Liquidate your Assets

Reasoning about resource usage in Liquid Haskell

M. Handley, **N. Vazou**, and G. Hutton
University of Nottingham and **IMDEA**

```haskell
insert :: Ord a => x:a -> xs:[a] -> [a]


insert x []   = [x]
insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise = y:insert x ys
```

Refinement types for length preservation

```haskell
insert :: Ord a => x:a -> xs:[a]
       -> {os:[a]|len os == 1 + len xs}

insert x []   = [x]
insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise = y:insert x ys
```

SMT automated checking.
Even for list sortedness.

Refinement types for length preservation

```
insert :: Ord a => x:a -> xs:[a]
       -> {os:[a]|len os == 1 + len xs}

insert x []   = [x]
insert x (y:ys)
 | x <= y     = x:y:ys
 | otherwise = y:insert x ys
```

What about resources?
(here number of comparisons)

Tracking resources

```
insert :: Ord a => x:a -> xs:[a]
       -> {os:[a]|len os == 1 + len xs}
        cost o <= len xs
insert x []   = [x]
insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise = y:insert x ys
```

Tracking resources using The Tick data type

```
insert :: Ord a => x:a -> xs:[a]
-> {o:Tick {os:[a]|len os == 1 + len xs}
      | tcost o <= len xs }
insert x []   = [x]
insert x (y:ys)
  | x <= y     = x:y:ys
  | otherwise = y:insert x ys
```

The Tick data type

```
data Tick a = Tick { tcost :: Int,
                     tval  :: a  }
```

The Tick data type

```
data Tick a = Tick { tcost :: Int,
                     tval  :: a   }
```

The Applicative Instance

```
pure :: x:a -> {t:Tick a | tcost t == 0}
pure x = Tick 0 x

(<*>) :: f:(Tick (a -> b)) -> x:Tick a
      -> {t:Tick b | tcost t == tcost x + tcost f}
Tick i f <*> Tick j x = Tick (i+j) (f x)
```

Zero resources using the Tick data type

```
insert :: Ord a => x:a -> xs:[a]
-> {o:Tick {os:[a]|len os == 1 + len xs}
      | tcost o <= len xs }
insert x []   = pure [x]
insert x (y:ys)
  | x <= y    = pure (x:y:ys)
  | otherwise = (pure (y:)) <*> insert x ys
```

The Tick data type

```
data Tick a = Tick { tcost :: Int,
                     tval  :: a   }
```

Resource tracking

```
step :: x:a -> {t:Tick a | tcost t == 1}
step x = Tick 1 x

(</>) :: f:(Tick (a -> b)) -> x:Tick a
      -> {t:Tick b | tcost t == 1 + tcost x + tcost f}
Tick i f </> Tick j x = Tick (1+i+j) (f x)
```

Actual resources using the Tick data type

```
 insert :: Ord a => x:a -> xs:[a]
-> {o:Tick {os:[a]|len os == 1 + len xs}
        | tcost o <= len xs }
 insert x []   = pure [x]
 insert x (y:ys)
  | x <= y     = step (x:y:ys)
  | otherwise = (pure (y:)) </> insert x ys
```

Let’s define insertion sort!

Let's define insertion sort!

```
isort :: Ord a => xs:[a]
      -> {o:Tick {os:[a]|len os == len xs}
           | tcost o <= (len xs)²  }
isort []     = pure []
isort (x:xs) = isort xs >>= insert x
```

Tick is a Monad!

The Tick data type

```
data Tick a = Tick { tcost :: Int,
                     tval  :: a  }
```

The Monad Instance

```
return :: x:a -> {t:Tick a | tcost t == 0}
return x = Tick 0 x

(>>=) :: x:Tick a -> f:(a -> Tick b)
   -> {t:Tick b | tcost t == tcost x
                           + tcost (f (tval x)) }
Tick i x >>= f = case f x of
                   Tick j y -> Tick (i+j) y
```

Resources of insertion sort

```
isort :: Ord a => xs:[a]
      -> {o:Tick {os:[a]|len os == len xs}
           | tcost o <= (len xs)² }
isort []     = pure []
isort (x:xs) = isort xs >>= insert x
```

❌

```
(>>=) :: x:Tick a -> f:(a -> Tick b)
   -> {t:Tick b | tcost t == tcost x
                           + tcost (f (tval x)) }
```

Problem: type level computations!

Solution I: ghost parameter

Solution I: ghost parameter

```
isort :: Ord a => xs:[a]
      -> {o:Tick {os:[a]|len os == len xs}
            | tcost o <= (len xs)² }
isort []     = pure []
isort (x:xs) = isort xs >>={len xs} insert x
```

```
(>>={n}) :: x:Tick a
   -> f:(a -> {t:Tick b | tcost t <= n})
   -> {t:Tick b | tcost t <= tcost x + n}
```

Solution I: ghost parameter

```
isort :: Ord a => xs:[a]
      -> {o:Tick {os:[a]|len os == len xs}
            | tcost o <= (len xs)² }
isort []       = pure []
isort (x:xs)   = isort xs >>={len xs} insert x
```

```
(>>={n}) :: x:Tick a
   -> f:(a -> {t:Tick b | tcost t <= n})
   -> {t:Tick b | tcost t <= tcost x + n}
```

Cost of monadic function is bound by n

Solution I: ghost parameter

```
isort :: Ord a => xs:[a]
      -> {o:Tick {os:[a]|len os == len xs}
          | tcost o <= (len xs)² }
isort []     = pure []
isort (x:xs) = isort xs >>={len xs} insert x
```

```
(>>={n}) :: x:Tick a
   -> f:(a -> {t:Tick b | tcost t <= n})
   -> {t:Tick b | tcost t <= tcost x + n}
```

No type level computations…

Solution I: ghost parameter

```
isort :: Ord a => xs:[a]
      -> {o:Tick {os:[a]|len os == len xs}
            | tcost o <= (len xs)² }
isort []       = pure []
isort (x:xs)   = isort xs >>={len xs} insert x
```

```
(>>={n}) :: x:Tick a
   -> f:(a -> {t:Tick b | tcost t <= n})
   -> {t:Tick b | tcost t <= tcost x + n}
```

... but explicit parameter should be provided.

Resource Tracking using Refinement Types

Tick monad lets you track resources in refinement types

Problem:

The bind operation breaks automatic verification

Solution I: ghost parameter

Solution II: extrinsic proofs

Extrinsic resource analysis proofs

```
isortCostSorted :: Ord a => xs:OList a
      -> { tcost (isort xs) <= len xs }
isortCostSorted []     = () - automated
isortCostSorted [_]    = () - automated
isortCostSorted (x:xs@(y:ys))
 =   tcost (isort (x:xs))
 ==. tcost (isort xs >>={len xs} insert x)
 ==. tcost (isort xs) + tcost (insert x (tval (isort xs)))
     ? isortSortedVal xs - tval (isort xs) == xs
 ==. tcost (isort xs) + tcost (insert x xs)
     ? isortCostSorted xs
 <=. len xs + tcost (insert x xs)
 <=. len xs + tcost (insert x (y:ys))
 <=. len xs + 1
 <=. len (x:xs)
 *** QED
```

Extrinsic resource analysis proofs can be used for arbitrary properties

Relational Properties,
e.g., sorted lists are sorted faster.

```
  xs:OList a
→ ys:{[a] | len xs == len ys}
→ { tcost (isort xs) <= tcost (isort ys) }
```

Extrinsic resource analysis proofs
can be used for arbitrary properties

Relational Properties,
e.g., sorted lists are sorted faster.

We encoded all the examples from the relational refinement types[1, 2, 3] work.

): But, required manual proofs :(

[1]Aguirre et al: A Relational Logic for Higher-Order Programs. ICFP'17.
[2]Çiçek et al: Relational Cost Analysis. POPL'17.
[3]Radiček et al:Monadic Refinements for Relational Cost Analysis. POPL'18.

Extrinsic resource analysis proofs
can be used for arbitrary properties

Relational Properties,
e.g., sorted lists are sorted faster.

Function optimization,
e.g., [] ++ xs is faster than xs ++ []

We developed operators to simultaneously reason about
1/ resource modification and 2/program equivalence.

Extrinsic resource analysis proofs can be used for arbitrary properties

Function optimization,
e.g., [] ++ xs is faster than xs ++ []

```
xs:[a] → { [] ++ xs <~< xs ++ [] }
```

value preservation &
cost improvement!

Extrinsic resource analysis proofs can be used for arbitrary properties

Relational Properties,
e.g., sorted lists are sorted faster.

Function optimization,
e.g., [] ++ xs is faster than xs ++ []

Higher Order Properties,
e.g., map fusion is an optimization

Extrinsic resource analysis proofs
can be used for arbitrary properties

Higher Order Properties,
e.g., map fusion is an optimization

```
mapM f xs >>= mapM g
  >== len xs ==>
 mapM (f >=> g) xs
```

value preservation &
exact cost improvement!

Extrinsic resource analysis proofs
can be used for arbitrary properties

Relational Properties,
e.g., sorted lists are sorted faster.

Function optimization,
e.g., [] ++ xs is faster than xs ++ []

Higher Order Properties,
e.g., map fusion is an optimization

In the paper:

More benchmarks

Soundness of the Tick library

What about laziness?
Lazy ADTs should be explicitly defined using Ticks*

*Danielsson: Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. POPL'08.

Resource Tracking using Refinement Types

Tick monad lets you track resources in refinement types

Problem:

The bind operation breaks automatic verification

Solution I: ghost parameter

Solution II: extrinsic proofs

Extrinsic proofs can prove arbitrary resource properties

Thanks!

The End

Benchmarks

	Property	Lines of code Exec.	Spec.	Proof
Laziness [Danielsson 2008]				
Insertion sort	$\textsc{Cost}(\textit{lisort xs}) \leqslant \lvert xs \rvert$	12	8	0
Implicit queues	$\textsc{Cost}(\textit{lsnoc q x}) = 5$, $\textsc{Cost}(\textit{view q}) = 1$	50	14	0
Relational [Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2017]				
2D count	$\textsc{Cost}(\textit{2DCount find}_1) \leqslant \textsc{Cost}(\textit{2DCount find}_2)$	16	3	24
Binary counters	$\textsc{Cost}(\textit{decr k tt}) = \textsc{Cost}(\textit{incr k ff})$	26	21	21
Boolean expressions	$\textsc{NoShort}(e) \Rightarrow \textsc{Cost}(\textit{eval}_1\ e) = \textsc{Cost}(\textit{eval}_2\ e)$	28	2	13
Constant-time comparison	$\textsc{Cost}(\textit{compare p u\_1}) = \textsc{Cost}(\textit{compare p u\_2})$	3	8	3
Insertion sort	$\textsc{Sorted}(xs) \Rightarrow \textsc{Cost}(\textit{isort xs}) \leqslant \textsc{Cost}(\textit{isort ys})$	16	17	44
Memory allocation of length	$\textsc{Cost}(\textit{length}_2\ xs) - \textsc{Cost}(\textit{length}_1\ xs) = \textit{length xs}$	10	4	6
Relational insertion sort	$\textsc{Cost}(\textit{isort xs}) - \textsc{Cost}(\textit{isort ys}) = \textit{unsortedDiff xs ys}$	16	11	69
Relational merge sort	$\textsc{Cost}(\textit{msort xs}) - \textsc{Cost}(\textit{msort ys}) \leqslant \lvert xs \rvert\ (1 + \log_2(\textit{diff xs ys}))$	23	25	59
Square and multiply	$\textsc{Cost}(\textit{sam t x } l_1) - \textsc{Cost}(\textit{sam t x } l_2) \leqslant t * \textit{diff } l_1\ l_2$	3	8	3
Datatypes [Vazou et al. 2018]				
Append's monoid laws	*see example 5 of section 2*	12	10	74
Appending	$\textsc{Cost}(xs \mathbin{\underline{++}} ys) = \lvert xs \rvert$	8	3	0
Flattening	$\textsc{Perfect}(t) \Rightarrow \textsc{Cost}(\textit{flattenOpt } t) = 2^{\lvert t \rvert} - 1$	5	18	45
Optimised-by-construction reverse	*reverse xs >~> fastReverse xs*	18	37	140
Reversing (naive)	$\textsc{Cost}(\textit{reverse xs}) = \frac{\lvert xs \rvert^2}{2} + \frac{\lvert xs \rvert + 1}{2}$	9	7	22
Reversing (optimised)	$\textsc{Cost}(\textit{fastReverse xs}) = \lvert xs \rvert$	5	8	0
Sorting				
Data.List.sort	$\textsc{Cost}(\textit{ssort xs}) \leqslant 4\ \lvert xs \rvert \log_2 \lvert xs \rvert + \lvert xs \rvert$	39	49	107
Insertion sort	$\textsc{Cost}(\textit{isort xs}) \leqslant \lvert xs \rvert^2$	8	10	33
Merge sort	$\frac{\lvert xs \rvert}{2} \log_2 \lvert xs \rvert \leqslant \textsc{Cost}(\textit{msort xs}) \leqslant \lvert xs \rvert \log_2 \frac{\lvert xs \rvert}{2} + \lvert xs \rvert$	22	69	139
Quicksort	$\textsc{Cost}(\textit{qsort xs}) \leqslant \frac{1}{2}(\lvert xs \rvert + 1)(\lvert xs \rvert + 2)$	15	8	27
Total		344	340	829

Metatheory

A corollary of monadic encapsulation +
metatheory of refinement types:

THEOREM (SOUNDNESS OF COST ANALYSIS). *Let $p :: Int \to Bool$ be a predicate over integers and $f :: x : \tau_x \to \tau$ a safe and terminating function.*

- **Intrinsic cost analysis** *If* $\emptyset \vdash f :: x : \tau_x \to \{t : Tick_\tau \mid p\ (tcost_\tau\ t)\}$, *then for all* $e_x \in [\![\tau_x]\!]$, $e_f\ e_x \hookrightarrow^\star Tick_\tau\ i\ e$ *and* $p\ i \hookrightarrow^\star true$.
- **Extrinsic cost analysis** *If* $\emptyset \vdash e :: x : \tau_x \to \{v : \tau \mid p\ (tcost_\tau\ f\ x)\}$, *then for all* $e_x \in [\![\tau_x]\!]$, $f\ e_x \hookrightarrow^\star Tick_\tau\ i\ e$ *and* $p\ i \hookrightarrow^\star true$.

Future Directions

Resource Bound Inference

Automate lifting or, at least, erasure

Turn Tick into a monad transformer
(e.g., to combine with Parallel Monad)