

Lazier Imperative Programming

Rémi Douence, Nicolas Tabareau

▶ **To cite this version:**

Rémi Douence, Nicolas Tabareau. Lazier Imperative Programming. [Research Report] RR-8569, INRIA. 2014. hal-01025633v2

HAL Id: hal-01025633
https://inria.hal.science/hal-01025633v2

Submitted on 23 Jul 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

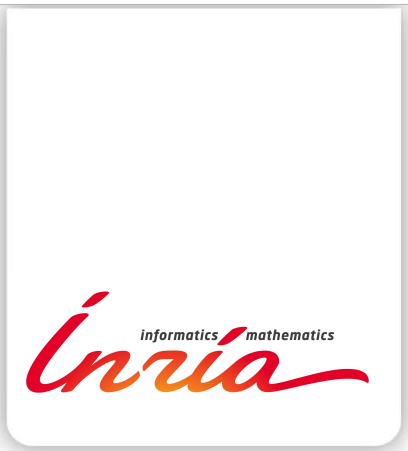


Lazier Imperative Programming

Rémi Douence, Nicolas Tabareau

RESEARCH REPORT
N° 8569
July 2014
Project-Teams Ascola

ISSN 0249-6399 ISRN INRIA/RR--8569--FR+ENG



Lazier Imperative Programming

Rémi Douence*, Nicolas Tabareau†

Project-Teams Ascola

Research Report n° 8569 — July 2014 — 34 pages

Abstract: Laziness is a powerful concept in functional programming that enables reusing general functions in a specific context, while keeping performance close to the efficiency of dedicated definitions. Lazy evaluation can be used in imperative programming too. Twenty years ago, John Launchbury was already advocating for lazy imperative programming, but the level of laziness of his framework remained limited: a single effect can trigger numerous delayed computations, even if those are not required for the correctness of the evaluation. Twenty years after, the picture has not changed. In this article, we propose an Haskell framework to specify computational effects of imperative programs as well as their dependencies. Our framework is based on the operational monad transformer which encapsulates an algebraic presentation of effectful operations. A lazy monad transformer is then in charge of delaying non-necessary computations by maintaining a trace of imperative closures. We present a semantics of a call-by-need λ-calculus extended with imperative strict and lazy features and prove the correctness of our approach. While originally motivated by a less rigid use of foreign functions, we show that our approach is fruitful for a simple scenario based on sorted mutable arrays. Furthermore, we can take advantage of equations between algebraic operations to dynamically optimize imperative computations composition.

Key-words: Lazyness, imperative programming, Haskell, monad transformer

* Mines de Nantes, France
† Inria, France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Programmation impérative paresseuse

Résumé : La paresse est un concept puissant en programmation fonctionnelle qui permet la réutilisation de fonctions générales dans un contexte spécifique, tout en conservant des performances proches de l'efficacité de définitions dédiées. L'évaluation paresseuse peut être utilisée dans la programmation impérativea aussi. Il y a vingt ans, John Launchbury préconisait déjà la programmation impérative paresseuse, mais le niveau de paresse de son cadre reste limité: un seul effet peut déclencher de nombreux calculs retardés, même si ces calculs ne sont pas nécessaires à l'exactitude de l'évaluation. Vingt ans après, la situation n'a pas changée. Dans cet article, nous proposons un cadre Haskell pour spécifier les effets des programmes impératifs ainsi que leurs dépendances. Notre cadre est basé sur le transformateur de monade opérationnelle qui encapsule une présentation algébrique des opérations à effets. Un transformateur de monade paresseux est alors en charge de retarder les calculs non nécessaires en maintenant une trace de fermetures impératives. Nous présentons une sémantique d'un λ-calcul en appel par besoin étendu avec des fonctionnalités impératives strictes et paresseuses et prouvons le bien-fondé de notre approche. Alors qu'à l'origine motivé par une utilisation moins rigide des fonctions étrangères, nous montrons que notre approche est fructueuse pour un scénario simple basé sur les tableaux mutables triés. En outre, nous pouvons profiter d'équations entre opérations algébriques afin d'optimiser dynamiquement la composition des calculs impératif.

Mots-clés : Paresse, programmation impérative, Haskell, transformateur de monades

Contents

1 Introduction **3**

2 Lazy Algebraic Operations **6**
2.1 A General Framework to describe commutation of algebraic operations 6
2.2 Lazier Imperative Arrays 8
2.3 Performance of lazy imperative arrays 10

3 Call-by-Need Calculus with Lazy Imperative Operations **11**
3.1 Call-by-Need Calculus with Strict Imperative Operations 12
3.2 Call-by-Need Calculus with Lazy Imperative Operations 14
3.3 Conservativity of Lazy Imperative Semantics 15

4 Lazy Evaluation of Algebraic Operations **19**
4.1 Heterogeneous Trace of Imperative Closures 19
4.2 Delaying computations 19
4.3 The Lazy Monad Transformer 20
4.4 Turning algebraic operations into monadic computations 21
4.5 Evaluation of Imperative Closures 21

5 Optimization **22**

6 Discussions **25**

A Code of Doubly Linked Lists **28**

B Code of the Lazy Monad Transformer **28**

1 Introduction

It has been advocated [Hughes, 1989] that:

> Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.

Laziness promotes code reuse. Functions are composed in a producer consumer scheme and a general producer can be used with different consumers. For each composition, the evaluation of the producer is driven by its consumer, hence only a subset of the producer result may be computed.

For instance, let us consider the function *minimum* that returns the lowest integer from a list. This function can be defined as the composition of a sorting function *bSort* with the function *head* that returns the first element of a list. In Haskell the corresponding definition is:

```
minimum l =
  let lOrdered = bSort l
  in head lOrdered
```

The bubble sorting function *bSort* calls the function *aBubble* that permutes locally the elements of its argument list so that the lowest element is at the end. This lowest element is the head of the result of *bSort* and the rest of the list is recursively sorted the same way:

```
bSort [] = []
bSort l = last l' : bSort (init l')
  where l' = aBubble l
```

Note that, a single call only to *aBubble* is required to compute the *minimum* of a list. Laziness ensures recursive calls to *bSort* are not performed in this case.

The producer consumer scheme can be used in imperative programming too. Let us consider the same example for mutable arrays.[1] The structure of the imperative program is quite similar to the functional version although a **do** introduces a sequence of imperative computations. First, we sort in place the full array *a* between indexes *from* and *to*, second we read at index *from* its first cell:

```
minimumM a = do
  (from, to) ← getBounds a
  bSortM a from to
  readArray a from
```

The behavior of the imperative program is quite different from the functional version. Imperative code is strict: the array has to be fully sorted by *bSortM*, before its first cell can be read. The goal of this article is to allow laziness for imperative programs so that the execution of *minimumM* mimics that of *mimimum*.

Lazy imperative programming. In [Launchbury, 1993], Launchbury has introduced a way to use functional laziness to perform imperative laziness for free. This particular form of laziness has been implemented in Haskell in the *Lazy.ST* monad.[2]

For instance, a call to *reset* in that monad to overwrite an array with 0 does not perform any write operation until a read is done on the array.

```
reset a = do
  (from, to) ← getBounds a
  forM_ [from .. to] (λi → writeArray a i 0)
```

However, as Launchbury noticed himself, such a laziness is not always enough:

> "If after specifying a computation which sets up a mutable array, only one element is read, then all preceding imperative actions will be performed, not merely those required to determine the particular element value."

Indeed, when the following code is run lazily in the *Lazy.ST* monad the single read operation requires one million write operations of *reset* to be previously performed:

```
Lazy.runST $ do
  a ← newArray (0, 1000000) 1
  reset a
  readArray a 0
```

The computation above is even slower that a strict run in the *Strict.ST* monad.

In the case of bubble sort, the situation is similar. The last line of *minimumM* that reads the first cell of *a* requires the array to be completely sorted: it triggers recursive calls to *bSortM*. So, as noted in [Launchbury, 1993]:

> "There may be room for the clever work found in imperative language implementations whereby imperative actions are reordered according to actual dependencies."

[1] In all the paper, we use the *MArray* type class for mutable arrays, defined in *Data.Array.MArray*.
[2] The monad *Lazy.ST* is provided by the (qualified as *Lazy*) module *Control.Monad.ST.Lazy*.

But twenty years later, very little has been done in this way... An interesting approach is iteratees [Kiselyov et al., 2012] that offers lazy evaluation for a consumer-programming style for streams. However dependencies between computations are explicitly expressed by the stream structure, hence they do not allow reordering (*e.g.*, in order to access the tenth value, the nine preceding values must be generated). So, the observation of Launchbury is the starting point of our work.

To understand why imperative laziness is not so lazy, let us develop our running example of the bubble sort. The imperative sorting function *bSortM* calls *aBubbleM a from to* in order to swap neighbour values if required, hence to place the lowest value of the subarray (*a* between *from* and *to*) in its cell *from*.[3] Then, the recursive call sorts the suffix of the array:

```
bSortM a from to = when (from < to) $ do
  aBubbleM a from to
  bSortM a (from + 1) to
```

When *bSortM* is used in *minimumM* the value of the first cell only is required. It can be read after the first call to *aBubbleM* and no recursive call to *bSortM* need to be evaluated.

But how to extract this property from the code? All we can say, without using involved static analysis, is that *bSortM* performs effects on its array argument, and so when a cell of this array needs later to be accessed, the entire sort operation must be performed. This is because it is very difficult to guess that *bSortM a* (*from* + 1) *to* has no effect on the cell at index *from*.

This article shows that fine-grained descriptions of computational effects and their dependencies enable the relaxation of this limitation. More precisely, we show that the following ingredients are sufficient to evaluate imperative computations in a lazier way: (i) an operational description of monadic computations that reifies computations; (ii) a lazy monad transformer that stores lazy reified computations in a trace of closures representing the sequence of delayed computations; (iii) a computable description of effects and their dependencies that enables triggering only a subset of delayed computations; (iv) an *unsafePerform* operation, in order to eventually evaluate a delayed computation. The use of an unsafe operation may look dangerous but this is not actually the case because we prove in Section 3 a conservativity result with respect to the strict semantics. Our argument is similar to the one of R. Lämmel and S. Peyton Jones in [Lämmel and Jones, 2003]: "we regard [an unsafe feature] as an implementation device to implement a safe feature".

By using annotation-like evaluation operators, the declaration of lazy imperative code is not invasive as it stays very close to the original code. Note that, our approach makes sense when the cost of laziness (creation of closures and checking their dependencies) is less than the cost of evaluating useless computations. Obvioulsy, it is not always the case, but expensive functions (*e.g.* some foreign functions) and code reuse are prone to create such opportunities.

This operational approach to computational effects can be related to the presentation of monads as algebraic operations and equations popularized by Plotkin and Power [Plotkin and Power, 2001]. In that setting, the dependency relation corresponds to commutativity equations between operations.

To take advantage of other kind of equations, lazy evaluation is then extended with a merge operation that allows replacing two consecutive operations by a new one. This corresponds for instance to an idempotent equation on an operation. We present the kind of optimization enabled by this extension for sorting operations, but also for read/write operations on files.

Overview. Section 2 presents from a user perspective our general framework to describe effects of computations and their dependencies, thus allowing the reordering of imperative operations according to actual dependencies. We instantiate it for the case of sorted arrays and compare

[3] To mimic the behavior of *bSort*, bubbles go from right to left.

efficiency of the lazy and strict quicksort functions in different scenarios. Section 3 defines formal semantics of call-by-need λ-calculi extended with strict and lazy imperative features. In this context, we prove the correctness of our approach. Section 4 gives an overview of the nuts and bolts of our framework. In particular, lazier imperative programming requires to build and lazily evaluate the delayed trace of execution. This trace is managed by a lazy monad transformer, that uses the description of effects to analyze which computations are triggered by the current computation to be evaluated. Section 5 introduces the possibility to dynamically merge composed imperative computations for optimization purpose. We conclude in Section 6. The full code can be found in annex.

2 Lazy Algebraic Operations

In functional programming, dependencies can be recovered directly from the structure of the computation. Thus, it is possible to execute what needs to be executed, and nothing more. In imperative programming, an operation may depend on another by *effect dependency*. Thus, without further information, all effects must be performed (in their declaration order) to guarantee the correctness of the execution. In [Launchbury, 1993], Launchbury presents a simple mechanism to delay imperative computations as long as the structure on which effects are performed is not accessed. But as soon as a value in that structure is required, the entire trace of delayed computations must be performed.

To go further, two problems need to be addressed: (i) we need to specify commutation between effectful operations; (ii) we need a direct access to atomic effectful operations.

2.1 A General Framework to describe commutation of algebraic operations

Reified monads are monads plus a means to compute when two effectful operations of the monad commute. Concretely, they are instances of the *Reified* type class: they must implement the *commute* method. To distinguish between lazy and strict computations, the *isStrict* method is also required by the *Reified* type class:

```
class Reified m where
  commute :: m a → m b → Bool
  isStrict  :: m a → Bool
```

Note that commutativity (of two effectful operations computed by *commute*), as well as strictness (of a single effectful operation computed by *isStrict*) are dynamic properties: they may depend on the parameters passed to effectful operations. This enables precise dynamic analysis at run-time.

In practice, instantiating the class *Reified* for an arbitrary monad can only be done by making everything strict because laziness would require to introspect any computation to get its effect. So to be able to reason about effects of computations, a monad may be advantageously described as set of operations and equations between them. This is the algebraic operations approach popularized by Plotkin and Power [Plotkin and Power, 2001]. But it is well known that many computational monads may be described as algebraic effects, i.e. by means of operations and equations [Plotkin and Power, 2002]. And that any algebraic effects induce a monad.

We take advantage of both worlds here and we describe our effectful computations by means of algebraic operations, but we embed it into a monadic computation using the operational monad transformer [Apfelmus, 2010].

The operational monad transformer. Algebraic operations are turned into computations using the operational monad transformer *ProgramT*.[4]

```
data ProgramT instr m a where
  Lift  :: m a → ProgramT instr m a
  Bind  :: ProgramT instr m a → (a → ProgramT instr m b)
    →   ProgramT instr m b
  Instr :: instr a → ProgramT instr m a
instance Monad m ⇒ Monad (ProgramT instr m) where
  return = Lift ∘ return
  (≫=)   = Bind
instance MonadTrans (ProgramT instr) where
  lift   = Lift
```

The *Lift* and *Bind* constructors reify operations provided by a monad transformer. This transformer expects a set of algebraic operations (or instructions), defined as a generalized algebraic datatype (GADT) *instr*, and turn them into a computation using the constructor *Instr*. By pattern-matching on the GADT *instr*, it becomes possible to specify the side effects of each instruction.

Turning algebraic operations into monadic computations. We now need to give a computational meaning to algebraic operations by interpreting them into the operational monad transformer. In this way, we marry the best of the two worlds: direct access to operations and modularity of monadic presentation. This interpretation is captured by the *EvalInstr* type class:

```
class EvalInstr instr m where
  evalInstr :: instr a → ProgramT instr m a
```

Defining an instance of *EvalInstr* requires building a computation in the operational monad transformer. This can be done in two ways: either by lifting an operation from the underlying monad; or by promoting an algebraic operation using the *Instr* constructor. To make the definition of computations in *ProgramT* more transparent to the user, we provide the following two synonym functions:

```
  -- (-!-) lift a computation in m
(-!-) :: (Monad m) ⇒ m a → ProgramT instr m a
(-!-) = lift
  -- (-?-) promotes an algebraic operation
(-?-) :: (Monad m) ⇒ instr a → ProgramT instr m a
(-?-) = Instr
```

They will be used to prefix lines in imperative definitions and they can be seen as strictness/laziness annotations.

Lazy evaluation of algebraic operations. To evaluate lazily a computation in *ProgramT instr m*, the computation has to be lifted to the lazy monad transformer *LazyT*. This transformer is presented in detail in Section 4, we just briefly overview its interface.

```
data LazyT m a = ...
```

LazyT is a monad transformer with run and lift operations. As working explicitly in this monad is not necessary for the programmer, we provide a *lazify* function that lifts a computation from *ProgramT* to *LazyT* and directly (lazily) runs it.

```
lazify :: ProgramT instr m a → ProgramT instr m a
lazify = runLazyT ∘ liftLazyT
```

[4] Provide by the *Control.Monad.Operational* module.

Describing commutation with effects. In practice, the commutation of two algebraic operations is given by non-interference of their underlying effects. To make this explicit, we introduce the notion of effect types, which are types plus a means to compute dependencies between two effects. Concretely, they are instances of the *Effects* type class:

```
class Effects eff where
  dependsOn :: eff → eff → Bool
```

To connect effects with computations, a monad m needs to provide a method *effect* that given a computation returns its effect[5]:

```
class Effects eff ⇒ Effectful m eff | m → eff where
  effect :: m a → eff
```

Then, any monad m that is an instance of *Effectful m eff* induces a notion of commutation given by

```
c `commute` c' = ¬ (effect c `dependsOn` effect c')
```

2.2 Lazier Imperative Arrays

Recall the use case of the introduction. We want to get the minimum value of an imperative array by sorting this array using *bSortM* and reading its first cell. But we do not need the whole array to be sorted. This section details what needs to be written to make *bSortM* lazy.

Effects as ranges of integers. The first thing to declare is the kind of effects performed by *bSortM*. A call to *bSortM* with arguments *a*, *from* and *to* sorts the subarray *a* between indexes *from* and *to*. The effects of this function are read and write operations on the cells *from*, *from* + 1... *to*. We specify such a set of contiguous cells using bounds of indexes:

```
data Bounds arrayof = Bounds (arrayof Int Int) Int Int
```

The type parameter *arrayof* abstracts over the type of mutable array (*e.g.* $\mathbb{IO}$*Array*, *STArray*, ...). In the sequel, we will abusively call sub-array a value of type *Bounds arrayof*.

We must specify when two calls to *bSortM* are independent, by instantiating the type class *Effects* for sub-arrays.

```
instance Eq (arrayof Int Int) ⇒
         Effects (Bounds arrayof)
where
dependsOn (Bounds a1 lb1 ub1) (Bounds a2 lb2 ub2) =
  a1 ≡ a2 ∧ lb2 ⩽ ub1 ∧ lb1 ⩽ ub2
```

Two subarrays are independent if their sets of indexes are disjoint or if they are subarrays of two different arrays.

Algebraic presentation of operations on arrays. To be able to reason about effects of computations, they have to be reified as algebraic operations. For simplicity, we only consider two kinds of operations in this example: sorting an array (using bubble sort) and reading a cell of an array. We turn this structure into the following GADT:

```
data ArrayC arrayof a where
  BSortM    :: arrayof Int Int → Int → Int →
```

[5] Using functional dependencies, we force that a monad m is described using at most one effect type.

```
                ArrayC arrayof ()
ReadArray :: arrayof Int Int → Int →
                ArrayC arrayof Int
```

Note that *ArrayC* must describe operations that will be performed on an array. If we need to use a new operation, *ArrayC* has to be extended accordingly (see *QSortM* below).

Description of each algebraic operation. We can now instantiate the class *Effectful* and *Reified* to specify, for the two kinds of computation, if they are strict/lazy and their side effects. In our example, *BSortM a from to* is lazy when *from* < *to* (trivial sorting operations are not delayed) and it sorts a subarray *a* bounded by *from* and *to*, *ReadArray* is strict and it reads a single array cell specified by its arguments.

```
instance Eq (arrayof Int Int) ⇒
         Effectful (ArrayC arrayof) (Bounds arrayof)
where
  effect (BSortM    a from to) = Bounds a from to
  effect (ReadArray a i)       = Bounds a i i
instance Eq (arrayof Int Int) ⇒
         Reified (ArrayC arrayof) where
where
  c `commute` c' = ¬ (effect c `dependsOn` effect c')
  isStrict (BSortM a from to) = from ⩾ to
  isStrict (ReadArray a i)    = True
```

We must also instantiate the type class *EvalInstr* to give meaning to those operations.

```
instance MArray arrayof Int m ⇒
         EvalInstr (ArrayC arrayof) m
where
  evalInstr (ReadArray a i) = (-!-) $ readArray a i
  evalInstr (BSortM a from to) = when (from < to) $ do
              (-!-) $ aBubbleM a from to
              (-?-) $ BSortM a (from + 1) to
```

The evaluation of *ReadArray* is given by performing the corresponding operation in the underlying monad *m*. The evaluation of *BSortM* is similar to the original *bSortM* except that the recursive call is done on the algebraic operation *BSortM*.

Note that, this description of algebraic operations is done once and for all. An end-programmer could then use the lazy imperative bubble sort as easily as the strict one (granted he uses only reified version of read to access the array).

Adding algebraic actions. The recursive scheme of *bSortM* is biased towards right most cells of arrays. This is fine for the function *minimumM* that returns the value of the leftmost cell. The dual function *maximumM* returns the value of the rightmost cell and so triggers all delayed computations. In that case, lazy evaluation is less efficient than the strict imperative version (closure building and dynamic dependencies analysis come with a cost).

It is therefore better to use a less biased recursion scheme such as the standard in-place quicksort algorithm that sorts the array dichotomically as follows:

```
qSortM a from to = when (from < to) $ do
  let pivot = from + div (to − from) 2
  pivot' ← partitionM a from to pivot
  qSortM a from (pivot' − 1)
  qSortM a (pivot' + 1) to
```

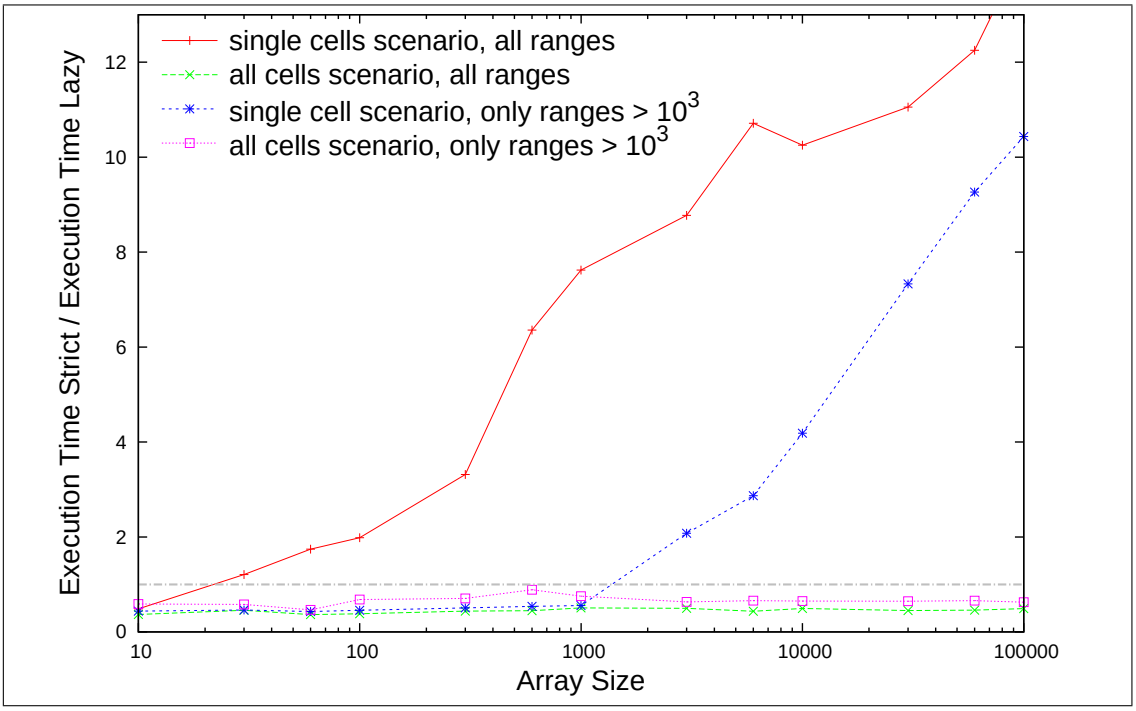


Figure 1: Time to quicksort an array strictly/lazily. Time to quicksort an array strictly/lazily only for range $> 10^3$

This algorithm selects the pivot in the middle of the subarray. The function *partitionM* returns the new index of the pivot once lower and greater value have been partitioned in place. In order to make *qSortM* lazy we only have to introduce an extra constructor *QSortM* in the GADT *ArrayC* and the same definitions for *strict* and *effect* as for *BSortM*. The only difference with bubble sort is in the definition of *evalInstr*

```
evalInstr (QSortM a from to) = do
  when (from < to) $ do
    let pivot = from + div (to − from) 2
    pivot′ ← (-!-) $ partitionM a from to pivot
    (-?-) $ QSortM a from (pivot′ − 1)
    (-?-) $ QSortM a (pivot′ + 1) to
```

Our dependencies dynamic analysis takes into account quite naturally this dynamic dichotomy scheme. The next section presents the performance of *QSortM* with respect to its strict version.

2.3 Performance of lazy imperative arrays

Laziness comes with a cost (of building closures, adding them to the trace, computing dependencies, updating closures and the trace). This section evaluates to which extent the use of imperative lazy evaluation may improve efficiency by comparing the lazy and strict quicksorts in two extreme cases. The first case is the access to a single cell of a large array that has been sorted to get the minimum value of that array.

```
firstScenario size = lazify $ do
  a ← (-!-) $ newListArray (0, size − 1) [size, size − 1 . . 1]
```

```
(-?-) $ QSortM a 0 (size − 1)
(-?-) $ ReadArray a 0
```

The other case is the access to all cells of the same array.

```
secondScenario size = lazify $ do
  a ← (-!-) $ newListArray (0, size − 1) [size, size − 1 . . 1]
  (-?-) $ QSortM a 0 (size − 1)
  forM_ [0 . . size − 1] ((-?-) ∘ (ReadArray a))
```

Thus, those two scenarios provide respectively the best and worst case with respect to imperative evaluation.

In both scenarios, the average case complexity of the strict quicksort algorithm is $O(n*log\ n)$, where n is the size of the array. But for lazy quicksort, the first scenario is advantageous because it only requires $O(n + (log\ n)^2)$ comparison. Indeed, a first pivot is selected and compared with all (n) elements, then this process is repeated only for subarrays that contain the cell of interest (in the average case, each subarray is split into two subarrays of equal size, hence $n/2 + n/4 + n/8 + \ldots = n$). This evaluation requires building $log\ n$ closures and to detect dependencies in a trace with less than $log\ n$ closures (hence the cost of dependencies checking is $(log\ n)^2$). So the theoretical gain of laziness is proportional to $log\ n$.

In the second scenario, the complexity is almost the same as for the strict case, whereas the complexity of the lazy case explodes similarly as in the worst case, the number of closure presents in the trace during computation is proportional to n, hence the cost of dependencies checking can be as big as n^2. In order to ensure the asymptotic complexity of an algorithm is not changed by our lazy framework, we can limit the size of the trace and decide to strictly evaluate a lazy computation when the trace is already full. In this case, which lazy computation should be strictly evaluated offers choices (for instance the more recent, or the oldest). Different strategies could be provided. Obviously, there is no best solution in general and this requires to mindfully configure each program.

Figure 1 shows the average result for both scenarios on a random set of 20 arrays of size between 10 and 100000. The plain red line with plus marks represents the ratio between execution time of strict vs lazy quicksort in the single cell access scenario. We can see that after a certain size of array, the gain is actually close to $log(n)$. The cost of laziness is described by the dashed green line with cross marks that shows that in the "all-cells access" scenario, the lazy algorithm is twice slower.

This makes clear that when all (or many) delayed computations are eventually needed, the strict version is indeed more efficient than the lazy one. But, laziness should be used, when few delayed computations are eventually needed.

Note also that the laziness of *QSortM* can be modulated dynamically, for instance by specifying that evaluation is strict as soon as the subarray is smaller than 1000:

```
strict (QSortM a from to) = to − from < 1000
```

This way, the programmer can set a trade-off between lazy and strict evaluation. Figure 1 shows (blue line with star marks) for the definition above that the first scenario is less efficient than the full lazy case, but still performs better than the strict case. For the worst case scenario (pink line with square marks), it is better than the full lazy evaluation as it is only 1.5 time slower than the strict evaluation.

3 Call-by-Need Calculus with Lazy Imperative Operations

This section presents the call-by-need calculus with lazy imperative operations which serves as a formalization of the lazy monad transformer *LazyT* described in Section 2.1. Note that, our

$$\textsc{Val}\ \frac{V \in \{\lambda x.E, \mathtt{return}\, E, \mathtt{ref}\, x, (), \mathtt{tt}, \mathtt{ff}\}}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash V \Downarrow \langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash V}$$

$$\textsc{Var}\ \frac{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash V}{\langle \Gamma_0' +\!\!+ [x \to E] +\!\!+ \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash x \Downarrow \langle \Gamma_0' +\!\!+ [x \to V] +\!\!+ \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash V}$$

$$\textsc{App}\ \frac{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \lambda x.F \quad \langle [x' \to E'] +\!\!+ \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash F[x'/x] \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V \qquad x' fresh}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash E\, E' \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V}$$

$$\textsc{Bnd}\ \frac{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \mathtt{return}\, G \quad \langle [x' \to G] +\!\!+ \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash F[x'/x] \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V \qquad x'\ fresh}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash x \leftarrow E\,;\, F \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V}$$

$$\textsc{New}\ \frac{\Gamma_1 = [x \to E] +\!\!+ \Gamma_0 \quad \Sigma_1 = \Sigma_0 \cup [a \to x] \quad x, a\ fresh}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash \mathtt{new}\, E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_0\rangle \vdash \mathtt{return}\, (\mathtt{ref}\, a)}$$

$$\textsc{Read}\ \frac{\langle \Gamma_0, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash \mathtt{ref}\, a}{\langle \Gamma_0, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash \mathtt{read}\, E \Downarrow \langle \Gamma_1, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash \mathtt{return}\, x}$$

$$\textsc{Write}\ \frac{\langle \Gamma_0, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash \mathtt{ref}\, a \qquad y\ fresh}{\langle \Gamma_0, \Sigma_0 \cup [a \to x], \Theta_0\rangle \vdash \mathtt{write}\, E\, F \Downarrow \langle [y \to F] +\!\!+ \Gamma_1, \Sigma_0 \cup [a \to y], \Theta_0\rangle \vdash \mathtt{return}\, ()}$$

Figure 2: Big-step semantics for the call-by-need calculus with strict imperative operations

semantics are more abstract than our implementation. In particular, the implementation reifies instructions in order to interpret them lazily. Instructions are not reified in the semantics, since semantics are interpreters. Moreover our semantics focus on laziness, so we do not detail here optimizations that merge closures. Our semantics are then used to prove a conservativity result of reductions under *LazyT* with respect to the usual semantics of Haskell—using only simple assumptions on the *commute* operation.

Section 3.1 defines the semantics of the call-by-need calculus with strict imperative operations. Section 3.2 presents the extension to lazy imperative operations and Section 3.3 outlines a conservativity between the two semantics under simple assumptions on the modeling of effects.

3.1 Call-by-Need Calculus with Strict Imperative Operations

The call-by-need calculus with strict imperative operations is a call-by-need λ-calculus [Maraist et al., 1998] extended with imperative features: monadic bind ($x \leftarrow E\,;\, E$) and return ($\mathtt{return}\, E$) and mutable memory cells [Ariola and Sabry, 1998]. Together with abstractions and returned computation, there are ground values unit (), boolean constants `tt` and `ff`, and memory cell references

`ref` x. Memory cells are similar to Haskell's `STRef`: they can be allocated and initialized with an expression E by `new` E, read by `read` E where E denotes a reference and written by `write` E_1 E_2 where E_1 denotes a reference. The full calculus is given by

$$E ::= x \mid \lambda x.E \mid E\,E \mid x \leftarrow E\,;\,E \mid \texttt{return}\,E \mid () \mid \\ \texttt{new}\,E \mid \texttt{read}\,E \mid \texttt{write}\,E\,E \mid \texttt{ref}\,x \mid \texttt{tt} \mid \texttt{ff}$$

The semantics of the calculus is similar to Haskell strict version of the `ST` monad, available in the library `ST.Strict`—i.e., side effects of store primitives are strictly (immediately) executed.

The big-step semantics is given in Figure 2, where configuration terms are reduced in one step to configuration values. A configuration is a triplet of components that represents a context and a term $\langle \Gamma, \Sigma, \Theta \rangle \vdash E$, where (i) Γ is an environment of mutable closures $[x \to E]$ (that can be updated with their normal form); (ii) Σ is a store of mutable cells $[a \to x]$ where a and x are variables; (iii) Θ is a trace of closure references. This trace is not required until lazy imperative computations are introduced (section 3.2). Thus, in the strict version of the calculus, it is threaded but never accessed. However, we introduce it upfront for uniformity with reduction rules of the lazy version. A closed term is evaluated in the originally empty context $\langle \emptyset, \emptyset, \emptyset \rangle$.

We now briefly review rules of Figure 2.

Purely Functional Rules. Every value is directly returned (Rule Val), no computation is performed. The rule for application (Rule App) sheds light on the laziness of the purely functional part of the calculus. The function E is evaluated to its normal form $\lambda x.F$, a closure is created for the unevaluated argument E', a (fresh) pointer to the closure is substituted in the body F of the function which is finally evaluated. When a variable is evaluated (Rule Var), the closure on which it points to in the environment is evaluated. Note that it is evaluated in environment Γ_0 because the more recent bindings Γ'_0 are temporarily/locally ignored. Then, the closure is updated with its normal form, and the more recent bindings are restored.

Monadic Rules. The evaluation of a bind expression (Rule Bnd) is quite similar to the evaluation of an application in the functional world, but the "argument" E is evaluated to its (monadic) normal form. Thus, effect are strictly executed while the remaining computation is lazily returned. The store primitives allocate, read and write memory cells denoted by their references `ref` x. A `new` E expression (Rule New) creates a new closure for E, allocates a fresh cell a in the store initialized with the closure pointer x and returns the cell reference. Note that, the expression E is not evaluated: the store is lazy. This models the Haskell library `Data.STRef`. A `read` E expression (Rule Read) evaluates its argument E to a reference to a cell store and it returns the content of the cells store (a closure pointer). Note that this does not evaluate the closure: the store is lazy. A `write` $E\,F$ expression (Rule Write) evaluates its first argument E to a cell store and creates a closure for F (the store is lazy) and the cell is updated with the newly created closure pointer.

A word on recursion. As our calculus is untyped, we can express recursion using a fixpoint combinator, which agrees with Wadsworth's original treatment [Wadsworth, 1971] of recursion in call-by-need calculi. However, implementations of fixpoint in lazy functional programming and in particular in Haskell make use of circularity in the environment. This mismatch is not problematic in our setting because the two points of view coincide for recursive function definitions—they only differ in the sharing behavior of circular data structure, which we will not consider in this paper.

$$\textsc{Eval}_{\textsc{L}}\ \frac{\Gamma_1 = [x \to \texttt{clo}\, x\, E] +\!\!+ \Gamma_0 \qquad \Theta_1 = [x] +\!\!+ \Theta_0 \qquad x\ fresh}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash \uparrow_{\texttt{L}} E \Downarrow \langle \Gamma_1, \Sigma_0, \Theta_1\rangle \vdash \texttt{return}\, x}$$

$$\textsc{Eval}_{\textsc{S}}\ \frac{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash E \Downarrow_{Force} \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \texttt{return}\, E}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash \uparrow_{\texttt{S}} E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \texttt{return}\, E}$$

$$\textsc{Clo}\ \frac{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \downdownarrows_x \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \qquad \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash x \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash \texttt{clo}\, x\, E \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V}$$

$$\textsc{Pop-Trace}\ \frac{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash E \Downarrow_{Force} \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \texttt{return}\, F}{\langle \Gamma'_0 +\!\!+ [x \to \texttt{clo}\, x\, E] +\!\!+ \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ \Theta_0\rangle \downdownarrows_x \langle \Gamma'_0 +\!\!+ [x \to F] +\!\!+ \Gamma_1, \Sigma_1, \Theta'_0 +\!\!+ \Theta_1\rangle}$$

$$\textsc{NotIn-Trace}\ \frac{x \notin \Theta_0}{\langle \Gamma_0, \Sigma_0, \Theta_0\rangle \downdownarrows_x \langle \Gamma_0, \Sigma_0, \Theta_0\rangle}$$

$$\textsc{Force-Comm}\ \frac{\begin{array}{c}\langle \Gamma'_0 +\!\!+ [x \to \texttt{clo}\, x\, F] +\!\!+ \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash \texttt{commute}\, F\, E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \texttt{tt} \\ \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash E \Downarrow_{Force} \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash V\end{array}}{\langle \Gamma'_0 +\!\!+ [x \to \texttt{clo}\, x\, F] +\!\!+ \Gamma_0, \Sigma_0, [x] +\!\!+ \Theta_0\rangle \vdash E \Downarrow_{Force} \langle \Gamma_2, \Sigma_2, [x] +\!\!+ \Theta_2\rangle \vdash V}$$

$$\textsc{Force-NoComm}\ \frac{\begin{array}{c}\langle \Gamma'_0 +\!\!+ [x \to \texttt{clo}\, x\, F] +\!\!+ \Gamma_0, \Sigma_0, \Theta_0\rangle \vdash \texttt{commute}\, F\, E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash \texttt{ff} \\ \langle \Gamma_0, \Sigma_0, [x] +\!\!+ \Theta_1\rangle \downdownarrows_x \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \qquad \langle \Gamma_2, \Sigma_2, \Theta_2\rangle \vdash E \Downarrow_{Force} \langle \Gamma_3, \Sigma_3, \Theta_3\rangle \vdash V\end{array}}{\langle \Gamma'_0 +\!\!+ [x \to \texttt{clo}\, x\, F] +\!\!+ \Gamma_0, \Sigma_0, [x] +\!\!+ \Theta_0\rangle \vdash E \Downarrow_{Force} \langle \Gamma_3, \Sigma_3, \Theta_3\rangle \vdash V}$$

$$\textsc{Force-Eval}\ \frac{\langle \Gamma_0, \Sigma_0, \emptyset\rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash V}{\langle \Gamma_0, \Sigma_0, \emptyset\rangle \vdash E \Downarrow_{Force} \langle \Gamma_1, \Sigma_1, \Theta_1\rangle \vdash V}$$

Figure 3: Big-step semantics for the call-by-need calculus with lazy imperative operations

3.2 Call-by-Need Calculus with Lazy Imperative Operations

In the call-by-need calculus with lazy imperative operations, imperative computation is allowed to be delayed. To distinguish between computations that must be delayed or not, we extend our language with lazy $\uparrow_{\texttt{L}} E$ and strict $\uparrow_{\texttt{S}} E$ versions of the imperative (*a.k.a.* monadic) computation E. This is the static correspondence to the dynamic *isStrict* method introduced in Section 2.1. We favor a static description in the formalization in order to avoid unnecessary complications because this ensures the amount of laziness has no impact on conservativity. What is more important for conservativity is that all imperative computations must be accessed by either $\uparrow_{\texttt{L}}$ or $\uparrow_{\texttt{S}}$. Indeed, it is now unsafe in general for the programmer to use directly strict store primitives `read` and `write` in this setting, because their semantics does not trigger evaluation in the trace. This is guaranteed by the following extension of the call-by-need calculus with strict imperative

operations, that introduced a stratification:

$$\begin{array}{l} E ::= E_L \mid \ldots \text{ (same as Section 3.1)} \\ E_L ::= x \mid \lambda x.E_L \mid E_L\, E_L \mid x \leftarrow E_L\,;\, E_L \mid \texttt{return}\, E_L \\ \qquad \mid \uparrow_{\texttt{L}} E \mid \uparrow_{\texttt{S}} E \mid \texttt{clo}\, x\, E \mid \texttt{commute}\, E\, E \end{array}$$

The lazy extension possesses two new primitive operations: (i) $\texttt{clo}\, x\, E$ that corresponds to a (self-referential) delayed imperative computation (*a.k.a.* imperative closure) that is introduced in the environment during the evaluation of a lazy computation, (ii) `commute` that corresponds to the *commute* method described in Section 2.1.

Lazy Semantics. Figure 3 introduces the new rules that allow lazy imperative computations (the rules of the strict semantics remain valid). This requires the memorization of the sequence of delayed computations in the trace Θ and to force some of the delayed computations when one is to be eventually performed.

A lazy computation $\uparrow_{\texttt{L}} E$ (Rule $\textsc{Eval}_L$) does not evaluate E but it stores it in a new self-referential imperative closure and it adds in the trace the reference to this closure.

A strict computation $\uparrow_{\texttt{S}} E$ (Rule $\textsc{Eval}_S$) evaluates E, but first it may force the evaluation of other closures in the trace as specified by the relation $\Downarrow_{Force}$.

Performing delayed effects. When a delayed closure $\texttt{clo}\, x\, E$ is to be evaluated (Rule Clo), the remaining effects of x are performed using $\downdownarrows_x$ and then x is evaluated. When x is absent from the trace, $\downdownarrows_x$ does nothing (Rule NotIn-Trace). This situation can happen for instance in Rule Force-NoComm when the evaluation of $\texttt{commute}\, F\, E$ triggers the evaluation of the delayed computation x. When x is present (Rule Pop-Trace), it forces the evaluation of E to $\texttt{return}\, F$, updates x in the environment (similarly to Rule Var) and pops x from the trace.

The predicate $\Downarrow_{Force}\, E$ evaluates an expression but it first checks if delayed computations should be evaluated before. If the first closure in the trace commutes with E (Rule Force-Comm), the evaluation of $\Downarrow_{Force}\, E$ is performed on the remaining trace Θ_0. If it does not commute with E (Rule Force-NoComm), its evaluation is required before evaluating E. This is performed by $\downdownarrows_x$. The evaluation of $\Downarrow_{Force}\, E$ is performed on the remaining trace Θ_2. Finally, if the trace is empty (Rule Force-Eval), there is no other closure to potentially evaluate, so the closure is evaluated using the original evaluation $\Downarrow$.

Particular Case: the Lazy ST Monad. The original proposal for lazy imperative programming of Launchbury [Launchbury, 1993]—implemented in Haskell as the Lazy ST monad—is the particular case when `commute` always returns false. Indeed, without any reification of the calculus, this is the only correct assumption.

3.3 Conservativity of Lazy Imperative Semantics

The main purpose of this section is to exhibit assumptions on `commute` to prove the conservativity of the lazy semantics over the strict one.

Assumptions on `commute`. First, we assume that evaluations of `commute` always terminate and return either `tt` or `ff`. Indeed, if `commute` diverges, then a computation could converge in the strict semantics and diverge in the lazy one. Evaluations of $\texttt{commute}\, F\, E$ can have side effects

in general, but, for the sake of correctness, it should not evaluate more than a strict evaluation. This can be formalized as, for every configuration $\langle \Gamma, \Sigma, \Theta \rangle$:

$$\langle \Gamma, \Sigma, \Theta \rangle \vdash (b \leftarrow \texttt{commute}\, F\, E\,;\, x \leftarrow F\,;\, E \equiv x \leftarrow F\,;\, E) \tag{1}$$

where $\equiv$ means that the two terms are (observationally) equivalent in the configuration $\langle \Gamma, \Sigma, \Theta \rangle$.

When $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle \vdash \texttt{commute}\, F\, E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1 \rangle \vdash \texttt{tt}$, then evaluation of E and F must commute. This means that for any future configuration $\langle \Gamma_2, \Sigma_2, \Theta_2 \rangle$, the following equality must be satisfied:

$$\langle \Gamma_2, \Sigma_2, \Theta_2 \rangle \vdash (x \leftarrow E\,;\, y \leftarrow F\,;\, G \equiv y \leftarrow F\,;\, x \leftarrow E\,;\, G) \tag{2}$$

for all monadic computations G.

Purging the trace. To relate the semantics of lazy and strict evaluations, we first need to define an operation that transforms a lazy term into a strict one. The strictification $[E]_s$ of a term E is defined as

$$[\uparrow_{\mathtt{L}} E]_s \stackrel{def}{=} \uparrow_{\mathtt{S}} [E]_s \quad \text{and} \quad [C E_1 \dots E_n]_s \stackrel{def}{=} C[E_1]_s \dots [E_n]_s$$

where C is any other constructor of the language. Strictification is extended on environments pointwise.

We also need to define an operation $\langle \Gamma, \Sigma, \Theta \rangle_p$ that purges the trace Θ, that is that performs all delayed effects stored in that trace.

$$\begin{array}{lcl} \langle \Gamma, \Sigma, \Theta \rangle_p & \stackrel{def}{=} & \texttt{purge}\, \langle [\Gamma]_s, \Sigma, \Theta \rangle \\ \\ \texttt{purge}\, \langle \Gamma, \Sigma, \emptyset \rangle & \stackrel{def}{=} & \langle \Gamma, \Sigma, \emptyset \rangle \\ \texttt{purge}\, \langle \Gamma, \Sigma, \Theta +\!\!+ [x] \rangle & \stackrel{def}{=} & \texttt{purge}\, \langle \Gamma', \Sigma', \Theta' \rangle \end{array}$$

with $\langle \Gamma, \Sigma, \Theta +\!\!+ [x] \rangle \downarrow\downarrow_x \langle \Gamma', \Sigma', \Theta' \rangle$. Note that $\langle \Gamma, \Sigma, \Theta \rangle_p$ is not always defined, in the same way as $\Downarrow$. This could be guaranteed by a type system as in [Ariola and Sabry, 1998], but in the proof of conservativity, we preserve the property that the purge is well defined, so it is not necessary.

For now, we have been careful to purge the trace of delayed computations in a chronological order, to ensure correctness. The following lemma says that the order in which the trace is purged does not matter, granted the assumptions on `commute` above.

Lemma 1. *Let $\langle \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ \Theta_0 \rangle$ be a configuration such that $\langle \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ \Theta_0 \rangle_p$ is well-defined. Then there exists $\langle \Gamma_1, \Sigma_1, \Theta_1 \rangle$ such that*

$$\begin{cases} \langle \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ \Theta_0 \rangle \downarrow\downarrow_x \langle \Gamma_1, \Sigma_1, \Theta'_0 +\!\!+ \Theta_1 \rangle \\ \langle \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ \Theta_0 \rangle_p = \langle \Gamma_1, \Sigma_1, \Theta'_0 +\!\!+ \Theta_1 \rangle_p. \end{cases}$$

Proof. By induction on the length of the reduction of the purge $\langle \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ \Theta_0 \rangle_p$ and on the size of Θ_0.

Base Case. This case is not possible because there is at least x to be purged.

Induction Case. First remark that when $\Theta_0 = \emptyset$, the property is direct by definition of purging. Now, let us suppose that $\Theta_0 = [y] +\!\!+ \Theta''_0$. Let $\texttt{clo}\, x\, E$ and $\texttt{clo}\, y\, F$ be the two delayed closures on which x and y are (self) pointing to. The evaluation of $\langle \Gamma_0, \Sigma_0, \Theta'_0 +\!\!+ [x] +\!\!+ [y] +\!\!+ \Theta''_0 \rangle \downarrow\downarrow_x$ leads to the evaluation of $\langle \Gamma_0, \Sigma_0, [y] +\!\!+ \Theta''_0 \rangle \vdash \texttt{commute}\, F\, E$. Using Equation 1, we know that this evaluation does not affect the resulting configuration. By assumption, this evaluation always terminates and returns either `tt` or `ff`. (1) If it evaluates to `ff`, then y is purged from the trace.

By induction hypothesis on y (because the size of Θ_0 is smaller), we know that there exists $\langle \Gamma_1, \Sigma_1, \Theta_0' +\!\!+ [x] +\!\!+ \Theta_1 \rangle$ satisfying

$$\langle \Gamma_0, \Sigma_0, \Theta_0' +\!\!+ [x] +\!\!+ [y] +\!\!+ \Theta_0'' \rangle_p = \langle \Gamma_1, \Sigma_1, \Theta_0' +\!\!+ [x] +\!\!+ \Theta_1 \rangle_p.$$

We can then apply the induction hypothesis on x (because the length of the reduction of the purge has decreased) to conclude. (2) If it evaluates to `tt`, then y is skipped and we get by induction

$$\langle \Gamma_0, \Sigma_0, \Theta_0' +\!\!+ [y] +\!\!+ [x] +\!\!+ \Theta_0'' \rangle_p = \langle \Gamma_1, \Sigma_1, \Theta_0' +\!\!+ \Theta_1 \rangle_p.$$

It remains to show that

$$\langle \Gamma_2, \Sigma_2, \Theta_0' +\!\!+ [y] +\!\!+ [x] \rangle_p = \langle \Gamma_2, \Sigma_2, \Theta_0' +\!\!+ [x] +\!\!+ [y] \rangle_p$$

where Γ_2 and Σ_2 are the resulting environment and store obtained after the purge of Θ_0. But this is exactly what Equation 2 tells us. □

We can now state the conservativity theorem which says that if the strictification of a lazy computation E_L reduces (in a purged context) into a value V_1, then the lazy computation reduces (in the unpurged context) to a value V_2, whose strictification is equivalent to V_1. In other words, evaluation and strictification commute.

Theorem 1 (Conservativity)**.** *Let E_L be a term of the lazy imperative calculus and $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle$ a context such that $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p$ is defined. If*

$$\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p \vdash [E_L]_s \Downarrow \langle \Gamma_1, \Sigma_1, \emptyset \rangle \vdash V_1$$

then

$$\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle \vdash E_L \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2 \rangle \vdash V_2$$

and

$$\langle \Gamma_2, \Sigma_2, \Theta_2 \rangle_p \vdash V_1 \equiv [V_2]_s.$$

Furthermore, there exists a context extension $\Gamma_1 \subseteq \Gamma_1^{ext}$ (which corresponds to the additional variables introduced by the delayed computations in E_L) such that

$$\langle \Gamma_1^{ext}, \Sigma_1, \emptyset \rangle = \langle \Gamma_2, \Sigma_2, \Theta_2 \rangle_p.$$

Proof. By induction on the derivation tree of $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p \vdash [E_L]_s \Downarrow \langle \Gamma_1, \Sigma_1, \emptyset \rangle \vdash V_1$ and on the length of the reduction of the purge $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p$.
Rule Var: $E_L = x$. Two cases must be distinguished: (1) The variable is not pointing to a delayed computation, this case follows directly by induction. (2) The variable is pointing to a delayed computation, i.e., $[x \to \texttt{clo}\, x\, E]$. The only applicable rule is Clo which pops the variable x from the trace and evaluates x. As $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p$ is well defined, using Lemma 1, we know that

$$\begin{cases} \langle \Gamma_0, \Sigma_0, \Theta_0 \rangle \downdownarrows_x \langle \Gamma_1, \Sigma_1, \Theta_1 \rangle \\ \langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p = \langle \Gamma_1, \Sigma_1, \Theta_1 \rangle_p \end{cases}$$

The result follows by induction hypothesis on

$$\langle \Gamma_1, \Sigma_1, \Theta_1 \rangle \vdash x \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2 \rangle \vdash V$$

because the length of the purge is smaller.

Rule Val: there is nothing to compute and we have directly that $[E_L]_s = V_1$ for some value V_1. As the strictification of a term is a value exactly when the term is a value, we have that $E_L = V_2$ with $[V_2]_s = V_1$, as expected. The equality required on contexts is direct as $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle = \langle \Gamma_2, \Sigma_2, \Theta_2 \rangle$ and $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p = \langle \Gamma_1, \Sigma_1, \emptyset \rangle$, so we just have to set $\Gamma_1^{ext} = \Gamma_1$.

Rule App: E_L is of the form $E\,E'$. We define $\langle \Gamma'_0, \Sigma'_0, \emptyset \rangle \stackrel{def}{=} \langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p$. Then, by Rule App on $[E_L]_s = [E]_s\,[E']_s$, we have

$$\begin{array}{l} \langle \Gamma'_0, \Sigma'_0, \emptyset \rangle \vdash [E]_s \Downarrow \langle \Gamma'_1, \Sigma'_1, \emptyset \rangle \vdash \lambda x.[F]_s \\ \langle [x' \to [E']_s] +\!\!+ \Gamma'_1, \Sigma'_1, \emptyset \rangle \vdash [F]_s[x'/x] \Downarrow \langle \Gamma'_2, \Sigma'_2, \emptyset \rangle \vdash V_1 \end{array}$$

with

$$\langle \Gamma'_0, \Sigma'_0, \emptyset \rangle \vdash [E]_s\,[E']_s \Downarrow \langle \Gamma'_2, \Sigma'_2, \emptyset \rangle \vdash V_1.$$

By induction hypothesis, we have

$$\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle \vdash E \Downarrow \langle \Gamma_1, \Sigma_1, \Theta_1 \rangle \vdash \lambda x.F$$

with $\langle \Gamma_1, \Sigma_1, \Theta_1 \rangle_p = \langle \Gamma_1^{ext}, \Sigma'_1, \emptyset \rangle$ for some context extension Γ_1^{ext} of Γ'_1. Again, by induction hypothesis, using the fact that

$$\langle [x' \to E'] +\!\!+ \Gamma_1, \Sigma_1, \Theta_1 \rangle_p = \langle [x' \to [E']_s] +\!\!+ \Gamma_1^{ext}, \Sigma'_1, \emptyset \rangle$$

because purging has no effect on variables defined on the left-hand side of the environment, we have

$$\langle [x' \to E'] +\!\!+ \Gamma_1, \Sigma_1, \Theta_1 \rangle \vdash F[x'/x] \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2 \rangle \vdash V_2$$

with $\langle \Gamma_2, \Sigma_2, \Theta_2 \rangle_p \vdash V_1 \equiv [V_2]_s$ and $\langle \Gamma_2, \Sigma_2, \Theta_2 \rangle_p = \langle \Gamma_2^{ext}, \Sigma'_2, \emptyset \rangle$ for some context extension Γ_2^{ext} of Γ_2. Using Rule App, we can conclude that

$$\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle \vdash E\,E' \Downarrow \langle \Gamma_2, \Sigma_2, \Theta_2 \rangle \vdash V_2.$$

Rule Bind: this rule is similar to Rule App.
Rules New, Read, Write : those rules do not have to be considered because a lazy computation can not perform effects directly.
Rule Eval$_{\mathrm{L}}$: by hypothesis, we know that

$$\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p \vdash \uparrow_{\mathtt{S}} [E]_s \Downarrow \langle \Gamma_1, \Sigma_1, \emptyset \rangle \vdash V$$

To show,

$$\langle [x \to \mathtt{clo}\,x\,E] +\!\!+ \Gamma_0, \Sigma_0, [x] +\!\!+ \Theta_0 \rangle_p \vdash V \equiv [\mathtt{return}\,x]_s$$

To evaluate $\uparrow_{\mathtt{S}} [E]_s$, Rule Eval$_{\mathrm{S}}$ has necessarily been applied, followed by Rule Force-Eval, with premise

$$\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p \vdash [E]_s \Downarrow \langle \Gamma_1, \Sigma_1, \emptyset \rangle \vdash \mathtt{return}\,F.$$

So that $V = \mathtt{return}\,F$. Let $\langle \Gamma_0^P, \Sigma_0^P, \emptyset \rangle$ be $\langle \Gamma_0, \Sigma_0, \Theta_0 \rangle_p$. By definition of purging, we know that

$$\begin{array}{rl} & \langle [x \to \mathtt{clo}\,x\,E] +\!\!+ \Gamma_0, \Sigma_0, [x] +\!\!+ \Theta_0 \rangle_p \\ = & \langle [x \to \mathtt{clo}\,x\,E] +\!\!+ \Gamma_0^P, \Sigma_0^P, [x] \rangle_p \end{array}$$

The last step of purging amounts to evaluate $[E]_s$ is environment $\langle \Gamma_0^P, \Sigma_0^P, \emptyset \rangle$, so

$$\begin{array}{rl} & \langle [x \to \mathtt{clo}\,x\,E] +\!\!+ \Gamma_0, \Sigma_0, [x] +\!\!+ \Theta_0 \rangle_p \\ = & \langle [x \to F] +\!\!+ \Gamma_1, \Sigma_1, \emptyset \rangle \end{array}$$

with $\langle [x \to F] +\!\!+ \Gamma_1, \Sigma_1, \emptyset \rangle \vdash \mathtt{return}\,F \equiv [\mathtt{return}\,x]_s$ as expected. Note that here, the context extension of Γ_1 is simply given by $[x \to F]$.
Rule Eval$_{\mathrm{S}}$: this rule is direct by induction as both evaluation start with Rule Eval$_{\mathrm{S}}$. □

4 Lazy Evaluation of Algebraic Operations

This section presents the definition of the lazy monad transformer, that allows the lazily evaluatation of imperative computation. The core idea of that transformer is to maintain a trace of delayed closures and, when a computation has to be performed, to evaluate all the pending closures on which it depends. This implements the semantics described in Section 3.2. The main difference with the semantics of the calculus is that using a transformer enables running a computation using multiple lazy transformers, thus allowing lazy evaluation for different kinds of effects to be done with separated traces of executions. Conceptually, though, one can still think there is a single stateful thread and these independent traces could be interwoven in a single trace.

Section 4.1 presents the formalization of the trace of imperative closures using doubly linked lists. Section 4.2 introduces type classes to delay and evaluate computation. Section 4.3 defines the lazy monad transformer and its interface. Instances for the operational monad transformer are shown in Section 4.4. Finally, Section 4.5 details the computation performed in the *LazyT* layer.

4.1 Heterogeneous Trace of Imperative Closures

Delaying evaluation of imperative computations requires a trace of imperative closures. An imperative closure *Comp c* is a piece of computation *c* that may be later evaluated to perform its side-effects and get its value.

```
data Clo m where Comp :: m a → Clo m
```

A trace can contain closures of different types. So, the type *a* of the value returned by a closure has to be existentially quantified and does not appear in *Clo*.

If an imperative closure is eventually evaluated, it is removed from the trace. To be efficient, this requires a doubly linked list structure:

```
type TraceElt m = DLList (Clo m)
```

The module *DLList*—implemented using $\mathbb{IO}$ references and available in the source code—offers four functions: *createEmpty* that creates an new empty *DLList*, *removeElt* that removes its argument from the *DLList*, *insertElt* that inserts an element before its argument in the *DLList* and *endOfDL* that says if its argument is the end of the *DLList*.

4.2 Delaying computations

Lazy imperative evaluation requires postponing the evaluation of a computation.

Delaying evaluation amounts to pretending to have a value whereas it is still a computation. This is exactly what $unsafePerformIO :: \mathbb{IO}\ a \to a$ does for the $\mathbb{IO}$ monad. (This unsafe function must be used, but our framework guarantees that it is used in a safe way as proven in Section 3). Here, this mechanism is generalized to a type class *Delay m* that requires the monad to have an *unsafePerform* function. Then, delaying a computation *c* amounts to returning the *unsafePerform c*.

```
class MonadIO m ⇒ Delay m where
  unsafePerform :: m a → a
  delay :: Delay m ⇒ m a → m a
  delay = return ∘ unsafePerform
```

The constraint that the monad is an instance of *MonadIO* guarantees that $\mathbb{IO}$ operations performed on imperative closures are available in the monad stack.

The *Delay* type class can be instantiated for the $\mathbb{IO}$ and *Lazy.ST* monad. It can be instantiated for monad transformers that are based on an internal "state" (for a loose definition of state) by simply transforming this state into a mutable reference. *Delay* can also be instantiated for the operational monad transformer.

To evaluate a delayed imperative computation, the underlying monad has to be an instance of *Reified* in order to decide which other delayed imperative computations must be triggered first. But it also needs to provide information on how to evaluate an imperative closure by instantiating the *Evaluable* type class.

```
class Reified m ⇒ Evaluable m where
  eval :: m a → ReaderT (TraceElt m) m a
  hasEffect :: m a → Bool
```

Because evaluating an imperative closure may produce residual imperative closures, evaluation occurs in an environment with a trace; this is why the return type of *eval* uses the reader monad transformer. The function *hasEffect* decides if an imperative closure may trigger dependencies.

4.3 The Lazy Monad Transformer

We have now all the basic blocks to describe the lazy monad transformer. *LazyT m a* is just a computation *m a* that expects a trace to get evaluated.

```
data LazyT m a = LazyT (ReaderT (TraceElt m) m a)
```

To hide the manipulation of the environment, *LazyT* is based on the *ReaderT* monad transformer. We provide two functions *readTraceElt* and *readPTraceElt* that return the "current" imperative closure and the previous imperative closure stored in the trace.

Monadic operations on *LazyT* mimic the underlying monadic operations on *m*, except that a computation is evaluated lazily using the function *lazyEval*—which requires the underlying monad to be evaluable and delayable.

```
instance (Delay m, Evaluable m) ⇒
         Monad (LazyT m) where
  return v = LazyT $ return v
  x >>= k  = LazyT $ do y ← lazyEval x
                        lazyEval (k y)
```

When a computation has to be evaluated, if it is strict (according to the function *isStrict*), the imperative closures on which it depends are (potentially recursively) evaluated then the computation is performed (using the *eval* function). In the case it is lazy, a corresponding imperative closure is inserted in the trace and a means to evaluate it is returned (using the *evalL* function).

```
lazyEval :: (Delay m, Evaluable m) ⇒
            LazyT m a → ReaderT (TraceElt m) m a
lazyEval (LazyT c) = do clo ← readTraceElt
                        let c' = runReaderT c clo
                        if isStrict c'
                        then evalS c'
                        else evalL c'
```

The definitions of *evalS* and *evalL* are given in Section 4.5 and correspond to Rules $\textsc{Eval}_{\textsc{S}}$ and $\textsc{Eval}_{\textsc{L}}$.

Laziness is injected only by the bind operation of the *LazyT* transformer. This means that in order to be lazy, a computation in the underlying monad cannot be lifted globally but more deeply at each point of laziness. This is because the monad transformer law

$$lift\ (m \gg\!= f) \quad = \quad lift\ m \gg\!= lift \circ f$$

does not hold when considering the degree of laziness. Even if the two computations evaluate to the same value, the one on the right hand side is lazier.

To alleviate the burden for the programmer, we provide a special *deep* lift operation in case the underlying monad is constructed over the operational monad transformer.

```
liftLazyT :: ProgramT instr m a →
             LazyT (ProgramT instr m) a
liftLazyT (m `Bind` f) = liftLazyT m >>= liftLazyT ∘ f
liftLazyT (Instr a)    = lift $ Instr a
liftLazyT (Lift c)     = lift $ Lift c
```

This function guarantees that every algebraic operation is evaluated lazily.

Finally, to run a lazy computation, one needs to create a fresh empty trace and evaluate the computation in this trace.

```
runLazyT :: Delay m ⇒ LazyT m a → m a
runLazyT (LazyT c) = do trace ← liftIO createEmpty
                        runReaderT c trace
```

4.4 Turning algebraic operations into monadic computations

Every set of algebraic operations *instr* that instantiates *Reified* and *EvalInstr* naturally gives rise to an operational monad transformer that is an instance of *Evaluable*.

First, the operational monad transformer can be made an instance of *Reified*. The definitions of *isStrict* and *commute* on the *Instr* constructor are directly inherited from the definition on algebraic operations. The *Lift* and *Bind* constructors are made strict.

```
instance Reified instr ⇒
         Reified (ProgramT instr m) where
  isStrict (Instr a)          = isStrict a
  isStrict (Lift c)           = True
  isStrict (i `Bind` k)       = True
  commute (Instr a) (Instr a') = commute a a'
```

Then, evaluation of an algebraic operation *Instr a* is obtained by lifting (using the deep lift *liftLazyT*) the *evalInstr a* and evaluating it lazily. Finally, we enforce that *Lift* and *Bind* constructors can not have effect on *eff*.

```
instance (EvalInstr instr m, Delay m, Reified instr) ⇒
  Evaluable (ProgramT instr m) where
  eval (Instr a) = lazyEval (liftLazyT (evalInstr a))
  eval f         = lift f
  hasEffect (Lift c)     = False
  hasEffect (i `Bind` k) = False
  hasEffect (Instr c)    = True
```

4.5 Evaluation of Imperative Closures

The function *evalL* inserts at the beginning of the trace an imperative closure containing the computation to be delayed and it returns a delayed computation that gives a means to evaluate it later (using function *evalClo*).

```
evalL :: (Delay m, Evaluable m eff) ⇒
         m a → ReaderT (TraceElt m) m a
evalL c = do
```

```
elt ← readClo
newClo ← liftIO $ insertElt (Comp c) elt
(lift ∘ delay) $ runReaderT evalClo newClo
```

When a computation must finally be evaluated, its dependencies are evaluated by calling *evalDependencies* prior to its evaluation.

```
evalS :: (Delay m, Evaluable m eff) ⇒
  m a → ReaderT (TraceElt m) m a
evalS c = do
  when (hasEffect c) $
    evalDependencies $ (Comp c)
  eval c
```

Evaluating an imperative closure (function *evalClo*) and forcing evaluation of older dependent imperative closures (function *evalDependencies*) implements precisely the big-step semantics of Figure 3. The detailed code is presented in Appendix.

5 Optimization

Producer-consumer functional composition can be dynamically simplified. This requires the reification of closures. For instance, [Gill et al., 1993] proposes to (statically but also) dynamically simplify a *foldr-build* composition by reifying calls to *build* with an extra list constructor (a list is either nil, or cons, or build) and *foldr* has also three cases. Our laziness scheme reifies computations. This paves the way for similar dynamic simplifications in an imperative context.

Conceptually, the dependency relation *commute* used to compute which delayed operations must be performed is directly connected to commutativity equations between operations.

Unfortunately, this dependency relation appears quickly to be rigid. Consider for instance a third scenario

```
maximumLazy a size = do    -- similarly miminumLazy
  (-?-) $ QSortM a 0 (size − 1)
  (-?-) $ ReadArray a (size − 1)
thirdScenario size = lazify $ do
  a ← (-!-) $ newListArray (0, size − 1) [size, size − 1 .. 1]
  min ← minimumLazy a size
  max ← maximumLazy a size
  return (min, max)
```

where after computing the minimum, we compute the maximum by using a function that also sorts the array. Taking only dependency into account, the second sorting operation will trigger all the remaining sorting operation on subarrays of a, ending up in a completely sorted array, whereas only the left-most and right-most cell must be sorted.

The situation can be improved by considering that two overlapping sorting operations can be merged. To that end, the type class *Reified* is extended with a function *merge* that expects two computations and returns a new computation when those computations can be merged.

```
class Reified m where
    -- same as before for isStrict and commute
  merge :: m a → m b → Maybe (m b)
  merge _ _ = Nothing
```

The default instance does not merge any computation. Note that even if merging is defined for any computation of type $m\ a$ and $m\ b$, it does only make sense when $a = ()$. In this case, the merged closure is of type $m\ b$ and the result of type b is ignored when the call site is expecting for ().

This function *merge* allows new kind of equations, *e.g.* idempotence or overwriting. The semantics of the call-by-need calculus with lazy imperative operations can be extended to take merging into account. The conservativity result can be extended straightforwardly. For space reason, we have omitted the new rules of the big-step semantics, but the implementation of merging on the trace (function *mergeClo*) is presented in Appendix. In the implementation, the environment is managed using Haskell primitive binding mechanism so it is not possible to remove reference to former delayed computations that have been merged. Thus, conservativity for the implementation requires the additional assumption that no reference to former deleted delayed computation are evaluated.

We illustrate optimizations in the rest of the section with the case of *QSortM*, and of input/output operations in a file system.

Merging sorting operations. Two *QSortM* executions on the same array can be merged into a single *QSortM* execution without changing the result, because the function is idempotent. More generally, when one array contains the other, the execution of *QSortM* on the smaller array can be removed. Instantiating the *merge* function, this equation can be expressed as

```
merge c@(QSortM a from to) c'@(QSortM a' from' to') =
  if (c `subsumes` c' ∨ c' `subsumes` c)
  then Just $ QSort a (min from from') (max to to')
  else Nothing
where subsumes (QSortM a from to) (QSortM a' from' to')
  = from' ⩾ from ∧ to' ⩽ to
```

Figure 4 shows the impact of merging on the third scenario. The average execution time on a random set of 20 arrays of size between 10 and 100000 is printed. The optimized version performs better than both the strict and the unoptimized versions, taking coarsely twice the time of a lazy computation of the minimum (first scenario) whereas the two other versions perform a complete sort of the array.

Optimizing read and write in a file. Input/output operations on a disk are time consuming because of disk access latency. Therefore, there is a native buffer mechanism to merge small operations. Using dynamic laziness and optimization, a similar mechanism can be done in our framework. We exemplify it with operations on files defined in *System*.𝕀𝕆.

Read/Write operation are a classical example of algebraic operations. They are part of the class of input/output operations that can be modeled generically as:

```
type Partition region = [region]
data IOEffects region = IOEffects
  {inputE  :: Partition region,
   outputE :: Partition region}
instance Eq region ⇒ Effects (IOEffects region) where
  dependsOn e1 e2 =
    ¬ (null (inputE e1  `intersect` outputE e2)) ∨
    ¬ (null (outputE e1 `intersect` inputE e2))  ∨
    ¬ (null (outputE e1 `intersect` outputE e2))
```

The description is parametric with respect to the *region* decomposition of the structure on which I/O operations are performed. The effect of an operation is given by a couple of list of regions, one for regions that may be modified, one for regions that may be accessed. The dependency relation says that two operations commutes when they only access but never modify their common region.

Read, write and append IO operations on files can be reified as:

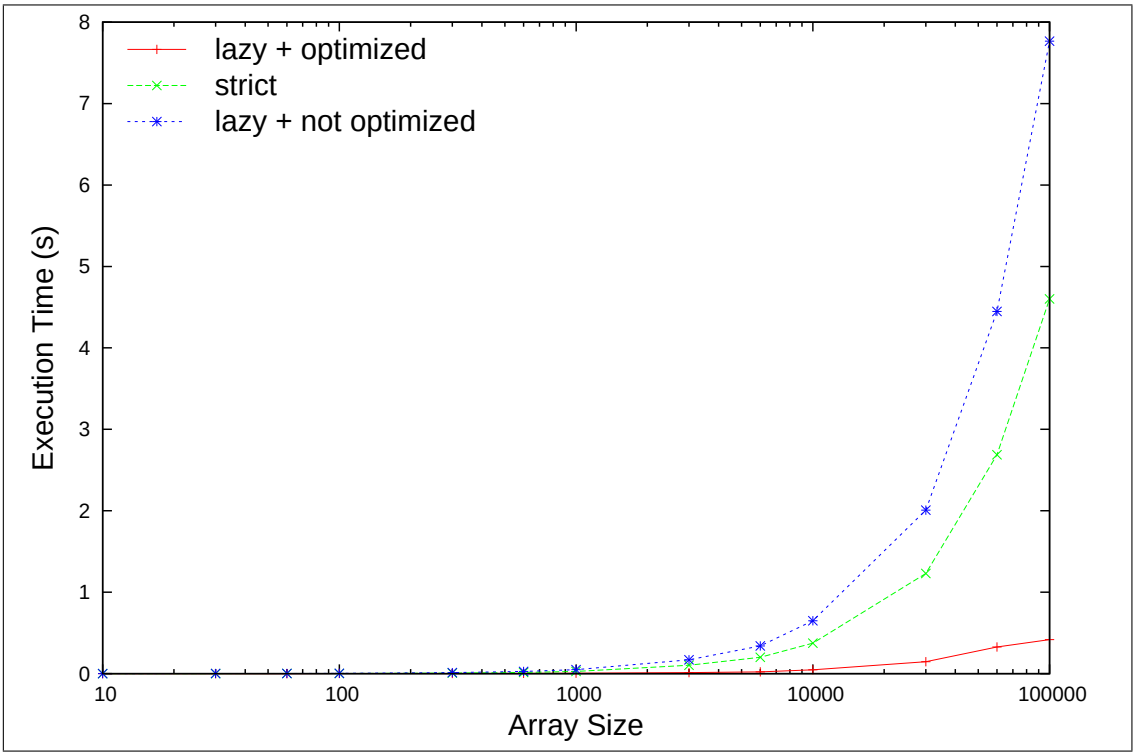


Figure 4: Impact of merging on the third scenario.

```
data RWFile a where
  ReadFile  :: FilePath → RWFile String
  WriteFile  :: FilePath → String → RWFile ()
  AppendFile :: FilePath → String → RWFile ()
  Flush      :: FilePath → RWFile ()
instance (MonadIO m) ⇒ EvalInstr RWFile m where
  evalInstr (ReadFile f)      = liftIO $ readFile f
  evalInstr (WriteFile f s)   = liftIO $ writeFile f s
  evalInstr (AppendFile f s) = liftIO $ appendFile f s
  evalInstr (Flush f)         = return ()
```

We also provide a *Flush f* operation whose purpose is to force remaining effects on f to be performed. *MonadIO* is a type class that guarantees that $\mathbb{IO}$ operations are available in the monad stack by using the *liftIO* function.

To simulate a writing buffer mechanism, we combine dynamic laziness (by making writing operation strict for big data) and optimization (by merging or overwriting small input operations).

```
instance Effectful RWFile (IOEffects FilePath) where
  effect (ReadFile f)      = IOEffects [f] []
  effect (WriteFile f s)   = IOEffects [] [f]
  effect (AppendFile f s) = IOEffects [] [f]
  effect (Flush f)         = IOEffects [] [f]
instance Reified RWFile where
  isStrict (ReadFile r)      = False
  isStrict (WriteFile f s)   = length s ⩾ 1000
  isStrict (AppendFile f s) = length s ⩾ 1000
```

```
isStrict (Flush f)          = True
commute c c' = ¬ $ (effect c) `dependsOn` (effect c')
merge (WriteFile f s) (WriteFile f' s') =
  maybeWhen (f ≡ f') $ WriteFile f s'
merge (AppendFile f s) (WriteFile f' s') =
  maybeWhen (f ≡ f') $ WriteFile f s'
merge (AppendFile f s) (AppendFile f' s') =
  maybeWhen (f ≡ f') $ AppendFile f (s ++ s')
merge (WriteFile f s) (AppendFile f' s') =
  maybeWhen (f ≡ f') $ WriteFile f (s ++ s')
merge _ _ = Nothing
```

where *maybeWhen* is syntactic sugar for

```
maybeWhen b x = if b then Just x else Nothing
```

To evaluate this optimization, we have compared the execution time of lazy and strict versions of a program that appends recursively to a file all the elements of a list. The test has been performed with lists of integers of different lengths.

```
recursiveWrite f [] = return ()
recursiveWrite f (x : xs) = do
  (-?-) $ AppendFile f (show x)
  recursiveWrite f xs
fileScenario length = lazify $ do
  liftIO $ writeFile "foo.tmp" ""
  recursiveWrite   "foo.tmp" [1 .. size]
  (-?-) $ Flush    "foo.tmp"
```

The lazy version appears to be slightly slower ($\sim 10\%$ for a list of size 100000) in the case of a fast direct access to the disk. To emulate a situation were the disk access is slower (for instance for a file on a distant machine), we have augmented the *appendFile* function with a delay of execution (using the *GHC.Conc* module)

```
appendFileDelay f s = do { threadDelay 50; appendFile f s }
```

With a delay of $50\mu s$, the lazy version already performs better ($\sim 20\%$ for a list of size 100000).

6 Discussions

In this article, we have proposed a framework that enables decoupling imperative producers and consumers: we specify effects of imperative blocks of code, delay their execution and dynamically detect dependencies between these blocks in order to preserve the original semantics when a delayed computation is eventually evaluated. Our approach can make classic sorting programs more efficient when a subset only of their results is required and this promotes imperative code reuse.

A prime candidate to apply our technique is foreign functions. Indeed, these functions have quite often an imperative semantics (they modify a state out of Haskell) and they can perform costly computations. For instance, consider a library of image filters. Pixel color information can be decomposed in hue, saturation, and lightness.[6] Some filters access or modify only a subset of the color components (for instance a threshold filter can generate a mask according to the lightness value only). Our framework would enable specifying color component effects of the different filters and complex composition of filters would be lazily executed according to the required color component dependencies.

[6] Other color systems provide different axes.

Finally, the correctness of our framework relies on effects definitions. These can be specified once and for all by an expert to produce a lazy version of an imperative library to be used by oblivious users. A recent work [Chang and Felleisen, 2014] equips a purely functional semantics with a metric that measures waste in an evaluation. This enables the implementatin of a profiling tool that suggests where to insert laziness. However, in an impure context improper laziness can change the semantics of programs. So, correction requires static analyses. For instance, it could be interesting to combine our approach with a type-and-effect system [Talpin and Jouvelot, 1994] in order to infer statically the *effect* function for algebraic operations. This way, it may be used to automatically and safely derive effects (for instance bounds of *QSortM*) from library codes.

Acknowledgments

The authors thank anonymous reviewers for their comments on a previous version of this article.

References

[Apfelmus, 2010] Apfelmus, H. (2010). The operational monad tutorial. *The Monad.Reader Issue 15*, page 37.

[Ariola and Sabry, 1998] Ariola, Z. M. and Sabry, A. (1998). Correctness of monadic state: An imperative call-by-need calculus. In *Proc. of Principles of Programming Languages*, pages 62–74. ACM.

[Chang and Felleisen, 2014] Chang, S. and Felleisen, M. (2014). Profiling for laziness. In *Proc. of the 41st Symp. on Principles of Programming Languages*, pages 349–360, New York, NY, USA. ACM.

[Gill et al., 1993] Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proc. of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA. ACM.

[Hughes, 1989] Hughes, J. (1989). Why functional programming matters. *Computer Journal*, 32(2):98–107.

[Kiselyov et al., 2012] Kiselyov, O., Jones, S. L. P., and Sabry, A. (2012). Lazy v. yield: Incremental, linear pretty-printing. In Jhala, R. and Igarashi, A., editors, *APLAS*, volume 7705 of *LNCS*, pages 190–206. Springer.

[Launchbury, 1993] Launchbury, J. (1993). Lazy imperative programming. In *Proc. of the ACM Sigplan Workshop on State in Programming Language*, pages 46–56.

[Lämmel and Jones, 2003] Lämmel, R. and Jones, S. P. (2003). Scrap your boilerplate: A practical design pattern for generic programming. In *Proc. of Types in Language Design and Implementation*, pages 26–37. ACM Press.

[Maraist et al., 1998] Maraist, J., Odersky, M., and Wadler, P. (1998). The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317.

[Plotkin and Power, 2001] Plotkin, G. and Power, J. (2001). Semantics for algebraic operations. In *Proc. of the 17th Mathematical Foundations of Programming Semantics, ENTCS*.

[Plotkin and Power, 2002] Plotkin, G. and Power, J. (2002). Notions of computation determine monads. In *Proc. FOSSACS 2002, Lecture Notes in Computer Science 2303*, pages 342–356. Springer.

[Talpin and Jouvelot, 1994] Talpin, J.-P. and Jouvelot, P. (1994). The type and effect discipline. *Information and Computation*, 111(2):245–296.

[Wadsworth, 1971] Wadsworth, C. P. (1971). *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, University of Oxford.

A Code of Doubly Linked Lists

```
module DLList where
import Data.IORef
data DLList a =
  Origin {future :: IORef (DLList a)} |
  Future {past   :: IORef (DLList a)} |
  DLList {past :: IORef (DLList a),
          elt  :: a,
          future :: IORef (DLList a)}
createEmpty :: IO (DLList a)
createEmpty = do
  rec iOrigin ← newIORef $ Origin iFuture
      iFuture ← newIORef $ Future iOrigin
  readIORef iFuture
removeElt :: DLList a → IO ()
removeElt (DLList iPastDLList iClo iFutureDLList) = do
  pastDLList ← readIORef iPastDLList
  futureDLList ← readIORef iFutureDLList
  writeIORef (future pastDLList) futureDLList
  writeIORef (past futureDLList) pastDLList
insertElt :: a → DLList a → IO (DLList a)
insertElt clo currentDLList = do
  let iPast = past currentDLList
    -- get the next older DLList
  pastDLList ← readIORef iPast
    -- create a new DLList
  iPastNew ← newIORef pastDLList
  iFutureNew ← newIORef currentDLList
  let newDLList = DLList iPastNew clo iFutureNew
    -- insert the new DLList
  writeIORef (past currentDLList) newDLList
  writeIORef (future pastDLList) newDLList
    -- return the new DLList
  return newDLList
commuteElt :: DLList a → IO ()
commuteElt l = do
  let iPast   = past l
  let iFuture = future l
  vPast ← readIORef iPast
  vFuture ← readIORef iFuture
  let iPast'  = past vPast
  let iFuture' = future vPast
  vPast' ← readIORef iPast'
  vFuture' ← readIORef iFuture'
  writeIORef (past l) vPast'
  writeIORef (past vPast) vFuture'
  writeIORef (future l) vPast
  writeIORef (future vPast) vFuture
endOfDL :: DLList a → Bool
endOfDL (Origin _) = True
endOfDL _ = False
```

B Code of the Lazy Monad Transformer

```
module LazyT where
```

```
import Data.List
import Data.Maybe
import DLList

import Data.IORef
import System.IO.Unsafe
import Unsafe.Coerce

import Control.Monad.Reader
import qualified Control.Monad.ST.Lazy as Lazy
import qualified Control.Monad.ST as Strict
import qualified Control.Monad.ST ∘ Lazy.Unsafe as LazyUnsafe
import qualified Control.Monad.ST.Unsafe as StrictUnsafe

import Data.Array.MArray
import GHC.Conc
  -- from operational monad transformer (could not directly import because of hidden constructors)
data ProgramT instr m a where
    Lift  :: m a → ProgramT instr m a
    Bind :: ProgramT instr m a → (a → ProgramT instr m b)
         → ProgramT instr m b
    Instr :: instr a → ProgramT instr m a
instance Monad m ⇒ Monad (ProgramT instr m) where
    return = Lift ∘ return
    (≫=) = Bind
instance MonadTrans (ProgramT instr) where
    lift   = Lift
  -- delays computation using unsafePerform (a generalization of unsafePerformIO)
class MonadIO m ⇒ Delay m where
    unsafePerform :: m a → a
    delay :: m a → m a
    delay = return ∘ unsafePerform
instance Delay IO where
    unsafePerform = unsafePerformIO
instance MonadIO (Lazy.ST s) where
    liftIO = LazyUnsafe.unsafeIOToST
instance Delay (Lazy.ST s) where
    unsafePerform = unsafePerformIO ∘ StrictUnsafe.unsafeSTToIO ∘ Lazy.lazyToStrictST
instance MonadIO (Strict.ST s) where
    liftIO = StrictUnsafe.unsafeIOToST
instance Delay (Strict.ST s) where
    unsafePerform = unsafePerformIO ∘ StrictUnsafe.unsafeSTToIO
  -- generic effect
class Effects eff where
  dependsOn :: eff → eff → Bool
  -- effectful monad
class Effects eff ⇒ Effectful m eff | m → eff where
  effect    :: m a → eff
class Reified m where
  isStrict  :: m a → Bool
  commute :: m a → m b → Bool
  merge     :: m a → m b → Maybe (m b)
  merge   _ _ = Nothing
class Reified m ⇒ Evaluable m where
  hasEffect :: m a → Bool
  eval      :: m a → ReaderT (SUC m) m a   --
  -- self updatable closure
```

```
  -- data Clo m where Comp :: m a -> Clo m
data Clo m where Comp :: m a → Clo m

type SUC m = DLList (Clo m)

mkSUC :: (Delay m, Evaluable m) ⇒
  Clo m → ReaderT (SUC m) m a
mkSUC clo = do
    currentSUC ← readSUC
      -- insert the closure in the trace
    newSUC ← liftIO $ insertElt clo currentSUC
      -- try to merge (recursively) the newly build closure with older closures in the trace
      -- withReaderT (_ -> newSUC) (mergeSUC newSUC)
      -- return a delayed computation that will evaluate and
      -- update the closure the first time it is accessed
    (lift ∘ delay) $ runReaderT evalSUC newSUC

mergeSUC :: (Delay m, Evaluable m) ⇒
            SUC m → ReaderT (SUC m) m ()
mergeSUC suc = do
    -- get suc' the next older closure of suc
  pastSUC ← readPastSUC
  when (¬ (endOfDL pastSUC)) $ do
       (Comp clo)  ← return $ elt suc
       (Comp clo′) ← return $ elt pastSUC
         -- if these two closures suc and suc' do not depends one on another,
         -- their evaluation is commutative
       if (clo `commute` clo′)
         -- so we can skip the next older closure suc' and try to merge suc
         -- with the next older closure of suc'
       then lift $ runReaderT (mergeSUC suc) pastSUC
         -- if these two closures suc and suc' depends one on another
       else let mergedComp = merge clo clo′ in
            when (isJust mergedComp) $ do
              newSUC ← liftIO $ insertElt (Comp $ fromJust mergedComp) suc
              liftIO $ removeElt suc
              liftIO $ removeElt pastSUC
                -- and remove the current closure suc from the trace
                -- (its is now merged and must occur only once in the trace)
                -- then try to merge the merged closure suc and suc'
                -- with the next older closure of suc'
              lift $ runReaderT (mergeSUC newSUC) newSUC

evalComp :: (Delay m, Evaluable m) ⇒
            m a → ReaderT (SUC m) m a
evalComp c = do
  when (hasEffect c) $
       evalDependencies (Comp c)
  eval c

  -- evalSUC uses unsafeCoerce to get a value of the right type
  -- from an existentially quantified value

evalSUC :: (Delay m, Evaluable m) ⇒ ReaderT (SUC m) m a
evalSUC = do
    -- access the current value of the self-updatable closure
  suc  ← readSUC
  myComp ← return $ elt suc
  case myComp of
    Comp c → do
        -- first evaluate older SUCs on which the current closure depends on
      when (hasEffect c) $ do
           evalDependencies $ myComp
             -- evalDependenciesClo myComp
        -- then evaluate the current closure
      a ← eval c
```

```
        -- remove it from the trace (for it is now evaluated)
      liftIO $ removeElt suc
        -- and return its value
      return $ unsafeCoerce a
  -- Compute the dependencies in the past trace
evalDependenciesClo :: (Delay m, Evaluable m) ⇒ Clo m → ReaderT (SUC m) m ()
evalDependenciesClo clo = do
  suc ← readSUC
  when (¬ (endOfDL suc)) $ do
      (Comp c) ← return $ clo
      (Comp c') ← return $ elt suc
      if commute c c'
      then do psuc ← readPastSUC
          withReaderT (\_ → psuc)
            (evalDependenciesClo clo)
      else do evalSUC
          currentSUC ← readSUC
          newSUC ← liftIO $ insertElt clo currentSUC
          withReaderT (\_ → newSUC) (mergeSUC newSUC)
          evalDependenciesClo clo
evalDependencies :: (Delay m, Evaluable m) ⇒ Clo m → ReaderT (SUC m) m ()
evalDependencies c = do
  pastSUC ← readPastSUC
  when (¬ (endOfDL pastSUC)) $ do
      (Comp clo) ← return c
      (Comp clo') ← return $ elt pastSUC
      if clo `commute` clo'
      then do withReaderT (\_ → pastSUC)
            (evalDependencies c)
      else do withReaderT (\_ → pastSUC) evalSUC
          evalDependencies c
  -- read effects on ReaderT (SUC m eff)
readSUC :: MonadIO m ⇒ ReaderT (SUC m) m (SUC m)
readSUC = ReaderT $ λsuc → return $ suc
readPastSUC :: MonadIO m ⇒ ReaderT (SUC m) m (SUC m)
readPastSUC = ReaderT $ λsuc → liftIO $ readIORef $ past suc
  -- The lazy imperative monad : lazyEval with a threaded trace state
data LazyT m a = LazyT (ReaderT (SUC m) m a)
lazyEval :: ∀eff m a.(Delay m, Evaluable m) ⇒
            LazyT m a → ReaderT (SUC m) m a
lazyEval (LazyT c) = do suc ← readSUC
  let c' = runReaderT c suc
  if isStrict c'
  then evalComp c'
  else mkSUC (Comp c')
instance (Delay m, Evaluable m) ⇒ Monad (LazyT m) where
  return v = LazyT $ return v
  x >>= k  = LazyT $ do y ← lazyEval x
              lazyEval (k y)
instance (Delay m, Evaluable m) ⇒ MonadIO (LazyT m) where
  liftIO = LazyT ∘ liftIO
instance MonadTrans LazyT where
  lift = LazyT ∘ lift
instance (Delay m, Evaluable m) ⇒ Delay (LazyT m) where
  unsafePerform = unsafePerform ∘ runLazyT
runLazyT :: Delay m ⇒ LazyT m a → m a
runLazyT (LazyT c) = do
    -- create a fresh suc for this thread
```

```
  suc ← liftIO createEmpty
  runReaderT c suc

lazify :: (EvalInstr instr m, Reified instr, Delay m) ⇒
          ProgramT instr m a → ProgramT instr m a
lazify = runLazyT ∘ liftLazyT

liftLazyT :: (EvalInstr instr m, Reified instr, Delay m) ⇒
             ProgramT instr m a → LazyT (ProgramT instr m) a
liftLazyT (c `Bind` k) = liftLazyT c ≫= liftLazyT ∘ k
liftLazyT (Instr a) = lift $ Instr a
liftLazyT (Lift c)  = lift $ Lift c

  -- An operational monad transformer with explicit effects

class EvalInstr instr m where
  evalInstr :: instr a → ProgramT instr m a

instance (MonadIO m) ⇒ MonadIO (ProgramT instr m) where
    liftIO = lift ∘ liftIO

instance Reified instr ⇒ Reified (ProgramT instr m) where
    isStrict (Instr a) = isStrict a
    isStrict _         = True

    commute (Instr a) (Instr a') = a `commute` a'

    merge (Instr a) (Instr a') = maybe Nothing (Just ∘ Instr)
      $ (merge a a')
    merge _ _ = Nothing

instance (EvalInstr instr m, Delay m, Reified instr) ⇒
         Evaluable (ProgramT instr m) where

    eval (Instr a)  = lazyEval $ liftLazyT $ evalInstr a
    eval f          = lift f

    hasEffect (Lift c)    = False
    hasEffect (i `Bind` k) = False
    hasEffect (Instr c)   = True

runProgramT :: (Delay m, EvalInstr instr m) ⇒ ProgramT instr m a → m a
runProgramT (Instr x)    = runProgramT (evalInstr x)
runProgramT (Lift c)     = c
runProgramT (c `Bind` k) = runProgramT c ≫= runProgramT ∘ k

(−!−) :: (Monad m) ⇒ m a → ProgramT instr m a
(−!−) = lift

(−?−) :: (Monad m) ⇒ instr a → ProgramT instr m a
(−?−) = Instr

instance (Delay m, EvalInstr instr m) ⇒ Delay (ProgramT instr m) where
    unsafePerform = unsafePerform ∘ runProgramT

  -- Range effect

data Bounds arrayOf = Bounds (arrayOf Int Int) Int Int

instance Eq (arrayOf Int Int) ⇒ Effects (Bounds arrayOf) where
  dependsOn (Bounds a1 lb1 ub1) (Bounds a2 lb2 ub2) = a1 ≡ a2 ∧ lb2 ⩽ ub1 ∧ lb1 ⩽ ub2

  -- Bubble sort example

data ArrayC arrayOf a where
    QSort :: arrayOf Int Int → Int → Int → ArrayC arrayOf ()
    BSort :: arrayOf Int Int → Int → Int → ArrayC arrayOf ()
    ReadArray :: arrayOf Int Int → Int → ArrayC arrayOf Int

instance Eq (arrayOf Int Int) ⇒ Effectful (ArrayC arrayOf) (Bounds arrayOf) where

    effect (BSort    a from to) = Bounds a from to
    effect (QSort    a from to) = Bounds a from to
    effect (ReadArray a i) = Bounds a i i

instance Eq (arrayOf Int Int) ⇒ Reified (ArrayC arrayOf) where

    c `commute` c' = ¬ (effect c `dependsOn` effect c')
```

```
    merge c@(BSort a from to) c'@(BSort a' from' to') =
            if (¬ (effect c `dependsOn` effect c'))
            then Nothing
            else Just $ BSort a (min from from') (max to to')
    merge c@(QSort a from to) c'@(QSort a' from' to') =
            if (c `contains` c' ∨ c' `contains` c)
            then Just $ QSort a (min from from') (max to to')
            else Nothing
      where contains (QSort a from to) (QSort a' from' to')
              = from' ⩾ from ∧ to' ⩽ to
    merge _ _ = Nothing

    isStrict (BSort a from to) = from ⩾ to
    isStrict (QSort a from to) = False   -- to-from < 10^3
    isStrict (ReadArray a i) = True
instance (MArray arrayOf Int m) ⇒ EvalInstr (ArrayC arrayOf) m where
    evalInstr (BSort a from to) = when (from < to) $ do
      (−!−) $ aBubble a from to
      (−?−) $ BSort a (from + 1) to
    evalInstr (QSort a from to) = do
      when (from < to) $ do
        let pivot = from + div (to − from) 2
        pivot' ← (−!−) $ partitionM a from to pivot
        (−?−) $ QSort a from (pivot' − 1)
        (−?−) $ QSort a (pivot' + 1) to
    evalInstr (ReadArray a i)  = (−!−) $ readArray a i

bSort a from to | from ≡ to = return ()
                | otherwise = do
  aBubble a from to
  bSort a (from + 1) to

aBubble a from to | from ≡ to = return ()
aBubble a from to = do
  bs ← getBounds a
  x1 ← readArray a $ to − 1
  x2 ← readArray a $ to
  when (x2 < x1) $ do writeArray a (to − 1) x2
    writeArray a to x1
  aBubble a from (to − 1)

qSort array left right = do
  when (left < right) $ do
    let pivot = left + div (right − left) 2
    pivot' ← partitionM array left right pivot
    qSort array left (pivot' − 1)
    qSort array (pivot' + 1) right

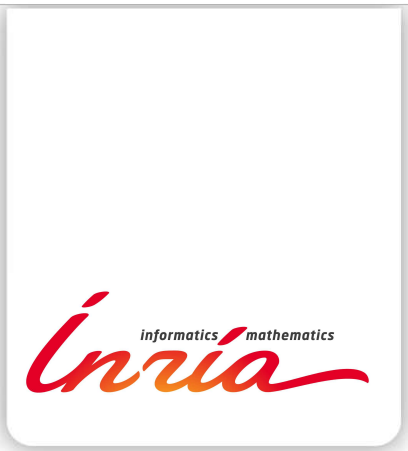
lQSort a from to = (−?−) $ QSort a from to
lBSort a from to = (−?−) $ BSort a from to
lReadArray a i   = (−?−) $ ReadArray a i

sQSort a from to = (−!−) $ qSort a from to
sBSort a from to = (−!−) $ bSort a from to
sReadArray a i   = (−!−) $ readArray a i

swap a i j = do
  iVal ← readArray a i
  jVal ← readArray a j
  writeArray a i jVal
  writeArray a j iVal

partitionM a left right pivot = do
  pivotVal ← readArray a pivot
  swap a pivot right
  store ← partitionM' a pivotVal left (right − 1) left
  swap a store right
```

```
    return store
    where
      partitionM' a p i to store | i > to = return store
      partitionM' a p i to store = do
        iVal ← readArray a i
        if (iVal < p) then do {swap a i store; partitionM' a p (i + 1) to (store + 1)}
                      else partitionM' a p (i + 1) to store
    -- Other examples ——
    -- Read/Write effects
type Partition a = [a]
data IOEffects cell = IOEffects {
        readE :: Partition cell,
        writeE :: Partition cell}
instance Eq e ⇒ Effects (IOEffects e) where
    dependsOn e1 e2 =
      ¬ (null (readE  e1 `intersect` writeE e2)) ∨
      ¬ (null (writeE e1 `intersect` readE e2)) ∨
      ¬ (null (writeE e1 `intersect` writeE e2))
data RWFile a where
      ReadFile   :: FilePath → RWFile String
      WriteFile  :: FilePath → String → RWFile ()
      AppendFile :: FilePath → String → RWFile ()
      Flush      :: FilePath → RWFile ()
instance (MonadIO m) ⇒ EvalInstr RWFile m where
      evalInstr (ReadFile f)    = liftIO $ readFile f
      evalInstr (WriteFile f s) = liftIO $ writeFile f s
      evalInstr (AppendFile f s) = liftIO $ appendFileDelay f s
      evalInstr (Flush f)       = return ()
maybeWhen :: Bool → a → Maybe a
maybeWhen b x = if b then Just x else Nothing
instance Effectful RWFile (IOEffects FilePath) where
      effect (ReadFile f)    = IOEffects [f] []
      effect (WriteFile f s) = IOEffects [] [f]
      effect (AppendFile f s) = IOEffects [] [f]
      effect (Flush f)       = IOEffects [] [f]
instance Reified RWFile where
      isStrict (ReadFile r)    = False
      isStrict (WriteFile f s) = length s ⩾ 10^3
      isStrict (AppendFile f s) = length s ⩾ 10^4
      isStrict (Flush f)       = True
      commute c c' = ¬ $ (effect c) `dependsOn` (effect c')
      merge (WriteFile f s) (WriteFile f' s') =
          maybeWhen (f ≡ f') $ WriteFile f s'
      merge (AppendFile f s) (WriteFile f' s') =
          maybeWhen (f ≡ f') $ WriteFile f s'
      merge (AppendFile f s) (AppendFile f' s') =
          maybeWhen (f ≡ f') $ AppendFile f (s ++ s')
      merge (WriteFile f s) (AppendFile f' s') =
          maybeWhen (f ≡ f') $ WriteFile f (s ++ s')
      merge _ _ = Nothing
appendFileDelay f s = do {threadDelay 50; appendFile f s}
```



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399