Liquidate Your Assets

Reasoning About Resource Usage in Liquid Haskell

MARTIN A.T. HANDLEY, University of Nottingham, UK
NIKI VAZOU, IMDEA Software Institute, Spain
GRAHAM HUTTON, University of Nottingham, UK

Liquid Haskell is an extension to the type system of Haskell that supports formal reasoning about program correctness by encoding logical properties as refinement types. In this article, we show how Liquid Haskell can also be used to reason about program efficiency in the same setting, with the system's existing verification machinery being used to ensure that the results are both valid and precise. To illustrate our approach, we analyse the efficiency of a wide range of popular data structures and algorithms, and in doing so, explore various notions of resource usage. Our experience is that reasoning about efficiency in Liquid Haskell is often just as simple as reasoning about correctness, and that the two can naturally be combined.

Additional Key Words and Phrases: refinement types, resource analysis, static verification

1 INTRODUCTION

Estimating the amount of resources that are required to execute a program is a key aspect of software development. Unfortunately, however, performance bugs are as difficult to detect as they are common [Jin et al. 2012]. As a result, the problem of statically analysing the resource usage, or execution cost, of programs has been subject to much research, in which a broad range of techniques have been studied, including resource-aware type systems [Çiçek et al. 2017; Hoffmann et al. 2012; Hofmann and Jost 2003; Jost et al. 2017; Wang et al. 2017], program and separation logics [Aspinall et al. 2007; Atkey 2010], and sized types [Vasconcelos 2008].

Another technique for statically analysing execution cost, inspired by the early work in [Moran and Sands 1999] on improvement theory, is to reify resource usage into the definition of a program by means of a datatype that accumulates abstract computation 'steps'. This technique has two main approaches: steps can either accumulate at the type level inside an index, or at the value level inside an integer field. Formal analysis at the type level has been successfully applied in Agda [Danielsson 2008], and more recently Coq [McCarthy et al. 2017], and recent work in [Radiček et al. 2018] developed the theoretical foundations of the value-level approach.

In this article, we take inspiration from [Radiček et al. 2018] and implement a monadic datatype to measure the abstract resource usage of pure Haskell functions. We then use Liquid Haskell's [Vazou 2016] refinement type system to statically prove bounds on resource usage. Our framework supports the standard approach to cost analysis, which is known as unary cost analysis and aims to establish upper and lower bounds on the execution cost of a single program, together with the more recent relational approach [Çiçek 2018], which aims to calculate the difference between the execution costs of two related programs or between one program on two different inputs.

Reasoning about execution cost using the Liquid Haskell system has two main advantages over most other formal cost analysis frameworks [Radiček et al. 2018]. First of all, the system allows correctness properties to be naturally integrated into cost analysis, which helps to ensure that cost analyses are valid. And secondly, the wide range of sophisticated invariants that can be expressed

Authors' addresses: Martin A.T. Handley, University of Nottingham, UK; Niki Vazou, IMDEA Software Institute, Spain; Graham Hutton, University of Nottingham, UK.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

and automatically verified by the system can be exploited to analyse resource usage in particular program contexts, which often leads to more precise and/or simpler analyses.

By way of example, Liquid Haskell can automatically verify that Haskell's standard `sort` function returns an ordered list (of type `OList a`) with the same length as its input, even when the result is embedded in the `Tick` datatype that we use to measure resource usage:

```
sort :: Ord a => xs:[a] → Tick { zs:OList a | length zs == length xs }
```

Applying our cost analysis to this function then allows us to prove that the maximum number of comparisons required to sort any list `xs` is $O(n \log_2 n)$, where $n \equiv length\ xs$:

```
sortCost :: Ord a => xs:[a] →
   { tcost (sort xs) <= 4 * length xs * log2 (length xs) + length xs }
```

Moreover, we can also combine correctness and resource properties, to show that the maximum number of comparisons becomes linear when the input list is already sorted:

```
sortCostSorted :: Ord a => xs:OList a → { tcost (sort xs) <= length xs }
```

The aim of this article is to develop, prove correct, and evaluate a system that supports the above form of reasoning. To this end, the article makes the following contributions:

- We design and implement a system that allows Liquid Haskell to be used to formally reason about the resource usage and correctness of pure Haskell programs. Our system takes the form of a library, and requires no modifications or extensions to the compiler.
- We prove that our library's approach to cost analysis is correct with respect of an underlying model of execution cost, using the metatheory of Liquid Haskell.
- We demonstrate the applicability of our library on a wide range of case studies, ranging from standard sorting algorithms to sophisticated relational cost properties, and including *all* examples from Aguirre et al. [2017]; Çiçek et al. [2017]; Radiček et al. [2018].

The article is aimed at readers who are familiar with the basic idea of using refinement types in a language such as Liquid Haskell [Vazou 2016], but no specialist knowledge on reasoning about efficiency is assumed. Section 2 introduces our approach with a number of examples; section 3 discusses how our library is implemented; section 4 evaluates its utility on a range of case studies; section 5 develops the supporting theory; and finally, sections 6 and 7 discuss related work and directions for further work. Our library is freely available on GitHub [Anonymised 2019], along with the source code for all of the examples presented throughout the article.

2 ANALYSING RESOURCE USAGE

In this section, we exemplify our library's intrinsic and extrinsic approaches to analysing resource usage, which support both unary and relational cost analysis. In addition, each example serves to demonstrate how correctness properties can be naturally integrated into each method of cost analysis. We conclude this section by discussing how to interpret such analyses in practice.

2.1 Intrinsic cost analysis

In the case of *intrinsic* cost analysis, the resources utilised by a function are declared *inside* the type signature of the function, and are *automatically* checked by Liquid Haskell.

Example 1. Time complexity We start by analysing the number of recursive calls made by Haskell's list append function `(++)`. We define the operator `(++)` that is similar to append, but lifted into our `Tick` datatype using applicative methods provided by our library (introduced in section 3):

```
(++) :: [a] → [a] → Tick [a]
[]       ++ ys = pure ys
(x : xs) ++ ys = pure (x :) </> (xs ++ ys)
```

That is, if the first argument list is empty, the second list `ys` is embedded into the `Tick` datatype using `pure`, which records zero cost. In turn, if the first list is non-empty, the partially applied result `(x :)` is embedded using `pure` and applied to the result of the recursive call. To record the cost of the recursive call we use the operator `(</>)`, a variant of the applicative operator `(<*>)` that sums the costs of the two arguments and then increases the total by *one*.

Now that we have defined the new operator, we can use Liquid Haskell to encode properties about append's execution cost by means of a refinement type specification, such as the following:

```
{-@ (++) :: xs:[a] → ys:[a] →
      { t:Tick { zs:[a] | length zs == length xs + length ys} | tcost t == length xs } @-}
```

This type states that the length of the output list is given by the sum of the lengths of the two input lists: a correctness property; and that the cost of appending two lists in terms of the number of recursive calls required is given by the length of the first list: an efficiency property. Liquid Haskell is able to automatically verify both properties without any assistance from the user.

Example 2. Memory allocation Next, we analyse a different resource: the number of thunks allocated when folding lists. As before, we lift the standard `foldl` function into the `Tick` datatype. However, this time we use `step` to manually increment `foldl`'s resource usage each time it allocates a thunk:

```
foldl f z []       = pure z
foldl f z (x : xs) = let w = f z x in 1 `step` foldl f w xs
```

Because `foldl`'s resource usage increases for each element in the input list, we can use Liquid Haskell to automatically check that the overall cost of folding is equal to the length of this list:

```
{-@ foldl :: (b → a → b) → b → xs:[a] → { t:Tick b | tcost t == length xs } @-}
```

In contrast, the strict variant of `foldl`, called `foldl'`, uses Haskell's `seq` primitive to force the evaluation of its intermediate results during execution:

```
{-@ foldl' :: (b → a → b) → b → xs:[a] → { t:Tick b | tcost t == 0 } @-}
foldl' f z []       = pure z
foldl' f z (x : xs) = let w = f z x in w `seq` foldl' f w xs
```

As before, the `Tick` datatype is used to record the number of allocated thunks. Because no thunks are allocated by `foldl'`, we do not increase the cost at each recursive step, and thus Liquid Haskell correctly verifies that the function's total execution cost is zero.

Both of these examples are simplified execution models, in the sense that they only account for the number of thunks allocated by the higher-order folding functions, and not the functions `f` being folded. In our subsequent case studies in section 4, we give a more accurate account of resource usage that incorporates the number of additional thunks allocated by each `f`.

Example 3. Cost analysis and verification In this example, we analyse the number of comparisons made when merging two ordered lists. As before, we lift the standard `merge` function into the `Tick` datatype, and use the `(</>)` operator to increase the cost each time a comparison is made:

```
merge xs [] = pure xs
merge [] ys = pure ys
merge (x : xs) (y : ys)
  | x <= y    = pure (x :) </> merge xs (y : ys)
  | otherwise = pure (y :) </> merge (x : xs) ys
```

The resource usage of the `merge` function depends on the values of the input lists, and so we cannot easily establish a precise bound on its execution cost. We can, however, use Liquid Haskell to automatically check upper and lower bounds on this cost:

```
{-@ merge :: Ord a => xs:OList a → ys:OList a →
       { t:Tick { zs:OList a | length zs == length xs + length ys }
       | tcost t <= length xs + length ys
      && tcost t >= min (length xs) (length ys) } @-}
```

That is, in the worse case, `merge` performs `length xs + length ys` comparisons as both input lists may need to be completely traversed to produce an ordered output. Conversely, in the best case, we only require `min (length xs) (length ys)` comparisons, as `merge` terminates as soon as one of the input lists becomes empty. Note that the above type uses the ordered list type constructor, `OList`, which is defined using abstract refinements [Vazou et al. 2013] as follows:

```
{-@ type OList a = [a]<{ λ x y → x <= y}> @-}
```

Hence, the refinement type for `merge` also states that merging two ordered lists returns an ordered list with length equal to the sum of the two input lengths. Once again, building cost analysis on top of existing Liquid Haskell features allows us to naturally combine correctness and efficiency properties.

2.2 Extrinsic cost analysis

In the case of *extrinsic* cost analysis, we use refinement types to express theorems about resource usage, and then define Haskell terms that inhabit these types to prove the theorems. In contrast to intrinsic cost analysis, this approach does not support fully automated verification, as proof terms must be provided by users. However, this method allows us to specify efficiency properties that are not intrinsic to the definitions of functions. For example, we can relate the costs of multiple functions, and analyse the resource usage of functions applied to specific subsets of their domains.

Example 4. Unary cost analysis Using the `merge` function from the previous example, we can define a function that implements merge sort, with the following refinement type:

```
{-@ msort :: Ord a => xs:[a] → Tick { zs:OList a | length zs == length xs } @-}
```

This type captures two correctness properties of merge sort, namely that the output list is sorted and has the same length as the input list. To analyse the cost of `msort` we can use the extrinsic approach. That is, we specify appropriate theorems outside of the function's definition and prove them manually. The following two theorems capture upper and lower bounds on execution cost:

```
{-@ msortCostUB :: Ord a => { xs:[a] | pow₂ (length xs) } →
        { tcost (msort xs) <= 2 * length xs * log₂ (length xs) } @-}

{-@ msortCostLB :: Ord a => { xs:[a] | pow₂ (length xs)} →
        { tcost (msort xs) >= (length xs / 2) * log₂ (length xs) } @-}
```

The first theorem states that the number of comparisons required by `msort` is bounded above by $O(n\ log_2\ n)$, where n is the length of the input list. The second states that the number of comparisons is bounded below by $O(n\ log_2\ n)$. In both cases, because merge sort proceeds by repeatedly splitting the input list into two parts, we assume the input length to be a power of two, specified by `pow₂ (length xs)`. This highlights the flexibility of the extrinsic method: even though it is reasonable to use this assumption for cost analysis, it would be unreasonable to impose this restriction on all of the inputs to which `msort` is applied. Proofs of these theorems can be constructed using the proof combinators introduced in section 3, and are available online [Anonymised 2019].

Note that if we assume the cost of comparison outweighs the cost of all other operations performed during merge sort's execution, we can use the above theorems to infer asymptotic upper and lower bounds on the algorithm's runtime performance, respectively.

Example 5. Relational cost analysis The extrinsic approach enables us to describe arbitrary program properties, including those that compare the relative cost of two functions, or the same function applied to different inputs. This is known as relational cost analysis [Çiçek 2018]. Here, we adapt an example from [Çiçek et al. 2017] to demonstrate how relational cost can be encoded in our setting.

In cryptography, a program adheres to the 'constant-time discipline' if its execution time is independent of secret inputs. Adhering to this discipline is an effective countermeasure against timing attacks, which can allow intruders to infer secret inputs by measuring variations in execution time. Using relational cost analysis, we can prove that a program is constant-time without having to show that it has equal upper and lower bounds on its execution cost (for example, by performing two separate unary analyses). To demonstrate this, we use our library to analyse the execution cost of a function that compares two equal-length password hashes represented as lists of binary digits:

```
{-@ type EqLen xs = { ys:[Bit] | length ys == length xs } @-}

{-@ compare :: xs:[Bit] → ys:EqLen xs → t:Tick Bool @-}
compare []       []       = pure True
compare (x : xs) (y : ys) = pure (&& x == y) </> compare xs ys
```

We assume that the equality `(==)` and conjunction `(&&)` functions are both constant-time, therefore, we only measure the number of recursive calls made during `compare`'s execution.

As we have assumed that the computations performed during each recursive step are constant-time, we can prove that `compare` is a constant-time function by showing that it requires the same number of recursive calls when comparing any stored password `pwd` against any two user inputs u_1 and u_2:

```
{-@ constant :: pwd:[Bit] → u1:EqLen pwd → u2:EqLen pwd →
        { tcost (compare pwd u1) == tcost (compare pwd u2) } @-}
constant []       _         _         = ()
constant (_ : ps) (_ : us1) (_ : us2) = constant ps us1 us2
```

The proof of this theorem proceeds by induction on the length of the input lists. Consequently, our proof has a trivial base case and an inductive case that recursively calls the inductive hypothesis. In this instance, Liquid Haskell can deduce the relative cost of both executions from the definition of `compare`. As such, all the details of the proof can be handled automatically by Liquid Haskell's proof by logical evaluation (PLE) tactic [Vazou et al. 2017].

Example 6. Cost improvement As a final example, we outline how extrinsic cost analysis can be used to calculate the difference in the execution costs of two *related* programs. This is also a primary application of relational cost analysis. Consider the familiar monoid laws for the append operator:

```
          [] ++ ys  ==  ys                    left identity
          xs ++ []  ==  xs                    right identity
(xs ++ ys) ++ zs    ==  xs ++ (ys ++ zs)      associativity
```

These properties can be proved correct in Liquid Haskell via equational reasoning [Vazou et al. 2018]. However, although the two sides of each property give the same results, each side does not necessarily require the same amount of resources. This observation can be made precise by proving the following properties of the annotated append operator, `(++)`:

```
[] ++ ys          <=>          pure ys
xs ++ []  >== length xs ==>  pure xs
(xs ++ ys) >>= (++ zs)  >== length xs ==>  (xs ++) =<< (ys ++ zs)
```

Recall from example 1 that the `(++)` operator records the number of recursive calls made during append's execution. Using this notion of cost, the first property above states that the left identity law is a *cost equivalence*. That is, `[] ++ ys` and `ys` evaluate to the the same result, and moreover, both require the same number of recursive calls to append. We make this precise by relating the annotated version of each side using the cost equivalence relation `<=>`. Note that `ys` must be embedded in the `Tick` datatype using `pure` in order for the property to be type-correct.

On the other hand, the right identity and associativity laws are *cost improvements* in the left-to-right direction. That is, both sides of each property evaluate to the same result, but in each case the right-hand side requires fewer recursive calls to append. Again, we make this precise by relating the corresponding annotated definitions. Moreover, we make the cost difference explicit using quantified improvement, written `>== n ==>` for a positive cost difference `n`, by showing that each right-hand side requires `length xs` fewer recursive calls than its corresponding left-hand side.

We return to the notions of cost equivalence, cost improvement, and quantified improvement in section 3, where we discuss our library's implementation and prove the second property as an example. Subsequently, in section 4, we use quantified improvement to construct a unified proof that shows the well-known map fusion technique preserves correctness and improves performance.

2.3 Interpreting cost analysis

Our library allows users to analyse a wide range of resources. Specifically, the `Tick` datatype can measure any kind of resource whose usage is additive, in the sense that the basic operation on costs is addition. Nonetheless, the correctness of a cost analysis relies on appropriate cost annotations being added to a program by the user. As such, it is the user's responsibility to ensure that such annotations correctly model the intended usage of a resource. In section 5, we use Liquid Haskell's metatheory to formally prove the correctness of specifications with respect to user-provided annotations.

Assuming that an annotated program does correctly model the intended usage of a particular resource, then the question is: how can a user relate its (intrinsic or extrinsic) cost analysis back to the execution cost of its unannotated counterpart? In other words, what is the interpretation of an annotated expression's cost bound in practice?

Haskell's lazy evaluation. As illustrated in the previous examples, any bound established on the execution cost of an annotated function that manipulates standard Haskell datatypes is a *worst-case* approximation of actual resource usage. For example, consider the annotated append function `(++)` that measures the number of recursive calls made by `(++)`. Then `tcost ([a,b,c] ++ ys) == 3` implies that the evaluation of `[a,b,c] ++ ys` makes three recursive calls to `(++)`. Three recursive calls to `(++)` corresponds to `[a,b,c] ++ ys` being *fully evaluated*.

An intuitive way to describe our library's cost analysis in this instance is to use terminology from Okasaki [1999]: the analysis assumes that functions applied to standard Haskell datatypes are *monolithic*. That is, once run, such functions are assumed to run until completion. This is not true in practice because Haskell's lazy evaluation strategy proactively halts computations to prevent functions from being unnecessarily fully applied.

Moreover, for efficiency, lazy evaluation allows computations to share intermediate results so that expressions are not unnecessarily re-evaluated when needed on multiple occasions. By default, however, annotated expressions do not model sharing, that is, memoisation. For example, the `square`

function below records the resource usage of its input `n :: Tick Int` twice, even though its unannotated counterpart, `square n = n * n`, only evaluates `n :: Int` once:

```
{-@ square :: n:Tick Int → { t:Tick Int | tcost t == 2 * tcost n } @-}
square n = pure (*) <*> n <*> n
```

Thus, overall, our library's default cost analysis assumes that computations are fully evaluated and overlooks memoisation, leading to worst-case approximations of actual execution costs in practice, that is, under Haskell's lazy evaluation strategy.

Explicit laziness. Our library can be used to precisely analyse the execution costs of computations that are *explicitly* lazy. This is achieved by encoding non-strictness into the definitions of datatypes and utilising a 'manual' memoisation function, similarly to the approach taken by Danielsson [2008]. We return to these ideas in a case study on insertion sort in section 4.

3 IMPLEMENTATION

In this section, we present the implementation of our library and discuss two soundness assumptions it makes. The library consists of two modules. The first, `RTick`, defines the `Tick` datatype and functions for recording and modifying resource usage, for example, `pure` and `(</>)` from section 2. The second module, `ProofCombinators`, defines combinators to encode steps of (in)equational reasoning about the values and resource usage of annotated expressions.

3.1 Recording resource usage

Our principle datatype, `Tick a`, consists of an integer to track resource usage and a value of type `a`:

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

The `Tick` datatype is a monad, with the following applicative and monad methods:

```
{-@ pure :: x:a → { t:Tick a | tval t == x && tcost t == 0 } @-}
pure x = Tick 0 x
{-@ (<*>) :: t1:Tick (a → b) → t2:Tick a →
        { t:Tick b | tval t  == (tval t1) (tval t2)
                   && tcost t == tcost t1 + tcost t2 }
Tick m f <*> Tick n x = Tick (m + n) (f x)

{-@ return :: x:a → { t:Tick a | tval t == x && tcost t == 0 } @-}
return x = Tick 0 x
{-@ (>>=) :: t1:Tick a → f:(a → Tick b) →
        { t:Tick b | tval t  == tval (f (tval t1))
                   && tcost t == tcost t1 + tcost (f (tval t1)) } @-}
Tick m x >>= f = let Tick n y = f x in Tick (m + n) y
```

The functions `pure` and `return` embed expressions in the `Tick` datatype and record zero cost, while the `(<*>)` and `(>>=)` operators sum up the costs of subexpressions.

We have formalised the applicative and monad laws for the above definitions in Liquid Haskell. The relevant proofs can be found on the library's GitHub page [Anonymised 2019].

3.2 Modifying resource usage

The `RTick` module defines various functions to record and modify resource usage. Inspired by the work in [Danielsson 2008], we refer to these (and the applicative and monad functions above) as *annotations*. Here we present the functions used throughout the article; the remainder can be found online [Anonymised 2019]. The most basic way to record resource usage is by using `step`:

```
{-@ step :: m:Int → t₁:Tick a →
        { t:Tick a | tval t == tval t₁ && tcost t == m + tcost t₁ } @-}
step m (Tick n x) = Tick (m + n) x
```

A positive integer argument to `step` indicates the consumption of a resource and negative production.

In many cases, we wish to sum up the costs of subexpressions and then modify the result. For this, we provide a number of resource combinators. One such combinator, `(</>)`, was used in section 2 and is a variation of the apply operator, `(<*>)`. Specifically, `(</>)` behaves just as `(<*>)` at the value level, but increases the total resource usage of its subexpressions by *one*:

```
{-@ (</>) :: t₁:Tick (a → b) → t₂:Tick a →
        { t:Tick b | tval  t == (tval t₁) (tval t₂)
                   && tcost t == 1 + tcost t₁ + tcost t₂ } @-}
Tick m f </> Tick n x = Tick (1 + m + n) (f x)
```

A similar combinator is defined in relation to the bind operator:

```
{-@ (>/=) :: t₁:Tick a → f:(a → Tick b) →
        { t:Tick b | tval  t == tval (f (tval t₁))
                   && tcost t == 1 + tcost t₁ + tcost (f (tval t₁)) } @-}
Tick m x >/= f = let Tick n y = f x in Tick (1 + m + n) y
```

Finally, we provide functions to embed computations in the `Tick` datatype while simultaneously consuming or producing resources. For example, `wait` and `waitN` [Danielsson 2008] act in the same manner as `return` at the value level, but consume one and `n` resources, respectively:

```
{-@ wait :: x:a → { t:Tick a | tval t == x && tcost t == 1 } @-}
wait x = Tick 1 x
{-@ waitN :: n:Int → x:a → { t:Tick a | tval t == x && tcost t == n } @-}
waitN n x = Tick n x
```

Note that `(>/=)` can be defined using `step` and `(>>=)`, specifically `(t >/= f) == step 1 (t >>= f)`. In fact, all of the functions provided by the `RTick` module, including `(<*>)` and `(</>)`, can be defined using `return`, `(>>=)`, and `step`. We make use of this fact in section 5 to simplify the proof of correctness of our cost analysis.

It is important to note that `Tick`'s cost parameter should *not* be modified by any means other than through the use of the functions in the `RTick` module, for example, by case analysis. Doing so breaks the encapsulation of `Tick`'s effects, potentially leading to invalid cost analysis. This is discussed in detail at the end of this section as part of the library's assumptions.

3.3 Proving extrinsic theorems

As exemplified in section 2, extrinsic cost analysis requires manually proving that bounds on resource usage hold. In Liquid Haskell, this is formalised as (in)equational reasoning, that is, a proof of an extrinsic theorem is a total and terminating Haskell function that appropriately relates the left-hand side of the theorem's proof statement to the right-hand side, for example, by unfolding and folding definitions [Burstall and Darlington 1977] and through the use of mathematical induction.

Next, we introduce a number of proof combinators from our library's `ProofCombinators` module that aid the development of extrinsic proofs. As a running example, we show that append's right identity law, `xs ++ [] == xs`, is an optimisation in the left-to-right direction, by proving properties about the annotated append function, `(++)`, from section 2.

3.3.1 Proof construction. We first review how to construct (in)equational proofs using Liquid Haskell. To exemplify both the equational and inequational styles of proof, we reason about the

Equal
```
{-@ (==.) :: x:a → { y:a | y == x }
          → { z:a | z == x && z == y } @-}
_ ==. y = y
```

Greater than or equal
```
{-@ (>=.) :: m:a → { n:a | m >= n }
          → { o:a | m >= o && o == n } @-}
_ >=. n = n
```

Theorem invocation
```
(?) :: a → Proof → a
x ? _ = x
```

Proof finalisation
```
(***) :: a → QED → Proof
_ *** QED = ()
```

QED definition
```
data QED = QED
```

Figure 1. Proof combinators introduced in [Vazou et al. 2018].

results and resource usage of append separately. Readers are referred to [Vazou et al. 2018] for a more detailed discussion on the following concepts.

Specifying theorems. The `Proof` type is simply the unit type, which is refined to express a theorem: `type Proof = ()`. For example, in order to show that append's right identity law is a denotational equivalence, we can express that the values of `xs ++ []` and `pure xs` are equal:

```
{-@ rightIdVal :: xs:[a] → { p:Proof | tval (xs ++ []) == tval (pure xs) } @-}
```

Here, the binder `p:Proof` is superfluous and so we can remove it:

```
{-@ rightIdVal :: xs:[a] → { tval (xs ++ []) == tval (pure xs) } @-}
```

Equational proofs. The above theorem expresses a *value* equivalence between two annotated expressions. In this case, Liquid Haskell cannot prove the theorem automatically. To prove it ourselves, we can define one of its inhabitants using a number of proof combinators from figure 1:

```
rightIdVal []
  =.  tval ([] ++ [])
  ==. tval (pure [])
  *** QED
```

```
rightIdVal (x : xs)
  =.  tval ((x : xs) ++ [])
  ==. tval (pure (x :) </> (xs ++ []))
    ? rightIdVal xs
  ==. tval (pure (x :) </> pure xs)
  ==. tval (Tick 0 (x :) </> Tick 0 xs)
  ==. tval (Tick 1 (x : xs))
  ==. tval (Tick 0 (x : xs))
  ==. tval (pure (x : xs))
  *** QED
```

Recall that the aim of the proof is to equate the left-hand side of the theorem's proof statement, `tval (xs ++ [])`, with the right-hand side, `tval (pure xs)`. We split it into two cases. In the base case, where `xs` is empty, the proof simply unfolds the definition of `(++)`. In the inductive case, where `xs` is non-empty, the proof unfolds `(++)` and `(</>)`, and unfolds and folds `pure`. It also appeals to the inductive hypothesis using `(?)`. In both cases, the `(==.)` combinator relates steps of reasoning by ensuring that both of its arguments are equal and returns its second argument to allow multiple uses to be chained together. The `(*** QED)` signifies the end of each proof.

Inequational proofs. Having proved that the values of `xs ++ []` and `pure xs` are equal, the next step is to compare their resource usage. From section 2, we know that the costs of both expressions are not equal. In particular, `xs ++ []` requires `length xs` more recursive calls to append than

Relations

Value equivalence	`t1 =!= t2 = tval t1 == tval t2`
Cost equivalence	`t1 <=> t2 = t1 =!= t2 && tcost t1 == tcost t2`
Cost improvement	`t1 >~> t2 = t1 =!= t2 && tcost t1 >= tcost t2`
Cost diminishment	`t1 <~< t2 = t1 =!= t2 && tcost t1 <= tcost t2`
Quantified improvement	`t1 >== n ==> t2 = t1 =!= t2 && tcost t1 == n + tcost t2`
Quantified diminishment	`t1 <== n ==< t2 = t1 =!= t2 && n + tcost t1 == tcost t2`

Combinators

Cost equivalence	`{-@ (<=>.) :: t1:Tick a → { t2:Tick a | t1 <=> t2} → { t:Tick a | t1 <=> t && t2 <=> t } @-}`
Cost improvement	`{-@ (>~>.) :: t1:Tick a → { t2:Tick a | t1 >~> t2} → { t:Tick a | t1 >~> t && t2 <=> t } @-}`
Cost diminishment	`{-@ (<~<.) :: t1:Tick a → { t2:Tick a | t1 <~< t2} → { t:Tick a | t1 <~< t && t2 <=> t } @-}`
Quantified improvement	`{-@ (.>==) :: t1:Tick a → n:Int → { t2: Tick a | t1 >== n ==> t2 } → { t:Tick a | t1 >== n ==> t && t2 <=> t } @-}`
Quantified diminishment	`{-@ (.<==) :: t1:Tick a → n:Int → { t2: Tick a | t1 <== n ==< t2 } → { t:Tick a | t1 <== n ==< t && t2 <=> t } @-}`

Combinators simply return their last arguments similarly to `(==.)` in figure 1.

Separators

Quantified improvement	`(==>.) :: (a → b) → a → b  f ==>. x = f x`
Quantified diminishment	`(==<.) :: (a → b) → a → b  f ==<. x = f x`

Figure 2. Cost relations, combinators, and separators.

`pure xs`. We can formalise this by proving that the execution cost of `xs ++ []` is greater than or equal to that of `pure xs` using the `(>=.)` combinator presented in figure 1:

```
{-@ rightIdCost :: xs:[a] → { tcost (xs ++ []) >= tcost (pure xs) } @-}
rightIdCost xs
  =.  tcost (xs ++ [])
  >=. tcost (pure [])
  *** QED
```

The resource usage of `pure xs` is zero as it requires no recursive calls to `(++)`. Furthermore, Liquid Haskell can automatically deduce that `tcost (xs ++ []) == length xs` and that `length xs >= 0`. Hence the theorem follows from a single use of `(>=.)`.

The `ProofCombinators` module includes numerous other numerical operators for reasoning about execution cost, including greater than `(>.)`, less than `(<.)`, and less than or equal `(<=.)`.

3.3.2 Proofs of cost equivalence, improvement, and diminishment.

Cost equivalence. Often it is beneficial to reason about the values and resource usage of annotated expressions simultaneously. For example, if we unfold the base case of the annotated append function, `(++)`, it is easy to show that both expressions are equal:

```
[] ++ ys == pure ys
```

Nonetheless, instead of relating the two expressions using equality, we prefer to use the notion of *cost equivalence*, which better clarifies our topic of reasoning. The cost equivalence relation is defined as

a Liquid Haskell predicate in figure 2, and states that two annotated expressions are cost-equivalent if they have the same values and resource usage. Clearly `[] ++ ys` and `pure ys` do:

```
[] ++ ys <=> pure ys
```

The above property is a 'resource-aware' version of append's left identity law, which formalises that both expressions, `[] ++ ys` and `pure ys`, evaluate to the same result and require the same number of recursive calls to append during evaluation (as shown in example 6 of section 2).

Cost improvement. Previously in this section, we proved that append's right identity law is a value equivalence: `tval (xs ++ []) == tval (pure xs)`, and a cost inequivalence: `tcost (xs ++ []) >= tcost (pure xs)`. Both of these properties are captured by the *cost improvement* relation defined in figure 2. Append's right identity law is thus an improvement—with respect to number of recursive calls—in the left-to-right direction. Following [Moran and Sands 1999], we say that "`xs ++ []` is improved by `pure xs`".

One way to prove that append's right identity law is a left-to-right improvement is to simply combine both sets of refinements from `rightIdVal` and `rightIdCost` using `(?)`:

```
{-@ rightIdImp :: xs:[a] → { xs ++ [] >~> pure xs } @-}
rightIdImp xs = rightIdVal xs ? rightIdCost xs
```

However, in general, this approach overlooks a key opportunity afforded by relational cost analysis, which is the ability to precisely relate intermediate execution steps [Çiçek 2018].

Crucially, unfolding (and folding) the definitions of annotated expressions makes resource usage explicit in steps of (in)equational reasoning. Not only does this allow savings in resource usage to be quantified in proofs, but it allows such savings to be localised. This approach fundamentally requires reasoning about the values and execution costs of annotated expressions simultaneously, however, and thus proofs relating values and costs independently simply cannot exploit it.

Quantified improvement. It is straightforward to show that `xs ++ []` is improved by `pure xs` by relating the expressions' intermediate execution steps using cost combinators from figure 2. However, we know the exact cost difference between `xs ++ []` and `pure xs`, namely `length xs`. This additional information allows us to relate the expressions more precisely using the *quantified improvement* relation, also defined in figure 2. Quantified improvement extends cost improvement by taking an additional argument, which is the cost difference between its first and last arguments.

Therefore, we can say "`xs ++ []` is improved by `pure xs`, by a cost of `length xs`", and make it precise by defining a corresponding theorem, as follows:

```
{-@ rightIdQImp :: xs:[a] → { xs ++ [] >== length xs ==> pure xs } @-}
```

To prove this theorem, we can simply extend the previous proof of value equivalence, `rightIdVal`, by replacing equality with cost equivalence and by making cost savings wherever possible. Readers are encouraged to note the strong connection between `rightIdVal` and the following proof:

```
rightIdQImp []
  =    [] ++ []
  <=>. pure []
  ***  QED
```

```
rightIdQImp (x : xs)
  =    (x : xs) ++ []
  <=>. pure (x :) </> (xs ++ [])
     ? rightIdQImp xs
  .>== length xs ==>. pure (x :) </> pure xs
  <=>. Tick 0 (x :) </> Tick 0 xs
  <=>. Tick 1 (x : xs)
  .>== 1 ==>. Tick 0 (x : xs)
  <=>. pure (x : xs)
  ***  QED
```

In the base case, where `xs` is empty, there is no cost saving. This is because `length [] == 0` and, therefore, `tcost ([] ++ []) == tcost (pure [])`. Hence it suffices to show that `[] ++ [] <=> pure []`, which follows immediately from the definition of `(++)`.

In the inductive case, where `xs` is non-empty, we must save `length (x : xs)` cost. We start by unfolding the definition of `(++)` and then replace `xs ++ []` with `pure xs` by appealing to the inductive hypothesis using `(?)`, which saves `length xs` resources. This saving is made explicit using the quantified improvement operator, `(.>== length xs ==>.)`, which is a combination of two functions, `(.>==)` and `(==>.)`, whereby the latter is a syntactic sugar for Haskell's `($)` operator, which allows `(.>==)` to be used infix. We save one further recursive call by unfolding the definition of `(</>)`. Finally, our goal follows from the definition of `pure`. The total resource saving is `1 + length xs`, which is equal to `length (x : xs)` by the definition of `length`.

By starting at the left-hand side of a resource-aware version of append's right identity law, we have used simple steps of inequational reasoning to derive the right-hand side. Each step of reasoning ensures denotational meaning is preserved while simultaneously maintaining or improving resource usage. Resource usage is made explicit in steps of reasoning by cost annotations. Furthermore, the location and quantity of each resource saving is made explicit through the use of quantified improvement. We remind readers that Liquid Haskell verifies the correctness of every proof step.

In this particular instance, quantified improvement shows that one recursive call is saved per inductive step of the proof, and hence append's right identity law is a left-to-right optimisation—with respect to number of recursive calls—precisely because `xs ++ []` evaluates to `xs`.

Cost diminishment and quantified diminishment. The notion of (*quantified*) *cost diminishment*, also presented in figure 2, is dual to (quantified) cost improvement. Using this notion, we can prove that `pure xs` is diminished by `xs ++ []`, by a cost of `length xs`: simply reverse the calculation steps of `rightIdQImp`, replacing instances of improvement with diminishment.

Note that `t1 >~> t2` if and only if `t2 <~< t1`, and likewise that `t1 >== n ==> t2` if and only if `t2 <== n ==< t1`. Similar relationships exist between other cost relations, for example, if `t1 >~> t2` and `t2 >~> t1`, then `t1 <=> t2`. All such relationships have been formalised using Liquid Haskell and the relevant proofs are available online [Anonymised 2019].

3.4 Library assumptions

To ensure its cost analysis is sound, the library makes two key assumptions: (1) expressions subject to cost analysis are not defined in terms of `tval` or `tcost`, nor do they perform case analysis on the `Tick` data constructor; and (2) Liquid Haskell's totality and termination checkers are active at all times. These assumptions are discussed in the remainder of this section.

3.4.1 Projections and case analysis. Expressions subject to resource analysis must not be defined in terms of `Tick`'s projection functions `tval` and `tcost`. This is to preserve the encapsulation of `Tick`'s accumulated cost. For example, `tval` can be used to (indirectly) show that two lists can be appended using `(++)` without incurring any cost:

```
{-@ freeAppend :: xs:[a] → ys:[a] → { t:Tick [a] |  tcost t == 0 } @-}
freeAppend xs ys = pure (tval (xs ++ ys))
```

From the type specification of `(++)`, we know that `tcost (xs ++ ys) == length xs`. However, the cost of `freeAppend` is zero as it uses `tval` to discards `(++)`'s incurred cost.

Similarly, user-defined functions can freely overwrite accumulated costs using the `tcost` projection or by performing case analysis on `Tick`'s data constructor.

Consequently, these primitives are not permitted in the definitions of expressions, as such expressions are not *safe* for cost analysis. (We formally define a safety predicate in section 5). Instead, users should always record resource usage *implicitly* using the functions provided by the `RTick` module.

Remark. Despite this assumption, the `tval` and `tcost` projection functions and the `Tick` data constructor are exported from the `RTick` module. This is because, as we've seen previously, these definitions are required in refinement specifications and extrinsic proof terms.

3.4.2 Totality and termination. Partial definitions, which Haskell permits, are not valid inhabitants of theorems expressed in refinement types [Vazou et al. 2018]. As such, the resource usage of partial definitions should not be analysed using the library. Similarly, partial definitions should not be used to prove theorems regarding the resource usage of other (total) annotated expressions.

Haskell can also be used to specify non-terminating computations. Divergence in refinement typing (in combination with lazy evaluation) can, however, be used to prove false predicates [Vazou et al. 2014]. Hence, the library's cost analysis is only sound for computations that require *finite* resources.

Liquid Haskell provides powerful totality and termination checkers that are active by default. Partial and/or divergent definitions will thus be rejected so long as these systems are not deactivated. The library, therefore, assumes that they remain active at all times.

4 EVALUATION

In this section, we present an evaluation of our library. The evaluation encompasses three detailed case studies: cost analyses of monolithic and non-strict implementations of insertion sort (sections 4.1 and 4.2), and a proof that the well-known map fusion technique is a correctness-preserving optimisation (section 4.3). The evaluation concludes with a summary of all of the examples we have studied (section 4.4), the majority of which have been adapted from the literature for comparison purposes.

4.1 Insertion sort

This case study analyses the number of comparisons required by the insertion sort algorithm. First, intrinsic cost analysis is used to prove a quadratic upper bound on the number of comparisons needed to sort a list of any configuration. We then use extrinsic cost analysis to prove a linear upper bound on the number of comparisons needed to sort a list that is already sorted.

To begin, we lift the standard insertion sort function into the `Tick` datatype:

```
isort []       = return []                insert x [] = pure [x]
isort (x : xs) = insert x =<< isort xs    insert x (y : ys)
                                            | x <= y    = wait (x : y : ys)
                                            | otherwise = pure (y :) </> insert x ys
```

According to the definition of `isort`, an empty list is already sorted: the result is simply embedded in the `Tick` datatype. To sort a non-empty list, its head is inserted into its recursively sorted tail. In this case, the flipped bind operator, `(=<<)`, sums up the costs of the insertion and the recursive sort. In turn, inserting an element into a sorted list is standard, with each comparison being recorded using the functions `wait` and `(</>)` introduced previously in section 3.

4.1.1 Intrinsic cost analysis. Refinement types can now be used to simultaneously specify properties about the correctness and resource usage of the above functions. In particular, abstract refinement types [Vazou et al. 2013] can be used to define sorted Haskell lists, that is, a list whereby the head of each sublist is less than or equal to every element in the tail:

```
{-@ type OList a = [a]<{ λ x y → x <= y }> @-}
```

The `OList` type constructor is used in the type of `insert` to ensure that its input `xs` is sorted:

```
{-@ insert :: Ord a => x:a → xs:OList a →
        { t:Tick { zs:OList a | length zs == 1 + length xs } | tcost t <= length xs } @-}
```

The result type of `insert` asserts that the function's output list `zs` is sorted and contains one more element than `xs`, and that an insertion requires at most `length xs` comparisons.

The specification for `isort` states that it returns a sorted list of the same length as its input, `xs`, and furthermore, that sorting `xs` requires at most $(length\ xs)^2$ comparisons:

```
{-@ isort :: Ord a => xs:[a] →
        { t:Tick { zs:OList a | length zs == length xs } | tcost t <= (length xs)^2 } @-}
```

Liquid Haskell automatically verifies `insert`'s specification. On the other hand, `isort`'s specification is rejected. This is because the resource usage of `insert x =<< isort xs` can only be calculated by performing type-level computations that are not automated by the system. At this point, we could switch to extrinsic cost analysis and perform the necessary calculations manually. However, we can also take a different approach that allows us to continue with our intrinsic analysis. The key to this approach is utilising the following function, which is a variant of `(=<<)`:

```
{-@ (=<<{·}) :: n:Int → f:(a → { tf:Tick b | tcost tf <= n }) → t:Tick a →
        { to:Tick b | tcost to <= tcost t + n } @-}
f =<<{n} t = f =<< t
```

From an operational stand point, the expression `f =<<{n} t` is equal to `f =<< t`. However, the refinement type of this 'bounded' version of `(=<<)` restricts its domain to functions `f :: a → Tick b` with execution costs no greater than `n`. Hence, the total resource usage of the expression `f =<<{n} t` cannot exceed the resource usage of `t` plus `n`.

Using `(=<<{·})` in the definition of `isort` allows Liquid Haskell to check the function's execution cost without performing any type-level computations. Thus, `isort`'s type can be automatically verified by specifying `length xs` as an upper bound on the cost of each insertion:

```
isort []       = return []
isort (x : xs) = insert x =<<{ length xs } isort xs
```

4.1.2 Extrinsic cost analysis. Next, we prove that the maximum number of comparisons made by `isort` is linear when its input is already sorted. We capture this property with the following extrinsic theorem that takes a sorted list as input. Therefore, `isort` does not need to be redefined.

```
{-@ isortCostSorted :: Ord a => xs:OList a → { tcost (isort xs) <= length xs } @-}
```

To prove this theorem, we consider three cases: when the input list is empty; when the list is a singleton, which invokes the base case of `insert`; and when the list has more than one element, which invokes the recursive case of `insert`. The first two cases follow immediately from the definitions of `isort` and `insert`, and thus can be automated by Liquid Haskell's PLE feature:

```
isortCostSorted []  = ()
isortCostSorted [x] = ()
```

When the input list contains more than one element, the proof begins by unfolding the definitions of `isort` and `(=<<{·})`, and then continues by appealing to the inductive hypothesis:

```
isortCostSorted (x : (xs@(y : ys)))
  =   tcost (isort (x : xs))
  ==. tcost (insert x =<<{ length xs } isort xs)
  ==. tcost (isort xs) + tcost (insert x (tval (isort xs)))
    ? isortCostSorted xs
  <=. length xs + tcost (insert x (tval (isort xs)))
```

At this point, we invoke a lemma that proves `tval (isort xs)` is an identity on `xs` when the list is sorted. (`isortSortedVal`'s proof is available online Anonymised [2019].)

```
    ? isortSortedVal xs
==. length xs + tcost (insert x xs)
==. length xs + tcost (insert x (y : ys))
```

As the input `(x : y : ys)` is sorted, we know that `x <= y`. Consequently, `insert x (y : ys)` will not recurse and unfolding the definitions of `insert` and `wait` completes the proof:

```
==. length xs + tcost (wait (x : y : ys))
==. length xs + 1
==. length (x : xs)
*** QED
```

Overall, this case study exemplifies how our library can be used to establish precise bounds on the resource usage of functions operating on subsets of their domains. In this instance, we imposed a 'sortedness' constraint on `isort`'s input using an extrinsic theorem, without needing to modify the function's definition. Furthermore, the proof relies on the fact that `isort`'s result is a sorted list in order to show that `tval (isort xs)` is an identity on `xs`. Hence, once more, we have demonstrated how correctness properties can be utilised for the purposes of precise cost analysis.

4.1.3 Resource propagation. The execution cost of any annotated function that utilises `isort` will in general be at least quadratic. For example, a minimum function defined by taking the head of a non-empty list that is sorted using `isort` also has a quadratic upper bound:

```
{-@ type NonEmpty a = { xs:[a] | length xs > 0 } @-}

{-@ minimum :: Ord a => xs:NonEmpty a → { t:Tick a | tcost t <= (length xs)^2 } @-}
minimum xs = pure head <*> isort xs
```

This is because, as discussed in section 2.3, `isort` is treated as a monolithic function given that it operates on standard Haskell lists. The cost of `pure head <*> isort xs`, therefore, includes the cost of *fully* evaluating `isort xs`. In practice, however, insertion sort does not need to be fully applied in order to obtain the least element in the input list. In particular, Haskell's lazy evaluation strategy will halt the sorting computation as soon the head of the result is generated. Next, we see how the `Tick` datatype can be used to explicitly encode this kind of non-strict behaviour.

4.2 Non-strict insertion sort

Our cost analysis treats functions operating on standard Haskell datatypes as monolithic. To encode non-strict evaluation, we include `Tick` in the definitions of datatypes in order to suspend computations. Datatypes defined using `Tick` are called *lazy* and functions that operate on them *non-strict*.

In this case study, which is adapted from [Danielsson 2008], we define a non-strict minimum function to calculate the least element in a non-empty lazy list that has been sorted using insertion sort. The execution cost of the new minimum function has a linear upper bound, which corresponds to the resources required by Haskell's on-demand evaluation.

4.2.1 Refined lazy lists. Following [Danielsson 2008], we define lazy lists to be either empty (`Nil`) or constructed (`Cons`) from a pair of a `lhead :: a` and a `ltail :: Tick (LList a)`, which is an annotated computation that returns a lazy list. Furthermore, to encode recursive properties into lazy lists, we use an abstract refinement `p` to capture invariants that hold between the head of a lazy list and each element of its tail, and moreover, that recursively hold inside its tail:

```
{-@ data LList a <p :: a → a → Bool> =
       Nil | Cons { lhead :: a, ltail :: Tick (LList <p> (a <p lhead>)) } @-}
```

Sorted lazy lists are defined similarly to `OList a`, by instantiating the abstract refinement to express that the head of each sublist is less than or equal to any element in the tail:

```
{-@ type OLList a = LList<{ λ x y → x <= y }> a @-}
```

4.2.2 Non-strict sorting. We can now define a non-strict version of the insertion function using lazy lists. The key distinction between `insert` and `linsert` is that in the definition below, the recursive call to `linsert` is *suspended* and stored in the tail of the resulting list.

```
{-@ linsert :: Ord a => a → xs:OLList a →
        { t:Tick { zs:OLList a | llength zs == length xs + 1 } | tcost t <= 1 } @-}
linsert x Nil =  return (Cons x (return Nil))
linsert x (Cons y ys)
  | x <= y    = wait (Cons x (return (Cons y ys)))
  | otherwise = wait (Cons y (ys >>= linsert x))
```

When analysing functions that operate on standard Haskell datatypes, we have seen that execution cost corresponds to such functions being fully applied. Now we see that the execution cost of non-strict functions corresponds to such functions returning the *first parts* of their results. In this instance, `linsert` returns the first element in its resulting lazy list by making one comparison when its input is non-empty and zero comparisons otherwise: `tcost t <= 1`.

Non-strict insertion sort, `lisort`, is analogous to `isort`, however its result is a sorted lazy list:

```
{-@ lisort :: Ord a => xs:[a] →
        { t:Tick { zs:OLList a | llength zs == length xs } | tcost t <= length xs } @-}
lisort []       = return Nil
lisort (x : xs) = linsert x =<<{1} lisort xs
```

Given a standard Haskell list as input, `lisort` returns a sorted lazy list. Hence, it is a non-strict function and its execution cost reflects the maximum number of comparisons required to produce the first element in its result. Notice that this execution cost has been intrinsically verified because `(=<<{·})` accurately approximates the execution cost of `linsert` at each recursive call.

4.2.3 Non-strict minimum. The following non-strict minimum function, `lminimum`, returns the first element in a non-empty list `xs` that is *partially* sorted using `lisort`. As `lminimum` only forces the first element of `lisort xs` to be calculated, it requires at most `length xs` comparisons:

```
{-@ lminimum :: Ord a => xs:NonEmpty a → { t:Tick a | tcost t <= length xs } @-}
lminimum xs = pure lhead <*> lisort xs
```

4.2.4 Explicit laziness. Lazy lists of type `LList a` are defined such that examining the head is zero-cost, but examining the last element has a cost equal to the sum total of the costs of each suspended computation in the tail. As discussed in section 2.3, if such a list is fully evaluated on multiple occasions during a computation, the library's default analysis records the cost of each evaluation independently. However, in practice, once a list is fully evaluated by Haskell's operational semantics, its value is memoised and thus subsequent uses are 'cost-free'.

To explicitly capture memoisation in our analysis, we use `pay` from [Danielsson 2008]:

```
{-@ pay :: m:Nat → { t₁:Tick a | tcost t₁ >= m } →
        { t:Tick (Tick a) | tcost (tval t) == tcost t₁ - m } @-}
pay m (Tick n x) = Tick m (Tick (n - m) x)
```

Evaluating `pay m x >>= f` allows `f` to use `x` numerous times while only paying `m` cost for it once. Therefore, if `m == tcost x` then this effectively models memoisation.

We repeated Danielsson's analysis Danielsson [2008] of Okasaki's queues as part of the library's evaluation (section 4.4). In this example, non-strictness is captured by defining a lazy queue datatype and sharing is modelled explicitly by defining *lazy* functions that are non-strict and use `pay`.

4.3 Map fusion

In this case study, we use the proof combinators from section 3.3 to simultaneously reason about the correctness and efficiency of *map fusion*. This well-known property states that mapping one function `f :: a → b` followed by another function `g :: b → c` over a list `xs :: [a]` gives the same result as mapping the composite function `g . f :: a → c` over the same list:

```
map g (map f xs) == map (g . f) xs
```

Although the two sides of this equation give the same result, they do not require the same amount of resources. In particular, the left-hand side traverses the list `xs` twice, whereas the right-hand side traverses `xs` only once. Thus, replacing the expression on the left-hand side with that on the right preserves correctness while saving `length xs` resources. To prove this, we first define annotated versions of the mapping and function composition operators, and then reason simultaneously about the correctness and efficiency of those definitions.

4.3.1 Definitions. First, we define an annotated mapping function, `mapM`, which takes as input a function `f :: a → Tick b` returning an annotated result and a list `xs`. The cost of `mapM`'s result, given by applying `f` to each element `x` in the list `xs`, includes the number of recursive calls made during `mapM`'s execution *and* the cost of each application `f x`:

```
mapM :: (a → Tick b) → [a] → Tick [b]
mapM _ []       = pure []
mapM f (x : xs) = step 1 (liftA2 (:) (f x) (mapM f xs))
```

In particular, when the input list is empty, no resources are consumed; and when the input list is non-empty, `step 1` is used to record the recursive call to `mapM`, and `liftA2` (defined subsequently) reconstructs the list while recording the cost of the application `f x`.

Remark: To the best of our knowledge, it is not possible to define a refinement type for `mapM` that precisely describes its resource usage. This is because `f` is applied to arbitrary inputs `x` from the list `xs`, and thus we cannot specify the cost of each `f x` in the general case. One way to approximate this cost is to employ the technique described in section 4.1, which is to establish an upper bound `n` on the cost of `f x` for any input `x`. The total execution cost of `mapM` would then be bounded above by `(n + 1) * length xs`. Nonetheless, we can prove that map fusion is an optimisation (a relational cost property) without needing to precisely compute the cost of each `f x` (a unary cost property). This is a notable advantage of relational cost analysis [Çiçek 2018].

The `liftA2` function is defined in the `RTick` library. Similarly to the applicative operator `(<*>)`, `liftA2` takes as input a binary function `f` and two annotated arguments, and returns an annotated result whose cost is equal to the sum of the costs of its arguments:

```
{-@ liftA2 :: f:(a → b → c) → t1:Tick a → t2:Tick b →
        { t:Tick c | tval t == f (tval t1) (tval t2) && tcost t == tcost t1 + tcost t2 } @-}
liftA2 f (Tick m x) (Tick n y) = Tick (m + n) (f x y)
```

To compose two annotated functions `f` and `g`, we define a composition function `(>=>)` that, when given an argument `x`, applies `g` to the value of `f x` and sums the costs of both applications:

```
(>=>) :: (a → Tick b) → (b → Tick c) → a → Tick c
(>=>) f g x = let Tick m y = f x in let Tick n z = g y in Tick (m + n) z
```

4.3.2 Specification. Using the above definitions and cost relations introduced in section 3.3, we can now state that the map fusion technique is a cost improvement in the left-to-right direction. Specifically, we can use *quantified improvement* to precisely capture the amount of resources saved by the optimisation, which is given by the length of the list `xs` being traversed:

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

The following extrinsic theorem formalises the above property in Liquid Haskell:

```
{-@ mapFusion :: f:(a → Tick b) → g:(b → Tick c) → xs:[a] →
       { (mapM f xs >>= mapM g) >== length xs ==> (mapM (f >=> g) xs) } @-}
```

4.3.3 Proof. To prove the `mapFusion` theorem, we must define a Haskell term that inhabits its type specification. In practice, we define such a term by performing (in)equational rewriting on the left-hand side of the proof statement, `mapM f xs >>= mapM g`, ultimately deriving the right-hand side, `mapM (f >=> g) xs`. By utilising the proof combinators from section 3.3, we ensure that each rewrite step preserves correctness (value equivalence). Furthermore, such combinators capture the total resource saving, which is calculated 'on the fly' as part of the derivation process. The proof proceeds in the standard manner by induction on the list argument.

In the base case, we begin by unfolding the definitions of `mapM` and `(>>=)`. The right-hand side of the proof statement then follows by folding the definition of `mapM`:

```
mapFusion f g []
  =.  mapM f [] >>= mapM g
  <=. pure [] >>= mapM g
  <=. mapM g []
  <=. pure []
  <=. mapM (f >=> g) []
  *** QED
```

In this case, we can see that the map fusion technique is a *cost equivalence*. That is, the costs of both sides of the property are equal. This is to be expected as the length of the input list `xs` is zero, and hence no resources are saved: `(mapM f [] >>= mapM g) <=> (mapM (f >=> g) []).`

The inductive case also begins by unfolding the definition of `mapM`:

```
mapFusion f g (x : xs)
  =.  mapM f (x : xs) >>= mapM g
  <=. (step 1 (liftA2 (:) (f x) (mapM f xs))) >>= mapM g
```

Next, we use quantified improvement to precisely capture the cost saved by eliminating `step 1`:

```
  .>== 1 ==>. liftA2 (:) (f x) (mapM f xs) >>= mapM g
```

To continue on with the proof, we must unfold the definition of `liftA2`. By deconstructing the results of `f x` and `mapM f xs` by pattern matching on the `Tick` datatype in a **`where`** clause, we can independently refer to the costs and values of `liftA2`'s arguments:

```
  where Tick cf  fx  = f x
        Tick cfs fxs = mapM f xs
```

Storing these parameters is particularly useful as the remaining cost savings and expenditures can be expressed entirely in terms of `cf`, `cfs`, and constants. This makes subsequent cost manipulations straightforward, and allows us to focus primarily on correctness.

Using the bindings from the **`where`** clause, we can unfold the definition of `liftA2` and continue rewriting: saving the cost of `liftA2`'s result and unfolding the definitions of `(>>=)` and `mapM`.

```
    <=>. Tick (cf + cfs) (fx : fxs) >>= mapM g
    .>== cf + cfs ==>. pure (fx : fxs) >>= mapM g
    <=>. mapM g (fx : fxs)
    <=>. step 1 (liftA2 (:) (g fx) (mapM g fxs))
```

Eliminating `step 1` saves an additional resource, however, we must then expend `cfs` resources in order to map `f` over the tail of the input list, `xs`:

```
    .>== 1 ==>. liftA2 (:) (g fx) (mapM g fxs)
    .<== cfs ==<. liftA2 (:) (g fx) (mapM f xs >>= mapM g)
```

At this point, we can appeal to the inductive hypothesis in order to save `length xs` resources by substituting `mapM (f >=> g) xs` for `mapM f xs >>= mapM g`:

```
      ? mapFusion f g xs
    .>== length xs ==>. liftA2 (:) (g fx) (mapM (f >=> g) xs)
```

To finalise the proof, we apply `f` to the head of the input list `x` and fold the definition of `mapM`:

```
    .<== cf ==<. liftA2 (:) ((f >=> g) x) (mapM (f >=> g) xs)
    .<== 1 ==<. step 1 (liftA2 (:) ((f >=> g) x) (mapM (f >=> g) xs))
    <=>. mapM (f >=> g) (x : xs)
    *** QED
```

As we have seen throughout the proof, the quantified cost operators are used to explicitly record resource saving, for example `(.>== cf + cfs ==>.)`, and expenditure, for example `(.<== cf ==<.)`. Overall, the costs savings and expenditures involving `cf` and `cfs` cancel out, as do the latter two costs involving `step 1`. The remaining costs are from the initial saving of `1` from `step 1`, and the saving of `length xs` from the inductive hypothesis. Hence, the resulting expression, `mapM (f >=> g) (x : xs)`, requires `length (x : xs)` *fewer* resources than the initial expression, `mapM f (x : xs) >>= mapM g`, as expected. As Liquid Haskell has SMT support for arithmetic, this overall cost saving is calculated automatically by the system.

In summary, this proof illustrates the power of relational cost analysis in our setting. In particular, the costs of `f x` and `mapM f xs` cannot be easily captured by a unary cost analysis of `mapM`. Nevertheless, our extrinsic approach overcomes this restriction by allowing such 'higher-order costs' to cancel out on both sides of the theorem's proof statement. Furthermore, storing the costs of `f x` and `mapM f xs` in a **`where`** clause allowed us to primarily focus on the correctness aspect of the proof. As such, we have not only shown how reasoning about resource usage can be as straightforward as reasoning about correctness, we have shown that the two can in fact *coincide*.

4.4 Summary of examples

To finalise the library's evaluation, we provide a summary of all of the examples we have surveyed during its development. Each example's source files are available online [Anonymised 2019].

Overview. Table 1 provides a quantitative summary of each example and is split into five categories. The first three categories include examples from the existing literature, the fourth category consists of higher-order examples, and the final category includes the complexity analysis of different sorting algorithms. An overview of the five categories is provided below.

– **Laziness** includes functions that manipulate lazy lists and lazy queues from [Danielsson 2008]. For example, in section 4.2, we proved that non-strict insertion sort on lazy lists is linear. We also encoded lazy queues and proved that viewing a lazy queue and appending at the end are constant-time operations. Danielsson [2008] reifies cost using a type-level index, namely `Thunk n a` where `n` is a type-level `Nat`, while we use a value-level integer field. Because of this distinction,

	Property	Lines of code		
		Code	Spec	Proof
Laziness [Danielsson 2008]				
Insertion sort	COST(lisort xs) $\leq \lvert xs \rvert$	12	8	0
Implicit queues	COST(lsnoc q x) = 5, COST(view q) = 1	50	14	0
Relational [Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2018]				
2D count	COST(2DCount find$_1$) $\leq$ COST(2DCount find$_2$)	16	3	24
Binary counters	COST(decr k t) = COST(incr k ff)	26	21	21
Boolean expressions	NOSHORT(e) $\Rightarrow$ COST(eval$_1$ e) = COST(eval$_2$ e)	28	2	13
Constant-time comparison	COST(compare p u$_1$) = COST(compare p u$_2$)	3	8	3
Insertion sort	SORTED(xs) $\Rightarrow$ COST(isort xs) $\leq$ COST(isort ys)	16	17	44
Memory allocation of length	COST(length$_2$ xs) - COST(length$_1$ xs) = $\lvert xs \rvert$	10	4	6
Relational insertion sort	COST(isort xs) - COST(isort ys) = unsortedDiff xs ys	16	11	69
Relational merge sort	COST(msort xs) - COST(msort ys) $\leq \lvert xs \rvert(1 + \log_2(\text{diff xs ys}))$	23	25	59
Square and multiply	COST(sam t x l$_1$) - COST(sam t x l$_2$) $\leq$ t * 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	COST(xs ++ys) = $\lvert xs \rvert$	8	3	0
Flattening	PERFECT(t) $\Rightarrow$ COST(flattenOpt t) = $2^{\lvert t \rvert} - 1$	5	18	45
Optimised-by-construction reverse	reverse xs >== $\lvert xs \rvert$ ==> fastReverse xs	18	37	140
Reversing (naive)	COST(reverse xs) = $\frac{\lvert xs \rvert^2}{2} + \frac{\lvert xs \rvert + 1}{2}$	9	7	22
Reversing (optimised)	COST(fastReverse xs) = $\lvert xs \rvert$	5	8	0
Higher order				
fold	COST(foldl xs) == $\lvert xs \rvert$, COST(foldl' xs) == 0	8	4	0
foldM	COST(foldlM xs) == (1 + n) $\lvert xs \rvert$, COST(foldlM' xs) == n $\lvert xs \rvert$	8	4	0
foldM relational	foldlM xs >== $\lvert xs \rvert$ ==> foldlM' xs	8	4	22
Map fusion	mapM f xs >>= mapM g >== $\lvert xs \rvert$ ==> mapM (f >=> g) xs	4	3	31
Sorting				
`Data.List.sort`	COST(ssort xs) $\leq 4 \lvert xs \rvert \log_2 \lvert xs \rvert + \lvert xs \rvert$	39	49	107
Insertion sort	COST(isort xs) $\leq \lvert xs \rvert^2$	8	10	33
Merge sort	$\frac{xs}{2} \log_2 \lvert xs \rvert \leq$ COST(msort xs) $\leq \lvert xs \rvert + \lvert xs \rvert$	22	69	139
Quicksort	COST(qsort xs) $\leq \frac{1}{2}(\lvert xs \rvert + 1)(\lvert xs \rvert + 2)$	15	8	27
Total		372	355	882

Table 1. Cost analysis using the `RTick` library. *Code* reports the lines of executable code, *Spec* reports the lines of specifications, and *Proof* reports the lines of proof terms.

Danielsson [2008] does not require ghost cost parameters (as per our `=<<{·}` of section 4.1). On the other hand, type-level costs cannot be abstracted, which is a requirement of our higher-order examples. Finally, as our analysis builds on top of Liquid Haskell's existing features, it incorporates additional (automated) correctness properties such as 'sortedness'.

– **Relational** comprises *all* the cost analysis examples from [Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2018]. These examples compare the resource usage of the same function on different inputs, for instance constant-time comparison from section 2.2, or different functions on the same input, for instance the memory allocation case study compares the memory required by the standard and tail recursive implementations of the `length` function.
This set of examples highlights a number of distinctions between our approach and relational refinement type systems developed for resource analysis. First of all, our system is agnostic to the resource being analysed, which means that the user has the flexibility to define arbitrary resources but also the responsibility to manually annotate resource usage. In comparison, the approach taken by [Çiçek et al. 2017] only analyses runtime complexity.
Secondly, unary cost analysis in our setting is automatically checked, but users must specify appropriate cost bounds manually, which adds some complexity to the analysis. In relational systems, such annotations are not required when the analysis is performed using 'synchronous'

rules. However, when synchronous rules fail, relational systems essentially replicate the unary analysis automatically performed by our system. Whether to use the synchronous or 'asynchronous' approach is dictated by heuristics [Çiçek et al. 2019].
And finally, in many of our examples, we must manually prove extrinsic theorems that can be automatically inferred by relational type systems. This is to be expected, as such systems are specialised for resource tracking. Nonetheless, these systems cannot encode the sophisticated correctness invariants, such as sortedness, that we frequently use to simplify our analysis.

– **Datatypes** includes properties concerning lists, trees, and function optimisations whose Liquid Haskell correctness proofs initially appeared in [Vazou et al. 2018]. We used the proof combinators of figure 2 to extend the correctness proofs with explicit resource tracking. Our experience, in accordance with the case study of section 4.3, is that because Liquid Haskell has SMT-automated integer arithmetic, reasoning about resource usage is as straightforward as reasoning about correctness. In fact, most of our proofs are very similar to their counterparts in [Vazou et al. 2018].

– **Higher-order** includes three higher-order examples. As per example 2 in section 2, we tracked the number of thunks allocated by `foldl` and `foldl'`. We then extended the analysis, considering `foldM` and `foldM'` whose arguments can also allocate thunks. For unary analysis, the cost of the function being folded is bounded above by a ghost cost parameter `n`. The relational comparison between `foldM` and `foldM'` does not require this bound and is greatly simplified using our proof combinators, similarly to the map fusion case study of section 4.3.
This category illustrates two key features of our analysis. Firstly, tracked resources can have arbitrary, user-defined meanings, such as number of allocated thunks. And secondly, our analysis supports higher-order functions, whose resource analysis is greatly simplified in a relational setting.

– **Sorting** includes the cost analysis of well-known sorting algorithms: `Data.List`'s *smooth* merge sort, insertion sort, merge sort, and quicksort. Other than the known upper bounds of the algorithms, we proved a lower bound for merge sort (section 2.2) and that both insertion sort (section 4.1) and smooth merge sort require at most linear comparisons when applied to sorted lists.
Two of the functions listed above have logarithmic bounds. We axiomatised logarithmic properties as Haskell functions using Liquid Haskell's **`assume`** feature. To prove these complexity bounds we used extrinsic reasoning, making explicit calls to the axioms when necessary. This showcases another feature of our analysis: despite Liquid Haskell only providing SMT automation for linear arithmetic, our analysis is still able to check arbitrarily expressive resource bounds.

Overall, we chose these examples because they: required both unary and relational cost analysis; often imposed constraints on the inputs to functions; were reasonably challenging to encode using our library; allowed us to draw comparisons against existing systems. Importantly, all of the examples demonstrate how correctness properties can be naturally integrated into our library's cost analysis.

Breakdown. Each line in table 1 describes an indicative property we have proved. In some cases, we have proved additional properties. In other cases, the desired property required proving a stronger theorem. Due to space limitations, these additional properties are not included in the table. However, the source files for all of the examples are available on the library's GitHub page [Anonymised 2019].

Synopsis. In total, we wrote 372 lines of executable code; 355 lines of Liquid Haskell specifications and 882 lines of proof terms. The total lines of code dedicated to specifications and proofs is approximately three times as much as executable code. Given the complexity of the properties we have proved, we consider this reasonable. Moreover, the size of many proof terms has been decreased by using Liquid Haskell's PLE feature [Vazou et al. 2017].

$$
\begin{array}{rrcl}
\textit{Constants} & c & ::= & 0, 1, -1, \ldots \mid \texttt{true}, \texttt{false} \\
 & & \mid & +, -, \ldots \mid =, <, \ldots \mid \texttt{crash} \\
\textit{Values} & w & ::= & c \mid \lambda x.e \mid D\ \overline{e} \\
\textit{Expressions} & e & ::= & w \mid x \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid & \texttt{case}\ x = e\ \texttt{of}\{D\ \overline{y} \rightarrow e\} \\
\textit{Refinements} & r & ::= & e \\
\textit{Basic types} & B & ::= & \texttt{Int}, \texttt{Bool}, \texttt{T} \\
\textit{Types} & \tau & ::= & \{v{:}B \mid r\} \mid x{:}\tau \rightarrow \tau \\
\textit{Contexts} & \mathbb{C} & ::= & \cdot \mid \mathbb{C}\ e \mid c\ \mathbb{C} \mid D\ \overline{e}\ \mathbb{C}\ \overline{e} \\
 & & \mid & \texttt{case}\ x = \mathbb{C}\ \texttt{of}\{D\ \overline{y} \rightarrow e\}
\end{array}
$$

Reduction

$$
\begin{array}{rcll}
\mathbb{C}[e] & \hookrightarrow & \mathbb{C}[e'] & \text{if } e \hookrightarrow e' \\
c\ w & \hookrightarrow & \delta(c, v) & \\
(\lambda x.e)e_x & \hookrightarrow & e[e_x/x] & \\
\texttt{let}\ x = e_x\ \texttt{in}\ e & \hookrightarrow & e[e_x/x] & \\
\texttt{case}\ x = D_j\ \overline{e}\ \texttt{of}\{D_i\ \overline{y_i} \rightarrow e_i\} & \hookrightarrow & e_j[D_j\ \overline{e}/x][\overline{e}/\overline{y_i}] &
\end{array}
$$

Figure 3. λ^U: Syntax and Operational Semantics as in [Vazou et al. 2014].

5 CORRECTNESS OF STATIC COST ANALYSIS

In this section, we prove the correctness of our cost analysis using the metatheory of Liquid Haskell.

5.1 Metatheory of Liquid Haskell

Figure 3 summarises the syntax and operational semantics of λ^U, the core language used to model Liquid Haskell [Vazou et al. 2014]. The language λ^U includes constants, abstractions, applications, let and case statements, and datatypes. Its operational semantics is defined as a contextual, small-step, call-by-name relation $\hookrightarrow$ whose reflective, transitive closure is denoted by $\hookrightarrow^\star$.

Types. The basic types in λ^U are integers, booleans, and type constructors. Types are either refinement types of the form $\{v{:}B \mid e\}$ where the basic type B, captured by the variable v, is refined by the boolean expression e; or dependent function types of the form $x{:}\tau_x \rightarrow \tau$, where the input x has the type τ_x and the result type τ may refer to the binder x.

Denotations. Each type τ denotes a set of expressions $[\![\tau]\!]$, defined by the dynamic semantics in [Vazou et al. 2014]. Let $\lfloor\tau\rfloor$ be the type obtained by erasing all refinements from τ, and $e{:}\lfloor\tau\rfloor$ be the standard typing relation for the λ-calculus. Then, we define the denotation of types as follows:

$$
\begin{array}{rcl}
[\![\{x{:}B \mid e_r\}]\!] & \doteq & \{e \mid e{:}B, \text{if } e \hookrightarrow^\star w, \text{then } e_r[w/x] \hookrightarrow^\star \texttt{true}\} \\
[\![x{:}\tau_x \rightarrow \tau]\!] & \doteq & \{e \mid e{:}\lfloor x{:}\tau_x \rightarrow \tau\rfloor, \forall e_x \in [\![\tau_x]\!].e\ e_x \in [\![\tau[e_x/x]]\!]\}
\end{array}
$$

Syntactic typing. The typing judgement $\Gamma \vdash e{::}\tau$ decides syntactically if e is a member of τ's denotation using the environment Γ that maps variables to their types: $\Gamma \doteq x_1{:}\tau_1 \ldots x_n{:}\tau_n$.

To analyse resource usage in λ^U we do not need to modify the typing rules [Vazou et al. 2014]. Instead, we can use λ^U constants to encode `Tick`'s annotation functions. This approach corresponds to our implementation, as we define `RTick` as a library without changing the underlying behaviour of Liquid Haskell. To type a λ^U constant c, we use the meta-function $\mathrm{Ty}(c)$ that returns the type of c:

$$\frac{}{\Gamma \vdash c{::}\mathrm{Ty}(c)}\ \textsc{T-Con}$$

To ensure the soundness, $\mathrm{Ty}(c)$ should satisfy denotational inclusion $c \in [\![\mathrm{Ty}(c)]\!]$. For example:

$$\begin{array}{rcl} \mathrm{Ty}(3) & \doteq & \{v{:}\texttt{Int} \mid v == 3\} \\ \mathrm{Ty}(+) & \doteq & x{:}\texttt{Int} \to y{:}\texttt{Int} \to \{v{:}\texttt{Int} \mid v == x + y\} \end{array}$$

Soundness of λ^U. The soundness of λ^U states that if each constant belongs to the denotation of its assumed type, then syntactic typing implies denotational inclusion:

THEOREM 5.1 (SOUNDNESS OF λ^U). *If for all c, $c \in [\![Ty(c)]\!]$, then if $\emptyset \vdash e::\tau$, then $e \in [\![\tau]\!]$.*

5.2 Correctness of cost analysis

As λ^U contains type constructors, data constructors, and constants, but does not support type polymorphism, we formalise our approach by defining the `Tick` datatype and a number of its annotation functions as a type family, where each function is a λ^U constant. The correctness of our cost analysis is then simply a corollary of the soundness of λ^U.

The Tick datatype. For each type τ, we define a datatype $\texttt{Tick}_\tau$ with a single data constructor: $\texttt{Tick}_\tau :: \texttt{Int} \to \tau \to \texttt{Tick}_\tau$. $\texttt{Tick}_\tau$ data constructors should not be used directly. Instead, each $\texttt{Tick}_\tau$ datatype should be accessed implicitly using the constants defined below.

Resource annotations. We define the following annotation functions from section 3.2 as λ^U constants: $return_\tau$, $bind_{\tau,\tau'}$, $step_\tau$, $tcost_\tau$, $tval_\tau$ for each types τ, τ'. We use λ^U to define the types and (type-specific) bodies of each constant just as in section 3.2. Because Liquid Haskell type-checks the previous definitions, we have $c \in [\![\mathrm{Ty}(c)]\!]$ for each constant $return_\tau$, $bind_{\tau,\tau'}$, and so on. Therefore, these constants can be used *safely* in λ^U while preserving soundness.

Safe expressions. Recall from section 3.4 that the library imposes the following restrictions on annotated expressions to correctly analyse their resource usage: firstly, expressions should not be defined using `tval` or `tcost`; secondly, expressions should not perform case analysis on the `Tick` data constructor. We formalise these restrictions by defining a safety predicate on λ^U expressions:

Definition 5.2 (Safety). A λ^U expression e is *safe iff*:

- $e{:}\tau$, that is, e is typeable;
- e's body is not defined in terms of any $tcost_\tau$ or $tval_\tau$ constants;
- e does not perform case analysis on any $\texttt{Tick}_\tau$ data constructors.

Execution cost. Consider a safe, terminating function f that returns a value of type $\texttt{Tick}_\tau$ for some type τ, that is, $f :: x{:}\tau_x \to \texttt{Tick}_\tau$. We define the execution cost of f on an input $e_x :: \tau_x$ to be the index of the returned value. In other words, the execution cost of $f\ e_x$ is i where $f\ e_x \hookrightarrow^\star \texttt{Tick}_\tau\ i\ v$. As f does not directly modify any $\texttt{Tick}_\tau$ datatypes, all resource consumptions or productions via applications of $step_\tau$ in f's definition accumulate in the cost i of the final value, $\texttt{Tick}_\tau\ i\ v$.

Static cost analysis. Finally, we use the soundness of λ^U to show that the library's intrinsic and extrinsic approaches are both correct with respect to the above definition of execution cost:

THEOREM 5.3 (CORRECTNESS OF COST ANALYSIS). *Let $p :: \texttt{Int} \to \texttt{Bool}$ be a predicate over integers and $f :: x{:}\tau_x \to \tau$ a safe and terminating function.*

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

The proof of this theorem follows immediately from the soundness of the core language λ^U, the denotations of dependent function types, and the definition of $tcost_\tau$.

Other annotations. Theorem 5.3 proves that the library's cost analysis is consistent for annotated expressions defined using `return`, `(>>=)`, and `step`. However, the `RTick` module provides many more annotation functions, for example, `pure` and `(<*>)` introduced in section 3.2. However, all such functions can be defined using `return`, `(>>=)`, and `step`: a proof of this fact can be found on the library's GitHub page [Anonymised 2019]. Thus, we implicitly extend theorem 5.3 to include expressions defined using any of the helper functions provided by the `RTick` module.

6 RELATED WORK

Our work has been strongly influenced by Danielsson's [2008] lightweight framework for cost analysis in Agda. Danielsson's library is based on the *Thunk* datatype, which is indexed with a dependent type used to measure the runtime complexity of purely functional algorithms and data structures in the style of Okasaki. Our `Tick` datatype is comparable to `Thunk`, but captures abstract resource usage at the value-level. Much of *Thunk*'s analysis requires basic equality proofs because Agda does not automatically prove arithmetic equalities. In contrast, our use of Liquid Types allows us to delegated all (linear) arithmetic necessary for our cost analysis to an SMT solver. Another notable distinction between the two approaches is that our library supports unary and relational cost analysis, whereas the *Thunk* library only supports the unary variant.

Indexed types have been widely used for resource analysis. [Crary and Weirich 2000] indexes the type of functions to compute the number of recursive calls required by their executions. Sized types [Hughes et al. 1996; Vasconcelos and Hammond 2003], which index types with natural numbers that denote the size of their values, have also been used to analyse runtimes. However, none of these approaches can express correctness properties, which as we have seen, allow for more precise analysis. Recent work [McCarthy et al. 2017; Wang et al. 2017] combines indexed types with functional correctness. [McCarthy et al. 2017] develops a Coq library that uses a monad indexed by a predicate to measure runtimes. The approach is comparable to [Danielsson 2008], however the predicate is used to express invariants of data structures. This enables more complex case studies (such as Okasaki's Braun Trees) to be examined. Another distinction is that cost annotations can be automatically inserted, and then erased when code is extracted. A method for automated annotation is part of our future work. Similarly to [Danielsson 2008], relational cost analysis is not supported. TiML [Wang et al. 2017] indexes the types of functions with their time bounds. A significant feature of this system is that it provides automated support for solving recurrence relations, by heuristically matching against cases of the Master Theorem. In comparison, we use extrinsic proofs to manually derive time complexity theorems. Similarly to our approach, TiML supports sophisticated invariants, however, they are only exploited for the purposes of cost analysis. Our library on the other hand uses invariants to simultaneously reason about correctness and resource usage.

Automatic Amortized Resource Analysis (AARA) [Hofmann and Jost 2003] aims to automatically derive amortised bounds on execution cost. This is achieved using a type system that generates resource-specific inequalities to be solved by a linear programming solver. The initial system [2003] supports linear bounds on monomorphic, first-order programs, but this has since been generalised to incorporate polynomial bounds [Hoffmann et al. 2011, 2012], higher-order functions [Jost et al. 2010], parallelism [Hoffmann and Shao 2015], and most recently, a Haskell-like lazy semantics [Jost et al. 2017]. As AARA focuses on automatically inferring bounds, its analysis is often less precise than ours. In particular, our library's extrinsic resource analysis can notionally compute resource bounds of any kind: examples of polynomial, logarithmic, and polylogarithmic bounds appear throughout the article. In comparisons, AARA is (at best) restricted to polynomial bounds, though such bounds can be automatically inferred. Correctness invariants are not supported by AARA.

RelCost [Çiçek et al. 2017; Çiçek 2018] is a refinement type and effect system for both relational and unary cost analysis. The main idea is to reason about structurally related expressions as much as

possible in order to calculate precise resource bounds via relational cost analysis. When programs or inputs are not structurally related, the system reverts back to performing unary cost analysis. This is achieved using two 'modes' of typing: one for similar expressions and one for unrelated expressions. Liquid Haskell only supports one mode of typing, nevertheless, our library fully supports relational cost analysis by way of extrinsic theorems. The refinements used by Liquid Haskell are more expressive than those used by *RelCost*, which allows us to consider additional examples.

BiRelCost [Çiçek et al. 2019] is a bidirectional type checker for *RelCost*, implemented in OCaml. This system, which appears to be the first of its kind, is able to type check all of the examples presented in [Çiçek et al. 2017], and does so automatically while only requiring minimal annotations from the user. However, the implementation is incomplete and relies on example-driven heuristics to avoid nondeterminism in its type checking process. Nondeterminism (and completeness) is not a concern for our system, but we have seen throughout the article that users are often required to provide manual proofs of resource usage, specifically for our extrinsic approach. Fundamentally, we see this as a compromise between expressiveness and automation.

[Radiček et al. 2018] develops theoretical frameworks for unary and relational cost analysis implemented in RHOL. The underlying language includes a monad used to encapsulate expressions with cost, much like our `Tick` datatype, which shares the same monadic implementation. Similarly to our approach, the frameworks can express correctness properties that allow for more precise analysis. Our library is equally as expressive given that the combined results of [Vazou et al. 2017] and [Aguirre et al. 2017] are equally as expressive as HOL. The authors of [Radiček et al. 2018] note that the use of a cost monad "syntactically separates reasoning about costs from reasoning about functional properties, thus improving clarity in proofs". From our experience, reasoning independently about correctness and resource usage (using the `tval` or `tcost` project functions, respectively) can indeed simplify steps of (in)equational reasoning, especially in the latter case. On the other hand, we have also demonstrated that reasoning about both simultaneously can be very helpful, for example, when considering higher-order properties such as map fusion.

[Madhavan et al. 2017] presents a system that can verify resource bounds for higher-order functional programs with lazy evaluation written in Scala. Similarly to our system, users must specify desired resource bounds. However, such bounds are *templates*, which may contain 'numerical holes' that are automatically inferred. During this verification process, programs are transformed to make their resource usage explicit. In particular, the forcing of thunks is made explicit as per our *pay* function. It is worth noting that this work contains examples involving logarithmic bounds (as well as polynomials), which appears to be fairly uncommon in the literature.

Improvement theory [Moran and Sands 1999] inspired our notions of improvement and quantified improvement. However, previously Sands [1995] introduced improvements as a semantic approach to relational cost analysis, which can be used to prove equivalences between programs. Similarly, improvements in this context only offer a qualitative guarantee that one program uses no more resource than another. In this work, we have extended this notion to quantify such guarantees.

7 CONCLUSION AND FURTHER WORK

This article has demonstrated how refinement types can be used to reason in a precise manner about execution cost. In particular, we have developed a Liquid Haskell library that can be used to analyse the resource usage of pure Haskell programs. Furthermore, by surveying a wide range of examples from the literature, we have shown how the system's existing support for correctness verification can be harnessed to ensure cost analysis is valid and precise.

There are three main avenues for further work. Firstly, we would like to use metaprogramming to automate code annotation prior to analysis, and furthermore, remove all cost annotations post analysis. Secondly, we wish to provide support for solving recurrence relations and Big O complexity

analysis. For this, we look to the TiML language for guidance. Finally, we plan to incorporate the cost analysis of monadic code, for example, the parallelised version of quicksort. We suspect this requires reimplementing the `Tick` datatype as a monad transformer.

REFERENCES

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A Relational Logic for Higher-order Programs. In *ICFP*. ACM.

Anonymised. 2019. GitHub Repository for Liquidate Your Assets. (2019).

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. 2007. A Program Logic for Resources. *Theoretical Computer Science* (2007).

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *ESOP*. Springer.

Rod M Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *JACM* (1977).

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *POPL*. ACM.

Ezgi Çiçek. 2018. *Relational Cost Analysis*. Ph.D. Dissertation. Saarland University, Saarbrücken, Germany.

Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional Type Checking for Relational Properties. In *PLDI*. ACM.

Karl Crary and Stephnie Weirich. 2000. Resource Bound Certification. In *POPL*. ACM.

Nils Anders Danielsson. 2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *POPL*.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *ACM SIGPLAN Notices*. ACM.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *CAV*. Springer.

Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *ESOP*. Springer.

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-order Functional Programs. In *ACM SIGPLAN Notices*. ACM.

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems using Sized Types. In *POPL*. ACM.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. *ACM SIGPLAN Notices* (2012).

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *ACM Sigplan Notices*. ACM.

Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-based Cost Analysis for Lazy Functional Languages. *Journal of Automated Reasoning* (2017).

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *ACM Sigplan Notices*. ACM.

Jay McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Findler. 2017. A Coq Library for Internal Verification of Running-times. *Science of Computer Programming* (2017).

A. K. Moran and David Sands. 1999. Improvement in a Lazy Context: An Operational Theory for Call-By-Need. In *POPL*.

Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic Refinements for Relational Cost Analysis. *PACMPL* POPL (2018).

David Sands. 1995. Total Correctness by Local Improvement in Program Transformation. In *POPL*. ACM.

Pedro B Vasconcelos. 2008. *Space Cost Analysis using Sized Types*. Ph.D. Dissertation. University of St. Andrews.

Pedro B Vasconcelos and Kevin Hammond. 2003. Inferring Cost Equations for Recursive, Polymorphic and Higher-order Functional Programs. In *IFL*. Springer.

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph.D. Dissertation. UC San Diego.

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving For All: Equational Reasoning in Liquid Haskell. In *Haskell Symposium*.

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *ESOP*. Springer-Verlag.

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *ICFP*. ACM.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *POPL* (2017).

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *OOPSLA* (2017).