Purely Functional Incremental Computing

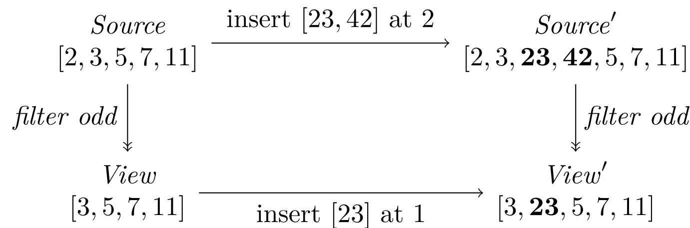
Denis Firsov and Wolfgang Jeltsch

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
{denis, wolfgang}@cs.ioc.ee

Abstract Many applications have to maintain evolving data sources as well as views on these sources. If sources change, the corresponding views have to be adapted. Complete recomputation of views is typically too expensive. An alternative is to convert source changes into view changes and apply these to the views. This is the key idea of incremental computing. In this paper, we use Haskell to develop an incremental computing framework. We illustrate the concepts behind this framework by implementing several example computations on sequences. Our framework allows the user to implement incremental computations using arbitrary monad families that encapsulate mutable state. This makes it possible to use highly efficient algorithms for core computations.

1 Introduction

Incremental computing is an approach to efficiently updating a view of some data source whenever the data source changes. For an explanation, let us look at the following diagram:



Initially, the source is the list $[2, 3, 5, 7, 11]$. We create a view of the source, defined as the list of odd numbers in the source, which is $[3, 5, 7, 11]$ initially. Next, we change the source by inserting the numbers 23 and 42 at index 2, resulting in $[2, 3, 23, 42, 5, 7, 11]$. We expect the view to adapt to the new source, that is, to become $[3, 23, 5, 7, 11]$. This can be done by fully recomputing the view from the source. However, a more efficient method is to turn the source change "insert $[23, 42]$ at 2" into a view change "insert $[23]$ at 1" and apply the latter to the view.

The most important trait of different approaches to incremental computing is the amount of automation they provide. One of the strongest achievements

in the field of incremental computing is the approach of self-adjusting computation developed by Acar [1]. Here, any function is incrementalized automatically using dependency tracking. However, the downside of full automation is that it provides less control over time and space complexity. For example, the trivial accumulator-based implementation of the *reverse* function requires linear time for change propagation when incrementalized automatically. One can achieve change propagation in logarithmic time by implementing *reverse* using a divide-and-conquer strategy. However, it is generally hard to come up with a function definition that results in efficient change propagation.

In this paper, we present a framework for incremental computing. This framework makes it possible to efficiently implement core computations using carefully crafted algorithms, and then build more complex computations from them by means of easy-to-use combinators. Furthermore, our framework offers composability at the type level, allowing notions of change for complex types to be derived from notions of change for simpler types. To illustrate our framework, we use sequences as our running example. We make the following contributions:

- In Sect. 2, we describe an interface to changeable values and associated changes.
- In Sect. 3, we introduce the notion of transformation. A transformation maps a source to a view. It allows for efficient updates of the view by propagating changes of the source to the view. (An example of a transformation is the *filter odd* in the above diagram.)
- In Sect. 4, we develop transformations that may use pure state to propagate changes. As a result, we can equip a wider range of operations with change propagation.
- In Sect. 5, we show that for some transformations efficient change propagation requires mutable state. We characterize a class of monad families that can embed different kinds of mutable state into pure computations. We generalize transformations such that they can use arbitrary monad families from this class.

The remaining sections are devoted to related work, conclusions, and further work.

Our developments use the Haskell programming language and are compatible with the Glasgow Haskell Compiler (GHC), version 7.8.3[1]. They are available as the Cabal package *incremental-computing* [6].

2 Changes and Changeables

The central notion of our framework is the change. A change describes a modification of values. Changes are typically implemented using algebraic data types. This way, they can be inspected during change propagation. We define a type class *Change* of all types of changes:

[1] Unfortunately, our code does not work with GHC 7.10 because of a bug in this version of the compiler. However, this bug will be fixed in the upcoming GHC 7.12.

```
class Change p where
  type Value p :: *
  ($$) :: p → Value p → Value p
```

Each type p of changes has an associated type *Value* p of values on which the changes can act. The $$-operator denotes change application. We can see a partial application (*change* $$) :: *Value* p → *Value* p as the meaning of *change*.

A change type may optionally be an instance of the *Monoid* class, in which case ε denotes the identity change, and $change_2 \bullet change_1$ denotes the change that consists of $change_1$ followed by $change_2$.

For any type of values, there is a primitive notion of change, where a change is either keeping the current value or replacing the current value by a new value:

```
data PrimitiveChange a = Keep | ReplaceBy a
instance Monoid (PrimitiveChange a) where
  ε = Keep
  Keep          • change = change
  ReplaceBy val • _      = ReplaceBy val
instance Change (PrimitiveChange a) where
  type Value (PrimitiveChange a) = a
  Keep          $$ val = val
  ReplaceBy val $$ _   = val
```

Each value type can have an arbitrary number of change types. However, we allow to specify a single notion of change as the default for a value type. We introduce a class *Changeable* of all value types with a default change type:

```
class (Monoid (DefaultChange a), Change (DefaultChange a),
       Value (DefaultChange a) ~ a) ⇒ Changeable a where
  type DefaultChange a :: *
  type DefaultChange a = PrimitiveChange a
```

As the code specifies, default changes have to form a monoid. If an instance declaration does not provide a declaration for *DefaultChange*, primitive changes are used as the default notion of change. Primitive changes are appropriate for primitive types, like *Bool* and *Integer*; so we can instantiate *Changeable* for primitive types easily:

```
instance Changeable Bool
instance Changeable Integer
```

In this paper, we want to illustrate the concepts of our framework taking lists as a running example. However, some operations on standard Haskell lists are inefficient. As a solution, the module *Data.Sequence* from the *containers* package provides a type *Seq* with operations that mostly run in $O(\log n)$ time. In particular, splitting a sequence at a given index and concatenating two sequences

takes $\Theta(\log n)$ time. So we base our illustration on the *Seq* type. However, we will still use list syntax in examples to avoid notational clutter.

First, we define a type *AtomicChange* whose elements are changes of sequences:

```
data AtomicChange a = Insert Int (Seq a)
                    | Delete Int Int
                    | Shift Int Int Int
                    | ChangeAt Int (DefaultChange a)
```

The semantics of changes and the time complexity of change application to sequences of size n are as follows:

- *Insert ix seq* inserts *seq* at index *ix*. It takes $O(\log n + |seq|)$ time.
- *Delete ix len* deletes the part of length *len* that starts at index *ix*. It takes $O(\log n)$ time.
- *Shift src len tgt* shifts the part of length *len* that starts at index *src* to index *tgt*. It takes $O(\log n)$ time.
  Applying *Shift src len tgt* is actually equivalent to first applying *Delete src len* and then *Insert tgt seq* where *seq* is the deleted part. We provide *Shift* nevertheless, because in certain situations, change propagation for *Shift* can be done more efficiently than change propagation for the corresponding *Delete*–*Insert* chain.
- *ChangeAt ix elemChange* applies *elemChange* to the element at index *ix*. It takes $O(\log n + m)$ time where m is the time cost for applying *elemChange* to the element.

Note that not all changes are applicable to a given sequence. For example, a change *Insert ix seq* can only be applied to a sequence of length *len* if $0 \leqslant ix \leqslant len$. In the *incremental-computing* package, we properly deal with this issue, but in this paper, we ignore it for the sake of simplicity.

We want to use lists of atomic changes as the default changes of sequences. This way, default sequence changes form a monoid. We introduce a type *MultiChange* such that *MultiChange a* is essentially $[a]$, but differs from it in the following points:

- Concatenation takes only $O(1)$ time, which is achieved by using difference lists.
- The monoid operator $\bullet$ is concatenation with arguments swapped to accommodate the above-mentioned argument order of change composition.
- For every instance p of *Change*, *MultiChange p* is an instance of *Change* as well (with an obvious implementation).

We instantiate *Changeable* for sequence types as follows:

```
instance Changeable a => Changeable (Seq a) where
  type DefaultChange (Seq a) = MultiChange (AtomicChange a)
```

For convenience, we define variants of *Insert*, *Delete*, *Shift*, and *ChangeAt*, called *insert*, *delete*, *shift*, and *changeAt*, that construct singleton multi changes instead of atomic changes.

3 Transformations without State

Transformations are functions equipped with means for change propagation. For very simple cases, a transformation can be seen as a pair of two functions: one that maps source values to view values and one that maps source changes to view changes:

$$\textbf{data}\ \mathit{Trans}\ p\ q = \mathit{Trans}\ (\mathit{Value}\ p \rightarrow \mathit{Value}\ q)\ (p \rightarrow q)$$

For transformations that work with default changes, we provide a convenience type alias:

$$\textbf{type}\ a \twoheadrightarrow b = \mathit{Trans}\ (\mathit{DefaultChange}\ a)\ (\mathit{DefaultChange}\ b)$$

As an example, we present a map combinator for sequences that works with transformations instead of functions. First, we implement a version of this combinator that only propagates atomic changes:

```
atomicMap :: (Changeable a, Changeable b)
           ⇒ (a ↠ b) → Trans (AtomicChange a) (AtomicChange b)
atomicMap (Trans elemFun elemProp) = Trans fun prop where
  fun = fmap elemFun
  prop (Insert ix seq)           = Insert ix (fmap elemFun seq)
  prop (Delete ix len)           = Delete ix len
  prop (Shift src len tgt)       = Shift src len tgt
  prop (ChangeAt ix elemChange) = ChangeAt ix (elemProp elemChange)
```

Since the default changes of sequences are multi changes, we now develop a combinator *map* for sequences that works with multi changes. Recall that multi changes are essentially just lists of changes. We introduce a function *MultiChange.map* of type

$$\mathit{Trans}\ p\ q \rightarrow \mathit{Trans}\ (\mathit{MultiChange}\ p)\ (\mathit{MultiChange}\ q)$$

that keeps the function on values and lifts the function on changes to a function on multi changes. The definition of *map* becomes simple now:

```
map :: (Changeable a, Changeable b) ⇒ (a ↠ b) → (Seq a ↠ Seq b)
map trans = MultiChange.map (atomicMap trans)
```

4 Transformations with Pure State

In many cases, we need additional information about the current source in order to propagate changes. An example is the *reverse* transformation. A source change *Insert ix seq*, for example, must be turned into the view change *Insert* (*len* − *ix*) (*reverse seq*) where *len* is the current length of the source. So the change propagator needs to know this length.

To remedy this problem, we extend the *Trans* type such that a transformation can use a state to track information about the source:

$$\textbf{data}\ \mathit{Trans}\ p\ q = \forall s\ .\ \mathit{Trans}\ (\mathit{Value}\ p \to (\mathit{Value}\ q, s))\ (p \to s \to (q, s))$$

The type s is the type of the state. Every value of a type *Trans p q* can use its own type s, which does not show up as a parameter of *Trans*. In a transformation *Trans init prop*, the function *init* turns an initial source to the corresponding view and the initial state, and the function *prop* turns a source change and a current state into the corresponding view change and the updated state. Note that *prop* is a computation in the state monad.

We can still represent transformations without state by setting the type s to (). However, we cannot use the previous implementation of *map* anymore, since the argument of type $a \twoheadrightarrow b$ that *map* receives has the new, more complex, structure. Nevertheless, it is possible to implement *map* for transformations with pure state. The only difficulty is the propagation of *ChangeAt* changes. To propagate a change of the form *ChangeAt ix elemChange*, we have to propagate *elemChange*. This requires access to the state of the element at index *ix*. For this reason, we store the sequence of all element states as the state of the result transformation of *map*.

Another example of a transformation that requires state is *concat*, which flattens a sequence of sequences. To propagate changes, *concat* needs to translate indexes and lengths that refer to the nested source sequence into indexes and lengths that refer to the flattened view sequence. For this, it needs to know the lengths of the elements of the source. The *concat* transformation stores these as its state.

Having *concat*, we can implement a *filter* combinator with only little effort. First, we implement a helper combinator

$$\mathit{gate} :: \mathit{Changeable}\ a \Rightarrow (a \twoheadrightarrow \mathit{Bool}) \to (a \twoheadrightarrow \mathit{Seq}\ a)\ \ ,$$

which is also useful in other contexts. The view of a transformation *gate prd* is the empty sequence whenever *prd* yields false for the source; otherwise, it is the singleton sequence containing just the source. Figure 1 shows an example run for *gate odd*. Note that the source is of type *Integer* and thus uses changes of type *PrimitiveChange Integer*, while the view is of type *Seq Integer* and thus uses changes of type *MultiChange* (*AtomicChange Integer*). We can now implement *filter* easily by composing *map*, *concat*, and *gate* appropriately:

$$\begin{aligned}&\mathit{filter} :: \mathit{Changeable}\ a \Rightarrow (a \twoheadrightarrow \mathit{Bool}) \to (\mathit{Seq}\ a \twoheadrightarrow \mathit{Seq}\ a)\\&\mathit{filter}\ \mathit{prd} = \mathit{concat} \circ \mathit{map}\ (\mathit{gate}\ \mathit{prd})\end{aligned}$$

The given implementation uses a composition operator $\circ$ of type $\mathit{Trans}\ q\ r \to \mathit{Trans}\ p\ q \to \mathit{Trans}\ p\ r$, which definition is straightforward. The example from the introduction illustrates the use of *filter*.

5 Transformations with Mutable State

The goal of incremental computing is to make views adapt quickly to source changes. In some cases, transformations with only pure state are not capable of

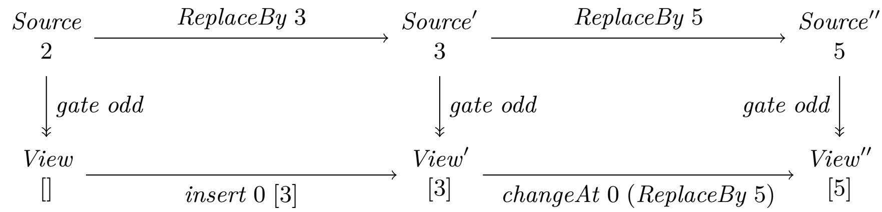


Figure 1. Example run for *gate odd*

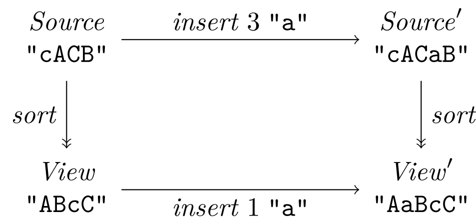
propagating changes with optimal time complexity. In Subsect. 5.1, we sketch an efficient solution to incremental stable sorting that relies on mutable state. Next, in Subsect. 5.2, we develop the notion of monadic transformation, which makes it possible to use mutable state in incremental computations. Finally, in Subsect. 5.3, we discuss how transformation combinators can be implemented safely in the presence of mutable state.

5.1 Incremental Stable Sorting

A sorting algorithm is stable if it retains the relative order of elements that are considered equivalent by the comparison function. Stability is especially important in an incremental setting, as it prevents equivalent elements from changing their relative order during application of unrelated changes.

There are several solutions to incremental sorting. Acar [1] presents a randomized merge sort that he incrementalizes using self-adjusting computation. With this approach, change propagation takes logarithmic expected time for single-element insertions and deletions, and sorting is stable. Furthermore, Acar et al. [2] describe a cleverly crafted heapsort implementation, for which self-adjusting computation provides single-element change propagation in logarithmic worst-case time. Unfortunately, the use of heapsort makes sorting unstable. We overcome the tradeoff between these two approaches by implementing incremental stable sorting with logarithmic worst-case time for single-element change propagation. In this subsection, we describe the main ideas behind our implementation.

Let us first discuss incremental unstable sorting. As an example, we want to look at sequences of letters. We assume that letters are ordered according to their position in the alphabet without taking case into account. The following diagram shows an example of change propagation:



The generated view change *insert* 1 `"a"` is also appropriate for stable sorting. Unstable sorting, however, additionally permits the view change *insert* 0 `"a"`, which leads to the updated view `"aABcC"`.

The crucial part of change propagation for unstable sorting is the translation of source indexes into view indexes. To facilitate this translation, we maintain the sorted sequence as the state of the sorting transformation. We use a search tree data structure for it, so that we can find the view index of a newly inserted element or an element to be deleted in logarithmic worst-case time.

Now let us try to turn this incremental unstable approach into an approach to incremental stable sorting. It is well known how to perform non-incremental stable sorting based on an unstable sorting algorithm. First, the elements of the unsorted sequence are tagged with their indexes. Afterwards, the resulting sequence of element–index pairs is sorted lexicographically. Finally, the indexes are dropped from the sorted sequence.

We cannot adapt this approach directly to incremental sorting. When propagating an insertion, we must come up with tags for the new elements that lie between the tags of the existing elements. For an explanation, let us look at the above diagram again. We first tag the initial source `"cACB"` with indexes and get the sequence $[(\texttt{'c'}, 0), (\texttt{'A'}, 1), (\texttt{'C'}, 2), (\texttt{'B'}, 3)]$. Then, we sort this sequence lexicographically, obtaining $[(\texttt{'A'}, 1), (\texttt{'B'}, 3), (\texttt{'c'}, 0), (\texttt{'C'}, 2)]$. For propagating the change *insert* 3 `"a"`, we have to create a tag that lies between the tags of `'C'` and `'B'`. We could use rational numbers as tags, so that the new tag could be 2.5. However with this approach, tag comparison would be linear in the worst case. Retagging the source sequence is also not an option, as it would take linear time in the worst case as well.

We solve the tagging issue by employing a solution to the order maintenance problem. In the order maintenance problem, the objective is to maintain a total order of tags subject to insertions, deletions, and tag comparison. Dietz and Sleator [5] show how to achieve constant worst-case time for all these operations. By using their solution, we are able to create tags between existing tags efficiently and still avoid linear time complexity for tag comparison. We keep the tagged sorted sequence in the form of a search tree. In addition, we maintain the tagged unsorted sequence, so that we can generate new tags based on the tags of neighboring elements.

Let us illustrate this with our running example. The initial source `"cACB"` leads to a tagged unsorted sequence $[(\texttt{'c'}, t_0), (\texttt{'A'}, t_1), (\texttt{'C'}, t_2), (\texttt{'B'}, t_3)]$ with $t_0 < t_1 < t_2 < t_3$ and the tagged sorted sequence $[(\texttt{'A'}, t_1), (\texttt{'B'}, t_3), (\texttt{'c'}, t_0), (\texttt{'C'}, t_2)]$, which together constitute the initial state. For propagating the change *insert* 3 `"a"`, we first use the tagged unsorted sequence to find the neighboring tags of the new element. We use order maintenance insertion to create a new tag t' between those tags, so that $t_2 < t' < t_3$. We insert the pair $(\texttt{'a'}, t')$ into the tagged sorted sequence, leading to $[(\texttt{'A'}, t_1), (\texttt{'a'}, t'), (\texttt{'B'}, t_3), (\texttt{'c'}, t_0), (\texttt{'C'}, t_2)]$. The index of this pair in the updated tagged sorted sequence is the view insertion index.

The crux is that the order maintenance solution by Dietz and Sleator relies on mutable state. So transformations with pure state are not powerful enough to

implement the above incremental stable sorting strategy. Therefore, we extend our notion of transformation once more to allow for state to be mutable.

5.2 Monadic Transformations

Haskell provides the ST type to implement computations that can work with mutable variables internally, but can still be used in a pure setting [9]. ST takes a phantom type parameter s and an ordinary type parameter a, where s represents a heap of mutable variables that the computation can access, and a is the result type of the computation. There is a function $runST :: (\forall s\ .\ ST\ s\ a) \to a$ that turns an ST computation into a pure value. The use of universal quantification ensures that a computation can only work with its own, private heap, so that state mutations cannot be observed from the outside. $ST\ s$ is a monad family indexed by s in the sense that for every particular s, $ST\ s$ is a monad.

We redefine *Trans* based on ST to enable transformations to use mutable state:

$$\textbf{newtype}\ \mathit{Trans}\ p\ q = \mathit{Trans}\ (\forall s\ .\ \mathit{Value}\ p \to ST\ s\ (\mathit{Value}\ q, p \to ST\ s\ q))$$

We represent a transformation by a computation that takes an initial source, sets up the initial state, and returns the initial view and a propagator. The propagator, in turn, is a computation that turns a source change into a view change. It can access the state that the initializer has set up, because it is created inside the initializer. Note that we can still express all transformations with pure state using this new definition of *Trans*, since we can store a pure state in a mutable variable.

The ST-based definition of *Trans* allows for arbitrary transformations with mutable state. However, it restricts code reuse. For example, there is an implementation of order maintenance in the form of the *order-maintenance* Cabal package [7], but we cannot use this package to implement a stable sorting transformation. To see why, we have to take a closer look at the interface of this package.

The *order-maintenance* package provides a type $OrderT$ for computations that have access to a mutable totally ordered set. $OrderT$ takes type parameters o, m, and a, where o is a phantom parameter that represents the ordered set, m is an inner monad, which provides additional effects, and a is the result type of the computation. There is a function

$$evalOrderT :: Monad\ m \Rightarrow (\forall o\ .\ OrderT\ o\ m\ a) \to m\ a$$

that turns an $OrderT$ computation into a computation in the inner monad. The use of universal quantification here is analogous to its use in $runST$. It ensures that a computation can only work with its own, private ordered set. For every monad m, $OrderT\ o\ m$ is a monad family indexed by o.

The *order-maintenance* package does not allow us to incorporate an $OrderT$ computation into an ST computation, as this would make the ordered set explicitly accessible via mutable variables and thus break the abstraction barrier. Therefore,

we cannot make use of the *order-maintenance* package with the above ST-based definition of *Trans*.

If we had a variant of *Trans* based on the monad family $\mathit{OrderT}\ o\ (ST\ s)$ with indexes o and s, we could use *order-maintenance* for implementing incremental stable sorting. The OrderT layer would provide us with a mutable totally ordered set for holding the tags, and the ST layer would provide us with a heap for storing the remaining state. Instead of providing a *Trans* variant for this particular monad family, we generalize *Trans* such that we can use every monad family that has the following properties:

- It is indexed by an arbitrary number of phantom type parameters that appear at arbitrary positions in the type.
- It comes with an evaluation function, that is, a function that turns a computation in the monad family into a pure value, using universal quantification for all the index parameters to keep mutable state private. (For $ST\ s$, this function is runST, and for $\mathit{OrderT}\ o\ (ST\ s)$, it is $\mathit{runST} \circ \mathit{evalOrderT}$.)

We introduce a type alias *TransProc* whose definition resembles the ST-based definition of *Trans*, but allows us to work in an arbitrary monad:

$$\textbf{type}\ \mathit{TransProc}\ m\ p\ q = \mathit{Value}\ p \to m\ (\mathit{Value}\ q, p \to m\ q)$$

We call a value of *TransProc* a transformation processor. We can represent a transformation by a value of a type $\forall \overline{w}\ .\ \mathit{TransProc}\ \mu\ p\ q$ where μ is a monad family with indexes $\overline{w}$. As special cases, we get $\forall s\ .\ \mathit{TransProc}\ (ST\ s)\ p\ q$, which corresponds to the ST-based *Trans*, and $\forall o\ s\ .\ \mathit{TransProc}\ (\mathit{OrderT}\ o\ (ST\ s))\ p\ q$, which is appropriate for implementing incremental stable sorting.

It would be straightforward to define *Trans* as a transformation processor, existentially quantifying the monad family μ. However, this would result in the following problems:

1. Since the number of indexes and the index positions depend on μ, we would have to bundle indexes as type tuples. This would require users of our framework to write considerable amounts of boilerplate code, and would require support for data type promotion, which is not a well-established language extension.
2. Since transformation processors can use different monad families, composition of transformation processors would be hard to implement.

Therefore, we represent a transformation by a pure function:

$$\textbf{newtype}\ \mathit{Trans}\ p\ q = \mathit{Trans}\ ((\mathit{Value}\ p, [p]) \to (\mathit{Value}\ q, [q]))$$

The representation of a transformation captures its behavior by turning any pair of an initial source value and a list of successive source changes into a corresponding pair of an initial view value and a list of successive view changes. In practice, we typically cannot provide the initial source value and all the source changes at once, since they only become available over time. However, we can obtain parts of the output based on only parts of the input by employing laziness.

Note that there are pure functions of the above-mentioned type that are not proper representations of transformations, for example, functions where a view change depends on future source changes. Therefore, we do not export the data constructor *Trans*. Instead, we introduce a function *trans* that constructs a transformation based on a given transformation processor.

Besides the transformation processor, the *trans* function needs to know the evaluation function of the monad family of the transformation processor. So it would be best if *trans* had the type

$$\forall \mu \,.\, (\forall \overline{w} \,.\, \mathit{TransProc}\ \mu\ p\ q) \rightarrow (\forall r \,.\, (\forall \overline{w} \,.\, \mu\ r) \rightarrow r) \rightarrow \mathit{Trans}\ p\ q \ \ .$$

Unfortunately, the use of universal quantification over monad families would involve a problem similar to Problem 1 described above. Therefore, we modify the interface of *trans* step by step until *trans* has a type that does not involve universal quantification over monad families. Some of our modifications make the interface more permissive for the user, but none of them makes it less permissive. Our modification steps are as follows:

1. We switch to continuation-passing style for the transformation processor, using $\mu\ r$ with an arbitrary r as the result type of the continuation. As a consequence, the first argument of *trans* has the type

$$\forall r \,.\, \forall \overline{w} \,.\, (\mathit{TransProc}\ \mu\ p\ q \rightarrow \mu\ r) \rightarrow \mu\ r \ \ .$$

2. We push the quantification $\forall \overline{w}$ under the arrow, so that the type of the first argument becomes

$$\forall r \,.\, (\forall \overline{w} \,.\, \mathit{TransProc}\ \mu\ p\ q \rightarrow \mu\ r) \rightarrow (\forall \overline{w} \,.\, \mu\ r) \ \ .$$

3. We merge the two arguments into one that is supposed to be the composition of the former two arguments. The type of this single argument is clearly

$$\forall r \,.\, (\forall \overline{w} \,.\, \mathit{TransProc}\ \mu\ p\ q \rightarrow \mu\ r) \rightarrow r \ \ .$$

4. We generalize the type of the continuation such that it covers all monads. The type of the *trans* argument becomes

$$\forall r \,.\, (\forall m \,.\, \mathit{Monad}\ m \Rightarrow \mathit{TransProc}\ m\ p\ q \rightarrow m\ r) \rightarrow r \ \ .$$

5. We drop the universal quantification of μ, which is not needed anymore, as there are no more uses of μ. Now *trans* has the type

$$(\forall r \,.\, (\forall m \,.\, \mathit{Monad}\ m \Rightarrow \mathit{TransProc}\ m\ p\ q \rightarrow m\ r) \rightarrow r) \rightarrow \mathit{Trans}\ p\ q \ \ .$$

Let us look how to construct a transformation from a transformation processor *transProc* and an evaluation function *eval*. First, we turn *transProc* into continuation-passing style, which results in the function $\lambda \mathit{cont} \rightarrow \mathit{cont}\ \mathit{transProc}$. Then, we compose this function with the *eval* function, leading to $\lambda \mathit{cont} \rightarrow \mathit{eval}\ (\mathit{cont}\ \mathit{transProc})$. Finally, we apply *trans* to this composed function.

By applying this technique to the monad family $\mathit{ST}\ s$, we can define a function *stTrans* that turns an *ST*-based transformation processor into a value of type *Trans*:

$$stTrans :: (\forall s\ .\ TransProc\ (ST\ s)\ p\ q) \rightarrow Trans\ p\ q$$
$$stTrans\ transProc = trans\ (\lambda cont \rightarrow runST\ (cont\ transProc))$$

We conclude this subsection with the presentation of the *trans* function implementation:

```
trans :: (∀r . (∀m . Monad m ⇒ TransProc m p q → m r) → r) → Trans p q
trans cpsProcAndEval = Trans conv where
  conv src = cpsProcAndEval $ λtransProc → monadicConv transProc src
  monadicConv transProc ~(val, changes) = do
    ~(val′, prop) ← transProc val
    changes′ ← mapM prop changes
    return (val′, changes′)
```

Note that the interface changes to *trans* have not prevented us from generating the pure function representation, despite them making the interface more permissive.

5.3 Transformation Combinators

The use of the pure function representation for transformations becomes a challenge for the implementation of transformation combinators, that is, functions that construct new transformations from existing ones. Examples of transformation combinators are *map* and *gate*, which are described in Sect. 3 and Sect. 4, respectively. Change propagation of a combinator's result may involve change propagation of this combinator's arguments. For example, when a transformation *map elemTrans* propagates a sequence change of the form *ChangeAt ix elemChange*, it has to use *elemTrans* to propagate *elemChange*.

A transformation combinator cannot directly use the *Trans* data constructor to construct its result, since *Trans* is private; it has to invoke the *trans* function instead. Therefore, the combinator must represent its result by a transformation processor, which it can feed to *trans*. In particular, it must implement change propagation via a propagator whose type has the form $p \rightarrow \mu\ q$. Such a propagator is called with one change at a time. To propagate a given change, it may need to propagate individual changes using the arguments of the combinator. The problem is that the arguments are represented by pure functions that take all their source changes at once and therefore cannot propagate changes individually.

As a solution, we develop a function *toSTProc* that turns a transformation into an *ST*-based transformation processor. Using *toSTProc*, a transformation combinator can obtain transformation processors for all its arguments and use them to individually propagate changes.

A transformation processor *toSTProc* (*Trans conv*) has to apply *conv* to the pair of the initial source and the list of all source changes in order to receive the initial view and the view changes. So it has to provide the list of all source changes immediately, although the source changes become known only by later propagator calls. To resolve this conflict, we let the propagator put the source changes into a channel and let the transformation processor construct a single lazy list of all future channel elements when it is initially invoked. For this purpose,

we implement channel support for the ST monad family. This support is inspired by the *Control.Concurrent.Chan* module, which works with the IO monad.

We introduce a type *Channel* such that a value of a type *Channel* s a is a channel that works with the monad $ST\ s$ and contains elements of type a. Furthermore, we provide a function $newChannel :: ST\ s\ (Channel\ s\ a, [a])$ that creates an empty channel and returns it together with the lazy list of its future elements, and a function $writeChannel :: Channel\ s\ a \to a \to ST\ s\ ()$ that puts an element into a channel.

We are now able to convert transformations into ST-based transformation processors:

```
toSTProc :: Trans p q → TransProc (ST s) p q
toSTProc (Trans conv) val = do
  (chan, changes) ← newChannel
  let (val′, changes′) = conv (val, changes)
  remainderRef ← newSTRef changes′
  let prop change = do
        writeChannel chan change
        next : further ← readSTRef remainderRef
        writeSTRef remainderRef further
        return next
  return (val′, prop)
```

A transformation processor constructed by *toSTProc* creates a channel for the source changes and passes its contents together with the initial source to the pure function that represents the given transformation. This way, the transformation processor obtains the initial view and a lazy list of future view changes. The propagator puts the given source change into the channel and fetches the corresponding view change from the list of view changes. For this purpose, the suffix of the view change list that contains the future view changes is kept in a mutable variable.

6 Related Work

Approaches to incremental computing differ by the amount of automation they provide. Generally, automation of change propagation relieves the programmer from manual design of propagation algorithms and the obligation to prove that these algorithms are correct. On the other hand, automation restricts control over incrementalization strategies, which may result in suboptimal time complexity.

We have tried to find a middle ground between automation and potential for manual intervention. A user of our framework can manually define notions of change for core data types and implement core transformations by specifically crafted algorithms. On the other hand, our framework offers composability of transformations and types of changeable data, allowing the construction of complex incremental programs from the hand-crafted building blocks.

Cai et al. [3] follow a similar approach. Like us, they allow the user to define change types and change propagation algorithms for core data types. Based on these, they can incrementalize arbitrary λ-terms by means of a static transformation. In one respect, their automation goes further than ours, since it can handle higher-order programs. In another respect, however, their automation is more restrictive, because it uses only transformations with a pure state that reflects the current source value. We conjecture that this restriction necessarily results in change propagation with suboptimal time complexity for some transformations, for example, incremental stable sorting.

Substantial contributions to the field of fully automatic incremental computing are due to Acar [1]. His key approach is executing an ordinary program in an incremental fashion by maintaining a dynamic dependency graph. Based on this idea, Acar has developed the technique of self-adjusting computation. This method allows a user to write a function in the usual way and then have it incrementalized automatically by the compiler. Unfortunately, this level of automation makes it complicated to analyze the complexity of change propagation. A user who is not satisfied with the result of automatic incrementalization has little clue about how to change the implementation of his function to make it perform better when incrementalized. For example, the typical accumulator-based implementation of the *reverse* function requires linear time for change propagation. One can achieve logarithmic time by implementing *reverse* using a divide-and-conquer strategy, but this is not obvious for the lesser experienced user.

Carlsson [4] has implemented adaptive functional programming, a subset of self-adjusting computation, in Haskell. The key contribution of his work is the use of monads for integrating incremental computing into a pure language.

Self-adjusting computation does not perform well in the presence of certain reuse patterns, particularly sharing (using a computation in different contexts), swapping (changing the order of subcomputations), and switching (toggling computations back and forth). As a solution, Hammer et al. [8] have developed the $\lambda_{\mathrm{ic}}^{\mathrm{cdd}}$-calculus and the Adapton library, which provide automatic incremental computing based on a demand-driven semantics.

Maier et al. [10] have developed the *Scala.React* framework, which supports functional incremental reactive lists. The authors use the notions of reversible and associative folds, which are usual folds with some additional constraints on their arguments. These folds can be used for implementing new incremental functions. Obtaining incremental operations on reactive lists means translating the linear recursion of sequential folds into tree recursion of associative folds.

7 Conclusions and Further Work

We have developed a framework for incremental computing in Haskell. This framework allows the user to associate different notions of change with different data types and implement change propagation based on arbitrary monad families whose computations can be turned into pure values. Furthermore, we have implemented incremental versions of several sequence operations. The user of

our framework has clear guidelines on how to implement new notions of change, transformations, and transformation combinators. By applying the described techniques, it is possible to achieve complex incremental computing behavior.

In the future, we want to develop a generic notion of change for inductive data types and use it to define generic transformations based on recursion schemes. We expect that general recursion schemes cannot be efficiently incrementalized. However, we plan to characterize recursion schemes that allow for efficient change propagation. All functions that are defined in terms of these recursion schemes can then be efficiently incrementalized automatically.

Acknowledgements

We want to thank Umut Acar, Yan Chen, Paolo Giarrusso, and Tarmo Uustalu for helpful discussions about the topics of this paper. This research was supported by the individual research grant PUT763 of the Estonian Research Council.

References

1. Acar, U.A.: Self-Adjusting Computation. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania (May 2005)
2. Acar, U.A., Blelloch, G., Ley-Wild, R., Tangwongsan, K., Turkoglu, D.: Traceable data types for self-adjusting computation. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10). pp. 483–496. ACM, New York (2010)
3. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 145–155. ACM, New York (2014)
4. Carlsson, M.: Monads for incremental computing. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. pp. 26–35. ACM, New York (2002)
5. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. Tech. Rep. CMU-CS-88-113, Carnegie Mellon University, Pittsburgh, Pennsylvania (1988)
6. Firsov, D., Jeltsch, W.: *incremental-computing-0.0.0.0* (Feb 2015), `http://hackage.haskell.org/package/incremental-computing-0.0.0.0`, Haskell Cabal package
7. Firsov, D., Jeltsch, W.: *order-maintenance-0.1.1.0* (Nov 2015), `http://hackage.haskell.org/package/order-maintenance-0.1.1.0`, Haskell Cabal package
8. Hammer, M.A., Phang, K.Y., Hicks, M., Foster, J.S.: Adapton: Composable, demand-driven incremental computation. In: Proceedings of the 35st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14). pp. 156–166. ACM, New York (2014)
9. Launchbury, J., Peyton Jones, S.: State in Haskell. LISP and Symbolic Computation 8(4), 293–341 (Dec 1995)
10. Maier, I., Odersky, M.: Higher-order reactive programming with incremental lists. In: Castagna, G. (ed.) Object-Oriented Programming, Lecture Notes in Computer Science, vol. 7920, pp. 707–731. Springer (2013)