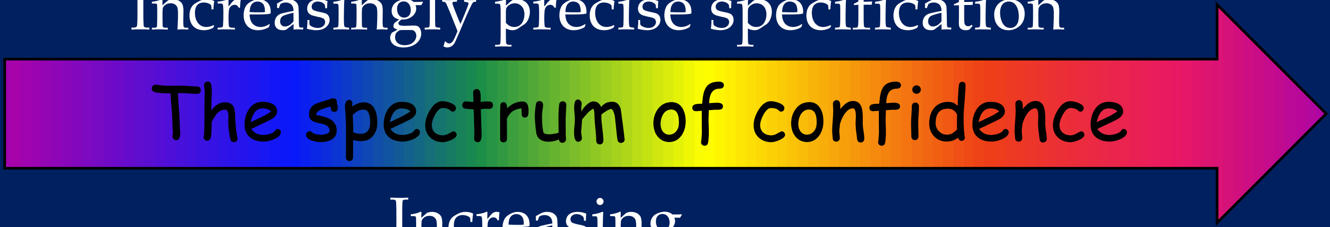
- “Program correctness is a basic scientific ideal for Computer Science”
- “The most widely used tools [in pursuit of correctness] concentrate on the detection of programming errors, widely known as bugs. Foremost among these [tools] are modern compilers for strongly typed languages”
- “Like insects that carry disease, the least efficient way of eradicating program bugs is by squashing them one by one. The only sure safeguard against attack is to pursue the ideal of not making the errors in the first place.”

“The ideal of program correctness”, Tony Hoare, BCS lecture and debate, Oct 2006

Tony was being cruel

- Static typing eradicates whole species of bugs
- The static type of a function is a partial specification: its says something (but not too much) about what the function does

  ```
  reverse :: [a] -> [a]
  ```



Increasingly precise specification

The spectrum of confidence

Increasing
confidence that the
program does what
you want

Tony was being cruel

- The static type of a function is like a weak specification: its says something (but not too much) about what the function does

  `reverse :: [a] -> [a]`

- Static typing is by far the most widely-used program verification technology in use today: particularly good cost/benefit ratio
  - Lightweight (so programmers use them)
  - Machine checked (fully automated, every compilation)
  - Ubiquitous (so programmers can't avoid them)

Tony was being cruel

- Static typing eradicates whole species of bugs
- Static typing is by far the most widely-used program verification technology in use today: particularly good cost/benefit ratio

Increasingly precise specification

The spectrum of confidence

Hammer
(cheap, easy to use, limited effectivenes)

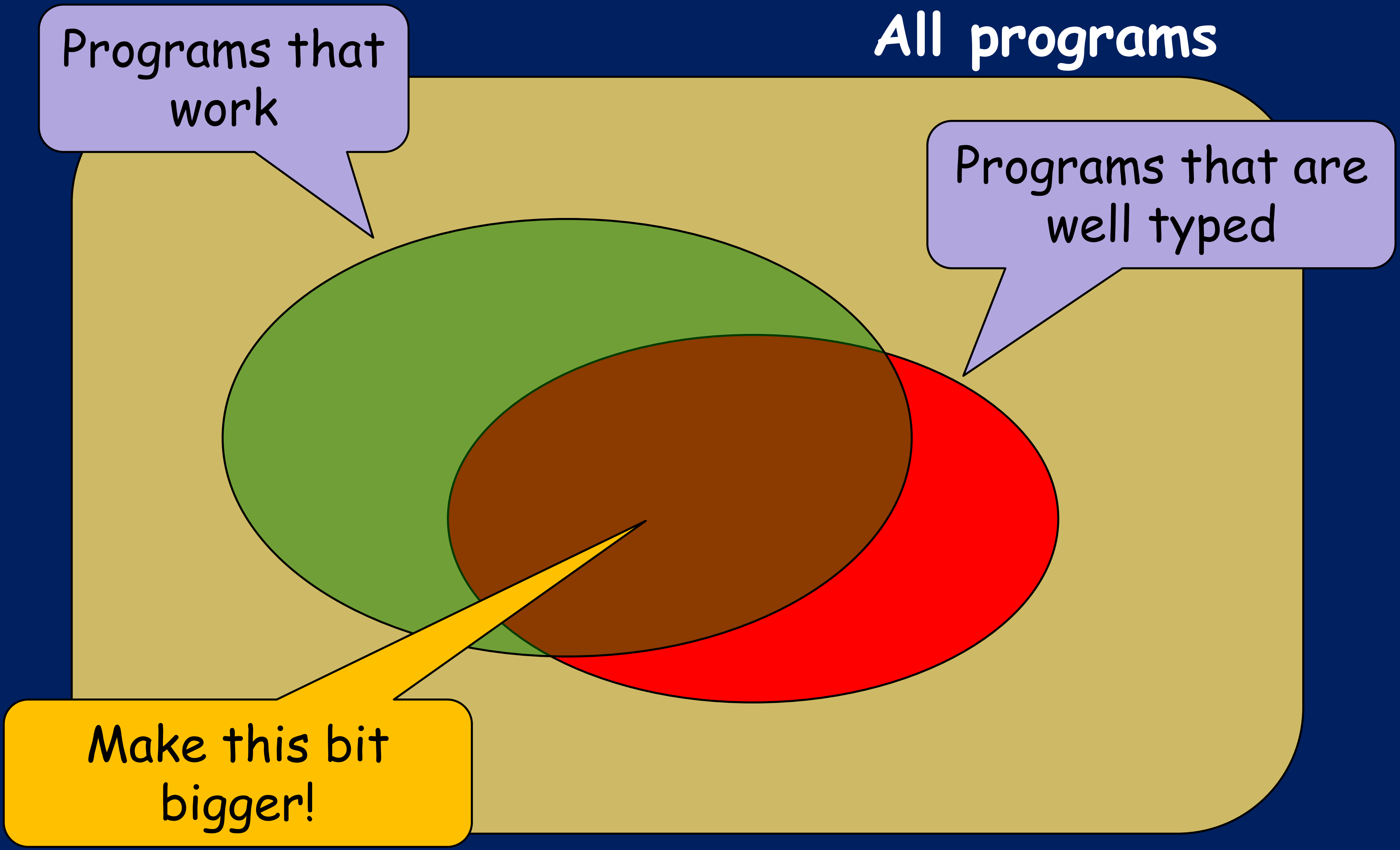
Increasing confidence that the program does what you want

Tactical nuclear weapon
(expensive, needs a trained user, but very effective indeed)

The type system designer

- The type system designer seeks to
- Retain the Joyful Properties of types
- While also:
  - making **more good programs** pass the type checker
  - making **fewer bad programs** pass the type checker

The business of types



All programs

Programs that work

Programs that are well typed

Make this bit bigger!

The type system designer

- The type system designer seeks to retain the Joyful Properties of types
- While also:
  - making **more good programs pass** the type checker
  - making **fewer bad programs pass** the type checker
- One such endeavour:

Extend Haskell with
Indexed type families

The type system designer

- The type system designer seeks to retain the Joyful Properties of ty
- While also:
  - making **more good programs** checker
  - making **fewer bad programs pass** the typ checker
- One such endeavour:

I fear that Haskell is doomed to succeed

Tony Hoare (1990)

Extend Haskell with Indexed type families

Type classes

Class decl gives type signature of each method

```
class Num a where
   (+), (*) :: a -> a -> a
   negate   :: a -> a

square :: Num a => a -> a
square x = x*x

instance Num Int where
   (+)      = plusInt
   (*)      = mulInt
   negate = negInt

test = square 4 + 5 :: Int
```

Instance decl gives a "witness" for each method, matching the signature

```
plusInt :: Int -> Int -> Int
mulInt  :: Int -> Int -> Int
negInt  :: Int -> Int
```

Generalising Num

```
plusInt    :: Int -> Int -> Int
plusFloat  :: Float -> Float -> Float
intToFloat :: Int -> Float
```

```
class GNum a b where
   (+) :: a -> b -> ???

instance GNum Int Int where
   (+) x y = plusInt x y

instance GNum Int Float where
   (+) x y = plusFloat (intToFloat x) y

test1 = (4::Int) + (5::Int)
test2 = (4::Int) + (5::Float)
```

Allowing more good programs

Generalising Num

```
class GNum a b where
   (+) :: a -> b -> ???
```

- Result type of (+) is a **function of the argument types**

```
class GNum a b where
   type SumTy a b :: *
   (+) :: a -> b -> SumTy a b
```

SumTy is an associated type of class GNum

- Each method gets a type signature
- Each associated type gets a kind signature

Generalising Num

```
class GNum a b where
  type SumTy a b :: *
  (+) :: a -> b -> SumTy a b
```

- Each instance declaration gives a "witness" for SumTy, matching the kind signature

```
instance GNum Int Int where
  type SumTy Int Int = Int
  (+) x y = plusInt x y

instance GNum Int Float where
  type SumTy Int Float = Float
  (+) x y = plusFloat (intToFloat x) y
```

Type functions

```
class GNum a b where
  type SumTy a b :: *
instance GNum Int Int where
  type SumTy Int Int = Int :: *
instance GNum Int Float where
  type SumTy Int Float = Float
```

- SumTy is a type-level function
- The type checker simply rewrites
  - SumTy Int Int --> Int
  - SumTy Int Float --> Float

  whenever it can
- But (SumTy t1 t2) is still a perfectly good type, even if it can't be rewritten. For example:

```
data T a b = MkT a b (SumTy a b)
```

Eliminate bad programs

- Simply omit instances for incompatible types

```
newtype Dollars = MkD Int

instance GNum Dollars Dollars where
  type SumTy Dollars Dollars = Dollars
  (+) (MkD d1) (MkD d2) = MkD (d1+d2)

-- No instance GNum Dollars Int

test = (MkD 3) + (4::Int)      -- REJECTED!
```

Optimising data structures

- Consider a finite map, mapping **keys** to **values**
- Goal: the data representation of the map depends on the type of the key
  - Boolean key: store two values (for F,T resp)
  - Int key: use a balanced tree
  - Pair key (x,y): map x to a finite map from y to value; ie use a trie!
- Cannot do this in Haskell...a good program that the type checker rejects

Optimising data structures

```
data Maybe a = Nothing | Just a
```

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ...insert, union, etc....
```

Map is **indexed** by k, but **parametric** in its second argument

Optimising data structures

```
data Maybe a = Nothing | Just a
```

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
  lookup :: k -> Map k v -> M ybe v
  ...insert, union, etc....

instance Key Bool where
  data Map Bool v = MB (Maybe v)  (Maybe v)
  empty = MB Nothing Nothing
  lookup True  (MB _ mt) = mt
  lookup False (MB mf _) = mf
```

Optional value for False

Optional value for True

Optimising data structures

```
data Maybe a = Nothing | Just a
```

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ...insert, union, etc....

instance (Key a, Key b) => Key (a,b) where
  data Map (a,b) v = MP (Map a (Map b v))
  empty = MP empty
  lookup (ka,kb) (MP m) = case lookup ka m of
                             Nothing -> Nothing
                             Just m2 -> lookup kb m2
```

Two-level map

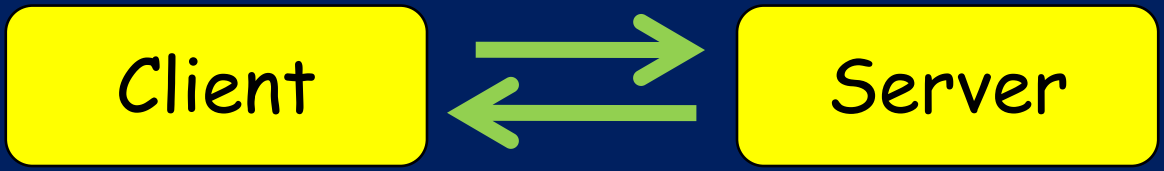
Two-level lookup

See paper for lists as keys: arbitrary depth tries

Optimising data structures

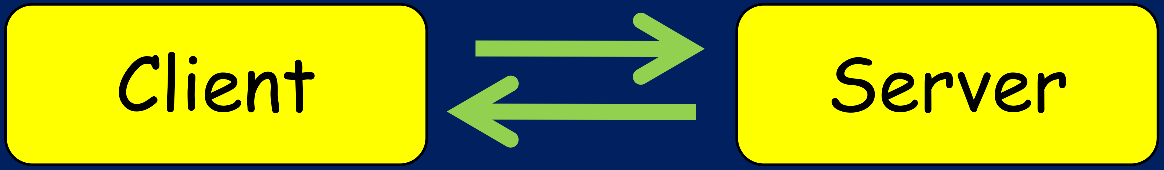
- Goal: the data representation of the map depends on the type of the key
  - Boolean key: SUM
  - Pair key (x,y): PRODUCT
  - List key [x]: SUM of PRODUCT + RECURSION
- Easy to extend to other types at will

Baby session types (BST)



- addServer :: In Int (In Int (Out Int End))
  addClient :: Out Int (Out Int (In Int End))
- Type of the process expresses its protocol
- Client and server should have dual protocols:
  run addServer addClient -- OK!
  run addServer addServer -- BAD!

Baby session types



- addServer :: In Int (In Int (Out Int End))
  addClient :: Out Int (Out Int (In Int End))

```
data In v p  = In (v -> p)
data Out v p = Out v p
data End     = End
```

NB punning

Baby session types

```
data In v p  = In (v -> p)
data Out v p = Out v p
data End     = End
```

```
addServer :: In Int (In Int (Out Int End))
addServer = In (\x -> In (\y ->
              Out (x + y) End))
```

- Nothing fancy here
- addClient is similar

But what about run???

```
run :: ??? -> ??? -> End
```

A process

A co-process

```
class Process p where
  type Co p
  run :: p -> Co p -> End
```

- Same deal as before: Co is a type-level function that transforms a process type into its dual

Implementing run

```
class Process p where
  type Co p
  run :: p -> Co p -> End
```

```
data In v p  = In (v -> p)
data Out v p = Out v p
data End     = End
```

```
instance Process p => Process (In v p) where
  type Co (In v p) = Out v (Co p)
  run (In vp) (Out v p) = run (vp v) p

instance Process p => Process (Out v p) where
  type Co (Out v p) = In v (Co p)
  run (Out v p) (In vp) = run p (vp v)
```

Just the obvious thing really

The paper: more examples

- The hoary printf chestnut
  printf “Name:%s, Age:%i” :: String -> Int -> String
  - Can’t do that, but we can do this:

```
printf (lit “Name:” <> string <> lit “, Age:” <> int)
   :: String -> Int -> String
```

- Machine address computation
  add :: Pointer n -> Offset m -> Pointer (GCD n m)

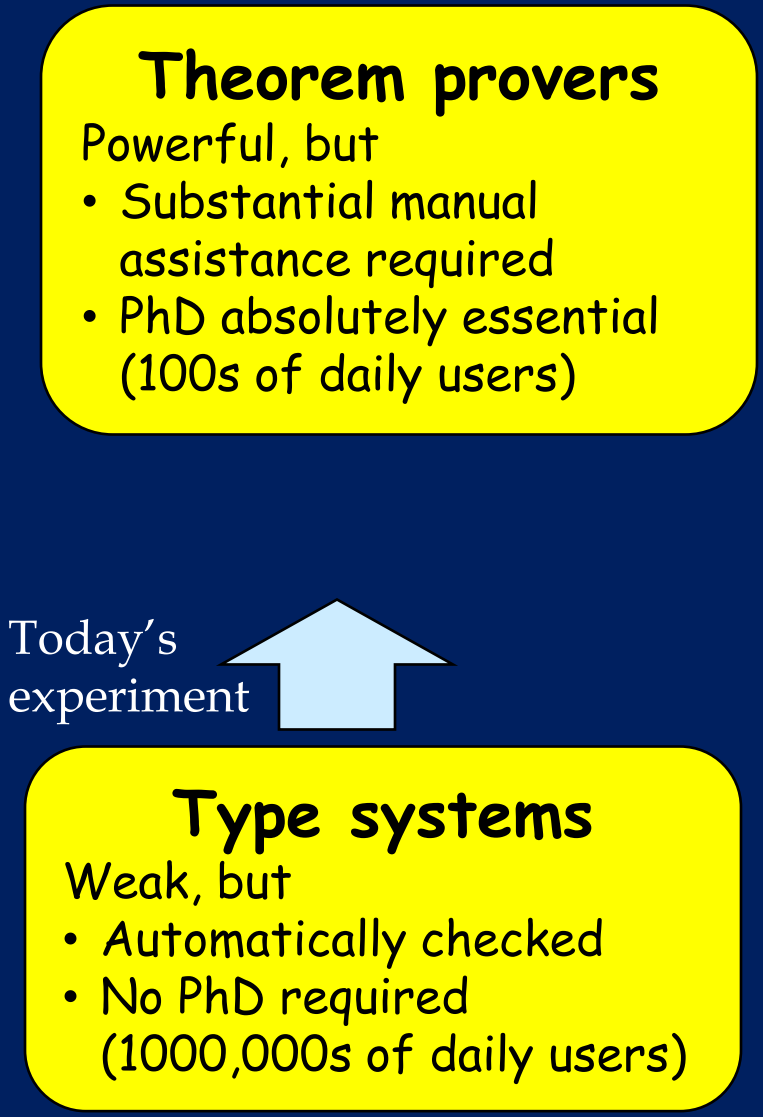
- Tracking state using Hoare triples

```
acquire :: (Get n p ~ Unlocked)
        => Lock n -> M p (Set n p Locked) ()
```

Lock-state before

Lock-state after

"Program correctness is a basic scientific ideal for Computer Science"



Theorem provers

Powerful, but

- Substantial manual assistance required
- PhD absolutely essential (100s of daily users)

Today's experiment

Type systems

Weak, but

- Automatically checked
- No PhD required (1000,000s of daily users)

- Types have made a huge contribution to this ideal
- More sophisticated type systems threaten both Happy Properties:
  1. Automation is harder
  2. The types are more complicated (MSc required)
- Some complications (2) are exactly due to ad-hoc restrictions to ensure full automation
- At some point it may be best to say "enough fooling around: just use Coq". But we aren't there yet
- Haskell is a great place to play this game