Consolidation of Haskell Programs

Semantic fusion of maps, filters and folds

Alberto Sadde

Pembroke College

University of Oxford

Supervised by Marcelo Sousa & Daniel Kroening

Submitted in partial completion of the

MSc Computer Science

Trinity 2016

Acknowledgements

Firstly, I would like to thank my supervisors Daniel and Marcelo for the invaluable comments and guidance during this project. I would also like to thank my friends Antonio, Francisco, Zac and Dave for all the much needed coffee breaks during this intense period. Finally, I would like to thank my parents without whom none of this would have been possible.

Consolidation of Haskell Programs

Semantic fusion of maps, filters and folds

Alberto Sadde

Department of Computer Science, University of Oxford

Abstract. Program consolidation is a recent, correct-by-construction program optimisation calculus that exploits semantic relations between terms in multiple programs to obtain a *consolidated* program that is a sound optimisation of the sequential execution of the original programs. Inspired by its success in the imperative setting, where the focus is on domain-specific languages without side effects, in this paper we explore its applicability in the setting of a functional language. Motivated by the latest developments in logical verification for Haskell such as refinement types, we present our *consolidation theory* as a set of inference rules annotated with semantic information. Furthermore, following the renewed interest in improvement theory, an operational theory that aims to bring rigour to efficiency improvement proofs, we present a cost-annotated operational semantics for a call-by-value lambda calculus that we use to formally show that the consolidation rules, when applicable, yield efficiency improvements. We also present the results of a survey of the Haskell ecosystem and a series of benchmarks that show that exploiting semantic dependencies between inputs is a viable direction to optimise performance and memory consumption of a broad range of functional programs.

Table of Contents

1 Introduction

One of the fundamental data structures used in functional languages, like Haskell or ML, are lists. This algebraic data type, often implemented as a "cons cell" (e.g., `List a = Nil | Cons a (List a)`), enables to pack multiple values into a bigger, combined value, and allows for a more idiomatic and concise style of programming. For example, in Haskell, a pure and lazy functional language, lists are the most frequently used data structure (see Sect. 5). They are commonly used as the glue that connects separate parts of the program, and enable a very simple and concise style of programming, as illustrated by Example 1.

Example 1.

```
sum_example :: Int
sum_example = sum (map (*2) [1,2,3,4])

map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = f x : map f x

sum :: [Int] → Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Here, the higher-order function (i.e., a function that takes other functions as arguments) `map`, which applies a given function to all the elements of a list, builds an intermediate list that is then used as the input of `sum` which adds up all the elements of a list. ◁

Consequently, much of the research on optimising compilation of Haskell has focused on lists and similar data structures. A particularly well-studied optimisation for this case is *deforestation* [40]. This optimisation relies on an algorithm that is able to eliminate intermediate trees and lists inside a pipeline of functions that use a specific syntactic form. The benefits of deforestation become apparent when we consider cases such as the following:

```
map f (map g xs)
```

Similarly to Example 1, the first application of `map` creates an intermediate list that is then consumed by the second application of `map`. Deforestation is able to eliminate the generation of the intermediate list by combining the two applications of `map`:

```
map (f ∘ g) xs
```

This yields a program that does not generate intermediate lists.

There exist many similar approaches that focus on optimising intermediate data structures. Instances include methods that eliminate the intermediate lists generated by common functions such as `(++)`, which concatenates two lists, and `reverse`, which reverses the list [37]. More recent developments include

"stream fusion" [8], a powerful deforestation technique, which eliminates intermediate data structures by exploiting the coalgebraic representation of lists (i.e., streams) and their unfoldings.

State-of-the-art compilers for Haskell such as GHC [21] implement many of these optimisations in the form of program transformations [27], i.e., a set of semantics-preserving operations that transform the original program into a more efficient one. Among other optimisations, GHC implements *short-cut fusion* [12], a variation of deforestation that focuses on eliminating intermediate lists in functions that can be expressed as a combination of a *fold* and an *unfold* [35,16]. However, there are many very common patterns that cannot be optimised using any of these variations of deforestation.

Example 2. Consider the `filter` function, which takes as input a predicate and a list and returns a list of all the elements that satisfy the predicate:

```
filter :: (a → Bool) → [a] → [a]
filter p [] = []
filter p (x:xs) = if p x
                  then x : filter p xs
                  else filter p xs
```

Suppose we have the following program that filters a list `xs` using two different predicates `p` and `q` and `sum` the results:

```
exampleFilters :: (Int → Bool) → (Int → Bool) → [Int] → Int
exampleFilters p q xs =
  let l1 = filter p xs
      l2 = filter q xs
  in sum l1 + sum l2
```

This program traverses the list `xs` twice: once to compute the `l1`, and one to compute `l2`. It is clear that it would be beneficial to traverse `xs` only once. We could achieve this by using `foldr`:

```
foldr :: (a → b → b) → [a] → b → [b]
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

This function consumes a list according to a function `f` and a starting value `z`. Using `foldr`, we can write the following function that traverses `xs` only once:

```
twoFilters :: (a → Bool) → (a → Bool) → [a] → ([a],[a])
twoFilters p q xs = foldr (select p q) ([],[]) xs
select p q x (l1,l2) = let l1' = if p x then (x:l1) else a
                           l2' = if q x then (x:l2) else b
                       in (l1',l2')
```

And generate the following optimised program:

```
exampleFiltersOpt :: (Int → Bool) → (Int → Bool) → [Int] → Int
exampleFiltersOpt p q xs =
  let (l1,l2) = twoFilters p q xs
  in sum l1 + sum l2
```

◁

Typically, without any further semantic information regarding the predicates, the program `exampleFiltersOpt` can be arguably considered the best optimisation of `exampleFilters`.

1.1 Consolidation

In this paper, by studying optimisations derived by exploiting semantic relations between the inputs of functions, we will show that we can speed up programs by at least 20%. For example, considering the previous function `twoFilters`, we could exploit a relation between the predicates that it consumes in order to reduce the number of boolean comparisons and thus obtain a faster program.

This insight follows a recent trend in verification of functional programs using logical relations such as refinement types [36] and optimisation of domain specific languages such as program consolidation [33], a recent development that exploits semantic relations between terms and is boosted by recent developments in SMT technology [10].

Example 3. As an example of how semantic relations between inputs can be used, consider the following predicates:

```
p = (λx → x > 10)
q = (λx → x > 5)
```

It is straightforward to conclude from the definitions of `p` and `q` that the following equation holds:

$$\forall\ x \in \mathrm{Int}\ .\ p\ x \Rightarrow q\ x \tag{1}$$

In Example 2, we used a relation between the lists in order to reduce the number of traversals over the data structures. In this case, by exploiting the semantic relations between p and q, we are able to improve upon the function `twoFilters`, and generate `twoFiltersCons`:

```
twoFiltersCons :: (a → Bool) → (a → Bool) → [a] → ([a],[a])
twoFiltersCons p q xs = foldr (select p q)  ([],[]) xs
  where select p q x (l1,l2) =
          | p x = (x:l1, x:l2)
          | otherwise = (l1, if q x then x:l2 else l2)
```

This new function only performs one traversal over the list and checks the predicate `q`, only on those elements that do not satisfy `p`. ◁

Inspired by program consolidation, our insight is to exploit this kind of semantic relations between the inputs of functions in order to derive optimisations in functional programs.

1.2 Contributions

This paper makes the following key contributions:

- We propose *semantic fusion* as a new technique to derive correct by construction improvements for programs in functional languages.
- We present an optimisation technique, semantic fusion, in the form of inference rules with semantic annotations that could be implemented as rewrite rules similar to those currently supported by the GHC Haskell compiler [19].
- We present the results of a large-scale survey of the Haskell ecosystem, and make it available online [30].
- We provide a set of 12 benchmarks to show that our technique produces runtime improvements and space improvements. We also make public the source code used for the tests [30].

Our evaluation and experiments strongly suggest that the rules we present are able to obtain speedups of 40% on average and reduce memory consumption of programs by at least 5%. This suggest that, indeed, program consolidation is a viable optimisation technique also in the case of functional programs.

1.3 Outline

The rest of this paper is divided as follows: in Section 2, we motivate the applicability of program consolidation by describing and running benchmarks on simple Haskell programs. In Section 3, we present a set of consolidation rules and provide an example of how to apply the rules in practice. In Section 4, we develop the theory behind the consolidation rules by means of a call-by-value lambda calculus and a cost-annotated operational semantics. In Section 5, we assess the improvements obtained by our rules over a set of benchmarks. Furthermore, we explain the process used to obtain these benchmarks and present the results of a large-scale survey of Haskell programs. In Section 6 we give a critical analysis of our theory and approach. Finally, in Section 7, we discuss related work and we conclude, in Section 8, with an overview of future work.

2 Motivating Examples

In this section, we give two examples and a small set of benchmarks, which motivate the applicability of program consolidation to programs written in a functional setting.

Example 2 shows that, in the absence of semantic relations between inputs, often, the best optimisation possible simply amounts to reduce the number of traversals over the data structure by using some version of tupling. Tupling [17] is a common technique used to optimise recursive functions. It reduces the number of traversals over a data structure or avoids redundant computations, by packing intermediate results in a tuple.

In the two examples that we present in this section, we will see how, by combining tupling and semantic knowledge about inputs, we can generate faster programs.

2.1 Filters and Partition

Consider a variation of the `exampleFilters` program from Example 2:

```
sumFilters :: (Int → Bool) → (Int → Bool) → [Int] → Int
sumFilters p q xs =
  let l1 = filter p xs
      l2 = filter q xs
  in sum l1 + mult l2

mult :: [Int] → Int
mult xs = foldr (λx y → x*2 + y) 0 xs
```

Here, we filter the same list according to two different predicates and then sum the results of adding up the first list and the result of adding up all the elements of the second list, which have been multiplied by two.

Similar to Example 2, without any further information regarding the predicates `p` and `q`, the natural optimisation is to pack together both filters into a single traversal of the list. However, lets assume that the two predicates are mutually exclusive, this is, `q == not ∘ p`. By using this fact, we can then replace both applications of `filter` by a single application of the well known `partition` function. Partition splits the input list into two lists, the first with all the elements that satisfy a predicate and second with the elements that do not satisfy it:

```
partition :: (a → Bool) → [a] → ([a],[a])
partition p xs = foldr (select p) ([],[]) xs
  where
  select p x (l1,l2)
  | p x       = (x : l1, l2)
  | otherwise = (a, l1 : l2)
```

It is straightforward to see that `partition` satisfies the following equality:

$$\frac{\texttt{(filter p xs, filter (not} \circ \texttt{p) xs)}}{\texttt{partition p xs}}$$

This means that we can replace both applications of `filter` in `sumFilters`, and obtain the *consolidated* program:

```
sumFiltersCons :: (Int → Bool) → [Int] → Int
sumFiltersCons p xs =
  let (l1,l2) = partition p xs
  in sum l1 + mult l2
```

By this reasoning, it would seem clear that, by reducing the number of traversals and predicate checks, we should obtain an optimised program.

To test this intuition and decide whether pursuing consolidation in a functional setting, more specifically Haskell, we ran a series of benchmarks to compare the two programs above. In order to obtain a meaningful conclusion, we generated the benchmarks with respect to three parameters: (i) the size of the

list, (ii) the distribution of elements in the list and (iii) the complexity of the predicates. We used lists of integers of various sizes starting from 100 up to 10 million elements. These lists were generated in sequence of increasing elements and also using a random number generator. We varied the predicates from simple integer comparison to modulus operations to account for memory accesses and reduce the noise associated with the garbage collector.

We present the results of this experiment in Figure 1. We used GHC 7.10.3 on a standard machine with a multicore Intel i7 processor, 16 GB of RAM, and running the latest version of OS X.

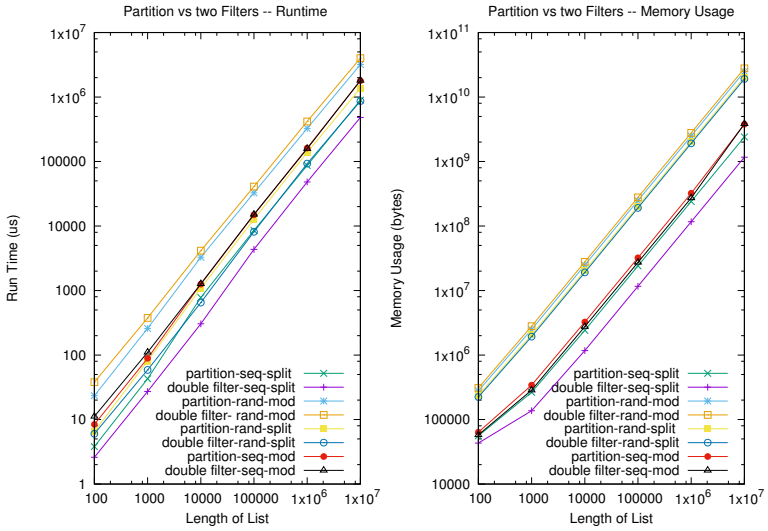

Fig. 1: Two `filters` vs. `partition`. Tested with integer comparison that split the lists in half (the predicate compares each elements against the length of the list / 2) and modulus operations, the predicate $x^3\ mod\ 5 == 0$ was generated at random (notice the log scales).

The results, in Figure 1 (left), show that it is not always the case that the intuition that a fewer number of traversals yields an optimisation. In fact, using simple integer comparison, two traversals over the list are faster than one traversal. Nevertheless, with computationally more expensive predicates (lines partition-rand-mod, partition-seq-mod, double filter-rand-mod and double filter-seq-mod), we observe that one traversal is an optimisation of several traversals. Therefore, we still live in a world where reducing the number of traversals over a data structure is an optimisation!

However, notice that, even when traversing the list only once, the optimised programs consumes, roughly, the same amount of memory as the original one (Figure 1 (right)). This effect is also seen in [17], where the authors state that "by

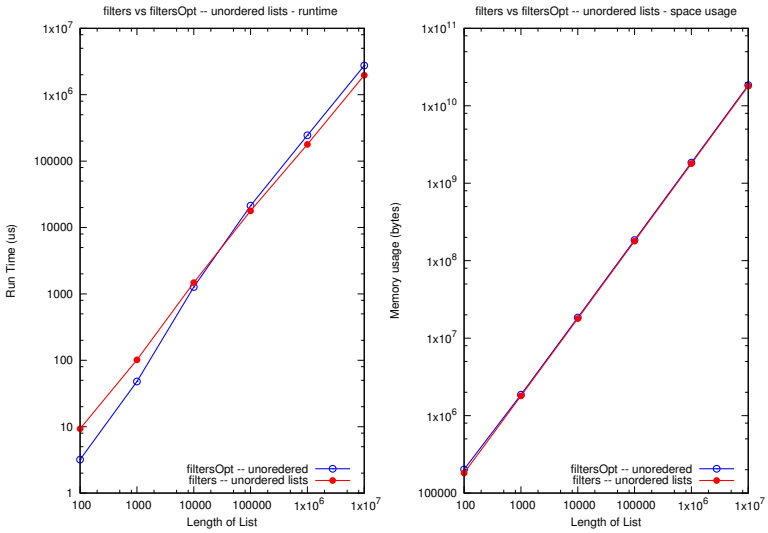

Fig. 2: `filters` vs. `filtersOpt`. Elements are generated at random (notice the log scale).

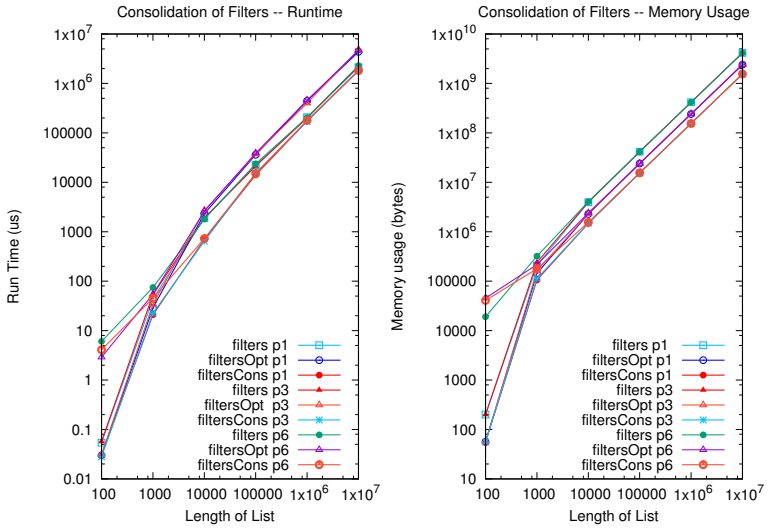

Fig. 3: `filters`, `filtersOpt` and `filtersCons`. Lists are ordered and the first predicate they satisfy is `p1`, `p3` and `p6` respectively (notice the log scale).

the ordinary implementation of the tuple data structure our algorithm does not guarantee spectacular efficiency improvements". In fact, in a recent discussion on StackOverflow [29], a popular Q&A site for developers, it was suggested that the problem is caused by the continuous construction and destruction of tuples (Haskell is a pure language so data structures are immutable) and the way they are handled by the garbage collector.

Interestingly, this example provides evidence to the famous quote (attributed to Hoare) "premature optimisation is the root of all evil" [20]. Indeed, in the case of integer comparison (lines partition-seq-split, double filter-seq-split in Figure 1), by applying the transformation *too early*, we end up with a program that is much more inefficient than the original one: using two filters is up to 2.4 times faster and 1.4 times faster on average.

Furthermore, most of the uses of `partition` that we have come across involve very simple predicates. Thus, a potential interesting practical optimisation could be to substitute a single application of `partition` by two filters. This could, in turn, open up avenues for parallelisation given that two filters are more easily parallelisable than partition. But, in this work, we do not address parallelism and focus solely on sequential programs.

2.2 Reducing n traversals to 1

In this example, we are going to illustrate the power of leveraging semantic information between inputs in a program. Thus, we are going to apply the intuition for consolidation to produce faster and more efficient programs. For sake of an example, consider the following toy program, `filters`:

```
filters (p1,p2,p3,p4,p5,p6) xs =
  (filter p1 xs, filter p2 xs, filter p3 xs,
   filter p4 xs, filter p5 xs, filter p6 xs)
```

This program receives a tuple of 6 predicates and filters the same list according to each of the predicates. It is clear that this can be generalised to a filter of n predicates (e.g. over a list). As usual, we are interested in reducing the number of traversals of the input list `xs` from n to 1 and in exploiting semantic relations between the predicates in order to reduce the number of boolean tests. A naive approach involves only reductions in the number of traversals, such as in the case of:

```
filters' (p1,p2,p3,p4,p5,p6) xs =
  foldr (λx (l1,l2,l3,l4,l5,l6) →
        ( check p1 x l1 , check p2 x l2
        , check p3 x l3 , check p4 x l4
        , check p5 x l5 , check p6 x l6)) ([],[],[],[],[],[]) xs
  where
    check p x l = if p x then x:l else l
```

In this case, `filters'` is not really an optimisation since, the overall amount of work will remain the same, all the predicates will still be checked in every iteration.

However, there are cases where the insights of program consolidation can help achieve dramatic speedups. As an example, suppose we want to use `filters` with the following predicates:

```
ps = ((> 500), (> 499), (> 400), (> 302) (> 2), (> 1))
```

In this case, we can see that the following equation holds:

$$\forall\, x \in \text{ Int } .(x > 500) \Rightarrow (x > 499) \Rightarrow (x > 302) \Rightarrow (x > 400) \Rightarrow (x > 2) \Rightarrow (x > 1) \tag{2}$$

i.e., if $x > 500$ it follows that x will satisfy the rest of the predicates as well. Using this information, we are able to generate the following optimised program:

```
filtersOpt ps@(p1,p2,p3,p4,p5,p6) xs =
foldr (select ps) ([],[],[],[],[],[]) xs
  where select (p1,p2,p3,p4,p5,p6) x (l1,l2,l3,l4,l5,l6)
          | p1 x      = (x:l1,x:l2,x:l3,x:l4,x:l5,x:l6)
          | p2 x      = (l1,x:l2,x:l3,x:l4,x:l5,x:l6)
          | p3 x      = (l1,l2,x:l3,x:l4,x:l5,x:l6)
          | p4 x      = (l1,l2,l3,x:l4,x:l5,x:l6)
          | p5 x      = (l1,l2,l3,l4,x:l5,x:l6)
          | p6 x      = (l1,l2,l3,l4,l5,x:l6)
          | otherwise = (l1,l2,l3,l4,l5,l6)
```

It is clear that this function not only reduces the number of traversals over `xs`, but also reduces the total number of boolean tests. In fact, in the best case, this function will only perform $length(xs)$ boolean checks and only in the worst case (when all the elements satisfy only the last predicate) it will perform the same number of boolean tests as `filters`.

In `filtersOpt` we can see a cascade effect: as soon as a predicate is satisfied, we simply prepend the element to the lists corresponding to the remaining predicates. This suggests that we can obtain further improvements by exploiting the ordering of the elements of the list. If `xs` is a list where the elements are sorted in an increasing order, then, as soon as an element satisfies a predicate, we know that all the remaining elements will satisfy the current and all the remaining predicates as well. Therefore, we can generate the following consolidation:

```
filtersCons _ [] = ([],[],[],[],[],[])
filtersCons ps@(p1,p2,p3,p4,p5,p6) xs'@(x:xs) =
  filterCons' ps x xs' (filterCons ps xs)

filterCons' ps@(p1,p2,p3,p4,p5,p6) x xs' (l1,l2,l3,l4,l5,l6)
  | p1 x         = (xs',xs',xs',xs',xs',xs')
  | p2 x         = (l1,xs',xs',xs',xs',xs')
  | p3 x         = (l1,l2,xs',xs',xs',xs')
  | p4 x         = (l1,l2,l3,xs',xs',xs')
  | p5 x         = (l1,l2,l3,l4,xs',xs')
  | p6 x         = (l1,l2,l3,l4,l5,xs')
  | otherwise    = (l1,l2,l3,l4,l5,l6)
```

`filtersCons` improves over both `filters` and `filtersOpt`. This is, by using semantic dependencies between the inputs, we are able to generate a program that will do the same amount of work as `filtersOpt` only in the case where no predicate is satisfied.

In Figures 2 and 3 we present the results of a series of benchmarks that we ran on these three programs in order to quantify and understand the extent of the gains that consolidation makes possible. In Figure 2 we compared `filters` and `filtersOpt`. The left plot shows the runtime of both programs while the right figure shows the memory consumption. In this case, we suffer from the same problem as in the example in Section 2.1: reducing the number of traversals does not guarantee a reduction in speed. However, Figure 3 (left) shows how, using more information about the inputs, we can get a meaningful speedup (compare, for example, the line "filtersCons p1 " against the other "p1" lines in the figure).

This example provides more evidence in favour of consolidation in a functional setting. In fact, `filterCons` is able to run more than three times faster than the original program, suggesting that, with an efficient implementation of tuples, we might achieve dramatic, super linear, speedups.

3 Consolidation Rules

In this section, we present the consolidation rules that can be used to optimise a set of common higher-order recursion schemes, which operate over lists, and can be implemented in any functional language.

All the rules are similar and focus on the optimisation of the following pattern, that we have encountered in real world programs:

$$(rec\ f\ \dagger\ xs,\ rec\ g\ \dagger\ ys) \tag{3}$$

Where $rec \in \{fold,\ map,\ filter\}$, f and g are possibly distinct functions, xs and ys are lists and $\dagger$ is the extra argument consumed by $fold$.

We present the rules as inference rules with judgements of the form

$$\Psi \vdash p$$

Here, Ψ is a context containing all the functions defined in Figures 4 and 5, plus a set of new definitions and p is a valid logic formula under the given context, a precondition, which needs to be satisfied for the consolidation rule to be applicable. As usual, the trivial precondition, denoted as $\Psi \vdash true$, is satisfied by any context.

Example 4. Consider the following judgement

$$\Psi\{\text{xs} = [1,2,3],\ \text{ys} = [1,2,3,4]\} \vdash \forall x \in xs.\ x \in ys$$

This says that, under the given context, if $xs = [1,2,3]$ and $ys = [1,2,3,4]$, it follows that all the elements in xs are also in ys, i.e., $xs \subseteq ys$. $\lhd$

Figure 4 presents a set of auxiliary functions used in the definition of the rules. Figure 5 defines a set of higher order functions and the data types that we use throughout our presentation. We define a type $P\ a\ b$ for tuples and use the shorthand notation (a, b) from now on. We also define an enhanced version of the common *List* data type, annotated with its length. We do this by means of a tuple, where the first element stores the length of the list, and the second element is the usual list. This allows to define a function *length* that is $O(1)$, i.e. $length = \pi_1$ where π_i defines the projections over tuples. When no ambiguity arises, we elide the first element of the pair in the presentation of the enhanced list. To simplify the presentation, we also define the $\otimes$ function. It takes two functions and a value and returns a tuple where each function has been applied to the given value. The set of consolidation rules is given in Figure 6.

Example 5. In order to understand how the rules can be used, lets consider rule (Map R1) and try to apply it. This rule says, that if under the given context Ψ the precondition $xs \sqsubseteq_{pre[Int]} ys$ is satisfied then, we can substitute the left-hand side of $\triangleright$ by the right-hand side of it (for a complete definition of $\triangleright$, see Definition 1 in Section 4).

Consider the following program

$$(map\ (\lambda x.\ x\ +\ x))\ [1..100], map\ (\lambda x.\ x\ +\ x))[1..150])$$

We use the shorthand notation $[m..n]$ to represent lists containing all the elements from m to n inclusive. This program maps a function, which adds an element to itself, to two different lists, xs and ys. In this case then, the context Ψ contains all the necessary definitions, i.e., definitions for tuple, *map* and lists, and contains also the definitions of the inputs $\lambda x.\ x\ +\ x$, xs and ys.

Now, rule (Map R1) is applicable only if the following judgement is satisfied:

$$\Psi\{xs : [Int],\ ys : [Int],\ f : Int \to Int\} \vdash xs \sqsubseteq_{pre[Int]} ys$$

In this we case we have:

$$\Psi\{(\lambda x.\ x\ +\ x),\ [1..100],\ [1..150]\} \vdash [1..100] \sqsubseteq_{pre[Int]} [1..150]$$

We assume that $(\lambda x.\ x\ +\ x) : Int \to Int$ and $xs,\ ys : [Int]$. The right-hand side of $\vdash$ is satisfied when all the elements of [1..100] appear in [1..150] and appear in the same order (see Section 4.2 for the complete definition of $\sqsubseteq_{pre[a]}$). Clearly, in this case, all the elements of xs are also in ys and they appear in the same order in both lists. Then, the precondition is satisfied so we can trigger the rule and obtain the program:

$$\otimes\ (take\ (length\ [1..100]))\ id\ (map\ (\lambda x.\ x\ +\ x)\ [1..150])$$

Finally, notice that this new program is an optimisation. Instead of mapping the same function two times over the list [1..100] (one time for xs and one time for ys), we simply map the function over the longer list and then collect the first 100 elements to get the final result. $\triangleleft$

In the next section, we will formally motivate and prove that this and the rest of the rules in Figure 6 are indeed optimisations.

$$
\begin{array}{ll}
take & :: \ Int \to [Int] \to [Int] \\
take\ n\ xs & = case\ n\ of \\
 & \quad 0 \to Nil \\
 & \quad n' \to case\ xs\ of \\
 & \quad\quad Nil \to Nil \\
 & \quad\quad Cons\ y\ ys \to Cons\ y\ (take\ (n-1)\ ys)
\end{array}
$$

$$
\begin{array}{ll}
drop & :: \ Int \to [Int] \to [Int] \\
drop\ n\ xs & = case\ n\ of \\
 & \quad 0 \to xs \\
 & \quad n' \to case\ xs\ of \\
 & \quad\quad Nil \to Nil \\
 & \quad\quad Cons\ y\ ys \to drop\ (n-1)\ ys
\end{array}
$$

$$
\begin{array}{ll}
(++) & :: \ [Int] \to [Int] \\
(++)\ xs\ ys & = case\ xs\ of \\
 & \quad Nil \to ys \\
 & \quad Cons\ x\ xs' = Cons\ x\ ((++)\ xs'\ ys)
\end{array}
$$

$$
\begin{array}{ll}
maps & :: \ (Int \to Int) \to (Int \to Int) \to [Int] \to ([Int], [Int]) \\
maps\ p\ q\ xs & = fold\ (\lambda x\ (l1, l2).\ (Cons\ (f\ x)\ l1, Cons\ (g\ x)\ l2))\ (Nil, Nil)\ xs
\end{array}
$$

$$
\begin{array}{ll}
filters2 & :: \ (Int \to Bool) \to (Int \to Bool) \to [Int] \to ([Int], [Int]) \\
filters2\ p\ q\ xs & = fold\ (\lambda x\ (l1, l2).\ case\ (p\ x)\ of \\
 & \quad \top \to (Cons\ x\ l1, Cons\ x\ l2) \\
 & \quad \bot \to (l1,\ case\ (q\ x)\ of\ \{\top \to (Cons\ x\ l2), \bot \to l2\}))(Nil, Nil)\ xs
\end{array}
$$

$$
\begin{array}{ll}
filters3 & :: \ (Int \to Bool) \to [Int] \to [Int] \to [Int] \\
filters3\ p\ xs\ ys\ l & = \ l\ (++)\ (filter\ p\ (drop\ (length\ xs)\ ys))
\end{array}
$$

$$
\begin{array}{ll}
filters4 & :: \ (Int \to Bool) \to (Int \to Bool) \to [Int] \to ([Int], [Int]) \\
filters4\ p\ q\ xs & = fold\ (\lambda x\ (l1, l2).\ case\ (p\ x)\ of \\
 & \quad \top \to (Cons\ x\ l1,\ case\ (q\ x)\ of\ \{\top \to (Cons\ x\ l2), \bot \to l2) \\
 & \quad \bot \to (l1, l2))(Nil, Nil)\ xs
\end{array}
$$

Fig. 4: Auxiliary functions used in consolidation rules.

$$
\begin{array}{ll}
\text{List a n} & = (0, Nil) \;\mid\; (n,\ L\ a) \\
\text{L a} & = \text{Cons a L a} \\
 & \\
length & :: [Int] \to Int \\
length\ xs & = \pi_1 \\
 & \\
\text{P a b} & = (a, b) \\
\text{P a b} & = (a, b) \\
 & \\
\pi_1 & :: (A, B) \to A \\
\pi_1(a, b) = a & \\
 & \\
\pi_2 & :: (A, B) \to A \\
\pi_2(a, b) = a & \\
 & \\
id & :: (A \to A) \\
id & = \lambda x.\ x \\
 & \\
\otimes & ::\ (A \to B) \to (A \to C) \to A \to (B, C) \\
\otimes & = \lambda f\ g\ xs.(f\ xs,\ g\ xs) \\
 & \\
map & :: (Int \to Bool) \to [Int] \to [Bool] \\
map\ f\ xs & = case\ xs\ of \\
 & \qquad (0, Nil) \to (0, Nil) \\
 & \qquad (n, Cons\ y\ ys) \to (n, Cons\ (f\ y)\ (map\ f\ ys)) \\
 & \\
filter & :: (Int \to Bool) \to [Int] \to [Int] \\
filter\ p\ xs & = case\ xs\ of \\
 & \qquad (0, Nil) \to (0, Nil) \\
 & \qquad (n, Cons\ y\ ys) \to case\ p\ x\ of \\
 & \qquad\quad \top \to (n', Cons\ y\ (filter\ p\ ys) \\
 & \qquad\quad \bot \to filter\ \text{p ys} \\
 & \\
fold & :: (A \to B \to B) \to B \to [A] \to B \\
fold\ \text{f z xs} & = case\ xs\ of \\
 & \quad (0, Nil) \to z \\
 & \quad (n, Cons\ y\ ys) \to f\ y\ (fold\ f\ z\ ys) \\
 & \\
partition & :: (Int \to Bool) \to [Int] \to [Int] \\
partition\ \text{p xs} & = fold\ (\lambda x\ (l1, l2).\ case\ p\ x\ of \\
 & \qquad \top \to ((n', Cons\ x\ l1),\ l2) \\
 & \qquad \bot \to (l1, (n'', Cons\ x\ l2)))\ ((0, Nil), (0, Nil))\ xs
\end{array}
$$

Fig. 5: Predefined functions.

$$\frac{\Psi\{xs : [Int],\ f : Int \to Int,\ rec \in \{map, filter, fold\}\} \vdash true}{(rec\ f\ \dagger\ xs,\ rec\ f\ \dagger\ xs) \triangleright \otimes\ id\ id\ (rec\ f\ xs)}\ \text{(Trivial)}$$

$$\frac{\Psi\{xs : [Int],\ ys : [Int],\ f : Int \to Int\} \vdash xs \sqsubseteq_{pre[Int]} ys}{(map\ f\ xs, map\ f\ ys) \triangleright \otimes(take\ (length\ xs))\ id\ (map\ f\ ys)}\ \text{(Map R1)}$$

$$\frac{\Psi\{xs : [Int],\ f : Int \to Int,\ g : Int \to Int\} \vdash f \neq g}{(map\ f\ xs,\ map\ g\ xs) \triangleleft \triangleright \otimes\ \ \pi_1\ \pi_2\ (maps\ f\ g\ xs)}\ \text{(Map R2)}$$

$$\frac{\Psi\{xs : [Int],\ p : Int \to Bool,\ q : Int \to Bool\} \vdash \forall x \in xs.\ p\ x = \neg\ q\ x}{(filter\ p\ xs, filter\ q\ xs) \triangleright \otimes\ \pi_1\ \pi_2\ (partition\ p\ xs)}\ \text{(Filter R1)}$$

$$\frac{\Psi\{xs : [Int],\ p : Int \to Bool,\ q : Int \to Bool\} \vdash \forall x \in xs.\ p\ x \Rightarrow q\ x}{(filter\ p\ xs, filter\ q\ xs) \triangleright \otimes\ \pi_1\ \pi_2\ (filters2\ p\ q\ xs)}\ \text{(Filter R2)}$$

$$\frac{\Psi\{xs : [Int],\ ys : [Int],\ p : Int \to Bool\} \vdash xs \sqsubseteq_{pre[Int]} ys}{(filter\ p\ xs, filter\ p\ ys) \triangleright \otimes\ \pi_1\ (\lambda l.\ filters3\ p\ xs\ ys\ l)\ (filter\ p\ xs)}\ \text{(Filter R3)}$$

$$\frac{\Psi\{xs : [Int],\ p : Int \to Int,\ q : Int \to Bool\} \vdash true}{\otimes\ id\ (\lambda l.\ filter\ q\ l)\ (filter\ p\ xs) \triangleleft \triangleright \otimes\ \ \pi_1\ \pi_2\ (filters4\ p\ q\ xs)}\ \text{(Filter R4)}$$

$$\frac{\Psi\{xs : [Int],\ ys : [Int],\ z : Int,\ f : Int \to Int\} \vdash xs \sqsubseteq_{pre[Int]} ys}{(fold\ f\ z\ xs, fold\ f\ z\ ys) \triangleright \otimes\ id\ (\lambda z'.\ fold\ f\ z'\ (drop\ (length\ xs)\ ys))\ (fold\ f\ z\ xs)}\ \text{(Fold R1)}$$

Fig. 6: Consolidation Rules.

4 Consolidation Rules Theory

In this section we formalise the theory behind the consolidation rules presented in the previous section and defined in Figure 6.

We do this by means of a typed functional language with call-by-value semantics. In order to justify the language's simplicity, we emphasise that the focus is on common patterns where we have multiple applications of a single function to a (possibly) distinct set of inputs. Moreover, we use call-by-value semantics, since we are interested in consolidating expressions where all the arguments are used and therefore need to be fully evaluated.

4.1 Theory Setup

The syntax of the language is given in Figure 7. Expressions in this language include simple integer and boolean expressions (*ie* and *be* respectively), variables, lambda abstractions, function application and fully saturated data constructors D. Values can only be integer or boolean values, lambda abstractions and fully evaluated data constructors. We also provide the basic types $Bool$, Int and use T to represent arbitrary data types, such as tuples (a,b), lists [a], etc.

$$
\begin{aligned}
\textit{Integer Exprs } ie &:= int \mid ie \oplus ie \ (\oplus \in \{+, -, *\}) \\
\textit{Boolean Exprs } be &:= \top \mid \bot \mid \neg be \\
&\quad \mid ie \odot ie \ (\odot \in \{\leq, <, =\}) \mid be \oslash be \ (\oslash \in \{\wedge, \vee\}) \\
\textit{Exps } e &:= x \mid \lambda x.\, e \mid ie \mid be \mid (e\ e) \mid D\ e \\
\textit{Basic Types } bt &:= Bool \mid Int \\
\textit{Types } \tau &:= bt \mid T\ bt \mid \tau \to \tau
\end{aligned}
$$

Fig. 7: Functional language used for the functional theory of consolidation.

Figure 8 gives a big-step, cost annotated operational semantics for the language, with judgements of the form

$$E,\ e \Downarrow_k c$$

where E is an environment that maps variables to expressions, e is an expression, c is a value or an expression and k is the cost associated with the evaluation of e.

The cost semantics is given using an abstract *cost* function that assigns a particular cost to the different operations and to the values of type $Bool$ and Int. We assume that the substitution of variables by their definitions, stored in

$$\frac{E, e \ \Downarrow_k \ e'}{E, \lambda x.\ e \ \Downarrow_k \ \lambda x.\ e'} \text{ (Lambda)} \qquad \frac{b \in \{\top, \bot\} \quad cost(bool) = k}{E, b \ \Downarrow_k \ b} \text{ (Cost-bool)}$$

$$\frac{c \in Int \quad cost(int) = k}{E, c \ \Downarrow_k \ c} \text{ (Cost-int)} \qquad \frac{E, e_1 \ \Downarrow_k \ \lambda x.\ e \quad E, e_2 \ \Downarrow_l \ e_2'}{E, (e_1\ e_2) \ \Downarrow_{k+l} \ e[e_2'/x]} \text{ (App)}$$

$$\frac{E, e_1 \ \Downarrow_{k_1} \ c_1 \quad E, e_2 \ \Downarrow_{k_2} \ c_2 \quad cost(\oplus) = m}{E, e_1 \oplus e_2 \ \Downarrow_{k_1+k_2+m} \ c_1 \oplus c_2} \text{ (Int-1)} \qquad \frac{E, e_1 \ \Downarrow_{k_1} \ c_1 \quad E, e_2 \ \Downarrow_{k_2} \ c_2 \quad cost(\odot) = n}{E, e_1 \odot e_2 \ \Downarrow_{k_1+k_2+n} \ c_1 \odot c_2} \text{ (Int-2)}$$

$$\frac{E, e_1 \ \Downarrow_{k_1} \ c_1 \quad E, e_2 \ \Downarrow_{k_2} \ c_2 \quad cost(\oslash) = m}{E, e_1 \oslash e_2 \ \Downarrow_{k_1+k_2+m} \ c_1 \oslash c_2} \text{ (Bool-1)} \qquad \frac{E, e_1 \ \Downarrow_k \ b \quad cost(\neg) = m}{E, e_1 \ \Downarrow_{k+m} \ \neg b} \text{ (Bool-2)}$$

$$\frac{E, e_i \Downarrow_{n_i} c_i \quad eval(f(e_1, \ldots, e_k)) = (v, c) \quad w = \sum_{i=1}^{k} n_i + c}{E, f(c_1, \ldots, c_k) \ \Downarrow_w \ v} \text{ (Eval)} \qquad \frac{E, e_i \ \Downarrow_{n_i} \ c_i \quad w = \sum_i n_i \quad cost(D) = d}{E, D\boldsymbol{e} \ \Downarrow_{d+w} \ D\boldsymbol{c}} \text{ (Data)}$$

Typing judgements

$$\frac{b \in \{\top, \bot\}}{\Gamma \vdash b : Bool} \text{ (T-Bool)} \qquad \frac{n \in Int}{\Gamma \vdash n : Int} \text{ (T-Int)}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\ e : \tau \to \tau'} \text{ (T-Lambda)} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'} \text{ (T-App)}$$

Fig. 8: Operational semantics and typing rules.

the environment, has no cost.

The rules (Cost-int) and (Cost-bool) evaluate integer and boolean values, while the rules (Int-1), (Int-2), (Bool-1) and (Bool-2) define the evaluation of integer and boolean expressions and assign a cost to each operation.

The rules (Lambda), (App), and (Data) give the call-by-value semantics.

Rule (Lambda) attempts to evaluate an expression inside a lambda abstraction, the rule (App) is the usual function application, and (Data) ensures that expressions inside data constructors are fully evaluated, when possible.

The rule (Eval) allows to evaluate external functions. Using an abstract function *eval*, this rule enables to reason about the costs of the functions that are defined externally. Thus, for an externally defined function f and language normal forms $e_1, \ldots, e_k$,

$$eval(f(e_1, \ldots, e_k)) = (v, c)$$

where, v is the output of the function and c its cost.

Finally, also in Figure 8, we provide a set of basic typing rules for the language. The *type context* Γ is defined as a, possibly empty, set of pairs $\{x_i : A_i\}$, where x_i are distinct terms and A_i are their assigned types. These rules, (T-Bool), (T-Int), (T-Lambda) and (T-App), enable to give types to integer and boolean expressions. Moreover, in order to keep the language as simple as possible, we do not provide rules for polymorphic types. But, for clarity of presentation, a function $f : A \to B$ can be given the type $Int \to Bool$ by annotating it as $f_{int-bool}$. We elide this notation when there is no ambiguity with respect to the types of the functions.

Given that the language lacks *let* and *case* expressions, we assume that all higher-order recursive functions are externally defined and always accessible in the environment E, contain no type errors and can operate over Integer and Boolean values.

Example 6. Consider the following simple expression:

$$map\ (\lambda x.\ x + 1)\ [1]$$

where *map* is as defined in Figure 5. [1] is an *enhanced* list of Ints, where we elide the first member of the pair, and $\lambda x.\ x + 1$ is a function that takes an integer and adds one to it. This expression adds one to every element of the list.

We can use the operational semantics to evaluate it, as follows:

$$\dfrac{\dfrac{\dfrac{E, 1 \ \Downarrow_l \ 1 \quad cost(Cons) = n}{E, [1] \ \Downarrow_l \ [1]}\ \text{(Data)}}{eval(map\ (\lambda x.\ x + 1)\ [1]) = ([1], c)}}{E,\ map\ (\lambda x.\ x + 1)\ [1] \ \Downarrow_{c+l+n} \ [1]}\ \text{(Eval)}$$

Also, we can use the typing rules to check that $map\ (\lambda x.\ x + 1)\ [1]$ has the correct type:

$$\dfrac{\dfrac{\Gamma \vdash map : (Int \to Int) \to [Int] \to [Int] \qquad \Gamma, \vdash (\lambda x.\ x + 1) : Int \to Int}{\Gamma \vdash map\ (\lambda x.\ x + 1) : [Int] \to [Int]}\ \text{t-app} \qquad \Gamma \vdash [1] : [1] \to Int}{\Gamma \vdash map\ (\lambda x.\ x + 1)\ [1] : [Int]}\ \text{t-app}$$

$\lhd$

4.2 Interpretations of Lists

Before continuing with the consolidation theory, it is worth to lay out the key algebraic properties of lists since, from now on, we assume that we operate over this kind of data structure only.

Lists are normally interpreted as *ordered multisets*. But there exist other interpretations that might prove useful. We now discuss four possible interpretations of lists.

Ordered multisets. This is the default interpretation of lists. It can be defined as a tuple $(\sqsubseteq, M)$ where $\sqsubseteq$ defines an ordering (a binary relation which is antisymmetric, transitive and reflexive) and $M = (S, m)$ where S is a set and $m : S \to \mathbb{N}_{\geq 1}$ defines the multiplicity of each element in S. With this interpretation we can define the prefix relation.

A list $xs = [x_1, \ldots, x_n]$ is a prefix of another list $ys = [y_1, \ldots, y_m]$ if

$$n \leq m \text{ and } \forall i = 1, \ldots, n\ .\ x_i = y_i$$

For example, the list $[1, 2, 3] \sqsubseteq_{pre[Int]} [1, 2, 3, 4, 5]$, where the notation $pre[a]$ defines the prefix relation over lists of type Int.

Now, recall that a partially ordered set $(L, \leq)$ forms a lattice if each pair of elements (x, y) has both a *least upper bound* (join) and *greatest lower bound* (meet). Then, we can see that $(L, \sqsubseteq_{pre[a]})$ forms a *meet semilattice* where, for every $x, y \in L$, the greatest lower bound is given by the longest prefix shared by both x and y. We will see that the prefix relation is very powerful and will enable us to apply consolidation to a variety of programs.

Ordered sets. Similar to the case of multisets, an ordered set is a pair $(\sqsubseteq, S)$ where S is a set and $\sqsubseteq$ defines an ordering on it. The prefix relation defined above can be used also for these sets without needing any modification.

Unordered sets. This is the usual definition of *set*. A straightforward relation, which is valid also for the ordered case, is the *subset* relation. For any two sets X and Y we have:

$$X \subseteq Y := \forall x \in X \Rightarrow x \in Y$$

Unordered multisets. This is the less restrictive interpretation we can get for lists. The simplest relation for multisets (also known as bags) is the *submultiset* relation. For any two multisets (M, m) and (N, m), we have:

$$(M, m) \subseteq_{multi} (N, n) := \forall x \in N \Rightarrow x \in M \wedge m(x) = n(x)$$

4.3 Soundness of the Rules

With these assumptions, we can proceed to describe the theory of consolidation rules for list-processing functional programs.

Definition 1 (Soundness). *We say that, under environment E, an expression e_1 is a consolidation of another expression e_2, denoted as $e_2 \triangleright e_1$, if*

$$E, e_1 \ \Downarrow_{k_1} \ c_1 \text{ and } E, e_2 \ \Downarrow_{k_2} \ c_2$$

then: $k_1 \leq k_2$, $\Gamma \vdash c_1 : \tau, c_2 : \tau$ and $c_1 = c_2$.

Accordingly, we use $e_2 \triangleleft e_1$ if $k_1 > k_2$, and $e_1 \triangleleft\triangleright e_2$ if $k_1 = k_2$.

In other words, a consolidation of an expression e is any expression, e', that has a smaller, or equal, cost than e, and yields the same result as the original expression.

In light of this definition and using the algebraic properties of lists (Section 4.2), lets reconsider Equation (3). According to this equation, all the rules follow a particular pattern and exploit information and relations between the four inputs f, g, xs and ys. By combining information between these inputs, we can generate the following four cases:

$$xs = ys \text{ and } f = g \tag{4}$$
$$xs \neq ys \text{ and } f = g \tag{5}$$
$$xs = ys \text{ and } f \neq g \tag{6}$$
$$xs \neq ys \text{ and } f \neq g \tag{7}$$

We now examine these in detail.

Equation (4) is the trivial case. The two input functions and the two lists are equal. The only thing that we can do is to avoid computing the value of the same expression twice. The rule for this case is the rule (Trivial).

If the input functions are equal, but the inputs lists are different (Equation (5)), we can try to exploit information between the two lists, by using one of the interpretations of lists defined in Section 4.2. In fact, both *map* and $filter$ consume the input list from left to right and the output preserves the order in which the elements are consumed. This suggests that, when using these functions, it is correct (and necessary) to view lists as *ordered multisets*. The same interpretation can be used with $folds$, but notice that, since this recursion scheme is more general, there might be other valid interpretations. For example, if the input to $fold$ is a symmetric function (such as addition) then, the order in which the elements of the list are consumed is irrelevant, so by interpreting lists as *unordered multisets*, we could open more avenues for consolidation. Then, we define three rules, (Map R1), (Filter R3) and (Fold R1), that exploit the prefix relation between the input lists in order to get a consolidation.

In the case of Equation (6), we are interested in relations between the input functions. For *map* there is little to do. The input functions must agree on their input type (since they both consume the same list), but the return types

might defer, so there are no guarantees that a semantic relation can be found. For this case, though, not all is lost. We define the rule (Map R2), a simple consolidation that packs both maps in a single traversal of the data structure. With $filters$, however, the situation is different. Filters enforce more restrictions on their inputs, given that they need to return a Boolean, so f and g must agree on their types. Then, we can exploit relations between them. Rule (Filter R1) replaces two filters, by a single application of the *partition* function, in cases where the input functions p and q are mutually exclusive. Rule (Filter R2), exploits the fact that *Bool* forms the trivial, two element, lattice (where the ordering is given by $\Rightarrow$), in order to reduce the number of times a predicate needs to be checked. This rule generates a consolidated expression that traverses the data structure only once and tests the predicate q only on those elements where p is not satisfied.

Equation (7) represents the most general case. In the case of *map*, there are no optimisations possible given that there are no possible relations between the inputs. For filters, there is a particular case, in which a filter is applied to the result of another filter. Similar to (Filter R3), rather than performing the two filters separately, we can traverse the data structure only once. Rule (Filter R4) gives the desired consolidation in this case.

4.4 Proofs of Consolidation Rules

We now motivate the correctness of the consolidation rules and show proofs for all the cases.

Firstly, notice that all the rules rely on the $\otimes$ combinator, which we conveniently defined in order to avoid the introduction of *let* expressions in the language. Then, given that we use call-by-value semantics, this combinator allows to mimic the sharing of computations. This is, for example, the expression $\otimes\ (map\ f)\ (map\ f)\ xs$ will compute $map\ f\ xs$ twice, but $\otimes\ id\ id\ (map\ f\ xs)$ will compute it only once.

In order to keep the correctness proofs simple, we use the cost model defined in Figure 9. We assume that integer and boolean values incur no cost, that the cost of tupling is also zero and that the cost of constructing a list is 1. We also assume that substitution of variables inside lambda abstractions has no cost. Figure 10 gives the costs of all the functions used in our rules. As an example, of how the costs are derived, consider the cost of $fold$. The (Eval) rule forces the evaluation of its inputs so we get $cost(f)$, $cost(xs)$ and $cost(z)$. Then, the cost of the case expression, in the body of the function, is approximated by adding the costs of applying the function to each element of the list and the initial value z.

By noticing that most of the functions that we use can be defined in terms of $fold$, the costs of all the other functions can be derived in a similar manner. For example, we can write *map* in terms of $fold$, by setting $f = \lambda x\ l.\ Cons\ (f'\ x)\ l$, where f' is the original mapping function, and $z = Nil$, then,

$$cost(map\ f'\ xs) = cost(fold\ f\ Nil\ xs)$$

$$
\begin{aligned}
&cost(Cons) = 1 \\
&cost(bool) \quad = cost(int) = 0 \\
&cost(\oslash) \quad\ = cost(\oplus) = cost(\neg) = 1 \\
&cost(Nil) \quad = 0 \\
&cost((,)) \quad = 0
\end{aligned}
$$

Fig. 9: Cost model used for proofs.

$$
\begin{aligned}
cost(fold\ f\ z\ xs) &= cost(f) + cost(xs) + cost(z) + \textstyle\sum_{x \in xs} cost(f\ x\ z) \\
cost(map\ f\ xs) &= cost(f) + cost(xs) + length(xs)cost(Cons) + \textstyle\sum_{x \in xs} cost(f\ x) \\
cost(filter\ p\ xs) &= cost(p) + cost(xs) + \textstyle\sum_{x \in xs} cost(p\ x) + \sum_{\substack{x \in xs \\ p\ x = \top}} cost(Cons) \\
cost(partition\ p\ xs) &= cost(p) + cost(xs) + \textstyle\sum_{x \in xs}(cost(p\ x) + cost(Cons)) \\
cost(take\ n\ xs) &= cost(xs) + ncost(Cons) \\
cost(drop\ n\ xs) &= cost(xs) + n \\
cost((++)\ xs\ ys) &= cost(xs) + cost(ys) + ncost(Cons) \\
cost(maps\ f\ g\ xs) &= cost(f) + cost(g) + cost(xs) + 2length(xs)cost(Cons) \\
&\quad + \textstyle\sum_{x \in xs}(cost(f\ x) + cost(g\ x)) \\
cost(filters2\ p\ q\ xs) &= cost(p) + cost(q) + cost(xs) + \textstyle\sum_{x \in xs} cost(p\ x) \\
&\quad + \textstyle\sum_{\substack{x \in xs \\ p\ x = \top}} 2cost(Cons) + \sum_{\substack{x \in xs \\ p\ x = \bot}} cost(q\ x) + \sum_{\substack{x \in xs \\ q\ x = \top \\ p\ x = \bot}} cost(Cons) \\
cost(filters3\ p\ xs\ ys\ l) &= cost(p) + cost(xs) + cost(ys) + cost(l) \\
&\quad + length(l) + \textstyle\sum_{x \in xs} cost(p\ x) + \sum_{\substack{x \in ys \\ x \notin xs \\ p\ x = \top}} cost(Cons) + length(xs) \\
cost(filters4\ p\ q\ xs) &= cost(p) + cost(q) + cost(xs) + \textstyle\sum_{x \in xs} cost(p\ x) \\
&\quad + \textstyle\sum_{\substack{x \in xs \\ p\ x = \top}} (cost(Cons) + cost(q\ x)) + \sum_{\substack{x \in xs \\ q\ x = \top}} cost(Cons)
\end{aligned}
$$

Fig. 10: Costs of the predefined higher order functions

Example 7 (Proof of Map R1). As an example, we show that, using the cost model defined in Figures 9 and 10, the rule (Map R1) is a sound consolidation.

Consider the inference rule for (Map R1):

$$\frac{\Psi\{xs : [Int],\ ys : [Int],\ f : Int \to Int\} \vdash xs \sqsubseteq_{pre[Int]} ys}{(map\ f\ xs, map\ f\ ys) \triangleright \otimes(\lambda z.\ take\ (length\ xs)\ z)\ id\ (map\ f\ ys)}$$

For the sake of presentation, we assume that the inputs have been already evaluated an do not contribute to the costs, this is

$$cost(xs) = cost(f) = cost(ys) = 0$$

First, we compute the cost of the left hand side of $\triangleright$:

$$\frac{\dfrac{eval(map\ f\ xs) = (v_1, k)}{E, (map\ f\ xs)\ \Downarrow_k\ v_1}\ (\text{Eval}) \quad \dfrac{eval(map\ f\ ys) = (v_2, l)}{E, (map\ f\ ys)\ \Downarrow_l\ v_2}\ (\text{Eval}) \quad cost((,)) = 0}{(map\ f\ xs, map\ f\ ys)\ \Downarrow_{k+l}\ (v_1, v_2)}\ (\text{Data})$$

Then, k and l are computed using the costs defined in Figure 10. So, we get that

$$cost(lhs) = k + l = length(xs) + \sum_{x \in xs} cost(f\ x) + length(ys) + \sum_{y \in ys} cost(f\ y)$$

Now, we compute the cost of the right hand side of $\triangleright$:

$$\frac{\dfrac{\dfrac{E, (length\ xs)\ \Downarrow_0\ n}{\lambda z.\ take\ (length\ xs)\ z\ \Downarrow_0\ \lambda z.\ take\ n\ z}\ (\text{App})}{\otimes\ ((\lambda z.\ take(length\ xs) z)\ id\ (map\ f\ ys)\ \Downarrow_0\ \lambda g\ l.\ ((\lambda z.(take\ n\ z))\ l, g\ l)\ id\ (map\ f\ ys)}\ (\text{App})}{\dfrac{E, (\lambda z.(take\ n\ z)\ l)\ \Downarrow_0\ (take\ n\ l) \quad E, id\ \Downarrow_0\ id}{\lambda g\ l.\ ((\lambda z.(take\ n\ z))\ l, g\ l)\ id\ (map\ f\ ys)\ \Downarrow_0\ (\lambda l.\ (take\ n\ l, id\ l))\ (map\ f\ ys)}\ (\text{App,Lambda,App})}$$

$$\frac{\dfrac{eval(map\ f\ ys) = (v_2, k_1)}{E, (map\ f\ ys)\ \Downarrow_{k_1}\ v_2}\ (\text{Eval})}{(\lambda l.\ (take\ n\ l, g\ l))\ (map\ f\ ys)\ \Downarrow_{k_1}\ (take\ n\ v_2, id\ v_2)}\ (\text{App})$$

$$\frac{\dfrac{eval(take\ n\ v_2) = (v_1, c)}{take\ n\ v_2\ \Downarrow_c\ v_1}\ (\text{Eval}) \qquad \dfrac{\dfrac{E, v_2\ \Downarrow_0\ v_2}{E, id\ v_2\ \Downarrow_0\ v_2}\ (\text{App})}{(take\ n\ v_2, id\ v_2)\ \Downarrow_c\ (v_1, v_2)}\ (\text{Data})$$

Finally, using the predefined costs for *map* and *take* and noticing that $n = length(xs)$, we get

$$cost(rhs) = k_1 + c = length(ys) + \sum_{y \in ys} cost(f\ y) + length(xs)$$

Therefore, it is straightforward to see that, as expected, $cost(lhs) > cost(rhs)$.

In order to see that both sides yield the same results, we check that both elements of the resulting tuple yield the same type (we just show the proof trees for the last stages of the derivation):

$$\vdots$$

$$\dfrac{\dfrac{\Gamma \vdash map : (Int \to Int) \to [Int] \to Int \quad \Gamma \vdash f : Int \to Int}{\Gamma \vdash map\ f : [Int] \to [Int] \quad \Gamma \vdash xs : [Int]}\ \text{(T-App)}}{\Gamma \vdash map\ f\ xs : [Int]}\ \text{(T-App)}$$

$$\dfrac{\dfrac{\Gamma \vdash take : Int \to [Int] \to [Int] \quad \Gamma \vdash n : Int}{\Gamma \vdash take\ n : [Int] \to [Int] \quad \Gamma \vdash v_2 : [Int]}\ \text{(T-App)}}{\Gamma \vdash take\ n\ v_2 : [Int]}\ \text{(T-App)}$$

$$\dfrac{\dfrac{\Gamma \vdash map : (Int \to Int) \to [Int] \to Int \quad \Gamma \vdash g : Int \to Int}{\Gamma \vdash map\ g : [Int] \to [Int] \quad \Gamma \vdash ys : [Int]}\ \text{(T-App)}}{\Gamma \vdash map\ g\ ys : [Int]}\ \text{(T-App)}$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash map : (Int \to Int) \to [Int] \to Int \quad \Gamma \vdash g : Int \to Int}{\Gamma \vdash map\ g : [Int] \to [Int] \quad \Gamma \vdash ys : [Int]}\ \text{(T-App)}}{\Gamma \vdash map\ g\ ys : [Int] \quad \Gamma \vdash id_{[Int]-[Int]} : [Int] \to [Int]}\ \text{(T-App)}}{\Gamma \vdash id\ map\ g\ ys : [Int]}\ \text{(T-App)}$$

$\triangleleft$

In Example 7, the only rules that contribute to the overall costs of the expressions are the ones in which we force the evaluation of the higher order functions. Then, instead of providing the full reduction trees for the rest of the rules, we reason in terms of the costs of the higher order functions.

For the remaining proofs we assume that the inputs to the functions have already been evaluated so they do not contribute to the cost in the proofs. Also, the costs are as defined in Figure 10.

Proof (Map R2).
We show that

$$cost(map\ f\ xs) + cost(map\ g\ xs) = cost(maps\ f\ g\ xs)$$

From the predefined costs we get:

$$\begin{aligned} cost(lhs) &= cost(map\ f\ xs) + cost(map\ g\ ys) \\ & \quad length(xs) + \textstyle\sum_{x \in xs} cost(f\ x) + length(xs) + \sum_{x \in xs} cost(g\ x) \\ & \quad 2length(xs) + \textstyle\sum_{x \in xs}(cost(f\ x) + cost(g\ x)) \end{aligned}$$

$$cost(rhs) = 2length(xs) + \textstyle\sum_{x \in xs}(cost(f\ x) + cost(g\ x))$$

We can see at once that $cost(lhs) = cost(rhs)$ □

Proof (Filter R1).
We show that

$$cost(filter\ p\ xs) + cost(filter\ (\neg\ p) xs) \geq cost(partition\ p\ xs)$$

From the predefined costs we get:

$$\begin{aligned} cost(lhs) &= \textstyle\sum_{x \in x} cost(p\ x) + \sum_{\substack{p\ x=\top}}^{x \in xs} cost(Cons) \\ & \quad + \textstyle\sum_{x \in x} cost(\neg\ p\ x) + \sum_{\substack{p\ x=\bot}}^{x \in xs} cost(Cons) \end{aligned}$$

$$cost(rhs) = \textstyle\sum_{x \in xs}(cost(p\ x) + cost(Cons))$$

Comparing the above, it is clear that $cost(lhs) \geq cost(rhs)$. □

Proof (Filter R2).
We show that $cost(filter\ p\ xs) + cost(filter\ q\ xs) \geq cost(filters2\ p\ q\ xs)$

$$\begin{aligned} cost(lhs) &= \textstyle\sum_{x \in x} cost(p\ x) + \sum_{\substack{p\ x=\top}}^{x \in xs} cost(Cons) \\ & \quad + \textstyle\sum_{x \in xs} cost(q\ x) + \sum_{\substack{q\ x=\top}}^{x \in xs} cost(Cons) \end{aligned}$$

$$\begin{aligned} cost(rhs) &= \textstyle\sum_{x \in xs} cost(p\ x) + \sum_{\substack{p\ x=\top}}^{x \in xs} 2cost(Cons) \\ & \quad + \textstyle\sum_{\substack{p\ x=\bot}}^{x \in xs} cost(q\ x) + \sum_{\substack{q\ x=\top \\ p\ x=\bot}}^{x \in xs} cost(Cons) \end{aligned}$$

Now, equating both sides, and simplifying, we get:

$$\textstyle\sum_{x \in xs} cost(q\ x) + \sum_{\substack{q\ x=\top}}^{x \in xs} cost(Cons)$$

$$\geq$$

$$\textstyle\sum_{\substack{p\ x=\top}}^{x \in xs} cost(Cons) + \sum_{\substack{p\ x=\bot}}^{x \in xs} cost(q\ x) + \sum_{\substack{q\ x=\top \\ p\ x=\bot}}^{x \in xs} cost(Cons)$$

Finally, simplyfing similar terms we get:

$$\sum_{\substack{x \in xs \\ p\ x=\top}} cost(q\ x) + \sum_{\substack{x \in xs \\ q\ x=\top \\ p\ x=\top}} cost(Cons) \geq \sum_{\substack{x \in xs \\ p\ x=\top}} cost(Cons)$$

Therefore, (Filter R2) is a sound consolidation. □

Proof (Filter R3).
We show that

$$cost(filter\ p\ xs) + cost(filter\ p\ ys) > cost(filters3\ p\ xs\ ys\ (filter\ p\ xs))$$

From the predefined costs we get:

$$\begin{aligned} cost(lhs) &= \sum_{\substack{p\ x=\top \\ x \in xs}} cost(Cons) + \sum_{x \in xs} cost(p\ x) \\ &\quad + \sum_{\substack{p\ y=\top \\ y \in ys}} cost(Cons) + \sum_{y \in ys} cost(p\ y) \end{aligned}$$

$$\begin{aligned} cost(rhs) &= \sum_{\substack{p\ x=\top \\ x \in xs}} cost(Cons) + \sum_{x \in xs} cost(p\ x) + length(l) \\ &\quad + \sum_{x \in ys \notin xs} cost(p\ x) + \sum_{\substack{x \in ys \\ x \notin xs \\ p\ x=\top}} cost(Cons) + length(xs) \end{aligned}$$

Where $l = filter\ p\ xs$. Now, simplyfing terms:

$$\sum_{\substack{p\ y=\top \\ y \in ys}} cost(Cons) + \sum_{y \in ys} cost(p\ y) \geq \sum_{x \in ys \notin xs} cost(p\ x) + \sum_{\substack{x \in ys \\ x \notin xs \\ p\ x=\top}} + length(l) + lengths(xs)$$

$$\sum_{\substack{p\ y=\top \\ y \in ys}} cost(Cons) + \sum_{x \in xs} cost(p\ x) \geq \sum_{\substack{x \in ys \\ x \notin xs \\ p\ x=\top}} cost(Cons) + length(l) + length(xs)$$

$$\sum_{\substack{x \in xs \\ p\ x=\top}} cost(Cons) + \sum_{x \in xs} cost(p\ x) \geq length(l) + length(xs)$$

Finally, note that $length(l) = \sum_{\substack{x \in xs \\ p\ x=\top}} cost(Cons)$, then

$$\sum_{x \in xs} cost(p\ x) \geq length(xs)$$

Therefore, (Filter R3) is a sound consolidation. □

Proof (Filter R4). We show that

$$cost(filter\ p\ xs) + cost(filter\ q\ l) = cost(filters4\ p\ q\ xs)$$

Where $l = filter\ p\ xs$.

Using the predefined costs we get:

$$\begin{aligned} cost(lhs) &= \textstyle\sum_{x \in xs} cost(p\ x) + \sum_{\substack{x \in xs \\ p\ x = \top}} cost(Cons) + \sum_{\substack{x \in xs \\ p\ x = \top}} cost(q\ x) \\ &\quad + \textstyle\sum_{\substack{x \in xs \\ p\ x = \top \\ q\ x = \top}} cost(Cons) \end{aligned}$$

$$\begin{aligned} cost(rhs) &= \textstyle\sum_{x \in xs} cost(p\ x) + \sum_{\substack{x \in xs \\ p\ x = \top}} cost(q\ x) + \sum_{\substack{x \in xs \\ q\ x = \top \\ p\ x = \top}} 2cost(Cons) \\ &\quad + \textstyle\sum_{\substack{x \in xs \\ p\ x = \top \\ q\ x = \bot}} cost(Cons) \end{aligned}$$

Simplifying equal terms we get:

$$\begin{aligned} &\textstyle\sum_{\substack{x \in xs \\ p\ x = \top}} cost(Cons) + \sum_{\substack{p\ x = \top \\ q\ x = \top}} cost(Cons) \\ &= \\ &\textstyle\sum_{\substack{x \in xs \\ q\ x = \top \\ p\ x = \top}} 2cost(Cons) + \sum_{\substack{x \in xs \\ p\ x = \top \\ q\ x = \bot}} cost(Cons) \end{aligned}$$

We end up with:

$$\sum_{\substack{x \in xs \\ p\ x = \top}} cost(Cons) = \sum_{\substack{x \in xs \\ q\ x = \top \\ p\ x = \top}} cost(Cons) + \sum_{\substack{x \in xs \\ p\ x = \top \\ q\ x = \bot}} cost(Cons)$$

Hence, (Filter R4) is a sound consolidation. □

Proof (Fold R1).

In this case we show that

$$cost(fold\ f\ z\ xs) + cost(fold\ f\ z\ ys) \geq cost(fold\ f\ (fold\ f\ z\ xs)\ (drop\ (length\ xs)\ ys))$$

Using the predefined costs we have that:

$$cost(lhs) = \textstyle\sum_{x \in xs} cost(f\ z\ x) + \sum_{y \in ys} cost(f\ z\ y)$$

$$cost(rhs) = \textstyle\sum_{x \in xs} cost(f\ z\ x) + length(xs) + \sum_{y \notin xs}^{y \in ys} cost(f\ z'\ y)$$

Now, notice that $z' = fold\ f\ z\ xs$, so for any $y \in ys$, $cost(f\ z'\ y)$ ultimately depends on $cost(f)$. Then we have $\forall y \in ys.\ cost(f\ z\ y) = cost(f\ z'\ y)$.

Therefore, simplifying, we have:

$$\sum_{x \in xs} cost(f\ z\ y) \geq length(xs)$$

So (Fold R1) is a sound consolidation. □

5 Experimental Evaluation

In this section, we describe the experimental setup used to test the consolidation rules (Figure 6) and present benchmarks for each rule. We end the section with a discussion of our findings.

5.1 Functions and Patterns

As a first step we analysed the current state of the Haskell ecosystem to filter out meaningful programs on which to test the consolidation rules defined in Section 4.

To achieve this, we surveyed Hackage[1], the main repository of Haskell packages and libraries. We processed 9581 packages (95.82% of Hackage at the time of writing) and sorted them by popularity. We used, as metrics, the number of dependencies (Figure 11) and number of downloads per package (Figure 12). We also extracted the most used data structures (Figure 13) and the most used functions (Figure 14). In the figures we only show the top 25 most used packages, modules and functions. A more complete set of results can be found online [30].

Overall, these results show that the lists are the most used data structure in Haskell programs and that the `filter` and `map` functions are amongst the 100 most used function while `foldr` is the 145th most used function. Next, we

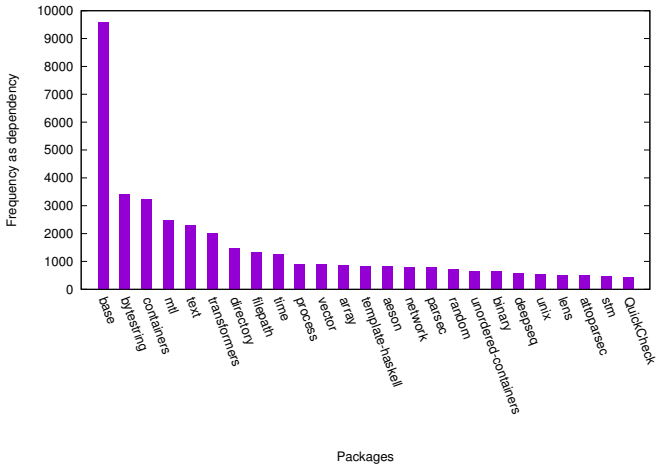

Fig. 11: Most used Haskell packages. Top 25.

turned to GitHub, a popular site where developers host, share and collaborate in software projects, to look for common programming patterns used in Haskell programs. We sorted the GitHub projects by their number of *stars* (stars count

[1] More information at http://hackage.haskell.org/

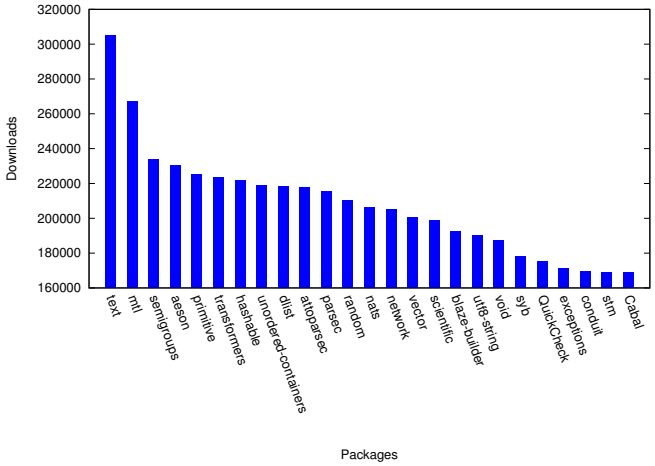

Fig. 12: Most downloaded Haskell packages. Top 25.

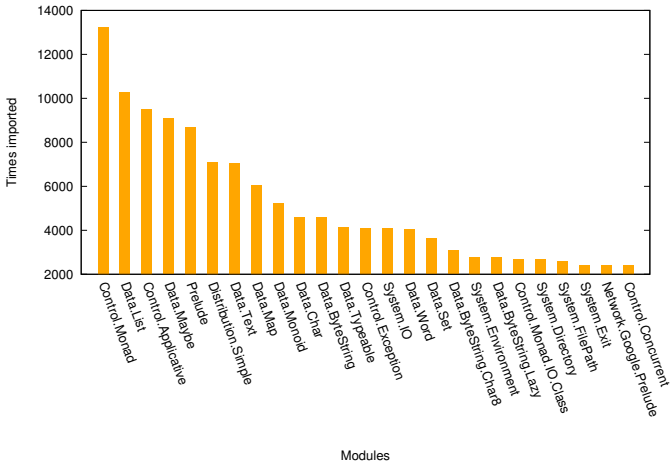

Fig. 13: Most imported Haskell modules. Top 25.

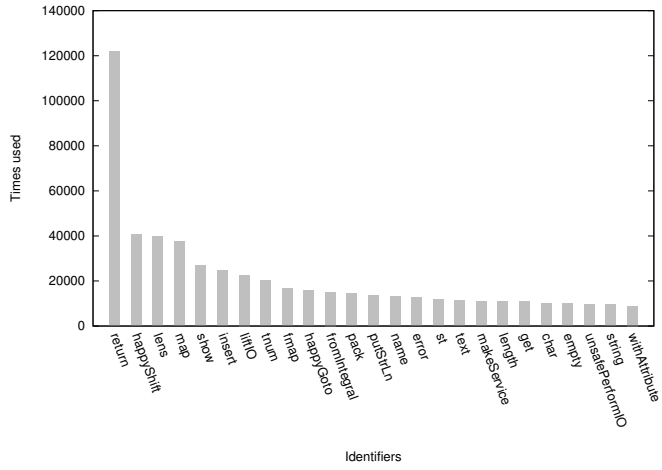

Fig. 14: Most called Haskell functions. Top 25.

how many times a project has been marked as favourite by a user) and manually extracted the common patterns involving `filter`, `partition`, `map` and `foldr`.

We found that the most common pattern comes in the form of two applications of `filter` to the same list, usually with different predicates. This suggest, that if the consolidation rules can be efficiently implemented, they should have a significant impact and would optimise a meaningful set of programs.

5.2 Implementation

We implemented the rules as simple Haskell programs and checked the preconditions manually before running each benchmark.

At this stage we were not able to automate the process of checking the preconditions. In some cases our rules only involve checking assertions such as $xs \sqsubseteq_{pre[a]} ys$ where $\sqsubseteq_{pre[a]}$ is a prefix relation over xs and ys. While this could be easily automated, it is less clear how to automate the verification of complex predicates such as the precondition for rule (Filter R3). However, recent developments such as the automatic verification of refinement types [36] and the translation of Haskell programs to first order logic [38], might provide a way for us to automate this process. We discuss these ideas in more detail in Section 7.

In order to mimic the call-by-value semantics expected by the rules, we used the language extension `{-# LANGUAGE BangPatterns #-}`, that enables us to make expressions strict by annotating them with a "bang" (`!`).

Example 8. As an example of the rules' implementation, consider following program that we used to benchmark the rule (Filter R1).

```
{-# LANGUAGE BangPatterns #-}
filter1 p vs = let !l1 = filter p vs
                   !l2 = filter (not ∘ p) vs
               in (l1,l2)

filter1_opt p vs = let !(l1,l2) = partition p vs
                    in (l1,l2)
```

◁

The source code for all the benchmarks, and instructions for how to execute them, can be found in the project's repository [30].

We also experimented with the current version of GHC's rewrite rules in order to automate the consolidation process but, without an automatic checker for the preconditions, this was not a good approach. Moreover we ran into some issues with inlining [26]. We hope to address these concerns in the near future.

A Problem with Tupling. In Section 2 we came across a surprising phenomenon: in certain cases, it appears that doing two traversals over the data structure is faster than traversing it only once (Figure 1). As we noted, this is not an isolated result but a wide ranging problem in GHC and has to do with the way that tuples are implemented and handled in the language. The problem is supposed to be solved by Wadler's "GC trick" [39] but this trick does not play nicely with inlining in the compiler, and in fact, there exist, a yet unresolved, bug[2] which might impede the efficiency gains that we promise.

In an attempt to sidestep this problem, at least with regards to the benchmarks, we implemented our experiments using `vectors`, a popular and efficient implementation of lists that, in most cases, does not suffer from these and other garbage collecting problems. Furthermore, `vectors` is the 11th most popular package by number of dependencies and the 15th by number of downloads, making it a good candidate to simulate the consolidation of real world programs.

5.3 Experiments

For the tests, our setup is the same as the one used in the examples presented in Section 2: we ran GHC 7.10.3 on an standard OS X environment featuring an Intel i7 multicore and 16GB of RAM.

We ran the benchmarks varying the predicates used, the size of the lists, and generated elements both in sequence and using a random number generator. To give a full assessment of the consolidation rules, we measured both speed and memory usage of all the programs. We measured speed using Criterion[3], a popular Haskell benchmarking library. We used Criterion's `nf` function in order

[2] GC and inlining bug still open: https://ghc.haskell.org/trac/ghc/ticket/2607.
[3] More information about Criterion at http://www.serpentine.com/criterion/

to force the evaluation of all the expressions to normal form, and avoid any wrong measurements that could arise due to the laziness of the language. Moreover, we set criterion to perform 1000 iterations in each case, so every result that we report here corresponds to the mean of 1000 runs. To record memory usage of the programs, we used Weigh,[4] a library inspired by Criterion that provides a small DSL to analyse the memory consumption of Haskell programs.

Maps. For rule (Map R1) we ran two tests. The first test maps a function that cubes each element of the list, while the second one maps a function that increments by one each element. Figures 15 and 16 show these results. In both cases we varied the size of the vectors and the method of generating the elements. Notice that the lists generated at random do not satisfy the precondition $xs \sqsubseteq_{pre[Int]} ys$, we only provide these benchmarks for completeness.

Both tests show that, according to our theory, the rule is able to generate more efficient programs. The biggest improvements are seen with vectors generated in sequence (lines m1 sequence-equal and m1-opt sequence-equal, m1 sequence-different, m1-opt sequence different in the left plot of each figure), where, on average the consolidated programs run 1.68 faster than the original ones. Furthermore, the original programs use, on average, 28% more memory than the optimised ones.

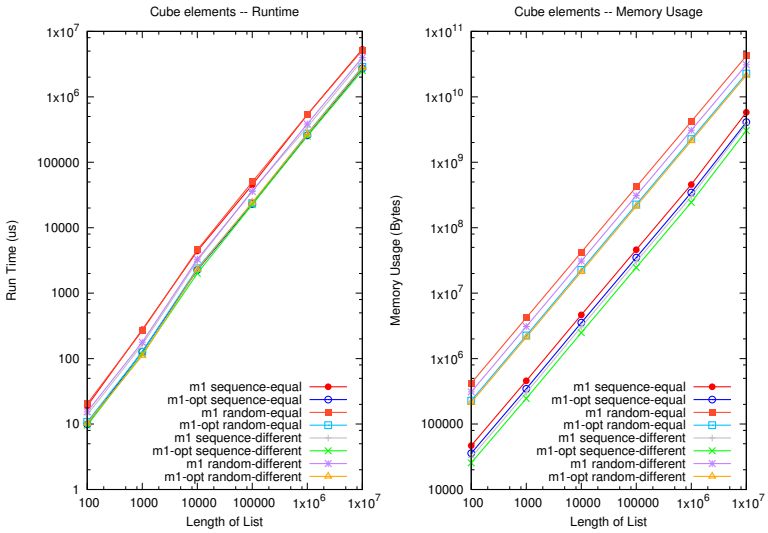

Fig. 15: Map Rule 1. Tested with a function that cubes all elements. We used vectors of the same length and vectors with different sizes (notice the log scale).

[4] More information about Weigh at https://hackage.haskell.org/package/weigh

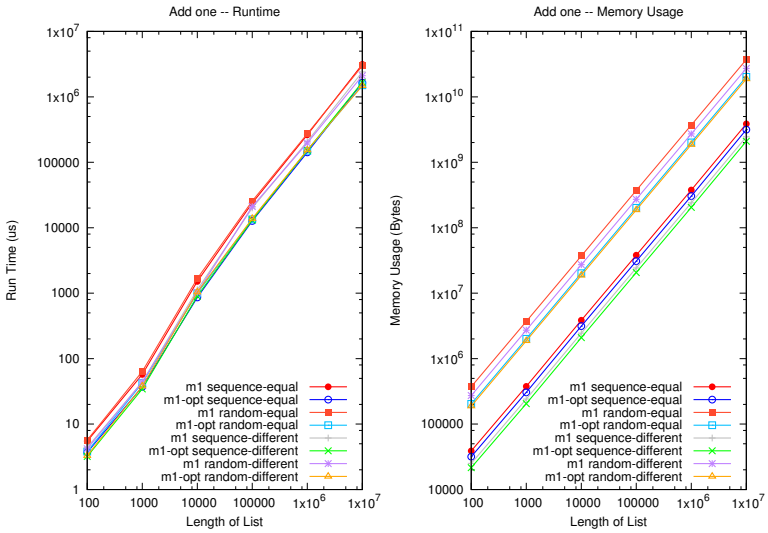

Fig. 16: Map Rule 1. Tested with a function that adds one to all elements. We used vectors of same length and vectors with different size (notice the log scale).

In the case of rule (Map R2), we experimented with simple integer comparison and modulus operations. In Section 4.4 we showed that, since this rule is not exploiting any deep semantic relations, the consolidated program will do the same amount of work as the original one. However, in the results, shown in Figure 17, we see that, in reality, the optimised program performs much worse. While this might suggest a problem with the theory, it is to note that the *Cons* operation takes $O(n)$ in the case of vectors since it involves a memory reallocation.

Filters. In the first example that we presented in Section 2, we already showed the results of a small set of benchmarks for the first rule of filters (Filter R1). In that case we used the standard list implementation and were victims of the "tupling effect". In Figure 18, we present the results of running the same tests but using vectors. Figure 18 (left) confirms our theory, and shows that, on average, the partition function is 1.9 times faster than using two filters. However, memory usage is not greatly reduced and in the case of using integer comparison with integers generated in sequence (lines fr1 sequence-split and fr1-opt sequence-split in Figure 18 (right)), the consolidated program uses around 10% more memory. Overall, this is not a bad outcome if we consider the gains in terms of runtime.

For the rule (Filter R2), we ran the tests using predicates that filter all the multiples of 4 and all the even elements, since these predicates satisfy the equation $\forall x.\ x\ mod\ 4 = 0 \Rightarrow even\ x$. We also used a variation of these predicates that multiply the given element by 100 before applying the predicate to it.

In this case, the situation is complicated. When running the benchmarks using vectors, we suffer from the same problem as in the benchmarks for (Map

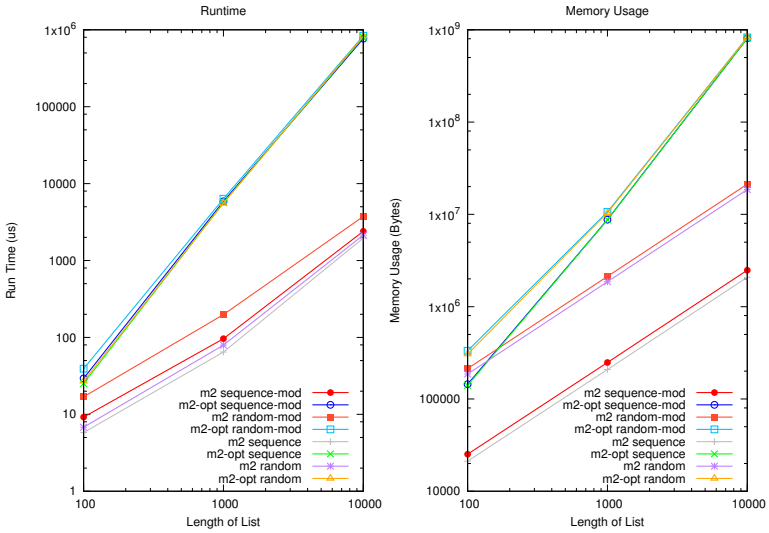

Fig. 17: Map Rule 2. Tested with integer comparison and mod operations (notice the log scale).

R2) (the *Cons* operation is $O(n)$ on the size of the vector). We can see this in Figure 19, where the runtime of the consolidated programs blows up fairly quickly, and memory consumption is around 30 times higher.

In an attempt to justify our theory, we switched back to plain Haskell lists where the *Cons* operator is O(1). The results in Figure 20 (left), show that when the lists have less than 1000 elements, the consolidated program is faster than the original program, but after that point, we are, once again, victims of the tupling effect. Nonetheless, in Figure 20 (right), we see that in certain cases the consolidated programs use less memory (e.g. lines f2-list mod and f2-list-opt mod). So, not all is lost.

For the rule (Filter R3), we varied the size of the vectors and varied the predicates. We used the `even` operator (Figure 21) and used a different `mod` operator together with multiplication to get a more complex predicate (Figure 22). We generated the elements in sequence and using a random number generator. Once again, note that the randomly generated vectors do not necessarily satisfy the precondition $xs \sqsubseteq_{pre[Int]} ys$, we only provide these benchmarks for comparison.

In all the cases, the consolidated programs are faster. The average speedup is of 1.6x, and the optimised programs consume, on average, 7% less memory.

Finally, for the rule (Filter R4), we ran the experiments with simple mod operations and more complex ones that involve also multiplication. We also compared between tupling and summing the results of the filters (Figures 23 and 24 respectively). This time both versions run almost identically and the average speedup is of only 2.4%. However, in a few cases, the consolidated version

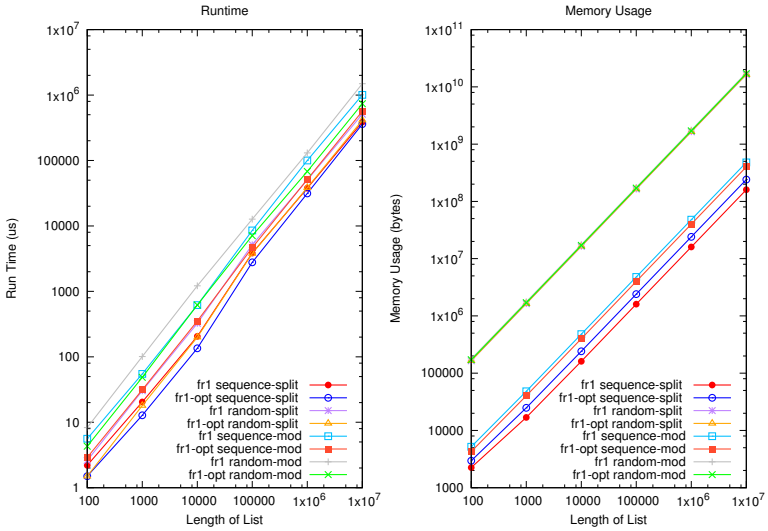

Fig. 18: Filter Rule 1. Tested with integer comparisons that split the list in half and mod operations on vectors with randomly generated elements (notice the log scale).

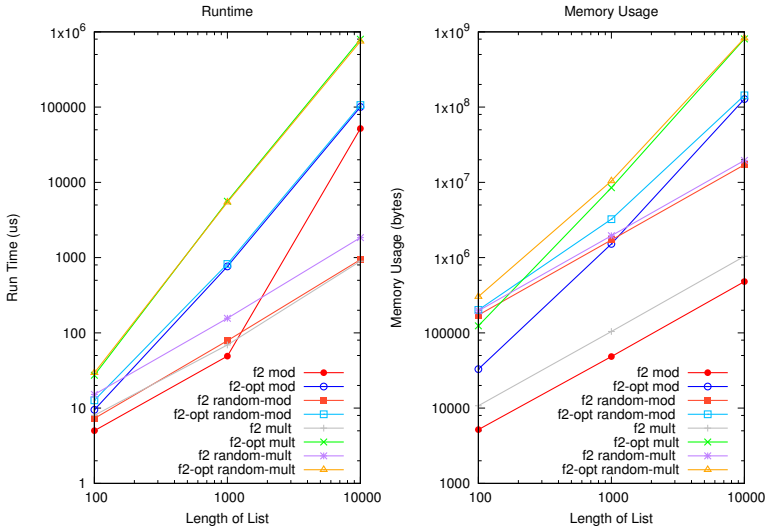

Fig. 19: Filter Rule 2 – Vectors. Tested with mod operations and with mod operations that multiply by 100 each element before applying the filter. Test with vectors of size $100, 1000$ and 10000 only. (notice the log scale).

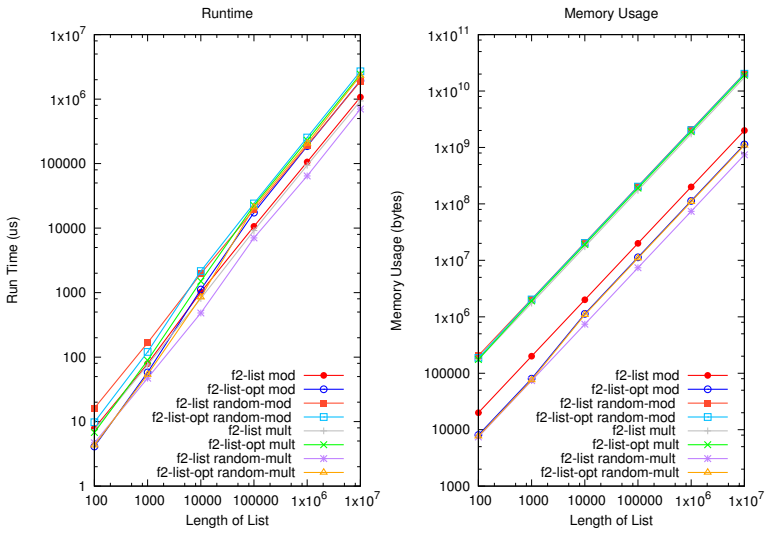

Fig. 20: Filter Rule 2 – Lists. Tested with mod operations and with mod operations that multiply by 100 each element before applying the filter. (notice the log scale).

is around 1% slower and doesn't show a reduction in memory consumption (compare for example the lines fr4 random and fr4-opt random in Figure 24(left)).

Folds. For the rule (Fold R1), we ran two different tests. One, shown in Figure 25 where simply add up all the elements of the vectors and the other, Figure 26 where we double each element before adding them up. As usual, we varied the size of the vectors and generated the elements both in sequence and using a random number generator. Note that as in the tests for rules (Map R1) and (Filter R3), the vectors with random elements do not necessarily satisfy the rule's precondition, we provide these cases only for completeness.

The results clearly show the benefits of using consolidation. The average speedup is of 1.82x and the average memory usage is 26% lower compared to the original programs.

5.4 Summary

The benchmarks, presented in this section, show that our rules give considerable efficiency improvements. Overall, the biggest gains are seen with the rules (Filter R1) and (Fold R1) where we see speedups of almost 2x and reduce memory consumption by more than 25%. This provides encouraging evidence in favour of our rules, given that both $fold$ and $filter$ are very common functions in functional programs.

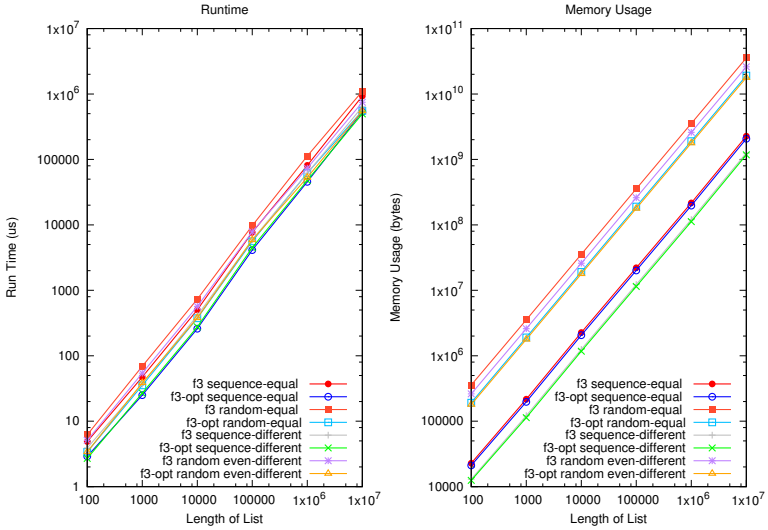

Fig. 21: Filter Rule 3 – Simple. Tested with a predicate that filters all even numbers. Varied the size of the vectors (notice the log scale).

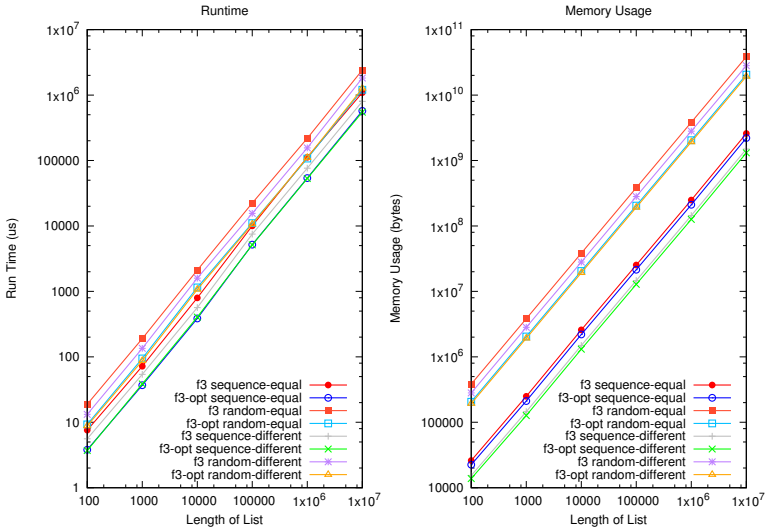

Fig. 22: Filter Rule 3 – Complex. Tested with predicates involving multiplication and mod operations. Varied size of vectors (notice the log scale).

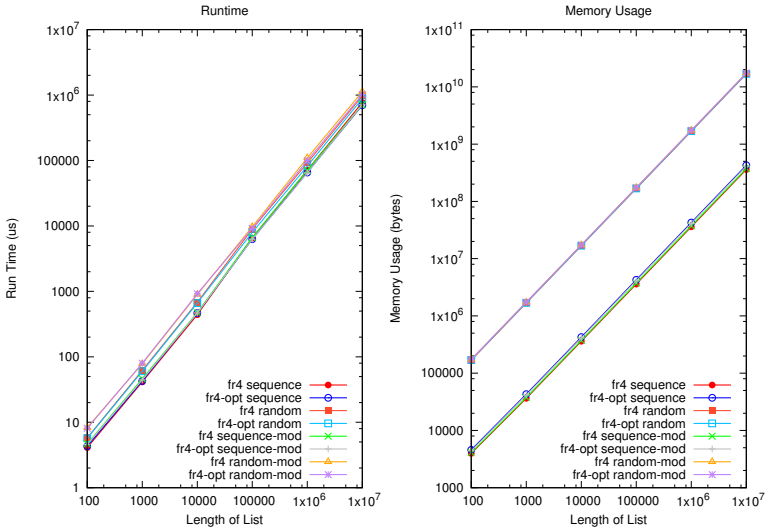

Fig. 23: Filter Rule 4 – Tuples. Tested with simple integer comparison and mod operations with exponentiation. Measuring the tupling of the resulting vectors (notice the log scales).

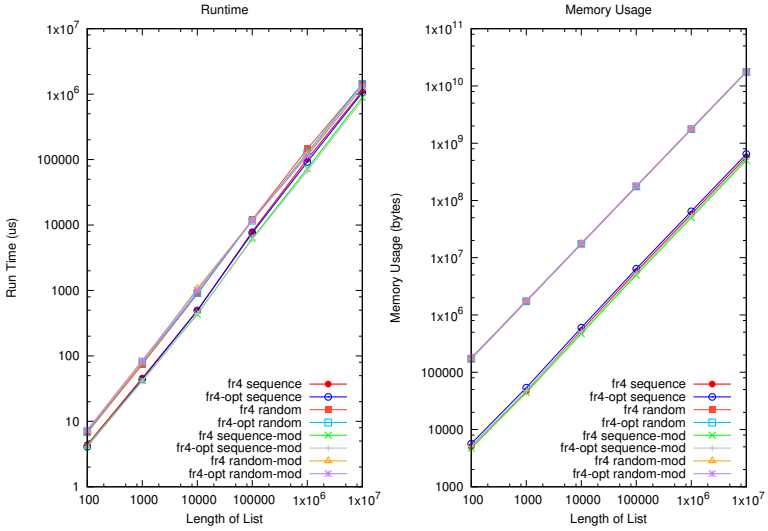

Fig. 24: Filter Rule 4 – Sum. Tested with simple integer comparison and mod operations with exponentiation. Measuring the sum of the resulting vectors (notice the log scales).

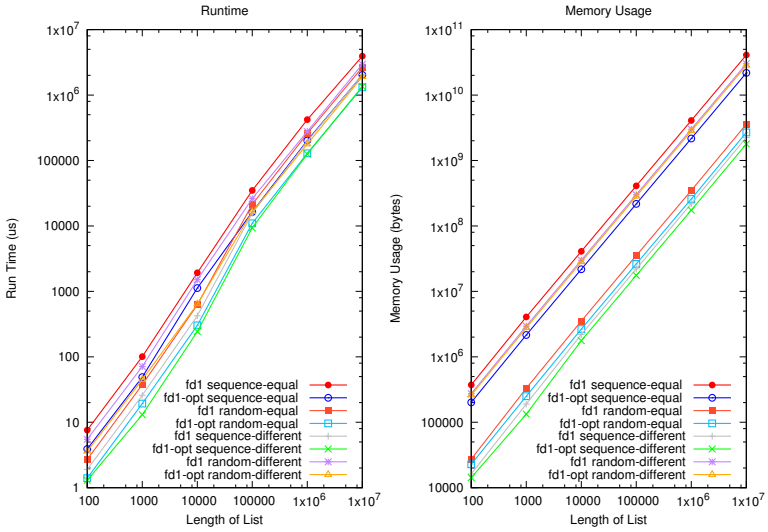

Fig. 25: Fold Rule 1. Tested with a fold that adds all the elements of the vector. Varied the size of the vectors. Used integers generated in sequence and using a random generator (notice the log scales).

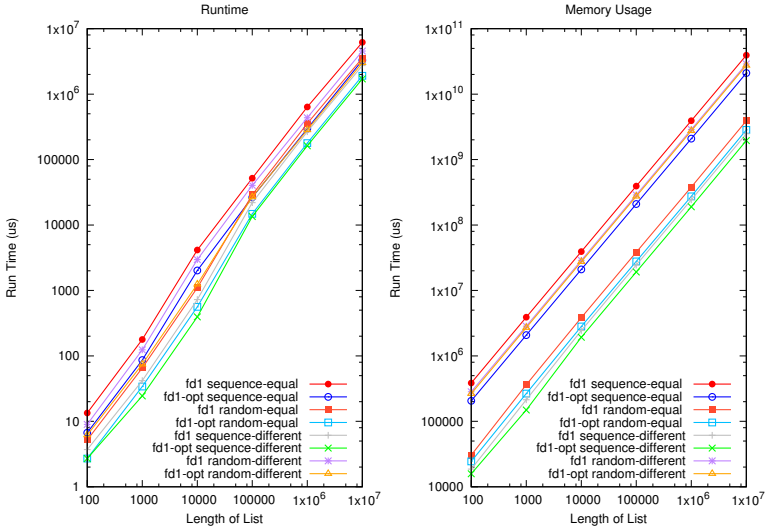

Fig. 26: Fold Rule 1. Tested with a fold multiples each element of the vector by 2 and then adds them up. Varied the size of the vectors. Used integers generated in sequence and using a random generator (notice the log scales).

Finally, an interesting phenomenon to note is that the biggest gains (such as the cases of (Filter R1) and (Fold R1) above) are seen with rules that manage to exploit a lot of semantic information. In fact, the rules that only use a minimal amount of semantic information, such as (Filter R4) and (Map R2), are able to achieve, at best, minor gains.

6 Threats to Validity

Threats to validity are "factors beyond our control that can affect the dependent variables" [2] of our experiments and research. In this section, we discuss two types of threats that could invalidate our theory and experiments. First, we analyse threats to *external validity*, an indicator of the generalisability of our theory. Then we describe potential *internal* threats to validity. These are threats that arise due to biases in the execution and interpretation of experiments.

6.1 External Validity

Data collection. Our work focuses on a small set of common programming patterns. These patterns were extracted by surveying Haskell libraries and by manually investigating popular projects on GitHub (see Section 5.1).

This data collection process poses a threat to our work. All the projects that we surveyed are open-source and community driven. Therefore, it is not clear if these projects can be considered a representative sample of all Haskell programs. In fact, we cannot measure how many close-source projects exist and we cannot test if the patterns that we are interested in are also common in commercial and/or close-source Haskell programs.

We have mitigated this threat by focusing on a set of structured recursion patterns (i.e., maps, filters, folds) and data structures (e.g., lists and vectors) that are core to Haskell and are, therefore, likely to appear in most projects, regardless of their status or use.

Scope of consolidation rules. Another threat to the possibilities of generalising our work stems from the set of proposed rules (Section 3). These rules can seem to have been created arbitrarily, and in fact, they attempt to only optimise a set of patterns that we found during our surveys. After seeing the potential threats with respect to the data collection, this means that our proposed rules might not be applicable in many cases.

This risk is mitigated by the fact that the theory and language that we proposed (see Section 4) is very simple. Then, following the trend of Haskell rewrite rules [19], we believe that our theory enables for the creation of new rules, on an ad hoc basis, with a minimal amount of effort.

6.2 Internal Validity

Cost model. A potential threat, that might render our theory invalid, are the arbitrary costs used in the soundness proofs (see Section 4.4). The cost model used

might seem too simplistic or even unrealistic in certain cases. For example, we assumed that the cost of tupling is zero and creating values is zero, assumptions that might not be realistic in all scenarios.

However, the cost semantics used is flexible enough that it does not inherently depend on our assumptions. In fact, the soundness proofs can be derived without our cost model or even using a different one.

Experiments. The biggest threat to the validity of our work relies on the experimental evaluation. We cannot ensure that the selected benchmarks are without errors or that they represent a meaningful sample that enables us to draw correct conclusions.

In this case, we minimised the risks of obtaining and interpreting wrong results by using an extensive benchmarking library commonly used to test Haskell programs. Moreover, all the benchmarks were generated with respect to three parameters that we varied in order to get a meaningful conclusion: (i) we used input functions with different complexities to reduce the noise produced by the garbage collector, (ii) we varied the distribution of the elements in the vectors and lists, by producing the elements in sequence and using a random number generator, and (iii) we varied the size of the input vectors and lists.

Finally, these risks are not completely eliminated. In order to be even more confident about our results, we should run the benchmarks in an ever bigger set of programs and industrial applications.

7 Related Work

Our work is based on program consolidation, a wealth of related topics is presented in the related work section of [33]. In what follows we focus instead only on related optimisations developed for functional languages, a large and wide ranging topic as well. In particular, we describe relations to well known, and widely studied, program transformations such as deforestation, tupling and common sub-expression elimination. We end with a description of recent developments with respect to the improvement theory for call-by-need [23] and how our work relates to it.

7.1 Deforestation

Deforestation is one of the most important optimisations in functional languages and has been widely studied since its introduction. The original work [40] describes an algorithm that is able to eliminate intermediate lists and trees in composition of functions that are written in a "treeless form".

In GHC, full-blown deforestation is not implemented. Instead there exist a variety of program transformations that exploit and extend Wadler's ideas. The basic case of deforestation in GHC is "short-cut fusion" [12,35]. This transformation is not as general as deforestation, and focuses on the deforestation of

Haskell lists. The main idea behind this technique exploits the so called "foldr/build" pattern, by which, a list-processing function is transformed into a consumer (fold) and an explicit builder of the list. Short-cut fusion is not able to eliminate intermediate lists in all list-consuming functions. For example, it is not able to deforest `foldr1` (`foldr` for non empty lists), since it does not "treat all the cons cells identically".

However, others have focused on extending this type of deforestation. Svenningsson [34] describes an approach that is able to deforest "zip-like" functions and functions with accumulating parameters (such as `foldr1`) by rewriting the list processing functions using the "destroy/unfoldr" pattern, which, opposed to short-cut fusion, manipulates lists using the lists's *co-structure*.

Another interesting development [37], describes a general methodology to eliminate not only intermediate lists but more general data structures in applications of the functions *map*, *concatenate* and *reverse*.

More recently, in an attempt to implement a more efficient representation of Strings in Haskell [9], Coutts et al. developed "stream fusion" [8], an approach that, similar to [34], uses the coalgebraic representation of lists (i.e. *streams*) to fuse a wider range of list-processing functions.

While our work also eliminates certain traversals over the consumed data structures, it does so from a different perspective. In fact, since we focus on a small set of common recursion schemes, in many cases we are able to benefit from deforestation and are also able to eliminate certain redundant traversals that deforestation and its variations fail to eliminate.

7.2 Tupling

Tupling is a very common optimisation strategy that enables to eliminate redundant computations and/or traversals of common inputs across calls of recursive functions. Most of the developments in this area stem from the pioneering work of Burstall and Darlington [5]. They explore different ideas to optimise recursive programs and hint at a possible way of mechanising the generation of efficient recursive programs by exploring the *computation tree* generated by the recursive calls. Along similar lines, Bird [3], describes a series of "tabulation" techniques that, by traversing and manipulating the *dependency graph* of the recursive calls, is able to optimise recursive functions.

Pettorossi and Burstall [24] introduced the concept of "eureka tuple", a special tuple, also found by searching the dependency graph, that groups together all the calls that need the result of a common computation. Inspired by this work, Chi [6] develops an algorithm that is able to find more cases of eureka tuples and therefore is able to apply the "tupling transformation" to a wider set of recursive programs. Finally, in more recent developments, Hu et al. [17] develop a more powerful technique, based on "constructive algorithmics" [22], that eliminates the need for eureka tuples and plays nicely with other program transformations such as deforestation.

In our work, we use tupling only as a mean to eliminate multiple traversals over the input data structure. However, while the consolidation rules also fo-

cus on sharing information and eliminating redundant computations across the recursive calls, they do so differently. We do not store or "tabulate" the intermediate results, but are able to eliminate, at once, certain redundant computations by exploiting semantic dependencies between the inputs.

7.3 Common Sub-Expression Elimination

Similar to tupling, common sub-expression elimination is a transformation that focuses on reducing the runtime of programs by avoiding the repeated computation of the same expression. In the case of lazy functional languages, this optimisation is not very common. In fact, in certain cases it is even considered harmful, since the shared computation might be a big thunk which would, unnecessarily, be kept in memory. In spite of this fact, and the fact that, as Chitil [7] argues, common sub-expressions are not very common in lazy languages, this optimisation is enabled by default in GHC.

Some of the consolidation rules that we presented in this paper, such as (Map R1), (Filter R3) and (Fold R1), are similar to common sub-expression elimination in the sense that, by exploiting a semantic relation between the input lists, they eliminate repeated computations over the common prefix.

7.4 Partial Evaluation

The process of transforming the evaluation of a function that takes multiple parameters into the evaluation of a series of functions that each take a single parameter, is known as *currying*. Partial evaluation [18] is for programs what currying is for functions. It is a technique that, given a program and *some* of its input data, produces a semantically equivalent program specialised for the case of the given data. For example, if a program `p` consumes two parameters `in1` and `in2`, where only `in2` changes during successive calls, partial evaluation can generate a specialised program `p'` that only takes `in2` as input and where `in1` is "freezed". Even though this technique sounds powerful in principle, in reality it needs to use a variety of complex heuristics to determine what parts of a program can be partially evaluated or "specialised".

In the case of Haskell, GHC implements a restricted form of partial evaluation, "call-pattern specialisation" [25], that is able to specialise recursive functions based on the *shape* of its inputs. In a similar spirit to this approach, the consolidation rules, presented in this paper, are able to *specialise* certain programs and expressions, not by checking the shape of the arguments, but by extracting semantical information about them.

7.5 Refinement Types

Even though Haskell's strong typing and powerful type system enables to get rid of a wide range of bugs at compile time, the type system is able to express all but the coarsest set of invariants, so using type information to make strong safety and/or correctness assertions is not possible.

In contrast to this, there exist a vast literature on languages with dependent types [4,1,41], which are types whose definition might depend on a value, thus allowing for a tighter and more expressive type, suitable to prove more powerful invariants and preconditions. While dependent types are not available in Haskell, an exciting development [36], has brought Liquid types [28], to the language. These types combine refinement types, a restricted form of dependent types in which types are annotated with conjunctions of logical predicates, with "liquid" type inference, a system that is able to automatically infer "dependent types precise enough to prove a variety of safety properties".

Even though the consolidation rules that we present are not baked into the type system, it is not hard to see the similarities with refinement types. In fact, our inference rules contain type information for all the expressions in the given context and are annotated with a logic predicate that needs to be satisfied in order for the rule to be applicable. Perhaps, as suggested by Freeman and Pfenning [11], we might explore the use of refinement types for program optimisation by encoding our rules in a suitable way.

7.6 Relations to Improvement Theory

"The goal of program transformation is to improve efficiency while preserving meanining"[31]. Yet, when developing new optimisation techniques, most developers focus on giving correctness proofs (i.e., proofs that show that their proposed optimisation doesn't alter the semantics of the original program), and only give empirical evidence about efficiency improvements.

In order to address this issue, Moran and Sands developed the *Improvement Theory for Call-By-Need* [23]. This is a calculus, together with an *improvement theorem*, that, by counting the steps that a program takes into a suitable abstract machine for a call-by-need language, is able to formally reason about runtime improvements. Moreover, there exists also an analogous theory for space improvement [14].

Until recently, improvement theory was largely neglected. Evidence of its use can be found in [37], where the author conjectures, but does not prove, that the proposed transformation yields an improvement under Moran and Sand's theory.

In an attempt to promote the use of more rigorous methods to study efficiency improvements, Hackett an Hutton [15] applied Moran and Sands's ideas to the worker/wrapper transformation [13]. This transformation, widely studied, and implemented in GHC, attempts to split a computation into a *worker* that executes the original computation, under a potentially more efficient data structure, and a *wrapper*, that bridges the result between the old and new computations. Using Moran and Sand's original ideas, Hackett and Hutton were able to show, as expected by most, that the worker/wrapper transformation is indeed an improvement.

Inspired by this renewed interest in improvement theory, Schimdt-Shauß and Sabel [32] extended the improvement calculus to a language that includes a *seq* operator to mimic strict evaluation, and showed that common sub-expression elimination is indeed an improvement.

In the early stages of this work, we also experimented with the improvement calculus for call-by-need. In fact, we implemented Moran and Sand's abstract machine (the source code is available online [30]) to interpret a small subset of GHC Core, a functional language used internally by GHC, and tested some of the rules using this machine. In our case, a call-by-value model seemed more appropriate so we did not pursue this idea further. However, note that, the cost semantics that we developed in Section 4, enables to reason rigorously about efficiency improvements for the consolidation rules. We will revisit our work with respect to the improvement theory for call-by-need in the future.

8 Conclusion and Future Work

In this paper, we have motivated the applicability of program consolidation in a functional setting. We focused on a set of common programming patterns found in Haskell programs and proposed a new optimisation technique, *semantic fusion*, defined as set of consolidation rules (Section 3) that are able to optimise programs involving maps, filters and folds. In support of our claims, we ran a series of 12 benchmarks (Section 5). The results show that the rules can have a meaningful impact in Haskell programs, yielding speedups of up to 2x and are able to reduce memory usage as well.

Moreover, our work has crystallised some facts about the Haskell ecosystem. We surveyed more than 9000 Haskell packages and libraries (Section 5.1). This survey shows that lists are, by far, the most widely used data structure in Haskell. It also provides evidence in favour of the scope of our rules. Maps, filters and folds are amongst the most used functions in Haskell applications. We make available all the survey results online [30].

Overall, we believe that our work provides another step towards the development of strong optimisation frameworks, based on semantic relations, for functional languages. However, there are many ways in which our work could be enhanced or extended.

In Section 6 we discussed some of the main limitations and weaknesses of our approach. A first step towards developing a more robust consolidation framework for functional languages would be to develop a full consolidation calculus similar to the one used in the original theory [33]. Another possibility would be to extend our set of rules and develop a more robust theory using more general concepts from Category Theory [16,22].

Also, in Section 7, we discussed the relations of our work with GHC's rewrite rules, the recent developments in SMT technologies and refinement types. A natural next step, would be then to automate the process of checking and applying our rules using rewrite rules and SMT solvers, or develop a type system, similar to the one proposed in [36], not for verification, but for optimisation of programs.

Finally, other directions could explore the development of a survey engine to automate the availability of the latest statistics about the Haskell ecosystem. In this way, we believe, many developers would be able to better target their efforts when developing new ideas and optimisations.

References

1. Augustsson, Lennart. Cayenne — A Language with Dependent Types. In S Doaitse Swierstra, José N Oliveira, and Pedro R Henriques, editors, *Advanced Functional Programming: Third International School, AFP'98, Braga, Portugal, September 12-19, 1998, Revised Lectures*, pages 240–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.
2. Basili, Victor R. and Green, Scott and Laitenberger, Oliver and Lanubile, Filippo and Shull, Forrest and Sørumgård, Sivert and Zelkowitz, Marvin V. The empirical investigation of Perspective-Based Reading. *Empirical Software Engineering*, 1(2):133–164, 1996.
3. Bird, Richard S. Tabulation Techniques for Recursive Programs. *ACM Computing Surveys (CSUR)*, 12(4):403–417, December 1980.
4. Bove, Ana and Dybjer, Peter and Norell, Ulf. A Brief Overview of Agda – A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 73–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
5. Burstall, R M and Darlington, John. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, January 1977.
6. Chin, Wei-Ngan. Towards an Automated Tupling Strategy. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 119–132, New York, NY, USA, 1993. ACM.
7. Chitil, Olaf. Common Subexpressions Are Uncommon in Lazy Functional Languages. In *Selected Papers from the 9th International Workshop on Implementation of Functional Languages*, pages 53–71, London, UK, UK, 1998. Springer-Verlag.
8. Coutts, Duncan and Leshchinskiy, Roman and Stewart, Don. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, New York, NY, USA, 2007. ACM.
9. Coutts, Duncan and Stewart, Don and Leshchinskiy, Roman. Rewriting Haskell Strings. In *PADL'07: Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*. University of New South Wales, Springer-Verlag, January 2007.
10. de Moura, Leonardo and Bjørner, Nikolaj. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2008.
11. Freeman, Tim and Pfenning, Frank. Refinement Types for ML. *ACM SIGPLAN Notices*, 26(6):268–277, May 1991.
12. Gill, Andrew and Launchbury, John and Peyton Jones, Simon L. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.
13. Gill, Andy and Hutton, Graham. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.
14. Gustavsson, Jörgen and Sands, David. Possibilities and Limitations of Call-by-need Space Improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 265–276, New York, NY, USA, 2001. ACM.

15. Hackett, Jennifer and Hutton, Graham. Worker/Wrapper/Makes It/Faster. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 95–107, New York, NY, USA, 2014. ACM.
16. Hinze, Ralf and Wu, Nicolas and Gibbons, Jeremy. Conjugate Hylomorphisms – Or: The Mother of All Structured Recursion Schemes. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 527–538, New York, NY, USA, 2015. ACM.
17. Hu, Zhenjiang and Iwasaki, Hideya and Takeichi, Masato and Takano, Akihiko. Tupling Calculation Eliminates Multiple Data Traversals. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. Hitachi, Ltd., ACM, August 1997.
18. Jones, Neil D. and Gomard, Carsten K. and Sestoft, Peter. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
19. Jones, Simon Peyton and Tolmach, Andrew and Hoare, Tony. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *In Haskell Workshop*, pages 203–233. ACM SIGPLAN, 2001.
20. Knuth, Donald E. Structured Programming with Go to Statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.
21. Marlow, Simon and Jones, Simon L Peyton. The Glasgow Haskell Compiler. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications Vol.II*. AOSA Books, 2012.
22. Meijer, Erik and Fokkinga, Maarten and Paterson, Ross. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
23. Moran, Andrew and Sands, David. Improvement in a Lazy Context: An Operational Theory for Call-by-need. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 43–56, New York, NY, USA, 1999. ACM.
24. Pettorossi, A and Burstall, R M. Deriving very efficient algorithms For Evaluating Linear Recurrence Relations Using The Program Transformation Technique. *Acta Informatica*, 18(2):181–206, 1982.
25. Peyton Jones, Simon. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337, New York, NY, USA, 2007. ACM.
26. Peyton Jones, Simon and Marlow, Simon. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.
27. Peyton Jones, Simon L. and Santos, André L. M. A Transformation-based Optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3):3–47, September 1998.
28. Rondon, Patrick M. and Kawaguci, Ming and Jhala, Ranjit. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.
29. Sadde, Alberto. Benchmarking Filter and Partition. [Online] Available: http://stackoverflow.com/questions/38671397/benchmarking-filter-and-partition [Accessed 17 August 2016].
30. Sadde, Alberto. Consolidation of Haskell Programs: Source Code and Survey. [Online] Available: http://aesadde.xyz/Consolidation [Accessed 29 August 2016].

31. D Sands. Improvement Theory and Its Applications . In Andrew D Gordon and Andrew M Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 275–306. Cambridge University Press, New York, NY, USA, 1998.
32. Schmidt-Schauß, Manfred and Sabel, David. Improvements in a Functional Core Language with Call-by-need Operational Semantics. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 220–231, New York, NY, USA, 2015. ACM.
33. Sousa, Marcelo and Dillig, Isil and Vytiniotis, Dimitrios and Dillig, Thomas and Gkantsidis, Christos. Consolidation of Queries with User-Defined Functions. In *PLDI '14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Microsoft Research, ACM, June 2014.
34. Svenningsson, Josef. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.
35. Takano, Akihiko and Meijer, Erik. Shortcut Deforestation in Calculational Form. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*. Hitachi, Ltd., ACM, October 1995.
36. Vazou, Niki and Seidel, Eric L. and Jhala, Ranjit and Vytiniotis, Dimitrios and Peyton-Jones, Simon. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM.
37. Voigtländer, Janis. Concatenate, Reverse and Map Vanish for Free. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 14–25, New York, NY, USA, 2002. ACM.
38. Vytiniotis, Dimitrios and Peyton Jones, Simon and Claessen, Koen and Rosén, Dan. HALO: Haskell to Logic Through Denotational Semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 431–442, New York, NY, USA, 2013. ACM.
39. Wadler, Philip. Fixing Some Space Leaks with a Garbage Collector. *Softw. Pract. Exper.*, 17(9):595–608, September 1987.
40. Wadler, Philip. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.*, 73(2):231–248, January 1988.
41. Xi, Hongwei and Pfenning, Frank. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, New York, NY, USA, 1999. ACM.