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 meaningful and precise. To illustrate our approach, we analyse the efficiency of a 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.

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 tick algebra, 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 both Agda [Danielsson 2008] and, more recently, Coq [McCarthy et al. 2017], and recent work in [Radiček et al. 2017] developed the theoretical foundations of the value-level approach.

In this article, we take inspiration from [Radiček et al. 2017] 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 both 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, and 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. 2017]. First of all, the system allows correctness properties to be naturally integrated into cost analysis, which helps to ensure that cost analyses are meaningful. And secondly, the wide range of sophisticated invariants that can be expressed and automatically verified by the system can be exploited to analyse resource usage in specific circumstances, which often leads to more precise and/or simpler analyses.

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

2019. XXXX-XXXX/2019/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For example, Liquid Haskell can automatically verify that the standard sorting function in the library *Data.List* returns a sorted 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:

$$\{-@\ sort :: Ord\ a \Rightarrow xs : [a] \rightarrow Tick\ \{zs : OList\ a \mid length\ s == length\ xs\}\ @-\}$$

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 $n \equiv length\ xs$:

$$\{-@\ sortCost :: Ord\ a \Rightarrow xs : [a] \rightarrow \{tcost\ (sort\ xs) <= 4 * length\ xs * log_2\ (length\ xs) + length\ xs\}\ @-\}$$

Moreover, we can also show that the maximum number of comparisons becomes linear in the case when the input list is already in a sorted order:

$$\{-@\ sortCostSorted :: Ord\ a \Rightarrow xs : OList\ a \rightarrow \{tcost\ (sort\ xs) <= length\ xs\}\ @-\}$$

The contributions of the article are as follows:

- We implement a new library, *RTick*, which can be used to formally reason about the resource usage and correctness of pure Haskell programs (section 3).
- We demonstrate the practical applicability of our library by providing a wide range of case studies, including sophisticated relational properties and four sorting algorithms. Three detailed case studies are presented in section 4. The remainder, which are summarised in Table 1 at the end of section 4, are freely available online.
- We prove that our cost analysis is sound using the metatheory of Liquid Haskell (section 5).

The article is aimed at readers who are familiar with the basic ideas of Liquid Haskell [Vazou 2016], but no previous experience with formal reasoning about efficiency is assumed. Section 2 introduces our approach using a number of examples, section 3 explains how the system is implemented, section 4 presents three case studies, section 5 develops the supporting theory, and finally, sections 6 and 7 compare with related work and conclude. Our system is freely available on GitHub [Handley and Vazou 2019], along with all of the examples from 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. Each example also demonstrates how correctness properties can be naturally integrated into cost analysis, and we conclude this section by discussing how to interpret our analyses in practice.

2.1 Intrinsic cost analysis

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

Example 1: Append. We start by analysing the number of recursive calls made by Haskell's list append operator (++). We define the operator ($\underline{++}$) that is similar to append, but lifted into our *Tick* datatype using applicative methods (which are formally defined in section 3):

$$\begin{aligned}
&(\underline{++}) :: [a] \rightarrow [a] \rightarrow Tick\ [a] \\
&[]\ \ \underline{++}\ ys = pure\ ys \\
&(x : xs)\ \underline{++}\ ys = pure\ (x:)\ </ >\ (xs\ \underline{++}\ ys)
\end{aligned}$$

That is, if the first argument list is empty, the second list *ys* is simply 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 increases the total by one.

Now that we have defined the new append operator, we can use Liquid Haskell to encode additional properties 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 further assistance from the user.

Example 2: Merge. Next, we analyse a different resource: the number of comparisons made when merging two ordered lists. As before, we lift the standard *merge* function into the *Tick* datatype. This time, however, we use the (</ >) operator to increase the cost each time a comparison is made:

```
merge :: Ord a ⇒ [a] → [a] → Tick [a]
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, 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 its 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]<{λx 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 the cost analysis on top of Liquid Haskell allows us to naturally combine correctness and efficiency properties.

2.2 Extrinsic cost analysis

In *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 this approach, verification is not automatic, as proof terms are manually provided by the user, but we can express efficiency properties that are not intrinsic to function definitions. For example, we can relate the costs of different functions, and analyse their resource usage on specific subsets of their domains.

Example 3: Merge sort. 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* in terms of the number of comparisons made, we need to use logarithmic knowledge that is not embedded in Liquid Haskell. Instead of augmenting the definition of *msort* with proof terms, 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 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 pow_2 (*length xs*). This highlights the flexibility of the extrinsic approach: even though it is reasonable to use this assumption for cost analysis, it would be unreasonable to impose this restriction for all inputs on which *msort* is applied. The proofs of these theorems can be constructed using the proof combinators of section 3, and are available online [Handley and Vazou 2019].

Example 4: Constant-time comparison. The extrinsic approach lets us 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 show 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 passwords represented as lists of binary digits:

```
{-@ compare :: xs : [Bit] → {ys : [Bit] | length ys == length xs}
    → {t : Tick Bool | tcost t == length xs} @-}
compare :: [Bit] → [Bit] → 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 p against any two user inputs u_1 and u_2:

```
{-@ ple compare @-}
{-@ compare :: p : [Bit] → {u1 : [Bit] | length u1 == length p} → {u2 : [Bit] | length u2 == length p}
    → {tcost (compare p u1) == tcost (compare p u2)} @-}
compare :: [Bit] → [Bit] → [Bit] → Proof
compare []       _ _ = ()
compare (_ : ps) (_ : us1) (_ : us2) = compare ps us1 us2
```

The proof of this theorem follows immediately from the definition of *compare*, in particular, from the fact that Liquid Haskell can automatically verify a precise bound on its execution cost (see *compare*'s type signature: *tcost t == length xs*). Consequently, our proof has a trivial base case and

an inductive case that recursively calls the inductive hypothesis, with all the details being automated using Liquid Haskell's proof by logical evaluation (PLE) tactic [Vazou et al. 2017].

Example 5: Append's monoid laws. As a final example, we outline how extrinsic cost analysis can be used to calculate the difference in execution cost for two related programs. This is also a primary application of relational cost analysis. Consider the familiar monoid laws for the append operator:

$$\begin{array}{rcll} [\,] \mathbin{+\!\!+} ys & == & ys & \textit{left identity} \\ xs \mathbin{+\!\!+} [\,] & == & xs & \textit{right identity} \\ (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs & == & xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs) & \textit{associativity} \end{array}$$

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, ($\underline{+\!\!+}$):

$$\begin{array}{rcl} [\,] \mathbin{\underline{+\!\!+}} ys & <\!\!=\!\!> & \textit{pure}\ ys \\ xs \mathbin{\underline{+\!\!+}} [\,] & >\!\!=\ \textit{length}\ xs\ \Longrightarrow & \textit{pure}\ xs \\ (xs \mathbin{\underline{+\!\!+}} ys) \gg\!= (\mathbin{\underline{+\!\!+}}\ zs) & >\!\!=\ \textit{length}\ xs\ \Longrightarrow & (xs \mathbin{\underline{+\!\!+}}) =\!\!\ll (ys \mathbin{\underline{+\!\!+}}\ zs) \end{array}$$

Recall from example 1 that the ($\underline{+\!\!+}$) 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, $[\,] \mathbin{+\!\!+} ys$ 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 $>\!\!=\ n\ \Longrightarrow$ 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 the next section, where we discuss our library's implementation and prove the second property as an example. Subsequently, in section 4, we use quantified improvement to systematically derive an optimised-by-construction reverse function from a high-level specification.

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 we only ever add (or subtract) costs. Nonetheless, the correctness of a cost analysis depends 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 soundness of the specifications with respect to the 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 upper, lower, and precise bounds on the resource usage of annotated expressions in practice?

Ordinary Haskell datatypes. It should be clear from the examples above that any bound (upper, lower, or precise) established on the execution cost of an annotated function that manipulates ordinary Haskell datatypes is a *worst-case* approximation of actual resource usage. For example, consider the

annotated append function ($\underline{++}$) that measures the number of recursive calls made by (++). Then $tcost\ ([a, b, c]\ \underline{++}\ ys) == 3$ implies that the evaluation of $[a, b, c]\ ++\ ys$ makes three recursive calls to (++). Three recursive calls to (++) corresponds to the function being *fully applied*.

With this in mind, 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 operating on ordinary 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. Hence, from this viewpoint it is clear that such analysis provides worst-case approximations of actual cost.

Memoisation. For efficiency, lazy evaluation relies on computations being able to share intermediate results so that expressions are not (unnecessarily) re-evaluated when they are needed on multiple occasions. By default, however, annotated expressions do not model sharing, that is, memoisation. This is true for annotated functions regardless of their input type. For example, the $\underline{square}$ 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 :: Tick Int → Tick Int
square n = pure (*) <*> n <*> n
```

Therefore, in this instance, the library's default cost analysis better approximates the resource usage of call-by-name evaluation, as opposed to Haskell's call-by-need evaluation.

Upper, lower, and precise bounds. It should now be clear that an upper bound on the cost of an annotated program is a true upper bound on actual resource usage, a lower bound is simply a smaller upper bound, and a precise bound is given by separate upper and lower bounds.

Explicit laziness. Our library can be used to perform more precise analysis on computations that are *explicitly* lazy, by analysing (functions on) datatypes that are defined in terms of *Tick*. We return to this idea in the case study on insertion sort in section 4.

3 IMPLEMENTATION

In this section, we discuss the implementation of our library and a number of assumptions it makes. The library consists of two modules. The first, *RTick*, includes the *Tick* datatype and numerous helper 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}
```

It should be clear that the value of an annotated expression is not necessarily in normal form, for example, the value of $pure\ (1 + 1) :: Tick\ Int$ is $1 + 1$, which is not in normal form.

The *Tick* datatype is a monad, with the following applicative and monad functions:

```
{-@ 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 [Handley and Vazou 2019].

3.2 Modifying resource usage

The *RTick* module provides a wide range of helper functions that can be used to record resource usage within function definitions. Inspired by the work in [Danielsson 2008], we refer to these (and the applicative and monad functions above) as *annotations*. The helper functions used throughout the article are introduced below, and the remainder can be found online [Handley and Vazou 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))} @−}
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 whilst 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 $n > 0$ resources, respectively:

```
{−@ wait :: x : a → {t : Tick a | tval t == x && tcost t == 1} @−}
wait x = Tick 1 x
{−@ waitN :: {n : Int | n > 0} → x : a → {t : Tick a | tval t == x && tcost t == n} @−}
waitN n x = Tick n x
```

From the definitions of (>/ =), *step*, and (>>=) it should be clear that the following equality holds: $(t >/ = f) == step\ 1\ (t >>= f)$. In fact, all of the helper functions provided by the *RTick* module, including (<*>) and (</ >), can be defined using *return*, (>>=), and *step*. We make use of this fact in section 5 when proving the soundness of the library's cost analysis.

It is important to note that *Tick*'s integer argument should *not* be modified by any means other than through the use of the helper functions in the *RTick* module, for example, by case analysis. The reasons for this are discussed at the end of this section as part of the library's assumptions.

Equal	$\{-@\ (==.) :: x : a \to \{y : a \mid y == x\} \to \{v : a \mid v == x\ \&\&\ v == y\}\ @-\}$ $\_ ==. y = y$
Greater than or equal	$\{-@\ (>=.) :: m : a \to \{n : a \mid m >= n\} \to \{o : a \mid m >= o\ \&\&\ o == n\}\ @-\}$ $\_ >=. n = n$
Theorem invocation	$(?) :: a \to Proof \to a$ $x\ ?\ \_ = x$
Proof finalisation	$(***) :: a \to QED \to Proof$ $\_ *** QED = ()$
QED Definition	**data** $QED = QED$

Eq a type class instances are assumed for (==.) and *Num a* for (>=.)

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

3.3 Manual proofs about resource usage

As exemplified in section 2, extrinsic cost analysis requires manually proving that bounds on resource usage hold using *deep reasoning* [Vazou et al. 2018]. In essence, deep reasoning is Liquid Haskell's formalisation of (in)equational reasoning about programs. To this end, 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 *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, ($\underline{++}$), from section 2.

3.3.1 Recap: proof construction. We first give an overview of how to construct (in)equational proofs using Liquid Haskell. To exemplify both styles of proof (equational and inequational), we reason about the results and resource usage of append separately. Readers may refer to [Vazou et al. 2018] for a 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 $xs\ \underline{++}\ [\,]$ and *pure xs* are equal:

$$\{-@\ rightIdVal :: xs : [a] \to \{p : Proof \mid tval\ (xs\ \underline{++}\ [\,]) == tval\ (pure\ xs)\}\ @-\}$$

Here, the binder $p : Proof$ is irrelevant and so we can remove it:

$$\{-@\ rightIdVal :: xs : [a] \to \{tval\ (xs\ \underline{++}\ [\,]) == tval\ (pure\ xs)\}\ @-\}$$

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

```
rightIdVal :: [a] → Proof
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*) function 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 *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 :: [a] → Proof
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:

$$[\,] \mathbin{\underline{++}} ys \;==\; pure\ ys$$

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:

$$[\,] \mathbin{\underline{++}} ys \;<=>\; pure\ ys$$

Relations

$$
\begin{array}{rlll}
\textit{Value equivalence} & t_1 \;\; =!= \;\; t_2 & = & \textit{tval}\ t_1 == \textit{tval}\ t_2 \\
\textit{Cost equivalence} & t_1 \;\; <=> \;\; t_2 & = & t_1 =!= t_2 \;\&\&\; \textit{tcost}\ t_1 == \textit{tcost}\ t_2 \\
\textit{Cost improvement} & t_1 \;\; >\sim> \;\; t_2 & = & t_1 =!= t_2 \;\&\&\; \textit{tcost}\ t_1 >= \textit{tcost}\ t_2 \\
\textit{Cost diminishment} & t_1 \;\; <\sim< \;\; t_2 & = & t_1 =!= t_2 \;\&\&\; \textit{tcost}\ t_1 <= \textit{tcost}\ t_2 \\
\textit{Quantified improvement} & t_1 >== n ==> t_2 & = & t_1 =!= t_2 \;\&\&\; \textit{tcost}\ t_1 == n + \textit{tcost}\ t_2 \\
\textit{Quantified diminishment} & t_1 <== n ==< t_2 & = & t_1 =!= t_2 \;\&\&\; n + \textit{tcost}\ t_1 == \textit{tcost}\ t_2
\end{array}
$$

Eq a type class instances are assumed for (=!=)

Combinators

Cost equivalence

$$\{-@\ (<=>.) :: t_1 : \textit{Tick}\ a \to \{t_2 : \textit{Tick}\ a \mid t_1 <=> t_2\} \to \{t : \textit{Tick}\ a \mid t_1 <=> t \;\&\&\; t_2 <=> t\}\ @-\}$$

Cost improvement

$$\{-@\ (>\sim>.) :: t_1 : \textit{Tick}\ a \to \{t_2 : \textit{Tick}\ a \mid t_1 >\sim> t_2\} \to \{t : \textit{Tick}\ a \mid t_1 >\sim> t \;\&\&\; t_2 <=> t\}\ @-\}$$

Cost diminishment

$$\{-@\ (<\sim<.) :: t_1 : \textit{Tick}\ a \to \{t_2 : \textit{Tick}\ a \mid t_1 <\sim< t_2\} \to \{t : \textit{Tick}\ a \mid t_1 <\sim< t \;\&\&\; t_2 <=> t\}\ @-\}$$

Quantified improvement

$$\{-@\ (.>==) :: t_1 : \textit{Tick}\ a \to n : \textit{Nat} \to \{t_2 : \textit{Tick}\ a \mid t_1 >== n ==> t_2\} \to \{t : \textit{Tick}\ a \mid t_1 >== n ==> t \;\&\&\; t_2 <=> t\}\ @-\}$$

Quantified diminishment

$$\{-@\ (.<==) :: t_1 : \textit{Tick}\ a \to n : \textit{Nat} \to \{t_2 : \textit{Tick}\ a \mid t_1 <== n ==< t_2\} \to \{t : \textit{Tick}\ a \mid t_1 <== n ==< t \;\&\&\; t_2 <=> t\}\ @-\}$$

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

Separators

$$
\begin{array}{rll}
\textit{Quantified improvement} & (==>.) :: (a \to b) \to a \to b & f ==>.\ x = f\ x \\
\textit{Quantified diminishment} & (==<.) :: (a \to b) \to a \to b & f ==<.\ x = f\ x
\end{array}
$$

Figure 2. Cost relations, combinators, and separators.

The above property is a "resource-aware" version of append's left identity law, which formalises that both expressions, $[\,] \mathbin{+\!\!+} ys$ and ys, evaluate to the same result and require the same number of recursive calls to append during evaluation (as shown in example 5 of section 2).

Cost improvement. Previously, we proved that append's right identity law is a value equivalence: $\textit{tval}\ (xs \mathbin{\underline{+\!\!+}} [\,]) == \textit{tval}\ (\textit{pure}\ xs)$, and a cost inequivalence: $\textit{tcost}\ (xs \mathbin{\underline{+\!\!+}} [\,]) >= \textit{tcost}\ (\textit{pure}\ xs)$. 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 "$xs \mathbin{\underline{+\!\!+}} [\,]$ is improved by $\textit{pure}\ xs$".

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 :: [a] -> Proof
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, 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 extra information allows us to relate the expressions more precisely using the *quantified improvement* relation, also defined in figure 2. Quantified improvement intuitively 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 correctness (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 :: [a] → 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 need to show a cost saving of *length* (*x* : *xs*). We start by unfolding the definition of (++), and then replace *xs* ++ [ ] with *pure xs* by referring 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 application operator ($) that 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*.

Although improvement and quantified improvement are notionally equivalent, the fact that the latter relation makes resource savings explicit in both theorems and proofs is a significant practical advantage. This is exemplified in section 4.3, where the derivation of an optimised-by-construction list-reverse function relies on essential use of quantified improvement.

Cost diminishment and quantified diminishment. The combinators introduced up until now can only be used to prove that one expression e_1 :: *Tick a* is improved by another e_2 :: *Tick a* by starting at

e_1 and deriving e_2. This is because (quantified) cost improvement enforces a positive cost difference in the left-to-right direction. However, in some cases it may be easier to derive e_1 from e_2. To support this, we use the notion of (*quantified*) *cost diminishment*, also presented in figure 2, which is dual to (quantified) cost improvement. For example, it is straightforward to prove that *pure xs* is diminished by $xs \mathbin{\underline{++}} [\,]$, by a cost of *length xs*. To achieve this, simply reverse the calculation steps of *rightIdQImp*: replacing instances of quantified improvement with quantified diminishment.

It should be clear that $e_1 \mathbin{>\sim>} e_2$ if and only if $e_2 \mathbin{<\sim<} e_1$, and likewise that $e_1 \mathbin{>==} n \mathbin{==>} e_2$ if and only if $e_2 \mathbin{<==} n \mathbin{==<} e_1$. Similar relationships exist between other cost relations, for example, if $e_1 \mathbin{>\sim>} e_2$ and $e_2 \mathbin{>\sim>} e_1$, then $e_1 \mathbin{<=>} e_2$. All such relationships have been formalised using Liquid Haskell and the relevant proofs are available online [Handley and Vazou 2019].

3.4 Library assumptions

To help users ensure their cost analysis is sound, the library makes two key assumptions: (1) any expression whose resource usage is being analysed is not defined using *tval* or *tcost*, and furthermore, it does not perform case analysis on the *Tick* data constructor; (2) Liquid Haskell's totality and termination checkers are active at all times. These assumptions are discussed further below.

3.4.1 Projections and case analysis. Expressions whose resource usage are being analysed must not be defined using the *tval* projection function. This is because projecting out the value of an annotated expression allows its resource usage to be arbitrarily modified. For example, *tval* can be used to (indirectly) show that two lists can be appended using ($\underline{++}$) without incurring any cost:

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

However, we know from the type specification of ($\underline{++}$) that *tcost* $(xs \mathbin{\underline{++}} ys)$ == *length xs*.

Although the above proof is valid: *pure* $(tval\ (xs \mathbin{\underline{++}} ys))$ evaluates to a result $t :: Tick\ [a]$ such that $tval\ t == tval\ (xs \mathbin{\underline{++}} ys)$ and $tcost\ t == 0$, the use of *tval* in this instance breaks the encapsulation of the *Tick* datatype's accumulated cost in order to discard it.

Performing case analysis on *Tick*'s data constructor allows its value to be projected out in much the same way. Moreover, *tcost* can simply overwrite any recorded cost.

Consequently, it is clear that these techniques are not in the spirit of genuine cost analysis, and hence are not permitted by the library. Instead, users should always record resource usage *implicitly* by using the helper functions defined in 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) *tval* and *tcost* are required to express theorems about correctness and resource usage in refinement types. In addition, steps of inequational reasoning in extrinsic proofs often refer to the *Tick* data constructor, for example, when unfolding the definitions of annotations such as (<*>) and (>>=).

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 case studies: cost analysis of monolithic (section 4.1) and non-strict (section 4.2) implementations of insertion sort, and a derivation of an optimised-by-construction list reverse function (section 4.3). It then concludes with a summary of all of the examples we have studied while developing the library, the majority of which have been adapted from the existing literature (section 4.4).

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:

```
insert :: Ord a ⇒ a → [a] → Tick [a]
insert x [ ] = pure [x]
insert x (y : ys)
  | x <= y    = wait (x : y : ys)
  | otherwise = pure (y:) </> insert x ys

isort :: Ord a ⇒ [a] → Tick [a]
isort [ ]       = return [ ]
isort (x : xs) = insert x =≪ isort xs
```

According to 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.

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 any element in the tail:

```
{−@ type OList a = [a]<{λx y → x <= y}> @−}
```

The *OList* type constructor is used in the specification 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 * length xs}@−}
```

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 on with our intrinsic analysis. The key to this approach is utilising the following function, which is a variant of (=≪):

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

It is clear that f =≪{n} t is operationally equal to f =≪ t. However, the refinement type of this "bounded" version of (=≪) restricts its domain to functions $f :: a \rightarrow Tick\ b$ with execution costs no greater than n. Hence, the resource usage of 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 type-level computations. Thus, *isort*'s specification is automatically verified for the following definition in which *length xs* is an upper bound on the cost of each insertion:

```
isort :: Ord a ⇒ [a] → Tick [a]
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 must consider three cases: when the input list is empty; when the input list is a singleton, which invokes the base case of *insert*; and when the input 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:

```
{−@ ple isortCostSorted @−}
isortCostSorted :: Ord a ⇒ [a] → Proof
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, which we know it is. (*isortSortedVal*'s proof is available online [Handley and Vazou 2019].)

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

As the input list ($x : y : ys$) 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 the input *xs* to *isort* using an extrinsic theorem, without needing to modify the function's definition. Furthermore, the above 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 uses *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:

```
{−@ minimum :: Ord a ⇒ xs : {[a] | length xs > 0} → {t : Tick a | tcost t <= length xs * length xs} @−}
minimum :: Ord a ⇒ [a] → Tick a
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* applying *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 the 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 that operate on standard Haskell datatypes as monolithic. To encode non-strict evaluation, we include *Tick* in the definitions of datatypes in order to suspend computations. Datatype 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 better reflects 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, such as "sortedness", 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>)) } @−}
data LList a = Nil | Cons { lhead :: a, ltail :: Tick (LList a) }
```

Sorted lazy lists can be 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<{λx 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* (as an annotated computation) 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 :: Ord a ⇒ a → LList a → Tick (LList a)
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 ordinary 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 :: Ord a ⇒ [a] → Tick (LList a)
lisort []     = return Nil
lisort (x : xs) = linsert x =≪{1} lisort xs
```

Given an ordinary 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 $(=\ll\{\cdot\})$ (accurately) approximates the execution cost of *linsert* at each recursive call.

4.2.3 Non-strict minimum. A non-strict minimum function, *lminimum*, can now return the first element in a non-empty list *xs* that has been *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 : [a] | length xs > 0} → {t : Tick a | tcost t <= length xs} @−}
lminimum :: Ord a ⇒ [a] → Tick a
lminimum xs = pure lhead <*> lisort xs
```

4.2.4 Explicit laziness. Lazy lists (of type *LList a*) are defined so 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 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 "free".

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

```
{−@ pay :: m : Nat → {t1 : Tick a | tcost t1 >= m}
  → {t2 : Tick (Tick a) | tcost (tval t2) == tcost t1 − m} @−}
pay :: Int → Tick a → Tick (Tick a)
pay m (Tick n x) = Tick m (Tick (n − m) x)
```

Evaluating *pay m t >>= f* allows *f* to use *t*'s value numerous times while only paying *m* cost for it once. Therefore, if *m == tcost t* 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 Optimised-by-construction reverse

In *Theorem Proving for All*, Vazou et al. [2018] systematically derive a linear-time list reverse function from a high-level specification concerning the well-known, naive list reverse function, which has a quadratic runtime performance. The fundamental goal of this calculation is to prove that the new implementation is *correct-by-construction*. In order words, that the denotational meaning of the initial specification is preserved during the calculation.

This case study goes one step further and proves that the derived implementation preserves the meaning and *improves* the initial function. To achieve this, we use the notion of quantified improvement to simultaneously reason about correctness and efficiency. The cost combinators from section 3.3 capture the resources saved at each calculation step and so the total resource saving and final resource usage are both calculated on the fly as part of the derivation process.

Readers should note that the definitions in the subsequent sections *Executable code*, *Specification*, and *Proof* exemplify the corresponding figures given in table 1 of section 4.4.

4.3.1 Executable code. Consider the naive list reverse function, which has been annotated to count the total number of recursive calls, that is, by itself and $(\underline{++})$ (from section 2.1):

```
reverse :: [a] → Tick [a]
reverse [ ]       = return [ ]
reverse (x : xs) = reverse xs >/= (++ [x])
```

The *reverse* function appends each element of the input list to the end of its reversed tail. As the cost of $(\underline{++})$ is linear in the length of its first argument, the total number of recursive calls made by *reverse* is quadratic. The precise cost of *reverse* is captured by the following extrinsic theorem:

```
{-@ reverseCost :: xs : [a]
    → { tcost (reverse xs) == ((length xs * length xs) / 2) + ((length xs + 1) / 2) } @-}
```

4.3.2 Specification. To *improve* reverse, we seek to fuse together the processes of appending and reversing. This can be achieved by defining a new function that reverses its first argument and appends its second. We express this requirement as a Liquid Haskell specification:

```
{-@ reverseApp :: xs : [a] → ys : [a] → { t : Tick [a] | reverse xs >>= (++ ys) >~> t } @-}
```

Since we plan to use *reverseApp* to improve *reverse*, any implementation we propose for this function *must* also record the total number of recursive calls. Furthermore, note that the above specification is a cost improvement. This means that *reverseApp* must give the same results as *reverse xs* $\gg\!=$ $(\underline{++}\ ys)$ by using no more resources. We start with a trivial definition for *reversApp* that satisfies the specification but makes no cost savings:

```
reverseApp :: [a] → [a] → Tick [a]
reverseApp [ ]       ys = reverse [ ]       >>= (++ ys)
reverseApp (x : xs) ys = reverse (x : xs) >>= (++ ys)
```

4.3.3 Initial resource usage. While deriving a new implementation for *reverseApp*, we will calculate the total resource saving, call it *s*, on the fly using quantified improvement. In order to calculate the resource usage of the final result, however, we must first calculate the initial resource usage of *reverse xs* $\gg\!=$ $(\underline{++}\ ys)$, call it *u*. The final resource usage is then simply $u - s$.

The resource usage of *reverse xs* $\gg\!=$ $(\underline{++}\ ys)$ is as follows:

```
{-@ reverseApp0Cost :: xs : [a] → ys : [a]
    → { tcost (reverse xs >>= (++ ys)) == ((length xs * length xs) / 2) + ((3 * length xs + 1) / 2) } @-}
```

The proof of this theorem follows immediately from the resource usage of *reverse* and (++), and is available online along with the proof of *reverseCost* [Handley and Vazou 2019].

4.3.4 Proof by inequational rewriting. The next step of the improvement process is to rewrite the right-hand sides of the trivial definition to more efficient forms. To do so, we use proof combinators introduced in section 3.3 to ensure that each rewrite preserves the result (denotation) of *reverseApp* while preserving or improving its execution cost.

The rewriting follows the general insight that a proof of improvement is very similar to its respective proof of correctness. As such, we initially focus on correctness by unfolding and folding definitions. Whenever a resource saving can be made, we use quantified improvement to precisely capture it. Finally, at the end of the calculation, we turn our attention to resource usage.

With this in mind, we begin by rewriting the base case of *reverseApp*: first by unfolding the definitions of *reverse*, (≫=), and (++); then by inlining the **let** binding and folding *return*:

```
reverseApp :: [a] → [a] → Tick [a]
reverseApp [ ] ys
  =      reverse [ ] ≫= (++ ys)
  <=>. Tick 0 [ ] ≫= (++ ys)
  <=>. (let Tick n y = [ ] ++ ys in Tick n y)
  <=>. (let Tick n y = Tick 0 ys in Tick n y)
  <=>. return ys
```

For the recursive case, we also begin by unfolding definitions as much as possible:

```
reverseApp (x : xs) ys
  =      reverse (x : xs) ≫= (++ ys)
  <=>. (reverse xs >/ = (++ [x])) ≫= (++ ys)
  <=>. (let Tick o w = reverse xs in Tick o w >/ = (++ [x])) ≫= (++ ys)
  <=>. (let Tick o w = reverse xs in let Tick p v = w ++ [x] in Tick (1 + o + p) v ≫= (++ ys))
  <=>. (let Tick o w = reverse xs in let Tick p v = w ++ [x] in let Tick q u = v ++ ys in Tick (1 + o + p + q) u)
```

Notice that in order to unfold the definition of (>/ =), we must introduce a **let** binding because *reverse xs* is not in "*Tick* normal form". This is a strategy we commonly employ.

At this point, there is an addition of a constant cost (of one) on the returned *Tick*. Since we are in the recursive case of the calculation, we do not save on this resource usage under the assumption that we will recurse on *reverseApp*. Consequently, we "bank" this recursive call with the *step* function.

```
  <=>. step 1 (let Tick o w = reverse xs in let Tick p v = w ++ [x] in let Tick q u = v ++ ys in Tick (o + p + q) u)
  <=>. step 1 (let Tick o w = reverse xs in let Tick p v = w ++ [x] ≫= (++ ys) in Tick (o + p) v)
```

We then folded the definition of (≫=) to expose the expression w ++ $[x]$ ≫= (++ ys), which is two appends associated to the left. In order to continue with the calculation, these appends must be re-associated to the right. From example 5 of section 2, we know that this is an improvement:

```
{-@ appendAssocQImp :: xs : [a] → ys : [a] → zs : [a]
  → {xs ++ ys ≫= (++ zs) >== length xs ==> (xs ++) =≪ ys ++ zs}@-}
```

Appealing to *appendAssocQImp* with xs := *tval* (*reverse xs*), ys := $[x]$, and zs := ys saves *length xs* resources. (Note that the specification of *reverse* proves that *length* (*tval* (*reverse xs*)) == *length xs*). We use quantified improvement to capture the cost saving: (.>== *length xs* ==>.).

```
    ? appendAssocQImp (tval (reverse xs)) [x] ys
  .>== length xs ==>. step 1 (let Tick o w = reverse xs in let Tick p v = [x] ++ ys ≫= (w ++) in Tick (o + p) v))
```

We then continue by unfolding the definitions of (++), *pure*, and (</ >):

```
  <=>. step 1 (let Tick o w = reverse xs in let Tick p v = pure (x:) </ > ([ ] ++ ys) ≫= (w ++) in Tick (o + p) v)
```

$$\ll\!\!=\!\!\gg. \; step\ 1\ (\textbf{let}\ Tick\ o\ w = reverse\ xs\ \textbf{in let}\ Tick\ p\ v = Tick\ 1\ (x : ys) \gg\!= (w\ \underline{++})\ \textbf{in}\ Tick\ (o + p)\ v)$$

Unfolding the definition of $(</>)$ has presented us with another opportunity for resource saving. We take it, and then continue by unfolding $(\gg\!=)$ and inlining the resulting **let**:

$$.\!>\!==\ 1\ ==\!\!\gg. \; step\ 1\ (\textbf{let}\ Tick\ o\ w = reverse\ xs\ \textbf{in let}\ Tick\ p\ v = Tick\ 0\ (x : ys) \gg\!= (w\ \underline{++})\ \textbf{in}\ Tick\ (o + p)\ v))$$
$$\ll\!\!=\!\!\gg. \; step\ 1\ (\textbf{let}\ Tick\ o\ w = reverse\ xs\ \textbf{in let}\ Tick\ p\ v = Tick\ 0\ (x : ys)\ \textbf{in let}\ Tick\ q\ u = w\ \underline{++}\ v\ \textbf{in}\ Tick\ (o + p + q)\ u)$$
$$\ll\!\!=\!\!\gg. \; step\ 1\ (\textbf{let}\ Tick\ o\ w = reverse\ xs\ \textbf{in let}\ Tick\ q\ u = (\underline{++}\ (x : ys))\ w\ \textbf{in}\ Tick\ (o + q)\ u)$$
$$\ll\!\!=\!\!\gg. \; step\ 1\ (reverse\ xs \gg\!= (\underline{++}\ (x : ys)))$$

The final step of the calculation is to rewrite the definition to be self-contained. Clearly replacing $reverse\ xs \gg\!= (\underline{++}\ (x : ys))$ with $reverseApp\ xs\ (x : ys)$ saves resources:

$$?\ \ reverseAppCost0\ xs\ (x : ys)$$
$$.\!>\!==\ ((length\ xs * length\ xs)\ `div`\ 2) + ((length\ xs + 1)\ `div`\ 2) ==\!\!\gg. \; step\ 1\ (reverseApp\ xs\ (x : ys))$$

This resource saving is calculated by subtracting $length\ xs$ from the resource usage of $reverse\ xs \gg\!= (\underline{++}\ (x : ys))$, which is given by $reverseApp0Cost$ (introduced previously). Note that we subtract $length\ xs$ because evaluating $reverseApp\ xs\ (x : ys)$ requires $length\ xs$ recursive calls, whereby each recursive call is recorded by the $step\ 1$ we "banked" earlier.

Having reached the end of the proof, we can now turn our attention to calculating final resource usage. Below is a table listing: the initial resource usage, u, calculated by $reverseApp0Cost$; the total resource saving, s, calculated by summing up the individual savings made explicit in steps of quantified improvement; and the final resource usage, which is simply $u - s$.

Initial usage (u)	$\frac{\lvert(x:xs)\rvert^2}{2} + \frac{3 * \lvert(x:xs)\rvert + 1}{2}$
Total saving (s)	$\lvert xs\rvert + 1 + \frac{\lvert xs\rvert^2}{2} + \frac{\lvert xs\rvert+1}{2}$
Final usage $(u - s)$	$\lvert(x : xs)\rvert$

By way of a simple subtraction, we have calculated that the final resource usage of $reverseApp$ is linear in the length of its first argument (as we might expect). By adding this resource bound to $reverseApp$'s specification, Liquid Haskell verifies that this property holds and the derivation of an *optimised-by-construction* implementation for $reverseApp$ is complete:

$$\{-@\ reverseApp :: xs : [a] \to ys : [a]$$
$$\to \{t : (Tick\ [a]) \mid reverse\ xs \gg\!= (\underline{++}\ ys) >\!\sim\!> t\ \&\&\ tcost\ t == length\ xs\}\ @-\}$$

In the above derivation process, quantified improvement makes the quantity and locality of each cost saving explicit. In particular, it shows a linear cost saving per recursive call. This corresponds precisely to evaluating $(\underline{++})$ in order to fuse together the processes of reversing and appending, which was our primary goal. Furthermore, the proof combinators used simply to return their last arguments. As such, at compile time (with optimisation turned on) GHC will remove all of the intermediate calculation steps, leading to the following concise definition of $reverseApp$:

$$reverseApp :: [a] \to [a] \to Tick\ [a]$$
$$reverseApp\ [\,]\ ys \quad\ = return\ ys$$
$$reverseApp\ (x : xs)\ ys = step\ 1\ (reverseApp\ xs\ (x : ys))$$

4.3.5 *Optimising reverse.*

Finally, we can use $reverseApp$ to improve the definition of $reverse$:

$$\{-@\ fastReverse :: xs : [a] \to \{t : Tick\ [a] \mid reverse\ xs >\!\sim\!> t\ \&\&\ tcost\ t == length\ xs\}\ @-\}$$
$$fastReverse :: [a] \to Tick\ [a]$$
$$fastReverse\ xs$$
$$= \quad reverse\ xs$$
$$\ll\!\!=\!\!\gg. \; (\textbf{let}\ Tick\ o\ w = reverse\ xs\ \textbf{in let}\ Tick\ p\ v = pure\ w\ \textbf{in}\ Tick\ (o + p)\ v)$$

```
    ? rightIdQImp (tval (reverse xs))
.<== length xs ==<. (let Tick o w = reverse xs in let Tick p v = w ++ [ ] in Tick (o + p) v)
<=>. reverse xs >>= (++ [ ])
    ? reverseApp0Cost xs [ ]
.>== ((length xs * length xs) `div` 2) + ((length xs + 1) `div` 2) ==>. reverseApp xs [ ]
```

Notice that by applying append's right identity (*rightIdQImp*) in the right-to-left direction, the resource usage of the resulting expression is greater than or equal to that of the initial expression. This is captured using quantified diminishment. A simple subtraction (as above) reveals that the final resource usage of *fastReverse xs* is *length xs*, which is verified by Liquid Haskell.

Similarly to *reverseApp*, the intermediate calculation steps of *fastReverse* will be removed at compile time when optimisation is turned on, leading to the following concise definition:

```
fastReverse :: [a] → Tick [a]
fastReverse xs = reverseApp xs [ ]
```

The following functions, *revApp* and *fastRev*, were the results of the derivation in *Theorem Proving for All* [Vazou et al. 2018]. Each can be calculated from *reverseApp* and *fastReverse*, respectively, by simply removing the cost annotations [Handley and Vazou 2019].

```
revApp :: [a] → [a] → [a]
revApp [ ]       ys = ys
revApp (x : xs) ys = revApp xs (x : ys)
```

```
fastRev :: [a] → [a]
fastRev xs = revApp xs [ ]
```

We have, therefore, arrived at the same end point as [Vazou et al. 2018], however, this time we have also formalised a reduction in the execution cost of the initial specification.

Summary. The derivation in this case study mirrors the calculation in section 4.1 of [Vazou et al. 2018] *step-for-step* (we encourage readers to check). Concretely, we have replaced, where necessary, equational reasoning with inequational reasoning, whereby the resource saving of each rewrite has been made explicit using the notion of quantified improvement. In short, we have taken a calculation aimed at deriving a correct-by-construction list reverse function and transformed it into a calculation aimed at deriving an optimised-by-construction list reverse function.

Furthermore, the reasoning initially focused exclusively on correctness (unfolding and folding definitions), while total resource savings were calculated on the fly. Only when a final result had been derived did our attention turn back to resource usage. Then, by a simple subtraction, we were able to calculate a final resource usage, which Liquid Haskell could verify as a consequence of our prior reasoning. Thus, 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 [Handley and Vazou 2019].

Overview. Table 1 provides a quantitative summary of each example and is split into four categories. The first three categories include examples from the existing literature, while the last includes the complexity analysis of different sorting algorithms. An overview of the four categories is as follows:

– ***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 is linear. We also encoded lazy queues and proved that viewing a lazy queue and appending at the end are constant. Our examples are highly comparable to [Danielsson 2008], however, we incorporate additional (automated) correctness properties, such as sortedness.

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

	Property	Lines of code		
		Exec.	Spec.	Proof
Laziness [Danielsson 2008]				
Insertion sort	$\textsc{Cost}(lisort\ xs) \leqslant \lvert xs \rvert$	12	8	0
Implicit queues	$\textsc{Cost}(lsnoc\ q\ x) = 5$, $\textsc{Cost}(view\ q) = 1$	50	14	0
Relational [Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2017]				
2D count	$\textsc{Cost}(2DCount\ find_1) \leqslant \textsc{Cost}(2DCount\ find_2)$	16	3	24
Binary counters	$\textsc{Cost}(decr\ k\ tt) = \textsc{Cost}(incr\ k\ ff)$	26	21	21
Boolean expressions	$\textsc{NoShort}(e) \Rightarrow \textsc{Cost}(eval_1\ e) = \textsc{Cost}(eval_2\ e)$	28	2	13
Constant-time comparison	$\textsc{Cost}(compare\ p\ u\_1) = \textsc{Cost}(compare\ p\ u\_2)$	3	8	3
Insertion sort	$\textsc{Sorted}(xs) \Rightarrow \textsc{Cost}(isort\ xs) \leqslant \textsc{Cost}(isort\ ys)$	16	17	44
Memory allocation of length	$\textsc{Cost}(length_2\ xs) - \textsc{Cost}(length_1\ xs) = length\ xs$	10	4	6
Relational insertion sort	$\textsc{Cost}(isort\ xs) - \textsc{Cost}(isort\ ys) = unsortedDiff\ xs\ ys$	16	11	69
Relational merge sort	$\textsc{Cost}(msort\ xs) - \textsc{Cost}(msort\ ys) \leqslant \lvert xs \rvert\ (1 + \log_2(diff\ xs\ ys))$	23	25	59
Square and multiply	$\textsc{Cost}(sam\ t\ x\ l_1) - \textsc{Cost}(sam\ t\ x\ l_2) \leqslant t * diff\ l_1\ l_2$	3	8	3
Datatypes [Vazou et al. 2018]				
Append's monoid laws	*see example 5 of section 2*	12	10	74
Appending	$\textsc{Cost}(xs\ \underline{++}\ ys) = \lvert xs \rvert$	8	3	0
Flattening	$\textsc{Perfect}(t) \Rightarrow \textsc{Cost}(flattenOpt\ t) = 2^{\lvert t \rvert} - 1$	5	18	45
Optimised-by-construction reverse	$reverse\ xs >\!\!\sim\!\!> fastReverse\ xs$	18	37	140
Reversing (naive)	$\textsc{Cost}(reverse\ xs) = \frac{\lvert xs \rvert^2}{2} + \frac{\lvert xs \rvert + 1}{2}$	9	7	22
Reversing (optimised)	$\textsc{Cost}(fastReverse\ xs) = \lvert xs \rvert$	5	8	0
Sorting				
Data.List.sort	$\textsc{Cost}(ssort\ xs) \leqslant 4\ \lvert xs \rvert \log_2 \lvert xs \rvert + \lvert xs \rvert$	39	49	107
Insertion sort	$\textsc{Cost}(isort\ xs) \leqslant \lvert xs \rvert^2$	8	10	33
Merge sort	$\frac{\lvert xs \rvert}{2} \log_2 \lvert xs \rvert \leqslant \textsc{Cost}(msort\ xs) \leqslant \lvert xs \rvert \log_2 \frac{\lvert xs \rvert}{2} + \lvert xs \rvert$	22	69	139
Quicksort	$\textsc{Cost}(qsort\ xs) \leqslant \frac{1}{2}(\lvert xs \rvert + 1)(\lvert xs \rvert + 2)$	15	8	27
Total		344	340	829

– ***Relational*** includes *all* the cost analysis examples from [Aguirre et al. 2017; Çiçek et al. 2017; Radiček et al. 2017]. These examples compare the resource usage of the same function on different inputs (for example, constant-time comparison from section 2.2) or different functions on the same input (for example, the memory allocation case study compares the memory required by the standard and tail recursive implementations of the *length* function).
From this set of examples, we conclude that even though our system does not have the synchronous and asynchronous rules of relational logic, extrinsic reasoning allows us to encode sophisticated relational properties whose proofs are simplified by Liquid Haskell's automation.
– ***Datatypes*** includes properties concerning lists and trees whose Liquid Haskell correctness proofs initially appeared in [Vazou et al. 2018]. Similarly to the calculation in section 4.3, we have extended the corresponding correctness proofs to further account for resource usage.
– ***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.

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

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

Reduction

$$
\begin{array}{rcll}
\mathbb{C}[e] & \hookrightarrow & \mathbb{C}[e'] & \text{if } e \hookrightarrow e' \\
c\ v & \hookrightarrow & \delta(c, v) & \\
(\lambda x.e_1)\ e_2 & \hookrightarrow & e_2[e_1/x] & \\
\textbf{let}\ x = e\_x\ \textbf{in}\ e & \hookrightarrow & e[e_x/x] & \\
\textbf{case}\ x = D_j\,\overline{e}\ \textbf{of}\ \{ D_i\,\overline{y_i} \rightarrow e_i \} & \hookrightarrow & e_j[D_j\,\overline{e}/x][\overline{e}/\overline{y_j}] &
\end{array}
$$

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

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 [Handley and Vazou 2019].

Recall that the case study in section 4.3 illustrates each figure reported in the *lines of code* column:

- ***Exec.*** the total lines of executable code relating to the property;
- ***Spec.*** the total lines of code used to express the property as a Liquid Haskell specification;
- ***Proof*** the total lines of code used to prove the property.

Synopsis. In total, we wrote 344 lines of executable code; 340 lines of Liquid Haskell specifications and 829 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].

5 THEORY

In this section, we prove the soundness 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, which is 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 we get if we erase all refinements from τ and $e : \lfloor \tau \rfloor$ be the standard typing relation for the typed λ-calculus. Then, we define the denotation of types as follows:

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

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:

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

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

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

Soundness of λ^U. The soundness of λ^U proves 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, $c \in [\![\mathit{Ty}(c)]\!]$, then $\emptyset \vdash e :: \tau$ implies $e \in [\![\tau]\!]$.*

5.2 Soundness of cost analysis

Since 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 soundness 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 Tick_τ with a single data constructor: $\mathit{Tick}_\tau :: \mathit{Int} \to \tau \to \mathit{Tick}_\tau$. Tick_τ data constructors should *not* be used directly. Instead, each Tick_τ 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_τ, $\mathit{bind}_{\tau_1,\tau_2}$, step_τ, tcost_τ, tval_τ for each type τ. We use λ^U to define the types and (type-specific) bodies of each constant just as in section 3.2. Since Liquid Haskell type-checks the previous definitions, it must be true that $c \in [\![\mathrm{Ty}(c)]\!]$ for each constant return_τ, $\mathit{bind}_{\tau_1,\tau_2}$, and so on. Therefore, these constants can be used *appropriately* in λ^U while preserving soundness.

Safe expressions. Recall from section 3.4 that the library imposes the following restrictions on annotated expressions in order 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 tval_τ or tcost_τ constants;
- e does not perform case analysis on any Tick_τ data constructors.

Execution cost. Consider a safe, terminating function f that returns a value of type Tick_τ for some type τ, that is, $f :: x : \tau_x \to \mathit{Tick}_\tau$. 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 $f\ e_x \hookrightarrow^\star \mathit{Tick}_\tau\ i\ v$. As f does not directly modify any Tick_τ datatypes, all resource consumptions or productions via applications of step_τ in f's definition accumulate in the cost i of the final value, $\mathit{Tick}_\tau\ i\ v$.

Static cost analysis. Finally, we use the soundness of λ^U to prove that the library's intrinsic and extrinsic approaches to analysing resource usage are both sound:

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

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

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 sound 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. It should be clear, however, that all such functions can be defined using *return*, (≫=), and *step*: a proof of this fact can be found on the library's GitHub page [Handley and Vazou 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 *Thunk* library [2008], which provides a lightweight framework for cost analysis in Agda. It 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, but captures abstract resource usage at the value-level. Much of *Thunk*'s analysis requires simple equality proofs because Agda does not automatically prove arithmetic equalities. Our use of Liquid Types avoids such problems as the underlying arithmetic necessary for our cost analysis can be delegated to an SMT solver. Another notable distinction is that our library supports both 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. 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 be used to analyse runtimes. None of these approaches, however, can express correctness properties, which (as we have seen) allow for more precise and meaningful 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 allows for more complex case studies (such as Okasaki's Braun Trees) to be examined. Another distinction is that cost annotations can be automatically inserted then erased when code is extracted to OCaml. Implementing this is part of our future work. Similarly to [Danielsson 2008], relational cost analysis is not supported. TiML [Wang et al. 2017] indexes the arrow 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 (but in a complete way) derive complexity theorems. TiML also supports sophisticated invariants, however, they are only exploited for the purposes of cost analysis. Our library uses invariants (such as sortedness) to simultaneously reason about program correctness and resource usage.

Automatic Amortized Resource Analysis (AARA) [Hofmann and Jost 2003], aims to automatically derive amortised bounds on resource usage. 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 may often be less precise than ours. In particular, our library's extrinsic resource analysis can (notionally) compute resource bounds of any type, for example, polynomial, logarithmic, and polylogarithmic. In comparisons, AARA is (at best) restricted to polynomial bounds and invariants are not supported.

Improvement theory [Moran and Sands 1999] inspired our notions of *improvement* and *quantified improvement*. However, previously Sands introduced *improvements* [Sands 1995] 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. In [Moran and Sands 1999], resource usage is measured by counting transitions in an abstract machine. Interestingly, the linear-time reverse function, *fastReverse* (from section 4.3), is not an improvement over the naive function, *reverse*, in this setting, because the base case of *fastReverse* requires more abstract machine transitions than the base case of *reverse*. By capturing resource usage at a higher level of abstraction, namely recursive calls, we have proved that the linear-time reverse function improves its quadratic counterpart in more a practical sense.

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 to calculate more 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 (and unary) 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 more examples.

[Radiček et al. 2017] theorises 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. 2017] 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 formally deriving a new and improved implementation from a specification.

7 CONCLUSION AND FURTHER WORK

This article has demonstrated how Liquid Haskell can be used to reason about efficiency, by introducing a library for analysing the resource usage of pure Haskell programs. Furthermore, by surveying a range of examples from the existing literature, we have shown how the system's existing support for correctness verification can be harnessed to ensure cost analysis is meaningful and precise.

There are three main avenues for further work. Firstly, we would like to develop a GHC plugin that can be used to automatically annotate standard Haskell code prior to analysis, and furthermore, remove all cost annotations post analysis. Secondly, we wish to provide support for solving recurrence relations. For this, we look to the TiML language for guidance. Finally, we plan to incorporate the cost analysis of monadic Haskell 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.

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.

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*.

Martin A. T. Handley and Niki Vazou. 2019. GitHub Repository for Liquidate Your Assets. (2019). https://github.com/mathandley/RTick.

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).

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. 2017. Monadic Refinements for Relational Cost Analysis. *PACMPL* POPL (2017).

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).