An Experimental Implementation of Self-adjusting Bidirectional Transformations

Huu-Phuc Vo, Hiroyuki Kato, Soichiro Hidaka, Zhenjiang Hu

Bidirectional transformations provide a novel mechanism for synchronizing and maintaining the consistency of information between source and view. Despite many advantages, bidirectional transformations are limited in achieving incremental computing that concerns maintaining the source-view relationship of a program as the source of a program is changed. Recent work on incremental computing such as self-adjusting computation developed useful techniques for writing programs which can adapt their view to any small or large change in the source. In this article, we realize that bidirectional transformations on trees can be made more efficient by using self-adjusting techniques. The underlying systems of forward and backward transformations are represented with dynamic dependence graphs that record the data dependencies and control dependencies. When the source or the view is modified, a change propagation algorithm will update the connected view or source, and the dynamic dependence graphs by propagating the changes through the graphs and re-executing code where necessary. The idea is to use forward and backward dynamic dependence graphs to identify and re-execute the parts of the computation that are affected by modifications on the source or the view. We refer to this approach as self-adjusting bidirectional transformations and show that it is practical and efficient.

1 INTRODUCTION

Bidirectional transformations [13][20] provide a novel mechanism for synchronizing and maintaining the consistency of information between source and view. Bidirectional transformations can be used in many interesting applications, such as the well-known view updating mechanism which has been intensively studied in the database community [10][15][21][22][28], interactive user interface design [32], coupled software transformation [27], the synchronization of replicated data in different formats [20], and presentation-oriented structured document development [26]. Bidirectional graph transformation approaches can be distinguished at least according to two different sorts [13]: reversible graph transformation languages and truly bidirectional graph transformation languages. *Triple Graph Grammars (TGGs)* [37][38] are in the class of bidirectional transformation languages [14]. *TGGs* define a language of related pairs of graphs and derive pairs of uni-directional forward and backward transformations.

Despite many advantages, bidirectional transformations are limited in achieving incremental computing that concerns maintaining the source-view relationship of a program as the source of a program is changed. In order to maintain and synchronize the consistency of source and view, a bidirectional transformation consists of a pair of a forward and a backward transformation in the typical

An Experimental Implementation of Self-adjusting Bidirectional Transformations.

Huu-Phuc Vo, Dept. of Informatics, The Graduate University for Advanced Studies, Japan.

Hiroyuki Kato, Soichiro Hidaka, Zhenjiang Hu, National Institute of Informatics, Tokyo, Japan.

setting, the changes will be updated to the view or reflected back to the source by completely re-computing the forward or backward transformation when the source or view is changed by updates. In such cases, the forward and backward transformations are re-computed from scratch.

Lenses are one the most popular approaches to define bidirectional transformations between data models, and have been proposed by Foster *et al.* [20] as a linguistic solution to the view-update problem. Every lens program can be read as a function mapping sources to views as well as one mapping updated views back to updated sources. Edit lenses are one of the few approaches that operate directly on edits and work with descriptions of changes to structures rather than with the structures themselves [24].

With this approach, no matter how small or large modification is applied to the source, the forward and backward transformations are re-computed from scratch. Given some source data, the forward and backward transformations, without any modifications to the view, transformations are computed in the initial computation. When the source or the view is modified, the only way to update the modifications to the view or reflect them back to the source is to re-compute from scratch without reusing the previous computations.

Recent work on incremental computing such as self-adjusting computation [5] developed useful techniques for writing unidirectional programs which can adapt their view to any small or large change in the source. The self-adjusting technique is useful in circumstances where a small change in the source will lead to a small change in the view. In some restricted cases, the small changes from the source can not avoid a complete re-computation of the view; but in most cases, the previous computations can be reused to produce the view with the relative changes from the source. The self-adjusting technique is used for programs which can respond to the modifications to their source data by updating the dynamic dependence graphs (*DDGs*) and carrying out the change propagation. *DDGs* are graphs that represent the relationships between the data and control dependencies. They can be built as the program executes and used to update the computation and the view when necessary.

A complete list of references on incremental computation can be found in the bibliography of Ramalingam and Reps [35]. The most effective techniques to solve the incremental computation problem are based on dependence graphs, memoization, and partial memoization [4]. Different techniques for incremental updates exist such as static dependence graphs (Demers et al. [16] and Reps [36]; Hoover [25]) and memoization (Pugh and Teitelbaum [34]; Bellman [11]; McCarthy [31]; Michie [33]) which can apply to any pure functional program. Building on Pugh and Teitelbaum's work, applications of various forms of memoization to incremental computation have been investigated [1]; Liu et al. [29]; Heydon et al. [23] [7].

A partial evaluation approach to incremental computation that requires users to fix the partition of the source on which the program will be specialized is introduced by Sundaresh and Hudak [42]. The limitation of this approach is that the modifications to the source must only within a predetermined partition. In the context of lambda calculus, Field and Teitelbaum have presented formal reduction systems which optimally use partially evaluated results [19] [18]. Incremental computation has also been studied in artificial-intelligence and logic-programming communities with general-purpose techniques (Stallman and Sussman [40]; Doyle [17]; McAllester [30]).

In this paper, we present the idea of self-adjusting lenses which applies the self-adjusting technique to bidirectional transformations. We follow linguis-

tic approach by building lenses with both a forward function and a backward function, with an underlying system of the self-adjusting computation. This system uses the dynamic dependence graphs to track the controls and data dependencies in an execution, combine with memoization by remembering function calls and their results. The modifications to the source or the view of bidirectional programs are translated one at a time. An additional challenging is how to update the dynamic dependence graphs and dually propagate the changes from source and view. This work is one step to experimentally combine the self-adjusting technique and lenses as well as to introduce the methods to construct self-adjusting lenses, that are practical and efficienct, and includes a test suite and experiments with self-adjusting lenses.

***Outline*:** In this paper, we realize that bidirectional transformations on trees can work more efficiently by using self-adjusting techniques. We model the bidirectional transformations on tree-structured data as lenses [20] which map the concrete tree into a simplified abstract view in one direction, and map the modified abstract view and the original concrete tree to a modified concrete tree in the appropriate direction. We give an introduction to bidirectional transformations, and the incremental computing problem in Section 1. We state preliminary result of adaptive programming and self-adjusting computations for the incremental computing problem and describe lenses which will be used for experiments and evaluation in Section 2. In Section 3, we talk about the self-adjusting lenses, and the method to transform the pure lenses to self-adjusting lenses. We show the implementation, experiments and the evaluation in Section 4. In section 5, we present related work on bidirectional transformations and self-adjusting computation. We conclude the paper and discuss the results in Section 6.

2 PRELIMINARY

2. 1 Self-adjusting computation

Self-adjusting computation has been applied to various application domains, including dynamic algorithms, motion simulations, machine learning, and incremental invariant checking [39]. These applications confirm that the self-adjusting approach can be effective for a broad range of practical applications.

In the adaptive functional programming and self-adjusting approaches, the key point is a *modifiable reference* (or *modifiable* in short), which holds the data that can change overtime. A constraint over modifiables is that they are not be written no more than once within the self-adjusting program.

Dynamic dependence graphs: The underlying of forward and backward transformations are represented with *dynamic dependence graphs* (*DDGs*) that record data dependencies and control dependencies. When the source or the view is modified, a change propagation algorithm will update the *DDGs* and the view by propagating the changes through the graphs and re-executing code where necessary. The idea is to use forward and backward *DDGs* to identify and re-execute the parts of the computation that are affected by modifications on source or view. The dynamic dependence graphs is described more precisely in [9]. Dynamic dependence graphs can be viewed as a representation of the data and control dependencies. *DDGs* of a program can be built as it executes by tracking the executed operations, and used to update the computation and the view when the sources are modified. The *core primitive* of self-adjusting programs for operating on changeable data are: the *mod* operation that creates a modifiable, the *read* and the *write* operations that provide access to the modifiable. The dynamic dependence graphs are used to identify and re-execute the parts of computations

which are affected by modifications.

Structure of dynamic dependence graph: The *DDGs* of computations can be represented as a dynamic function call tree with nodes and edges. The nodes of the *DDGs* represent function calls. The edges are relations between computation data and function calls that depend on them and represent the caller-callee relationship between function calls as well. Edges between computation data (i.e., sources, intermediate results) and function calls represent data dependencies: if a function call reads a piece of data, then there is an edge from such data to that call. With this structure, the *DDGs* can monitor source data, function calls and the relations between them.

Change-propagation algorithm: With the given dynamic dependence graphs of a program and set of modified source modifiables. The change-propagation can track the controls and data dependencies in an execution of a program. The idea of the change-propagation is to re-evaluate the reads which are affected by the source modifications in the sequential execution order.

Self-adjusting program: Given a purely functional program, the *DDGs* can be constructed by executing it with the source. The self-adjusting program is defined when the source is changed during the computation, the computation and view will be updated by performing the change propagation through the dynamic dependence graphs.

2. 2 Lenses

In this section, we introduce several lenses such as the *identity lens, the head lens, the tail lens, the filter lens, the hoist lens*, and *the plunge lens.* Those lenses have all been defined by Nate Foster *et al.* [20].

First, we phrase basic lens definitions which are specified together with a type of the form $C \rightleftharpoons A$, where C is a set of concrete source structures and A is a set of abstract view structures. Second, the notion of well-behavedness is defined so that it can convey how the *get* and *putback* parts of a lens should behave in concert. The forward and backward evaluations should satisfy the bidirectional properties. Such properties are called PutGet and GetPut laws in the lens framework. For example, if we use *get* to extract an abstract view a from a concrete view c and then use the putback part to push the same a back into c, we should get c back. The meaning of the notations $l \nearrow$ and $l \searrow$ is that the get part of a lens lifts an abstract view out of a concrete one, while the putback part pushes down a new abstract view into an existing concrete view. When f is a partial function, we write $f(a) \sqsubseteq b$ if f is not defined on argument a or $f(a) = b$.

Well-behaved lenses: Let l be a lens, C and A be subsets of fixed set V. We say that l is a well-behaved lens from C to A, written $l \in C \rightleftharpoons A$, if it maps arguments in C to results in A and vice versa.

$$l \nearrow(C) \subseteq A \qquad \text{(GET)}$$
$$l \searrow(A \times C) \subseteq C \qquad \text{(PUT)}$$

and its *get* and *putback* functions obey the following laws:

(GETPUT)

$$l \searrow(l \nearrow c,c) \sqsubseteq c \quad \text{for all } c \in C$$

(PUTGET)

$$l \nearrow(l \searrow(a,c)) \sqsubseteq a \quad \text{for all } (a,c) \in A \times C$$

In this article we represent lists as trees by using a standard cons-cell encoding, and introduce some derived lenses which are used for making self-adjusting lenses. Then we define and implement those lenses. There are many different lenses such as generic lenses (i.e. identity, composition, conditional, and recursion), structure manipulation lenses which modify the shape of the tree near the root (i.e. hoist, plunge), tree navigation lenses which apply different lenses to different parts of the tree, or one lens deeper in the tree (i.e. map, fil-

ter), “database-like” lenses (i.e. flatten, join), and structure replication lenses (i.e. merge, copy).

In practice, there will be cases when we want to apply the *putback* function but the concrete views (or source) is not available. In this case, a placeholder Ω, pronounced missing, handles missing concrete views. For example, $l \searrow (a, \Omega)$ means that "lens l creates a new source from the information in the abstract view (or view)"

Identity lens: Identity lens is the simplest one. The identity lens is a bijective lens from any set to itself. The concrete view c and the abstract view will be copied in the get direction and and in the putback direction, respectively.

表 1 Identity lens

$\mathrm{id}\nearrow c$	$=$	c
$\mathrm{id}\searrow(a, c)$	$=$	a
$\forall C \subseteq V.id \in C \overset{\Omega}{\Longleftrightarrow} C$		

Head and tail projections: This is very simple lenses for projecting the head and tail of the list. The first list lenses extract the head or tail of a cons cell list. The lens *hd* expects a default tree, which it uses in the *putback* direction as the tail of the created tree when the concrete tree is missing; the *get* direction, it returns the tree under *h. The lens *tl* works analogously.

表 2 Head tail projections

$\mathrm{hd}\ d = \mathrm{focus}\ *h\{*t \mapsto d\}$
$\forall C, D \subseteq T.\forall d \in D.\ \mathrm{hd}\ d \in (C :: D) \overset{\Omega}{\Longleftrightarrow} C$
$\mathrm{tl}\ d = \mathrm{focus}\ *t\{*h \mapsto d\}$
$\forall C, D \subseteq T.\forall d \in C.\ \mathrm{tl}\ d \in (C :: D) \overset{\Omega}{\Longleftrightarrow} D$

Filter lens: Given a predicate p and the source, in the *get* direction, this lens produces the view by keeping all the elements of the source that satisfy the predicate p. On the other hand, the putback direction, the putback function of filter lens takes the view and the source, then it restores the filtered part of the source. In case the source is missing, the putback function restores the source and propagates the changes made to the view using Ω.

表 3 Filter lens

$\mathrm{filter}\ pd = \mathrm{fork}\ p\ \mathrm{id}\ (\mathrm{const}\ \{\}d)$
$\forall C \subseteq T.\forall p \subseteq N.\forall d \in C\backslash_p.$ $\mathrm{filter}\ pd \in (C\ |_p\ .C\backslash_p) \overset{\Omega}{\Longleftrightarrow} C\ |_p$

Hoist lens: The hoist lens is used to shorten a tree by removing an edge at the top. In the get direction, given a edge n, it remove the edge n and returns the child of n. On the other hand, the putback function restores the edge n, the value of old source is ignored and a new one is created with the edge n pointing to the given view.

表 4 Hoist lens

$(\mathrm{hoist}\ n)\nearrow c$	$=$	$c(n)$
$(\mathrm{hoist}\ n)\searrow(a, c)$	$=$	$\{n \mapsto a\}$
$\forall C \subseteq T.\forall n \in N.\ \mathrm{hoist}\ n \in \{n \mapsto C\} \overset{\Omega}{\Longleftrightarrow} C$		

Plunge lens: Conversely, the plunge lens is used to deepen a tree by adding an edge n at the top of the tree. In the get direction, a single edge n points to the source tree to form a new tree. In the putback direction, the value of a source tree is ignored and the result of the plunge is the view tree which has exactly one subtree.

表 5 Plunge lens

$(\mathrm{plunge}\ n)\nearrow c$	$=$	$\{n \mapsto c\}$
$(\mathrm{plunge}\ n)\searrow(a, c)$	$=$	$a(n)$
$\forall C \subseteq T.\forall n \in N.\ \mathrm{plunge}\ n \in C \overset{\Omega}{\Longleftrightarrow} \{n \mapsto C\}$		

3 SELF-ADJUSTING LENSES

In previous sections, we introduce the self-adjusting technique which is very useful for the

incremental problem when the modifications on source leads to relative changes from view, as well as the lenses. In this section, we present a brief overview of the self-adjusting lenses. The idea for self-adjusting lenses is to apply the self-adjusting technique to both the get direction and the putback direction in bidirectional transformation. The underlying system of the self-adjusting computation is using the dynamic dependence graphs to track the controls and data dependencies in an execution combining with the memoization by remembering function calls and their results.

3. 1 The Interface

We present the self-adjusting library which is implemented from Acar *et al.* work [8]. The implementation of the self-adjusting library is used for encoding the self-adjusting lenses and testing them with several sources as well as modifications. All the implementations are written in OCaml programming language.

First, in order to show the practical use of the self-adjusting lenses, we implement identity, head, tail, hoist, plunge, and filter lenses as self-adjusting lenses by using the self-adjusting library. Second, we test and compare the pure functional lenses and self-adjusting lenses to present the efficiency of the self-adjusting lenses with several test suite. The source data that will be the source for the test suites is encoded as cons cell list of integer.

3. 2 Transformation

The lenses are implemented as pure functional programs which are written in Ocaml. Then we transform those pure functional programs into self-adjusting programs by using the Self-adjusting library which is re-implemented in Ocaml (the original self-adjusting library is implemented in SML). This transformation consists of two steps.

First, we need to determine what parts of the source data will be changed over time and place it into modifiable references. By doing this, the source will be monitored to any changes. Second, we specify the read of the modifiables from the original programs. We place the results of each expression into modifiable. The new modifiables which hold all the changeable data in the computation will be created because the reads are change computation.

After the initial run, the source data can be changed and then the program need to update its view by performing change propagation. The source modification process can be repeated as many times as desired. The effectiveness of the dynamic dependence graphs and the change-propagation in the self-adjusting lens vary from the position of modifications. If the modifications take place at the beginning of source, the change propagation will require a from scratch execution. If the new element is insert into the source list at the end or in the middle, the change propagation will take $O(\log\ n)$ or $O(n)$ expected time, respectively.

3. 3 Encoding of Self-adjusting Lenses

For the self-adjusting lenses, one of the difficulties is how to monitor the modifications to both source and view. Given a pure functional lens, we first apply a two step transformation to get the self-adjusting lens.

In the forward transformation, the source of the self-adjusting lens is monitored as the changeable source by placing it into the modifiables. As the forward transformation of self-adjusting lens executes, the underlying system represents the data and control dependencies in the execution via a dynamic dependence graphs. When the source to the lens changes, a change-propagation algorithm updates the dependence graphs and the view by propagating the modifications through the dynamic dependence graphs and re-executing the computa-

tions where necessary.

In the backward transformation, the original source and the view of the self-adjusting lens are used to produce the new modified source. In this case, suppose that no modifications are made to the view, the *putback* function executes to produce the same source. In such direction, we consider the modified view and the original source as the source and the modified source is the view.

The problem is how to represent the dynamic dependence graphs and the change-propagation for both forward and backward transformations so that with any modifications to the source or view, they can be reflected back to the source or updated to the view.

With the self-adjusting approach Acar *et al.* 2009 [4], we can separately transform a pure functional lens to self-adjusting lens. The main idea to solve this problem is to represent two dynamic dependence graphs, one for the forward transformation and another for the backward transformation. Given the lens and the source, the forward dynamic dependence graphs (F*DDGs*) represent the source of the self-adjusting lens as source. In the other hand, without any modifications to the view, we represent the original source and the view of the lens as the backward dynamic dependence graphs (B*DDGs*) to monitor the changes. In the first case, when the source of the lens is modified, the F*DDGs* are updated and then propagate the changes to the view. Because the view is updated from the source modifications, the B*DDGs* need to be updated from the modified source and modified view in the forward transformation. In the second case, on the contrary, when the view of the lens is modified, the B*DDGs* are updated and then propagate the changes to the source. The F*DDGs* need to be updated from the modified view and original source because the source is updated from the view modifications.In both forward and backward transformations, the F*DDGs* and B*DDGs* are consistent and synchronized.

4 IMPLEMENTATION AND EXPERIMENTS

In this section, we will explain how to implement the self-adjusting lenses such as identity lens, head lens, tail lens, hoist, plunge and filter lens. We will demonstrate experiments of those lenses, evaluate and discuss the experimental results in later parts of this section.

4.1 Implementation of Pure Lenses and Self-adjusting Lenses

We prepare two different versions for each lens that we use as examples in the implementation section. The first version is the implementation of the pure functional lenses, while the second version is of self-adjusting lenses.

First, we introduce the construction and some examples of the pure functional lens which are implementented in straightforward way. We choose the head lens for the example in this section. Given a source list the forward transformation of the head lens produces the first element of the source list as view in case the source list is non empty or returns an empty list in case the given source list is empty. In the backward transformation, the new source is created by concatenating the view and the tail of the original source without the first element.

Second, we will explain how to implement an self-adjusting lens from the pure functional lens. Since the self-adjusting technique is used to place the sources to the modifiables for monitoring, we distinguish the source of the forward transformations as original source from the source of the backward transformation as original source and view.

Considering the code for the pure functional head lens and a self-adjusting lens, as shown in Figure 1. The transformation from pure functional lens

```
type 'a lst = NIL | CONS of 'a * 'a lst

let rec filterG f src = match src with

  | NIL         -> NIL
  | CONS(x,xs) ->
    if (f x) then
        CONS(x, filterG f xs)
    else
        filterG f xs



  let rec filterP f (view,src) =



    match (view,src) with
    | (NIL, CONS(s,ss)) ->
      if (f s) then
        filterP f (NIL,ss)
      else
        CONS(s, filterP f (NIL,ss))

    | (CONS(v,vs), NIL) -> CONS(v,vs)

    | (CONS(v,vs),CONS(s,ss)) ->
      if (f s) then
        filterP f (view,ss)
      else if (false = f s) then
        CONS(s,filterP f (view,ss))
      else if (f v) then
        CONS(v,filterP f (vs,src))
      else
        filterP f (vs,src)

    | (NIL,NIL) -> NIL
```

```
type 'a lst' = NIL | CONS of 'a  * 'a lst' mods

let filterG f src =
  let rec filterM src d = read src (fun src -> match src with
    | NIL         -> write d NIL
    | CONS(x,xs) ->
      if (f x) then
          write d (CONS(x, (modref (fun d -> filterM xs d))))
      else
          filterM xs d)

  in modref (fun d -> filterM src d)

  let rec filterP f (view,src) =
    let rec filterM (view,src) d =
      read src (fun src ->
      read view (fun view ->
      match (view,src) with
      | (NIL, CONS(s,ss)) ->
        if (f s) then
          filterM (NIL,ss) d
        else
          write d (CONS(s,(modref (fun d -> filterM (NIL,ss) d))))

      | (CONS(v,vs), NIL) -> write d (CONS(v,vs))

      | (CONS(v,vs),CONS(s,ss)) ->
        if (f s) then
          filterM (view,ss) d
        else if (false = (f s)) then
          write d (CONS(s,modref (fun d -> filterM (view,ss) d)))
        else if (f v) then
          write d (CONS(v,modref (fun d -> filterM (vs,src) d)))
        else
          filterM (vs,src) d

      | (NIL,NIL) -> write d NIL))

    in modref (fun d -> filterM (view,src) d)
```

図 1　The implementation of pure functional and self-adjusting version of filter lens.

to self-adjusting lens is done in two steps. 1) We First generate a lists of modifiables by placing each element of the list into a modifiable. By designing this kind of structure, the modifiable list can be inserted or deleted elements to and from by users. 2) The second, we change the program so that by using the read, value of the modifiable can be accessed. In order to check whether the source list is empty, the self-adjusting head lens uses a read, then write the result to desitnation d. A call to mod (through modl) creates the modifiables that destination d belongs to. These modifiables form the view list which now is a modifiable list.

Similarly, we can transform all the lenses described in this paper from pure functional lenses

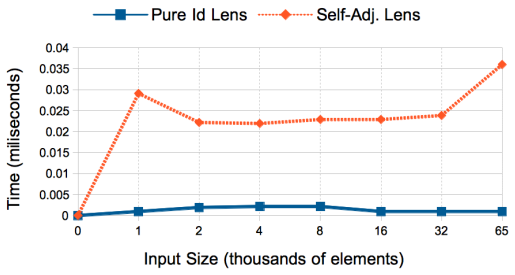

図 2　Initial forward transformation of Identity lens

to self-adjusting lenses using the two above steps.

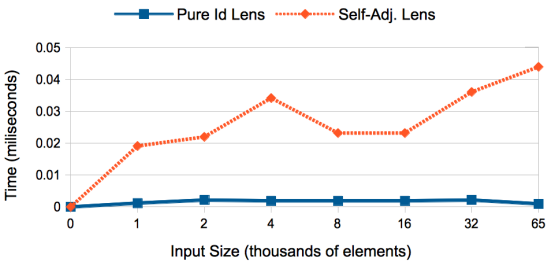

図 3 Initial backward transformation of Identity lens

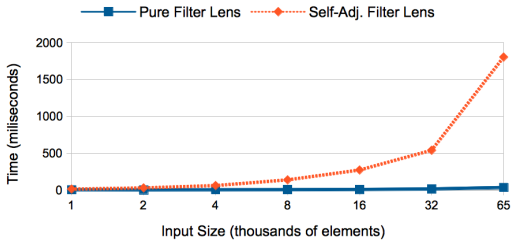

図 4 Initial forward transformation of Filter lens

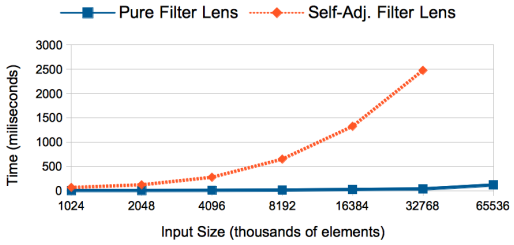

図 5 Initial backward transformation of Filter lens

4. 2 Experiments

We conduct experiments of self-adjusting lenses by using sources of different sizes. The source lists of integer for the lenses are randomly generalized in many different sizes. The lenses sequentially consume the sources of sizes (from 1 to 65,000). In general, we use the same sources for the pure and self-adjusting lenses in the experiments. As the self-adjusting lenses must execute at the first time to

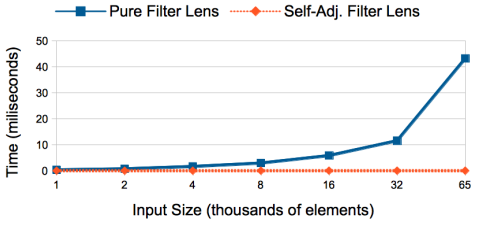

図 6 Insertion into source at the end of Filter lens in forward transformation

construct the dynamic dependence graphs, we compare the initial run of pure functional lenses and self-adjusting lenses.

In order to measure the effectiveness of the self-adjusting lenses, we modify the sources of both versions of lenses with the insertion into several positions. With the modified source, the pure functional lenses re-execute and the self-adjusting lenses propage the changes to update their views. The effectiveness of the self-adjusting techinique was proved in the publication of Acar et al. [8].

We demonstrate the result of the initialization of the forward and backward transforamtions of the identity lens and the filter lens. We prepare the random source in different sizes, from 1 thousand to 65 thousands of elements. In Figure 2 and Figure 3, we present the experimental results of the forward and backward transformations of filter lens. Likewise, results of initial bidirectional transformations of filter lens are showed in Figure 4 and Figure 5.

In Figure 6 and Figure 7, we show that the self-adjusting bidirectional transformations are practical and efficient. The performance of self-adjusting lens is better than the pure functional lens when insertion into the source or view at the end. The self-adjusting lens is more effective than the pure functional lens because it represents all source data and control dependencies in *dynamic dependence graphs* in the initialization phase while the pure functional lens does not save any information during its exe-

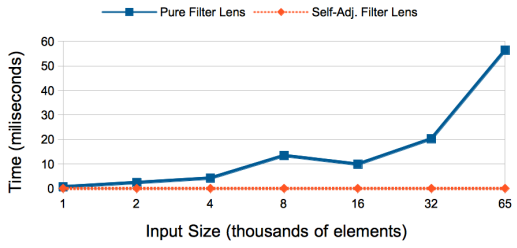

図 7 Insertion into source at the end of Filter lens in backward transformation

cutions. As a result, when the source or the view of the lens is modified, instead of re-computing from scratch, the self-adjusting lens can re-use the previous computations and show the better performance than pure functional lens.

4.3 Evaluation and Discussion

In the experimental section, we compare the initialization of the pure functional lenses and self-adjusting lenses. In the first run, the pure functional lenses are more effective than the self-adjusting lenses because the self-adjusting lenses save computations and data for later use. From the second runs, the self-adjusting programs are more effective than the pure programs theoractically. However, depend on the postion of modifications, the effectiveness of the self-adjusting programs can be vary. For instance, when we insert new value to the beginning of the source, it means that from the next run, the program will be re-run from scratch. Because in the initial run, we construct the dynamic dependence graphs which represent the data dependence and control dependencies as function call tree. The modification to the beginning of the source affects shallow function call in tree, so the change propagation is slow in such case. In other case, if we modify the source to the end, this affects the deep function call tree, the change propagation in this case is faster than previous case.

5 RELATED WORK

Bidirectional transformations have been discussed and studied in many different areas and communities including heterogeneous data synchronization [20], software model transformation [41], constraint maintenance for graphical user interfaces [32], interactive structure editing [26] and relational databases [12].

In the area of programming languages, Foster *et al.* [20] proposed bidirectional transformation approach, called lens, to solve the classical view-updating problem: when a concrete data model is abstracted into a view, the question is how modifications made to the view can be propagated back as modifications to the original model. Lenses must include two functions, one for extracting the abstract view from the concrete view, called *get* function, and another for putting back the modified abstract view back into the original concrete view to produce the updated concrete view, called *putback* function. Their linguistic approach was developed to support the development of bidirectional transformation on strings and trees.

The concept of adaptive computation was first introduced by Acar *et al.* 2002 [6], called adaptive functional programming (AFP), generalized dependence-graph approaches by introducing dynamic dependence graphs (*DDGs*) and providing a change-propagation algorithm for them. AFP can be applied to any purely functional program. A change-propagation algorithm that can update the dependence structure of the dynamic dependence graph by inserting and deleting dependencies where necessary, makes applications of adaptive functional programs be possible. Adaptive functional programs can be written by using type-safe linguistic facilities that guarantee safety and correctness of change propagation. The two papers, one on an SML library for self-adjusting compu-

tation (Acar *et al.* 2006a [2]) and the other on efficient implementation of the library (Acar *et al.* 2006b [5]), are based on Acar's thesis (Acar *et al.* 2005 [3]). The later self-adjusting work is on experimental analysis (Acar *et al.* 2009 [4]) which combines two papers above.

Our work was greatly inspired by work on self-adjusting computation (Acar *et al.* 2006 [2]). Unlike ordinary functional programs, the bidirectional programs allow modifications to either modify the source or the view before updating to or reflecting back to the view or source, respectively. How to update the dynamic dependence graph and propagate the changes from source and view dually is a challenge. This work is one step to experimentally combine the self-adjusting technique and lenses.

6 CONCLUSION

This article reports our first experiments on self-adjusting lenses using self-adjusting computation approach to the problem of incremental computation. The self-adjusting lenses can respond the source data quickly and effectively. We show that the pure functional lenses can transform to self-adjusting lenses in separate transformations. This article describes the method to construct self-adjusting lenses by using the Ocaml self-adjusting library and demonstrates the experiments of the self-adjusting lenses. The future work is how to unify separate self-adjusting transformations when constructing self-adjusting lens so that the modifications in either source or view can update or reflect back to the view or source in forward and backward transformation, respectively.

参 考 文 献

[1] Abadi, M., Lampson, B., and Lévy, J.-J.: Analysis and caching of dependencies, *SIGPLAN Not.*, Vol. 31, No. 6(1996), pp. 83–91.

[2] Acar, U., Blelloch, G., Blume, M., Harper, R., and Tangwongsan, K.: A Library for Self-Adjusting Computation, *Electron. Notes Theor. Comput. Sci.*, Vol. 148, No. 2(2006), pp. 127–154.

[3] Acar, U. A.: Self-Adjusting Computation, Technical report, In ACM SIGPLAN Workshop on ML, 2005.

[4] Acar, U. A., Blelloch, G. E., Blume, M., Harper, R., and Tangwongsan, K.: An experimental analysis of self-adjusting computation, *ACM Trans. Program. Lang. Syst.*, Vol. 32, No. 1(2009), pp. 3:1–3:53.

[5] Acar, U. A., Blelloch, G. E., Blume, M., and Tangwongsan, K.: An experimental analysis of self-adjusting computation, *SIGPLAN Not.*, Vol. 41, No. 6(2006), pp. 96–107.

[6] Acar, U. A., Blelloch, G. E., and Harper, R.: Adaptive functional programming, *SIGPLAN Not.*, Vol. 37, No. 1(2002), pp. 247–259.

[7] Acar, U. A., Blelloch, G. E., and Harper, R.: Selective memoization, *SIGPLAN Not.*, Vol. 38, No. 1(2003), pp. 14–25.

[8] Acar, U. A., Blelloch, G. E., and Harper, R.: Adaptive functional programming, *ACM Trans. Program. Lang. Syst.*, Vol. 28, No. 6(2006), pp. 990–1034.

[9] Acar, U. A., Blelloch, G. E., Harper, R., Vittes, J. L., and Woo, S. L. M.: Dynamizing static algorithms, with applications to dynamic trees and history independence, *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, 2004, pp. 531–540.

[10] Bancilhon, F. and Spyratos, N.: Update semantics of relational views, *ACM Trans. Database Syst.*, Vol. 6, No. 4(1981), pp. 557–575.

[11] Bellman, R.: *Dynamic Programming*, Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[12] Bohannon, A., Pierce, B. C., and Vaughan, J. A.: Relational lenses: a language for updatable views, *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, New York, NY, USA, ACM, 2006, pp. 338–347.

[13] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F.: Bidirectional Transformations: A Cross-Discipline Perspective, *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, Berlin, Heidelberg, Springer-Verlag, 2009, pp. 260–283.

[14] Czarnecki, K. and Helsen, S.: Classification of Model Transformation Approaches, *OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[15] Dayal, U. and Bernstein, P. A.: On the correct translation of update operations on relational views, *ACM Trans. Database Syst.*, Vol. 7, No. 3(1982),

pp. 381–416.

[16] Demers, A., Reps, T., and Teitelbaum, T.: Incremental evaluation for attribute grammars with application to syntax-directed editors, *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, New York, NY, USA, ACM, 1981, pp. 105–116.

[17] Doyle, J.: Readings in nonmonotonic reasoning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987, chapter A truth maintenance system, pp. 259–279.

[18] Field, J. and Teitelbaum, T.: Incremental reduction in the lambda calculus, *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, New York, NY, USA, ACM, 1990, pp. 307–322.

[19] Field, J. H.: *Incremental reduction in the lambda calculus and related reduction systems*, PhD Thesis, Ithaca, NY, USA, 1991. UMI Order No. GAX91-31429.

[20] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Trans. Program. Lang. Syst.*, Vol. 29, No. 3(2007).

[21] Gottlob, G., Paolini, P., and Zicari, R.: Properties and update semantics of consistent views, *ACM Trans. Database Syst.*, Vol. 13, No. 4(1988), pp. 486–524.

[22] Hegner, S.: Foundations of canonical update support for closed database views, *ICDT '90*, Abiteboul, S. and Kanellakis, P.(eds.), Lecture Notes in Computer Science, Vol. 470, Springer Berlin Heidelberg, 1990, pp. 422–436.

[23] Heydon, A., Levin, R., and Yu, Y.: Caching function calls using precise dependencies, *SIGPLAN Not.*, Vol. 35, No. 5(2000), pp. 311–320.

[24] Hofmann, M., Pierce, B., and Wagner, D.: Edit lenses, *SIGPLAN Not.*, Vol. 47, No. 1(2012), pp. 495–508.

[25] Hoover, R.: *Incremental graph evaluation (attribute grammar)*, PhD Thesis, Ithaca, NY, USA, 1987. UMI Order No. GAX87-24200.

[26] Hu, Z., Mu, S.-C., and Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations, *Higher Order Symbol. Comput.*, Vol. 21, No. 1-2(2008), pp. 89–118.

[27] Lämmel, R.: Coupled Software Transformations (Ext. Abstract), *Proc. Int'l Workshop on Software Evolution Transformations (SETra)*, Nov. 2004.

[28] Lechtenbörger, J. and Vossen, G.: On the computation of relational view complements, *ACM Trans. Database Syst.*, Vol. 28, No. 2(2003), pp. 175–208.

[29] Liu, Y. A., Stoller, S. D., and Teitelbaum, T.: Static caching for incremental computation, *ACM Trans. Program. Lang. Syst.*, Vol. 20, No. 3(1998), pp. 546–585.

[30] McAllester, D.: Truth maintenance, *Proceedings of the eighth National conference on Artificial intelligence - Volume 2*, AAAI'90, AAAI Press, 1990, pp. 1109–1116.

[31] Mccarthy, J.: A Basis for a Mathematical Theory of Computation, *Computer Programming and Formal Systems*, North-Holland, 1963, pp. 33–70.

[32] Meertens, L.: Designing Constraint Maintainers for User Interaction, Technical report, 1998.

[33] Michie, D.: "Memo" Functions and Machine Learning, *Nature*, Vol. 218, No. 5136(1968), pp. 19–22.

[34] Pugh, W. and Teitelbaum, T.: Incremental computation via function caching, *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, New York, NY, USA, ACM, 1989, pp. 315–328.

[35] Ramalingam, G. and Reps, T.: A categorized bibliography on incremental computation, *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, New York, NY, USA, ACM, 1993, pp. 502–510.

[36] Reps, T.: Optimal-time incremental semantic analysis for syntax-directed editors, *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, New York, NY, USA, ACM, 1982, pp. 169–176.

[37] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars, *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '94, London, UK, UK, Springer-Verlag, 1995, pp. 151–163.

[38] Schürr, A. and Klar, F.: 15 Years of Triple Graph Grammars - Research Challenges, New Contributions, Open Problems, *4th International Conference on Graph Transformation*, Vol. 5214, Springer Verlag, Heidelberg, Lecture Notes in Computer Science (LNCS), Sep 2008, pp. 411–425.

[39] Shankar, A. and Bodík, R.: DITTO: automatic incrementalization of data structure invariant checks (in Java), *SIGPLAN Not.*, Vol. 42, No. 6(2007), pp. 310–319.

[40] Stallman, R. M. and Sussman, G. J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence*, Vol. 9, No. 2(1977), pp. 135–196.

[41] Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions, *Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems*, MODELS'07, Berlin, Heidelberg, Springer-Verlag, 2007, pp. 1–15.

[42] Sundaresh, R. S. and Hudak, P.: Incremental Compilation via Partial Evaluation, *Conference Record of the 18th Annual ACM Symposium on POPL*, January 1991, pp. 1–13.