Type-Directed Automatic Incrementalization

Yan Chen Joshua Dunfield Umut A. Acar
Max Planck Institute for Software Systems
{chenyan, joshua, umut}@mpi-sws.org

Abstract

Application data often changes slowly or incrementally over time. Since incremental changes to input often result in only small changes in output, it is often feasible to respond to such changes asymptotically more efficiently than by re-running the whole computation. Traditionally, realizing such asymptotic efficiency improvements requires designing problem-specific algorithms known as dynamic or incremental algorithms, which are often significantly more complicated than conventional algorithms to design, analyze, implement, and use. A long-standing open problem is to develop techniques that automatically transform conventional programs so that they correctly and efficiently respond to incremental changes.

In this paper, we describe a significant step towards solving the problem of automatic incrementalization: a programming language and a compiler that can, given a few type annotations describing what can change over time, compile a conventional program that assumes its data to be static (unchanging over time) to an incremental program. Based on recent advances in self-adjusting computation, including a theoretical proposal for translating purely functional programs to self-adjusting programs, we develop techniques for translating conventional Standard ML programs to self-adjusting programs. By extending the Standard ML language, we design a fully featured programming language with higher-order features, a module system, and a powerful type system, and implement a compiler for this language. The resulting programming language, LML, enables translating conventional programs decorated with simple type annotations into incremental programs that can respond to changes in their data correctly and efficiently.

We evaluate the effectiveness of our approach by considering a range of benchmarks involving lists, vectors, and matrices, as well as a ray tracer. For these benchmarks, our compiler incrementalizes existing code with only trivial amounts of annotation. The resulting programs are often asymptotically more efficient, leading to orders of magnitude speedups in practice.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

Keywords Self-adjusting computation; incrementalization; type annotations; compiler optimization; performance

1. Introduction

Much modern software is highly dynamic: it continually receives input and responds by computing the corresponding output. This dynamic nature of computation creates both opportunities and challenges. Opportunities arise because the input data, which changes over time as a result of interactions, often changes *incrementally*—by a small amount at a time. Such incremental changes make it possible to compute the new output efficiently by reusing intermediate computations. In the common case, updating the output dynamically instead of recomputing it leads to asymptotically more efficient response times, which can dramatically improve practical efficiency. The challenges stem from the difficulty of realizing this potential: designing, analyzing, and implementing software systems that can operate efficiently on dynamically changing data.

In the algorithms and programming-languages communities, considerable work has been done on dynamic systems; for surveys, see Chiang and Tamassia [1992]; Ramalingam and Reps [1993]; Agarwal et al. [2002]; Demetrescu et al. [2005]. The algorithms community devises ad hoc dynamic algorithms for specific problems. While these algorithms can be very efficient, often achieving optimal complexity, they are generally hard to design, analyze, and implement. Dynamic algorithms are also difficult to compose in a modular fashion, limiting their applicability in large software systems. The programming-languages community develops languages for expressing dynamic programs and compilation techniques for translating these high-level programs into executables that respond efficiently to dynamic changes. This approach is often called *incremental computation.* Incremental computation can dramatically simplify developing dynamic software, but achieving optimal efficiency has remained elusive. This is perhaps unsurprising, because the problem is inherently challenging: the compiler is ultimately expected to generate code that significantly outperforms the source code, often by an asymptotically significant margin.

Recent advances in *self-adjusting computation* made important progress on the problem of incremental computation. By proposing dynamic dependency graphs, and a change propagation algorithm [Acar et al. 2006] that utilizes a particular form of memoization techniques [Acar et al. 2009, 2008], the approach enables the programmer to express dynamic computations via several language abstractions. Previous work extended existing languages including C [Hammer et al. 2009, 2011] and ML [Acar et al. 2009; Ley-Wild et al. 2008] to support self-adjusting computation. Evaluations showed that the approach can achieve asymptotically efficient updates for a reasonably broad range of benchmarks [Acar et al. 2009; Hammer et al. 2011], and even help solve major open problems in a range of domains including computational geometry [Acar et al. 2010] and machine learning [Sümer et al. 2011]. More recent work generalized the approach to support parallel computation on multicores, taking advantage of the performance benefits of parallelism and incremental computation at the same time by exploiting structural similarities between them [Burckhardt et al. 2011; Acar et al. 2011]. The approach has also been applied to large-scale MapReduce computations in distributed systems [Bhatotia et al. 2011].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06. . . $10.00

While these advances chart a viable approach to incremental computation by eliminating the need to design and implement sophisticated algorithms for dynamic problems, they still require significant programmer involvement. For example, writing a self-adjusting program in ΔML [Ley-Wild et al. 2008] requires carefully annotating the program with specific primitives that determine how to construct the dynamic dependence graph and perform change propagation.

In this paper, we design a language, LML, that allows the programmer to derive self-adjusting software from conventional programs through simple type annotations and implement a compiler for LML. Specifically, LML extends the Standard ML language, which has a range of features including higher-order functions, an advanced module system, and imperative references, with level type qualifiers, which enable marking certain data as *changeable*, that is, subject to modifications over time. For example, when programming a ray tracer, we mark the surface properties of objects as changeable. We implement a compiler for our language that generates code that can be executed with fixed input as usual, but also can automatically respond to changes by updating its output via change propagation.

Our approach to automatic incrementalization builds on recent advances in self-adjusting computation. Specifically, Chen et al. [2011] develop an algorithm for translating purely functional programs decorated with type annotations into self-adjusting programs. Their work, however, is purely theoretical; it considers only a simplified language, and provides no implementation or empirical results. We extend their algorithm for full Standard ML including a key imperative feature (mutable references), and extend the MLton compiler [MLton] for Standard ML (Section 3) to generate efficient self-adjusting executables from type annotations by using the extended translation algorithm.

Our compiler takes the user annotations and propagates them through various phases of the compiler to intermediate code, where it applies the translation algorithm. Assuming a runtime system that provides primitives for self-adjusting computation, the translation algorithm generates code by minimally inserting the self-adjusting primitives via type-directed, local rewrites. Such local rewrites, however, can lead to globally suboptimal code by inserting redundant calls to self-adjusting primitives. We therefore formulate a global rewriting system for eliminating such redundant code, and prove that the rewriting rules are terminating and confluent (Section 3.4). We implement the run-time system for self-adjusting primitives directly in SML.

We evaluate our implementation (Section 4) by considering benchmarks including various primitives on lists, sorting functions, vector operations, matrix operations, and a ray tracer. For each of these, we only need to insert some keywords into the program to specify the desired behavior. Specifically, most benchmarks require trivial decorations, often amounting to inserting type qualifiers in one or two lines of code. No changes to the structure of the types, or any part of the code itself, are necessary. The executables generated by the compilers respond automatically and efficiently to small changes to their data. We frequently observe significant asymptotic improvements in efficiency and obtain significant speedups ranging from 10% (for large changes that affect a significant portion of the output) to several orders of magnitude.

2. Overview

We illustrate our approach through a simple example. First, we describe matrix multiplication in ordinary SML. Next, we describe a self-adjusting version, which could be written by hand, that can update its output asymptotically more efficiently (by nearly a quadratic factor) than a complete re-computation. We then describe our approach, where we obtain the same code automatically by adding a single type qualifier to the code and compiling it with our compiler. In the example, we consider a particular kind of change to the input and briefly outline other possibilities in Section 2.4. In all cases, the executable generated by our compiler updates the output correctly and efficiently, leading to orders of magnitude speedups in practice (Section 4).

2.1 Matrix Multiplication in SML

The SML code in Figure 1 multiplies two matrices, where a matrix is represented as a vector of rows, and each row is a vector of integers. Omitting the annotation `$C`, this is the usual $\Theta(n^3)$ matrix-multiplication algorithm. Using the `map` function over vectors, it iterates over the rows of the first matrix and the columns of the second (transposed for faster access). By using the `map2` function, it multiplies each element of the row with the corresponding element of the column, and using `reduce`, adds these results to generate one element of the output matrix.

The function `transpose`, whose code is not shown here, transposes a matrix. The functions `map`, `map2` and `reduce` perform standard operations on vectors: Given a vector $a = \langle a_1, \ldots, a_n \rangle$ and a function `f`, the call `map(a,f)` returns a vector $\langle f(a_1), \ldots, f(a_n) \rangle$). Similarly, given two vectors `a` and `b` and a binary function `f`, the call `map2(a,b,f)` returns $\langle f(a_1, b_1), \ldots, f(a_n, b_n) \rangle$). Given a vector $a = \langle a_1, \ldots, a_n \rangle$, an identity element `Z` and an associative binary function $\oplus$, the call `reduce(a,Z,⊕)` returns $a_1 \oplus \cdots \oplus a_n$ if $n > 0$, and `Z` if $n = 0$.

2.2 Self-Adjusting Matrix Multiplication

Suppose that we are interested in changing the elements of the matrix incrementally and updating the result of multiplication. One approach would be to develop an algorithm specific to this problem. For matrices with integers or floating-point numbers, this does not seem particularly difficult: we could devise an algorithm that undoes the effect of the changed input element and updates the sum by factoring in its new value. However, such an algorithm assumes addition and multiplication are commutative and have inverses; in reality, floating-point arithmetic does not have these properties. Moreover, matrix multiplication can be used for a whole array of computations, some of which don't admit such inverses. In these cases, it is significantly more challenging to come up with an efficient incremental algorithm.

More generally, the history of research on incremental algorithms demonstrates that they can be extremely challenging to design and implement. For example, a paper on an advanced dynamic algorithm for planar convex hull exceeds 100 pages [Jakob 2002]; that algorithm appears very difficult to implement, and to the best of our knowledge, has never been implemented. In contrast, standard non-incremental algorithms for convex hull can be implemented in less than 50 lines of SML.

Writing self-adjusting matrix-multiply. Self-adjusting computation offers a way to write efficient incremental algorithms by modifying the code for the standard, non-incremental algorithm. The idea is to distinguish between *stable* and *changeable* data and insert operations that manipulate changeable data. Incremental changes can be made to the changeable parts of the input and a *change propagation* algorithm can be used to update the computation. Figure 2 shows code written in a direct style similar to previous work [Acar et al. 2009]. Stable data is handled as usual: the type of a stable integer is simply `int`. Changeable data, however, is stored in *modifiables*: a changeable integer becomes an `int mod`. We declare `matrix` as `((int mod) vector) vector`: a vector of vectors of integers, where the integer elements are wrapped by modifiables. This allows changing the individual elements of the matrix and updating the computation automatically.

```
type matrix = ((int $C) vector) vector
multiply : matrix * matrix → matrix

fun multiply (A, B) =
  let val Tb = transpose B
  in
    map (A, fn row ⇒
      map (Tb, fn col ⇒
        reduce (map2 (row,
                      col,
                      fn (a,b) ⇒ a*b),

                0,
                fn (x, res) ⇒ x+res)))

  end
```

Figure 1. Matrix multiplication in SML.

```
type matrix = ((int mod) vector) vector
multiply : matrix * matrix → matrix

fun multiply (A, B) =
  let val Tb = transpose B
  in
    map (A, fn row ⇒
      map (Tb, fn col ⇒
        reduce (map2 (row,
                      col,
                      fn (a,b) ⇒ Mod (Read a (fn a' ⇒
                                      Read b (fn b' ⇒ Write (a'*b')))))),
                Mod (Write 0),
                fn (x, res) ⇒
                  Mod (Read x (fn x' ⇒ Read res (fn res' ⇒
                    Write (x'+res')))))))
  end
```

Figure 2. Compiler-generated self-adjusting matrix multiplication.

When writing a self-adjusting program by hand, we first determine the changeable parts of the data, and then edit the code to explicitly manipulate such data through `Read`, `Write` and `Mod` primitives. This process is relatively cumbersome and error-prone: we must identify the sections of code that depend on changeable data, conforming to a modal type system. Specifically, a `Read` primitive can be used only inside a "changeable" section of code, which must be contained within the dynamic scope of a `Mod` operation, and each `Mod` operation must end with a `Write` that places a changeable value in it.

As an example, consider the function passed to `map2`, **fn** `(a,b) ⇒ a*b`. This function directly operates on elements of the matrix, which are changeable integers. Since they are changeable integers, they must be stored in modifiables and have type `int mod` in the self-adjusting program. Thus, the function cannot multiply `a` and `b` directly. Instead, we write a `Read` that passes the contents of the modifiable `a` to a function **fn** `a' ⇒ Read b ...`. A second `Read` gets the contents of the modifiable `b`. The `a'` and `b'` appearing in `Write (a'*b')` have type `int`, so we can multiply them normally. Because `a'*b'` depends on changeable data (information flows from a and b, which are changeable), it is also changeable, and must be written to a fresh modifiable: `Mod (... Write (a'*b'))`.

In contrast, calls to `transpose`, `map`, `map2` and `reduce` need not be treated specially, because they operate on stable data (vectors); note that the elements of the vector are changeable but the vectors themselves are not. The bodies of these functions, however, still need to be modified to accommodate changes in the types: they all now operate on vectors whose elements are changeable, and in some cases, their other arguments also change (e.g., the function argument of `map2`).

Change propagation. Given the self-adjusting matrix multiplication function, we can run it in much the same way as running the standard version. Such a complete run takes asymptotically as long as the complete run but incurs some constant-factor overhead in practice.

After a complete run, we can change any or all of the changeable data and update the output by performing change propagation. As an example, consider changing one element of the first matrix and performing change propagation. This propagation will trigger execution of the n multiplication operations that use this element and the `reduce` operation that computes the new sum. When multiplying $n \times n$ matrices, it is not difficult to show that change propagation takes $O(n \log n)$ time. Since at least n entries in the output matrix must be updated, this is within a $\log n$ factor of the trivial lower bound $\Omega(n)$; in fact, it is likely that $O(n \log n)$ is tight in the general case. Consequently, change propagation is nearly quadratically faster than a complete execution, a huge gain in efficiency. In writing this self-adjusting program, we realized this efficiency without designing and implementing an incremental algorithm, but we nevertheless had to make significant changes to the code.

2.3 Type-Directed Self-Adjustment

As the description of the self-adjusting matrix multiply suggests, writing self-adjusting programs can require rather invasive changes to the code. In our approach, our compiler can derive self-adjusting programs automatically, based on simple type annotations. For example, given the code in Figure 1 and the annotation `$C` (the first line) that makes the element type `int` changeable, our compiler derives the code in Figure 2 automatically. As a result, starting with a trivial annotation to the type declarations, our compiler yields a near-quadratic-time improvement in run time.

We refer to the type annotations as *levels* and the resulting types as *level types*. Programmers need only annotate the types of changeable data with `$C`; all other types remain stable. For example:

- `int` is a stable integer;
- `int $C` is a changeable integer;
- `(int $C) vector` is a stable vector of changeable integers.

The last type may look odd—how can the vector be stable when its elements are changeable? But it simply expresses that each element is individually changeable, while the vector as a whole is not changeable: it would not be possible to replace the entire vector except by changing each element individually.

2.4 Remarks

In the above example, we allow changes to individual elements by representing matrices as stable vectors with changeable elements: `((int $C) vector) vector`. Our approach also makes it possible to use many other representations, again deriving self-adjusting code automatically. For example, if we expect entire rows of the input matrices to change at once, we could choose to represent matrices as changeable rows of stable elements, `((int) vector $C) vector`. This would track the computation only at the granularity of entire rows and result in optimal updates for full row changes of the input. Using our approach, we could

derive a self-adjusting program suitable for this case by changing the type annotation, with no changes to the source code itself at all.

Similarly, it can be beneficial to use a blocked representation, where the matrix is represented with blocks, smaller matrices of size $m \times m$ for some constant m (typically between 10 and 100). This representation has superior locality, often resulting in better practical efficiency. Our compiler easily generates self-adjusting code that treats the blocks, and not individual elements, as changeable. As we show in our experimental evaluation (Section 4), this blocking technique leads to extremely time- and space-efficient self-adjusting computations.

3. Design and Implementation

To support type-directed, automatic incrementalization, we extended Standard ML with a single keyword `$C`, a type qualifier, and extended the MLton compiler [MLton] for Standard ML to generate self-adjusting executables. This extension to SML does not restrict the language in any way, allowing all its features to be used freely, including the module language.

The most important additions to the compiler are a translation phase, an optimization phase, and a run-time system. In this section, we describe the structure of our system, and discuss some of its key components in more detail.

3.1 Structure of the Compiler

The compiler pipeline is shown in Figure 3. Although the surface language has only one type annotation, `$C`, in all intermediate languages, we explicitly mark all types as stable or changeable using two type qualifiers `$S` and `$C`. We modified all of MLton's phases that come before the new translation phase (**Translate**) to propagate these type annotations. The translation phase uses type qualifiers to generate self-adjusting code in the SXML intermediate language of MLton, as discussed in the next section. The optimization phase eliminates some important redundancies of the code produced by the translation phase. For technical reasons related to MLton's dead code elimination (see Section 3.5), we stop the compiler after the optimization phase and run an unmodified version of MLton on the SXML output, producing an executable.

3.2 Pre-Translation Phases

We extended the MLton lexer and parser to handle types with `$C` annotations, producing abstract syntax in which types include level information. We systematically added levels to several typed intermediate languages (CoreML, "XML", and SXML), and modified each pre-translation phase to accept and propagate levels in types. Thus, we leverage MLton's broad scope—it accepts full Standard ML—and its various code transformations: elaboration, defunctorization, linearization (conversion to A-normal form), dead code elimination, monomorphization of ML-polymorphic code, etc.

3.3 Self-Adjusting Translation

We apply our translation on MLton's SXML intermediate language, a monomorphic subset of SML in A-normal form. SXML is suitable for the transformation to self-adjusting code because our transformation algorithm expects, and produces, monomorphic code in A-normal form.

Our translation algorithm extends the algorithm of Chen et al. [2011] to support full SML, including (recursive) data types and imperative references. The translation algorithm is relatively technical, making its presentation difficult in the context of this paper, but we give a high-level overview and briefly illustrate some of the extensions needed to handle full SML. The translation algorithm takes SXML code and transforms it into SXML code containing self-adjusting computation primitives, whose implementations will be supplied by the run-time system. The self-adjusting primitives include **mod**, **read**, and **write** functions for creating, reading from, and writing to modifiable references. At a high level, the translation rules inspect the code locally, insert **read**s where changeable data is used (according to type information), and ensure that each **read** takes place within the dynamic scope of a call to **mod**. To ensure this and other correctness properties, the rules distinguish stable and changeable modes.

$$\frac{\Gamma \vdash x : \tau \;\underset{\$S}{\hookrightarrow}\; x'}{\Gamma \vdash (\mathbf{ref}\ x) : (\tau\ \mathbf{ref}^{\$C}) \;\underset{\$S}{\hookrightarrow}\; \mathbf{mod}\ (\mathbf{write}(x'))} \text{ (Ref)}$$

$$\frac{\Gamma \vdash x_2 : \tau'\ \mathbf{ref}^{\$C} \;\underset{\$S}{\hookrightarrow}\; x_2 \qquad \Gamma, x_1 : \tau' \vdash e : \tau \;\underset{\$C}{\hookrightarrow}\; e'}{\Gamma \vdash \mathbf{let}\ x_1 = !x_2\ \mathbf{in}\ e : \tau \;\underset{\$C}{\hookrightarrow}\; \mathbf{read}\ x_2\ \mathbf{as}\ x_1\ \mathbf{in}\ e'} \text{ (Deref)}$$

$$\frac{\begin{array}{l}\Gamma \vdash x_1 : \tau'\ \mathbf{ref}^{\$C} \;\underset{\$S}{\hookrightarrow}\; x_1 \\ \Gamma \vdash x_2 : \tau' \;\underset{\$S}{\hookrightarrow}\; x_2' \qquad\qquad \Gamma \vdash e : \tau \;\underset{\$C}{\hookrightarrow}\; e'\end{array}}{\Gamma \vdash \mathbf{let}\ \_ = (x_1 := x_2)\ \mathbf{in}\ e : \tau \;\underset{\$C}{\hookrightarrow}\; \mathbf{impwrite}\ x_1 := x_2'\ \mathbf{in}\ e'} \text{ (Assign)}$$

Figure 4. Translation rules for mutable references

Figure 4 shows the translation rules for mutable references, which we translate to modifiables. The translation judgment $\Gamma \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$ is read "in environment Γ and mode ε, source expression e at type τ translates to e'". In stable mode `$S`, the translation produces stable code that cannot inspect changeable data or directly use changeable code; in changeable mode `$C`, the translation produces changeable code that can appear within the body of a **read** and can manipulate references. For translation of imperative references, we add another primitive **impwrite** that updates the value of a modifiable directly.

Stable functions may be called with either stable or changeable arguments. For example, the program might use the built-in SML `+` function on changeable integers. Our translation algorithm handles such polymorphic usage by inserting coercions, which read changeable arguments and create a modifiable from the result.

3.4 Optimization

Our translation algorithm follows a system of inductive rules, which are guided only by local information—the structure of types and terms. This locality is key to simplifying the algorithm and the implementation but it often generates code with redundant operations. For example, translating **fst** x, where $x : (\mathbf{int}^{\$C} \times \tau_2)^{\$S}$, in changeable mode generates the term **read** (**mod** (**let** r = **fst** x **in write**(r))) **as** x' **in write**(x'), which is redundant: it creates a temporary modifiable for the first projection of x and immediately reads its contents. A more efficient translation, **let** x' = **fst** x **in write**(x'), avoids the temporary modifiable.

Such redundancies turn out to be common, because of the local nature of the translation algorithm. We therefore developed a post-translation optimization phase to eliminate redundant operations. Figure 5 illustrates the rules that drive the optimization phase. Each rule eliminates three operations: reading from a modifiable, writing to a modifiable, and creating a modifiable. As we show in Section 4.8, this optimization phase reduces the execution time for self-adjusting programs by up to 60%.

Eliminating write-create-read. The left-hand side of rewrite rule (1) evaluates an expression e_1 into a new modifiable, then immediately reads the contents of the modifiable into x'. The right-hand side evaluates e_1 and binds the result to x' with no extra modifiable.

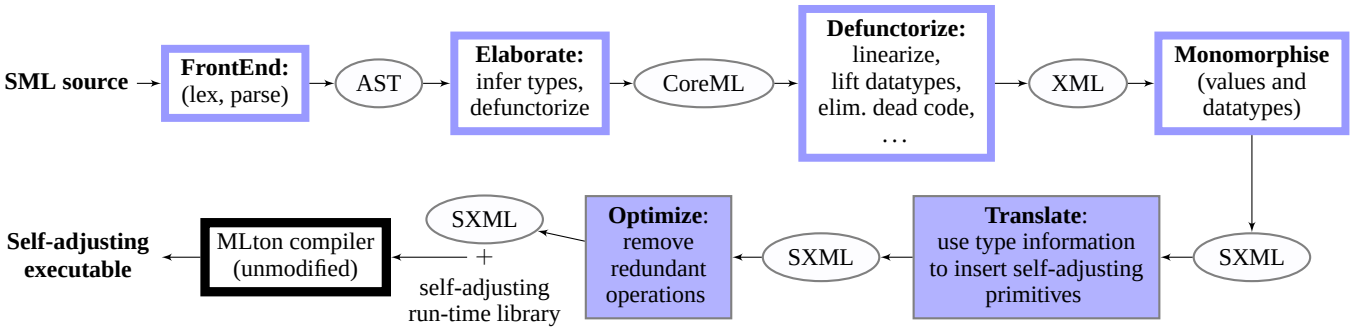

Figure 3. The structure of our compiler: new phases, modified phases, unmodified phases, intermediate languages

$$\textbf{read}\ (\textbf{mod}\ (\textbf{let}\ r = e_1\ \textbf{in write}(r)))\ \textbf{as}\ x'\ \textbf{in}\ e_2 \longrightarrow \textbf{let}\ x' = e_1\ \textbf{in}\ e_2 \quad (1)$$

$$\textbf{read}\ (\textbf{mod}\ e)\ \textbf{as}\ x'\ \textbf{in write}(x') \longrightarrow e \quad (2)$$

$$\textbf{mod}\ (\textbf{read}\ e\ \textbf{as}\ x'\ \textbf{in write}(x')) \longrightarrow e \quad (3)$$

Figure 5. Optimization rules

Eliminating create-read-write. The left-hand side of (2) evaluates e (which, since it is the body of a **mod**, must end in a **write**), creates a modifiable, reads the just-written value into x', and writes it again. The right-hand side just evaluates e.

Eliminating read-write-create. Rule (3) is similar to rule (2): the left-hand side reads some modifiable e into a variable x', and immediately writes x' back to a new modifiable; the right-hand side only evaluates e.

These rules are shrinking reduction rules guaranteed to make the program smaller [Appel and Jim 1997]. The rules are also terminating and confluent. Termination is immediate, because in each rule, the right-hand side is smaller than the left-hand side: rule (1) replaces one **let** with another and drops a **read**, a **mod** and a **write**. In rules (2) and (3), the right-hand side is a proper subterm of the left-hand side. Confluence (the property that all choices of rewrite sequences eventually yield α-equivalent terms) is not immediate, but is straightforward:

Theorem 3.1. *Rules (1)–(3) are locally confluent.*

Proof. First, the left-hand sides of rules (1) and (2) may overlap exactly: either rule can be applied to $\textbf{read}\ (\textbf{mod}\ (\textbf{let}\ r = e_1\ \textbf{in write}(r)))\ \textbf{as}\ x'\ \textbf{in write}(x')$, but the right-hand sides of (1) and (2) are $\textbf{let}\ x' = e_1\ \textbf{in write}(x')$ and $\textbf{let}\ r = e_1\ \textbf{in write}(r)$, which are α-equivalent. Rules (2) and (3) may overlap critically, but in all cases yield α-equivalent terms. One case (the other is similar) is:

$$\underline{\textbf{read}\ (\textbf{mod}\ (\textbf{read}\ e_3\ \textbf{as}\ x_3'\ \textbf{in}\ x_3'))\ \textbf{as}\ x_2'\ \textbf{in write}(x_2')} \xrightarrow{(2)} \underline{\textbf{read}\ e_3\ \textbf{as}\ x_3'\ \textbf{in write}(x_3')}$$

$$\textbf{read}\ (\underline{\textbf{mod}\ (\textbf{read}\ e_3\ \textbf{as}\ x_3'\ \textbf{in}\ x_3')})\ \textbf{as}\ x_2'\ \textbf{in write}(x_2') \xrightarrow{(3)} \textbf{read}\ \underline{e_3}\ \textbf{as}\ x_2'\ \textbf{in write}(x_2')$$

Otherwise, redexes overlap only when an entire left-hand side is a subterm of the e in another left-hand side (possibly of the same rule). Such *non-critical overlap* cases follow as in Baader and Nipkow [1998, pp. 137–138]. □

Since the rules are terminating and locally confluent, by Newman's lemma [Newman 1942], they are globally confluent. Thus, we can safely apply them in any order, to arbitrary subterms, until no rules apply. In practice, it suffices to traverse the program only once: if we traverse it in preorder, we apply rules near the leaves of the tree first. That means the subterms of the left-hand sides have already been rewritten, so the right-hand sides will contain no more candidate subterms.

3.5 Final Stage

The translated, optimized SXML code has explicit self-adjusting computation primitives, suitable for use with a self-adjusting run-time library implementing these primitives. Our implementation of this library is written in SML, so we might expect to run the full MLton pipeline on the library and source program together to produce an executable. Since, however, no calls to the library appear in the user's source program, dead code elimination deletes the library code from the program during MLton's Defunctorize phase. Instead of re-engineering MLton's dead code elimination to specially treat library code as live, we take a simpler approach. Taking advantage of the fact that SXML is a subset of SML, we cut off the pipeline after producing optimized self-adjusting SXML, combine the SXML output of the translation phase with the library, and generate an executable by running the unmodified MLton compiler.

3.6 Runtime Environment

As described above, we compile the translated user program together with a self-adjusting run-time library. This library implements the self-adjusting primitives (`Mod`, `Read`, `Write`) used by the translated code. When executed, the library constructs a dependency graph during the complete run. The library also provides facilities for changing the input and invoking change propagation to reflect changes to the output.

4. Experiments

We present an experimental evaluation of our approach and compare it to previous work.

4.1 Benchmarks

We implemented a number of benchmarks in our LML language, including standard self-adjusting-computation benchmarks from previous work [Acar et al. 2009; Ley-Wild et al. 2008], additional benchmarks on vectors and matrices, and a ray tracer. LML makes it relatively straightforward to derive self-adjusting versions of programs. Specifically, we simply wrote the standard code for our benchmarks and changed the type declarations to allow for changes to the input data. For the ray tracer, we used an unmodified SML implementation of a sphere ray tracer [King 1998].

Our benchmarks include some standard list primitives (`map`, `filter`, `split`), quicksort, and mergesort (`qsort`, `msort`). These include simple iteration (`map`, `filter`, `split`), accumulator passing (`qsort`), and divide-and-conquer algorithms (`qsort`, `msort`). All of these list benchmarks operate on integers: `map` applies $f(i) = i \div 3 + i \div 5 + i \div 7$ to each element; `filter` keeps the elements when $f(i)$ is even; `split` partitions its input; `qsort` and `msort` implement sorting algorithms. Similarly, our vector benchmarks include `vec-reduce`, `vec-mult` (dot product), `mat-vec-mult` (matrix-vector multiplication), `mat-add`, `transpose` (matrix transpose), `mat-mult`, and `block-mat-mult` (matrix multiplication on matrices that use a simple blocked representation).

The vector and matrix benchmarks implement the corresponding vector or matrix algorithm with double-precision (64-bit) floating point numbers; when multiplying two doubles, we normalize the result by their sum to prevent overflows when operating on large matrices. For our matrix benchmarks, we consider two different representations of matrices: the standard representation where the elements are laid out in memory in row-major order, and the blocked representation where elements are blocked into small submatrices. Our final benchmark is an off-the-shelf ray tracer that supports point and directional lights, sphere and plane objects, and diffuse, specular, transparent, and reflective surface properties.

To support flexible changes to the input data, our list benchmarks permit insertion and deletion of any element from the input; in LML, this requires simply specifying the "tail" of the lists as changeable. Our vector and matrix benchmarks permit changing any element of the input; in LML, this requires simply specifying the vector and matrix elements as changeable. Our blocked matrix benchmark permits changing any block (and thus any element) of the input. Our ray tracer permits changing the surface properties of objects in the image; thus, for a fixed input scene (lights and objects) and output image size, we can render multiple images via change propagation.

The type annotations needed to enable these changes in our self-adjusting versions of the benchmarks were trivial. Each benchmark, including the ray tracer, required changes to no more than a few lines of code—in fact, never more than two lines.

For each benchmark, we evaluate a conventional implementation and four self-adjusting versions. Three of these are hand-coded versions from previous work, "CPS" [Ley-Wild et al. 2008], "CEAL" [Hammer et al. 2011] and "AFL" [Acar et al. 2009]. We use these benchmarks exactly as published, except for setting the test parameters and input data consistently to enable comparison. The last set, labeled "Type-Directed", consists of the self-adjusting programs generated by our LML compiler. Our LML list benchmarks use the same memoization strategy as the "AFL" versions of the list benchmarks; the rest of the benchmarks do not need memoization in LML.

4.2 Experimental Setup

We used a 2 GHz Intel Xeon with 64 GB memory running Linux. The machine has multiple cores but all benchmarks are sequential. We compile all our benchmarks using our LML compiler. For comparison to previous work on ΔML, we use the publicly available ΔML compiler [DeltaML]. We execute all benchmarks with the "`gc-summary`" option to report garbage collection statistics, but exclude garbage-collection times from our experiments to focus on the actual computation time; we separately discuss garbage collection in Section 4.10.

For our measurements, we generate all inputs and all data changes uniformly randomly and sample over all possible changes. More specifically, the inputs to our integer benchmarks are random permutations of integers from 1 to n, where n is the input size. The inputs to our floating-point benchmarks are randomly generated floating-point numbers via the SML library. To increase the coverage of our evaluation, for each measurement, we average over four different input instances, as well as all input changes over each of these inputs.

For each benchmark we measure the complete running time of the conventional and the self-adjusting versions. All reported times are in seconds or milliseconds, averaged over four independent runs. Timings exclude creation of the initial input; in change-propagation timings, we also exclude the initial, pre-processing run (construction of the test data plus the complete run). To measure efficiency in responding to small data changes, we compute the *propagation time* for responding to an incremental change. The nature of the change depends on the benchmarks. For list benchmarks, we report the average time to insert or delete an element from the input list (average over all elements). For vector and matrix benchmarks, we report the average time to change an element of the vector or matrix by replacing it with a randomly generated element (averaged over all positions in the vector or one position per row in the matrix). For the ray tracer, we consider a range of changes which we describe later when discussing the ray tracer in detail.

4.3 Experiments: Correctness

To verify that our compiler generates self-adjusting executables that can respond to changes to their data correctly, we used three approaches: type checking, manual inspection, and extensive testing. Our compiler generates self-adjusting code to a text file, which we then type-check and compile along with a stand-alone self-adjusting-computation library, which we have separately implemented. SML's type system verifies that the translated code satisfies certain invariants. Since we can inspect the translated code manually, we can also spot-check the code, which is not a foolproof method but increases confidence. We have used this facility extensively when implementing the compiler.

Additionally, we have developed a testing framework, which makes a massive number of randomly generated changes to the input data, and checks that the executable responds correctly to each such change by comparing its output with that of a "verifier" (reference implementation) that computes the given output using a straightforward, non-incremental algorithm. Using this framework, we have verified the correctness of all the self-adjusting executables generated by our compiler.

4.4 Experiments: Timings Summary

Table 1 shows a summary of the timings that we collected for our benchmarks at fixed input sizes (written in parentheses after the benchmark's name). All times are reported in seconds. The first column ("Conv. Run") shows the run time of the conventional (reference) implementation with an input of specified size. The conventional version cannot self-adjust, but does not incur the overhead of trace construction as self-adjusting versions do. The second column ("Self-Adj. Run") shows the run time of the self-adjusting version with an input of specified size. Such a self-adjusting run constructs a trace as it executes, which can then be used to respond automatically to incremental changes to data via change propagation. The third column ("Self-Adj. Avg. Prop.") shows the average time for a change propagation after a small change to the input (as described in Section 4.1, the specific nature of the changes depend on the application).

The last two columns of the table, "Overhead" and "Speedup", report the ratio of the self-adjusting run to the conventional run, and the ratio of the conventional run to change propagation. The overhead is the slowdown that a self-adjusting run incurs compared to a run of the conventional program. The speedup measures the speedup that change propagation delivers compared to re-

Application (Input size)	Conv. Run (s)	Self-Adj. Run (s)	Self-Adj. Avg. Prop. (s)	Overhead	Speedup
map(10^6)	0.05	0.83	1.1×10^{-6}	16.7	4.6×10^4
filter(10^6)	0.04	1.25	1.4×10^{-6}	27.7	3.2×10^4
split(10^6)	0.14	1.63	3.2×10^{-6}	11.6	4.4×10^4
msort(10^5)	0.30	5.83	3.5×10^{-4}	19.5	850.92
qsort(10^5)	0.05	3.40	4.9×10^{-4}	64.2	108.17
vec-reduce(10^6)	0.05	0.26	4.4×10^{-6}	5.5	1.1×10^4
vec-mult(10^6)	0.18	1.10	6.7×10^{-6}	5.9	2.8×10^4
mat-vec-mult(10^3)	0.17	0.81	1.4×10^{-5}	4.6	1.3×10^4
mat-add(10^3)	0.10	0.36	4.9×10^{-7}	3.7	2.0×10^5
transpose(10^4)	2.14	2.15	5.1×10^{-8}	1.0	4.2×10^7
mat-mult(400)	10.65	90.22	5.8×10^{-3}	8.5	1.8×10^3
block-mat-mult(10^3)	7.03	8.38	4.6×10^{-3}	1.2	1.5×10^3

Table 1. Summary of benchmark timings.

computing with the conventional version. An analysis of the data shows that the overheads are higher for simple benchmarks such as list operations (which are dominated by memory accesses), but significantly lower for other benchmarks. The overheads for `qsort` are traditionally high, commensurate with the previous work. In all benchmarks, we observe massive speedups, thanks to the asymptotic improvements delivered by change propagation. In the common case, the overhead is incurred only once: after a self-adjusting run, we can change the input data incrementally and use change propagation, with massive speedups.

4.5 Experiments: Merge Sort

Although it is not apparent from the summary in Table 1, our experiments show that for all our benchmarks, the overheads of self-adjusting versions are constant and do not depend on the input size, whereas speedups, being asymptotically significant, increase with the input size. To illustrate this property, we examine our merge sort benchmark; in Appendix A, we show the corresponding data for four more representative benchmarks.

The plot on the left in Figure 6 shows the time (in seconds) for the complete run of self-adjusting merge sort compared to the conventional version, for a range of input sizes (x axis). The figure suggests that, in both the conventional and self-adjusting versions, the complete-run time grows almost linearly, and they exhibit the same asymptotic complexity, $O(n \log n)$. The plot in the middle of Figure 6 shows the time—in milliseconds—for change propagation after inserting/deleting one element for each input size (x axis). As can be seen, the time taken by change propagation grows sublinearly as the input size increases. This is consistent with the $O(\log n)$ bound that we can show analytically. The plot on the right in Figure 6 shows the speedups for different input sizes (x axis): the time for a run of the conventional algorithm divided by the time for change propagation. The plots show that speedups increase linearly with the input size, consistent with the theoretical bound. To obtain this asymptotic improvement, we only insert one keyword in the code and use our compiler to generate the self-adjusting version.

4.6 Experiments: Matrix Multiplication

Our type-based approach gives the programmer great flexibility in specifying changeable elements in different granularity. The difference in the data representation can lead to dramatically different time and space performance. For example, `mat-mult` performs matrix multiplication using the standard matrix representation where each element of the matrix is changeable, while `block-mat-mult` considers the blocked representation where elements are blocked into groups of 20×20 submatrices. As we can see from Table 1, although the blocked version has a smaller speedup, as changing one element requires recomputing the whole submatrix, it has much less overhead compared to the standard representation. To further explore this trade-off, Figure 7 shows the time and space of blocked matrix multiplication with block sizes from 20×20 to 50×50.

Run time. In Figure 7, the leftmost plot shows the time for the complete run of self-adjusting blocked-matrix multiply with different block sizes, as well as the conventional version. The figure suggests that all benchmarks exhibit the same asymptotic complexity, $O(n^3)$. We also observe that as the block size increases, the overhead becomes smaller. This is because we treat each block as a single modifiable, reducing the number of modifiables tracked at run time. The second plot in Figure 7 shows the time for change propagation after changing a block for each input size (x axis) with different block sizes. The time taken by change propagation grows almost linearly as the input size increases, which is consistent with the $O(n \log n)$ bound that we can compute analytically. Changing any part of a block requires recomputing the whole block, so propagation is faster with smaller block sizes. The third plot in Figure 7 shows speedups for different input sizes. Speedups increase asymptotically with input size, consistent with the theoretical bound of $O(n^2/\log n)$. Smaller blocks have higher speedups. For example, for a 1000×1000 matrix, the 20×20 block enables $1200\times$ speedup, while the 50×50 block has $280\times$ speedup.

Space. The rightmost plot in Figure 7 shows the memory used by change propagation with different block sizes (the complete run never uses more memory than change propagation). As with all other approaches to self-adjusting computation, our approach exploits a trade-off between memory (space) and time (we compare the space usage of other approaches in Section 4.9). We store computation traces in memory and use them to respond to incremental data changes, resulting in an increase in memory usage and a decrease in response time. Typically, self-adjusting programs use asymptotically as much memory as the run-time of the computation. The plot shows results consistent with the theoretical bound of $O(n^3)$. Although a smaller block size leads to larger speedups, it requires more memory, because the total number of modifiables created is proportional to the number of blocks in the matrix.

As can be seen from the plot, memory consumption can be high. However, it can be reduced by programmer control over dependencies, which self-adjusting computation provides [Acar et al. 2009; Hammer et al. 2011]: The programmer can specify larger chunks of data, instead of single units, as changeable. Our approach further simplifies such control by requiring only the types to be changed. As a concrete example, our matrix-multiplication benchmark with 50×50 blocks consumes about 300 MB as we change each element of the input matrix and update the output. This is about 10 times the space needed to store the two input matrices and the result matrix, but enables a $280\times$ speedup when re-computing the output.

4.7 Experiments: Ray Tracer

Many applications are suitable for incremental computation, because making a small change to their input on average causes small changes to their output. But some applications are not. Arguably, incremental computation techniques should be avoided or used cautiously in such applications. To evaluate the effectiveness of our approach in such limiting cases, we considered ray tracing, where a small change to the input can require a large fraction of the output image to be updated. In our experiments, we rendered an input scene of 3 light sources and 19 objects with an output image size of 512×512 and then repeatedly changed the surface properties

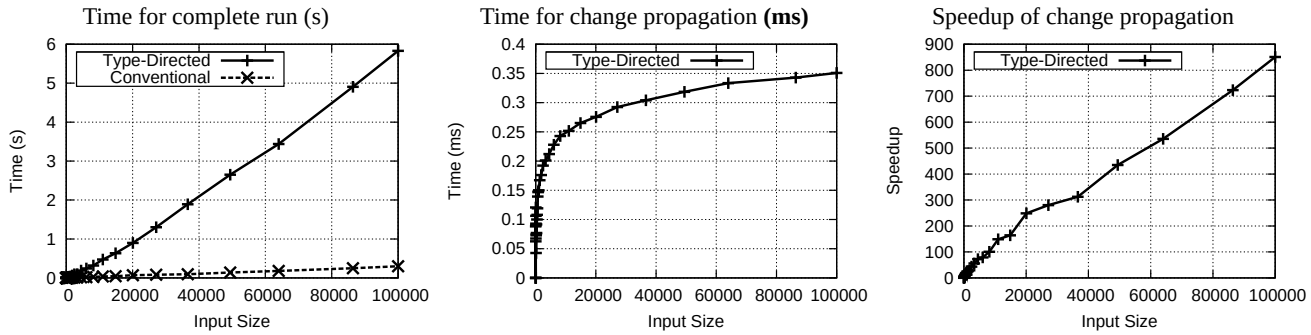

Figure 6. Time for complete run; time and speedup for change propagation for `msort`

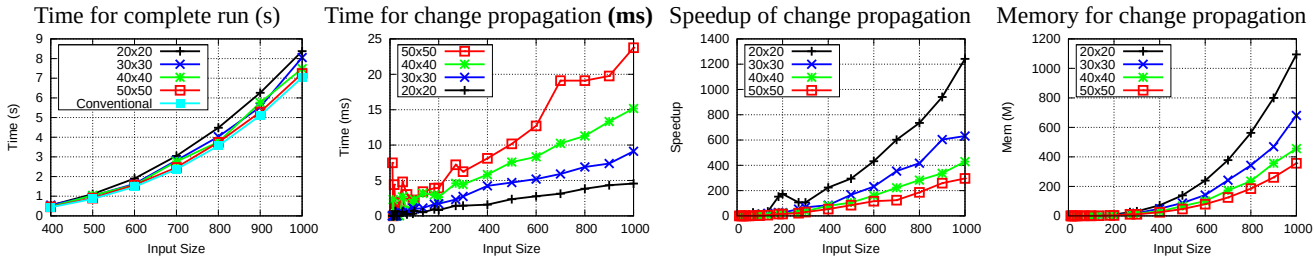

Figure 7. Time for complete run; time, speedup and memory for change propagation for blocked matrix multiply

Surface Changed	Image Diff. (% pixels)	Conv. Run (s)	Self-Adj. Run (s)	Self-Adj. Avg. Prop. (s)	Overhead	Speedup
A^D	57.22%	4.07	6.32	3.04	1.55	1.34
A^M	57.22%	1.91	5.75	8.48	3.02	0.22
B^D	8.43%	2.37	4.87	0.55	2.05	4.29
B^M	8.43%	2.44	4.42	1.00	1.81	2.44
C^D	9.20%	2.43	3.97	0.59	1.64	4.09
C^M	9.20%	2.16	3.86	1.12	1.79	1.92
D^D	1.85%	2.44	3.83	0.12	1.57	20.21
D^M	1.85%	2.19	3.85	0.20	1.76	10.74
E^D	11.64%	4.10	6.28	1.27	1.53	3.22
E^M	11.74%	2.79	5.83	1.87	2.09	1.49
F^D	19.47%	2.85	5.78	1.57	2.03	1.82
F^M	19.47%	2.83	3.92	2.97	1.38	0.95
G^D	27.37%	2.85	3.92	2.58	1.38	1.11
G^M	27.47%	2.82	5.36	4.64	1.90	0.61

Table 2. Summary of ray tracer timings.

of a single surface (which may be shared by multiple objects in the scene). We considered three kinds of changes: a change to the color of a surface, a change from a diffuse (non-reflective) surface to a mirrored surface, and a change from a mirrored to diffuse surface. We measured the time for a complete run of both the conventional and the self-adjusting versions, and the average propagation time for a single toggle of a surface property. For each change to the input, we also measured the change in the output image as a fraction of pixels. Figure 8 illustrates an example.

Table 2 shows the timings for various kinds of changes. The first column shows the percentage of pixels changed in the output. Each pair of rows corresponds to changing the surface properties of a set of objects (sets labeled A through G) to diffuse (non-reflective)

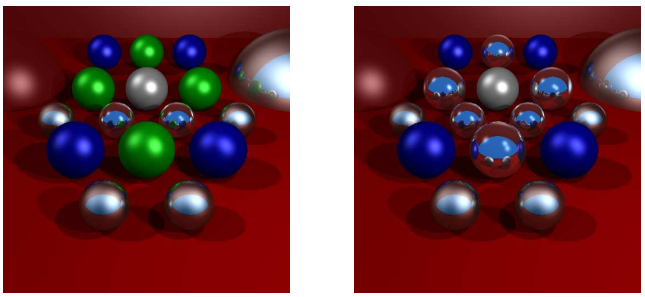

Figure 8. Two images produced by our ray tracer. To produce the right-hand image, we change the surfaces of the four green balls from diffused surfaces to mirrored surfaces. Change propagation yields about a 2× speedup over re-computing the output.

and mirror surfaces in that order, as indicated by superscripts $\cdot^D$ and $\cdot^M$ respectively. Our measurements show that even when a large fraction of the output image must change, our approach can perform better than recomputing from scratch. We also observe that since mirrors reflect light, making a surface mirrored often requires performing more work during change propagation than making the surface diffuse. Indeed, we observe that the speedups obtained for mirror changes are consistently about half of the speedups for diffuse changes.

4.8 Experiments: Compiler Optimizations

In Section 3.4 we described some key optimizations that eliminate redundancies in the code. To measure the effectiveness of these optimizations, we measured the running time for our benchmarks compiled with and without these optimizations. Since the optimizations always eliminate redundant calls, we expected them to improve efficiency consistently, and also quite significantly. As can be seen in Figure 9 by comparing the bars labeled "Unopt." (green) and "Type-Directed" (black), our experiments indeed show that the optimizations can improve the time of the complete run and the

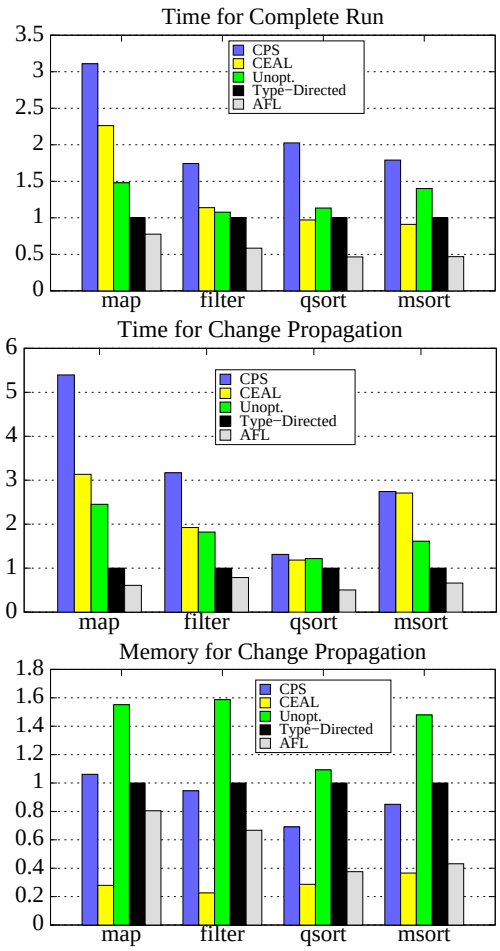

Figure 9. Time and memory for complete run and change propagation at fixed input sizes, normalized to "Type-Directed" (set to 1). We note for readability that the top-down order of the legend corresponds to the horizontal ordering of the bars.

time and space for change propagation by as much as 60%. The complete run never uses more space than change propagation does. We will discuss the rest of Figure 9 in Section 4.9.

4.9 Experiments: Comparison to Previous Work

We compare our results with previous work: the combinator library (AFL) in SML [Acar et al. 2009], the continuation-passing style (CPS) approach in SML [Ley-Wild et al. 2008], and the C-based CEAL system [Hammer et al. 2011], which is a carefully engineered and highly optimized system that can be competitive with hand-crafted algorithms [Demetrescu et al. 2004]. Figure 9 shows this comparison for the common benchmarks with fixed input sizes of 1 million keys for list operations and 100,000 keys for sorting, with results normalized to "Type-Directed" (= 1.0).

The comparison shows that, for both time and space, our approach is within a factor of two of AFL, a carefully engineered hand-written library. The principal reason for AFL's performance is its multiple interfaces to the self-adjusting primitives, which the programmer selects by hand. For example, AFL provides an unsafe interface that the quicksort benchmark uses to speed up the partition, creating half as many modifiables as with the standard interface. Our compiler does not directly support these low-level primitives, so we cannot perform the same optimizations. In AFL, programmers need to restructure programs in monadic style, explicitly

Surface Changed	Image Diff. (% pixels)	Type-Dir. Run (s)	CPS Run (s)	Speedup vs. CPS	Type-Dir. Prop (s)	CPS Prop (s)	Speedup vs. CPS
A^D	57.22%	6.32	5.88	0.93	3.04	4.36	1.43
A^M	57.22%	5.75	7.35	1.28	8.48	13.86	1.64
B^D	8.43%	4.87	8.06	1.66	0.55	1.01	1.82
B^M	8.43%	4.42	7.75	1.75	1.00	1.80	1.80
C^D	9.20%	3.97	7.97	2.01	0.59	1.15	1.93
C^M	9.20%	3.86	7.63	1.98	1.12	1.93	1.72
D^D	1.85%	3.83	7.95	2.07	0.12	0.21	1.75
D^M	1.85%	3.85	7.57	1.97	0.20	0.28	1.39
E^D	11.64%	6.28	5.88	0.94	1.27	2.52	1.98
E^M	11.74%	5.83	12.41	2.13	1.87	3.44	1.84
F^D	19.47%	5.78	11.96	2.07	1.57	3.00	1.91
F^M	19.47%	3.92	9.44	2.41	2.97	5.41	1.82
G^D	27.37%	3.92	9.64	2.46	2.58	4.69	1.82
G^M	27.47%	5.36	11.02	2.06	4.64	8.60	1.85

Table 3. Comparison of ray tracer with CPS

constructing the dependency graph. This process is similar to doing type inference and translation by hand. Our approach makes self-adjusting programs much easier to write, yet their performance is competitive with carefully-engineered self-adjusting programs.

Compared to CPS, our approach is approximately twice as fast, even though the CPS approach requires widespread changes to the program code and ours does not. The primary reason for the performance gap is likely that the CPS-based transformation relies on coarse approximations of true dependencies (based on continuations); our compiler identifies dependencies more precisely by using a type-directed translation. Additionally, we compared our ray tracer with one based on CPS, where our approach ("Type-Dir.") is approximately twice as fast as CPS (Table 3). Our approach often uses slightly more space than the CPS based approach, probably because of redundancies in the automatically generated code that can be eliminated manually in the CPS approach.

Compared to CEAL [Hammer et al. 2011], our approach is usually faster but occasionally slightly slower. We find this very interesting because the CEAL benchmarks use hand-written, potentially unsound optimizations, such as selective destination-passing and sharing of trace nodes [Hammer et al. 2011, Sections 7.2 and 8.1], that can result in incorrectly updated output. We also compared our approach to sound versions of CEAL benchmarks, which were a factor of two slower than the unsound versions. We use up to five times as much space; given that our approach uses space consistent with the other ML based approach ("CPS"), this is probably because of differences between ML, a functional, garbage-collected language, and C. When compared to sound versions of the CEAL benchmarks, which is arguably the more fair comparison, CEAL's space advantage decreases by a factor of two.

To summarize, even though our approach accepts conventional code with only a few type annotations, the generated programs perform better than most hand-written code in two programming languages, and are competitive with hand-written code in AFL. Memory usage is comparable to other ML-based approaches.

4.10 Experiments: the Effect of Garbage Collection

In our evaluation thus far, we did not include garbage-collection times because they are very sensitive to garbage collection parameters, which can be specified during run time. For example, our compiler allows us to specify a heap size when executing a program. If this heap size is sufficiently large to accommodate the live data of our benchmarks, then the timings show that essentially no time is spent in garbage collection. When we do not specify a heap size, memory is managed automatically, taking care not to over-expand the heap unnecessarily, by keeping the heap size close to the size of

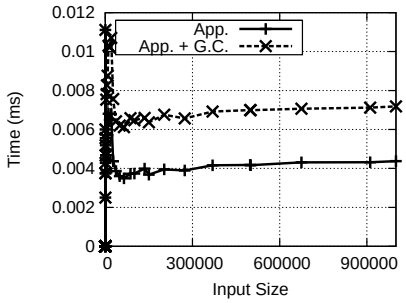

Figure 10. Propagation time for `vec-reduce` including GC time.

the live data. With this setting, our timings show that garbage collection behaves differently in the complete run and change propagation. The time can vary from negligible to moderate during complete runs of self-adjusting executables. For example, in blocked matrix multiplication, garbage collection times are less than 10%, but in vector multiplication, garbage collection takes nearly half of the total running time. Previous work on self-adjusting computation shows similar tradeoffs [Hammer and Acar 2008; Acar et al. 2009]. During change propagation, however, we observe that garbage-collection times are relatively small even when not using a fixed heap. Figure 10 shows the garbage collection time for vector reduce, which is close to the worst-case typical behavior that we obtain in our benchmarks. In some applications such as ray-tracing and blocked matrix multiplication, garbage collection times are negligible.

5. Related Work

The problem of enabling computation to respond efficiently to changes has been studied extensively. We briefly examine some earlier techniques for incremental computation and information flow. Detailed background can be found in several excellent surveys [Ramalingam and Reps 1993; Chiang and Tamassia 1992; Agarwal et al. 2002; Demetrescu et al. 2005; Sabelfeld and Myers 2003].

Incremental and self-adjusting computation. Earlier work on incremental computation, which took place in the '80s and '90s, was primarily based on dependence graphs and memoization. Dependence graphs record the dependencies between data in a computation and use a change-propagation algorithm to update the computation when the input is modified [Demers et al. 1981; Hoover 1987]. Dependence graphs have been effective in applications such as syntax-directed computations, but are not general-purpose because change propagation cannot update the dependence structure. As an alternative, researchers have proposed memoization (also called function caching) [Pugh and Teitelbaum 1989; Abadi et al. 1996; Heydon et al. 2000]. A classic idea [Bellman 1957; McCarthy 1963; Michie 1968], memoization applies to any purely functional program. It improves efficiency when executions of a program with similar inputs involve similar function calls, but such calls are relatively rare: small modifications to input can prevent reuse by changing the arguments to many function calls.

More recent work on self-adjusting computation proposed a particular technique for incremental computation that combines dynamic dependence graphs [Acar et al. 2006] and a form of computational memoization [Acar et al. 2009] to achieve efficient updates. Variants of self-adjusting computation have been implemented in several host languages such as C [Hammer et al. 2011], Java [Shankar and Bodik 2007], Haskell [Carlsson 2002], and SML [Ley-Wild et al. 2008]. Self-adjusting computation often achieves asymptotically efficient updates for a reasonably broad range of benchmarks [Acar et al. 2009; Hammer et al. 2011], can help verify runtime invariants [Shankar and Bodik 2007], and even help solve major open problems in many domains including computational geometry [Acar et al. 2010] and machine learning [Sümer et al. 2011]. More recent work shows that the approach can be generalized to parallel computations, taking simultaneous advantage of parallelism and incremental computation time by exploiting structural similarities between them [Hammer et al. 2007; Burckhardt et al. 2011; Acar et al. 2011], as well as large-scale distributed systems [Bhatotia et al. 2011].

Of all these previous approaches, DITTO [Shankar and Bodik 2007] and Incoop [Bhatotia et al. 2011] have the advantage of being completely transparent—they require no programmer annotations or changes to the code. But they only target specific domains—invariant checking for DITTO and large-scale MapReduce computations for Incoop—making them unsuited to general-purpose computations. Of the general approaches, library-based systems in SML and in C# [Acar et al. 2009; Burckhardt et al. 2011] require the programmer to guarantee certain invariants for correctness. These invariants are nontrivial to check statically or dynamically, and motivated the approaches taken by ΔML and CEAL.

ΔML [Ley-Wild et al. 2008] and the most recent version of the CEAL language system [Hammer et al. 2011] can ensure that self-adjusting programs respond to changes to their input correctly. As described in Section 2, ΔML requires writing self-adjusting programs in an explicit style by inserting several primitives that can require substantial changes to the code. Recent work on CEAL shows that a large fraction of annotations can be eliminated, but at the cost of tracing and recording all dependencies, which can lead to significant loss of time efficiency and space blowup.

The approach we describe in this paper uses a type-directed translation to enable selective dependency tracking, recording only the parts of the computation that can be affected by the changes. It guarantees that the output is updated correctly and efficiently under any changes to the data. The approach is based on the theoretical work of Chen et al. [2011]. That work, however, considered a minimal language and provided no implementation or practical evidence that the approach can be realized in practice. As our experimental evaluation shows, our implementation performs very well, usually outperforming ΔML and CEAL even though they require heavy programmer involvement, while our approach required only tiny changes to type declarations and no changes to the code itself. Our approach thus allows taking advantage of the benefits of self-adjusting computation without the burden of major program restructuring.

Information flow. A number of information flow type systems have been developed to check security properties, including the SLam calculus [Heintze and Riecke 1998], JFlow [Myers 1999] and a monadic system [Crary et al. 2005]. To save energy by approximating subcomputations, Sampson et al. [2011] use information flow to analyze dependencies. The type system of Chen et al. [2011] used many ideas from Pottier and Simonet [2003].

Coco [Swamy et al. 2011] transforms constructions such as effects from impure style (as in ML) to an explicit monadic style (as in Haskell). In other words, it translates effects in lightweight style into effects in a heavyweight style. But it does not support implicit self-adjusting computation: uses of effects, though lightweight compared to monadic style, must be explicit in the source program. Even such relatively lightweight constructs are pervasive in explicit self-adjusting computations and, compared to implicit self-adjusting computation, very tedious to program with.

6. Conclusion

We present the design, implementation, and evaluation of a programming language and compiler that enable programmers to write

programs that respond automatically to changes to their data, using only simple type annotations. By design, our approach guarantees that the compiled programs respond to data changes correctly. Through self-adjusting-computation techniques and a carefully designed type-directed mechanism for identifying dependencies, the compiler yields executables that can respond to incremental changes efficiently. Conventional benchmarks such as matrix multiplication and ray tracers can be compiled to efficient incremental executables, with only tiny changes to their type specifications. Our language and compiler take an important step in solving the long-standing problem of creating high-level languages for developing software that can react to incremental change correctly and efficiently with minimal programmer involvement.

Acknowledgments We thank the anonymous PLDI reviewers for their very useful comments on the submitted version of this paper, and Matthew A. Hammer for help with the CEAL benchmarks.

References

M. Abadi, B. W. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006.

U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, 2008.

U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–53, 2009.

U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010.

U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Parallelism in dynamic well-spaced point sets. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.

P. K. Agarwal, L. J. Guibas, H. Edelsbrunner, J. Erickson, M. Isard, S. Har-Peled, J. Hershberger, C. Jensen, L. Kavraki, P. Koehl, M. Lin, D. Manocha, D. Metaxas, B. Mirtich, D. Mount, S. Muthukrishnan, D. Pai, E. Sacks, J. Snoeyink, S. Suri, and O. Wolfson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *J. Funct. Program.*, 7(5):515–540, Sept. 1997.

F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011.

S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

M. Carlsson. Monads for incremental computing. In *International Conference on Functional Programming*, pages 26–35, 2002.

Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, pages 129–141, Sept. 2011.

Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2):249–291, 2005.

DeltaML. DeltaML web site. `http://ttic.uchicago.edu/~pl/sa-sml/`.

A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages*, pages 105–116, 1981.

C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 369–378, 2004.

C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005.

M. Hammer and U. A. Acar. Memory management for self-adjusting computation. In *International Symposium on Memory Management*, pages 51–60, 2008.

M. Hammer, U. A. Acar, M. Rajagopalan, and A. Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

M. Hammer, G. Neis, Y. Chen, and U. A. Acar. Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Principles of Programming Languages (POPL '98)*, pages 365–377, 1998.

A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Programming Language Design and Implementation*, pages 311–320, 2000.

R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Department of Computer Science, Cornell University, May 1987.

R. Jakob. *Dynamic Planar Convex Hull*. PhD thesis, Department of Computer Science, University of Aarhus, 2002.

D. J. King. A ray tracer for spheres, 1998. `http://www.cs.rice.edu/~dmp4866/darcs/nofib/spectral/sphere/`.

R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Int'l Conference on Functional Programming*, 2008.

J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

MLton. MLton web site. `http://www.mlton.org`.

A. C. Myers. JFlow: practical mostly-static information flow control. In *Principles of Programming Languages*, pages 228–241, 1999.

M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Prog. Lang. Sys.*, 25(1):117–158, Jan. 2003.

W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages*, pages 315–328, 1989.

G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pages 502–510, 1993.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), 2003.

A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Programming Language Design and Implementation*, pages 164–174, 2011.

A. Shankar and R. Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *Programming Language Design and Implementation*, 2007.

O. Sümer, U. A. Acar, A. Ihler, and R. Mettu. Fast parallel and adaptive updates for dual-decomposition solvers. In *Conference on Artificial Intelligence (AAAI)*, 2011.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *International Conference on Functional Programming (ICFP)*, Sept. 2011.

A. Appendix

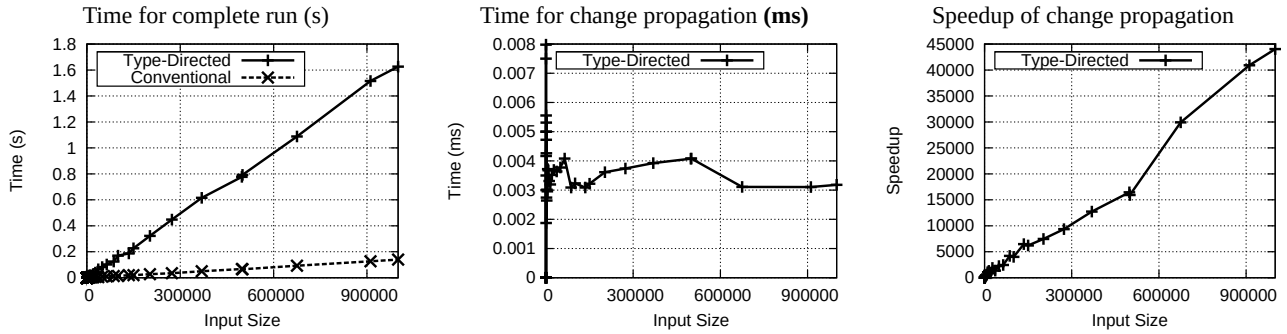

Figure 11. Time for complete run; time and speedup for change propagation for `split`

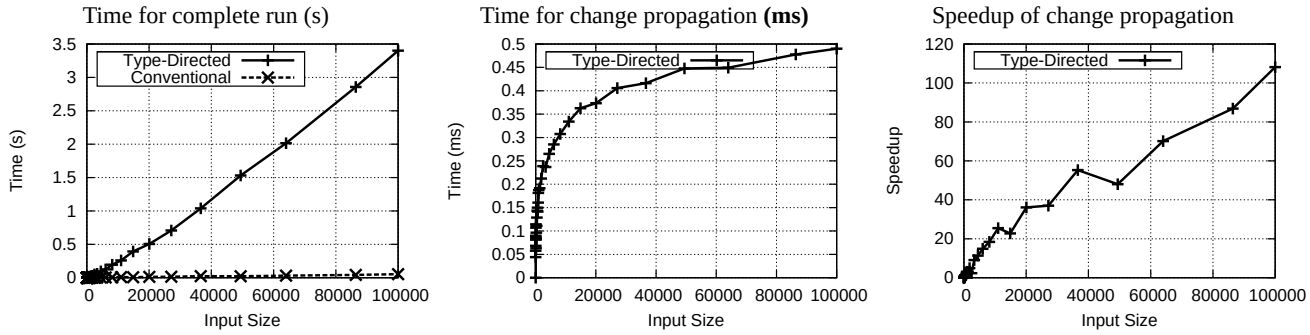

Figure 12. Time for complete run; time and speedup for change propagation for `qsort`

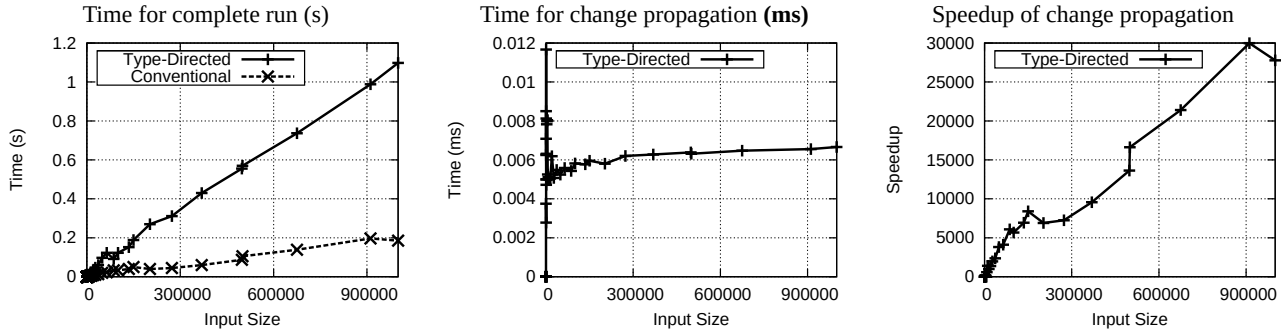

Figure 13. Time for complete run; time and speedup for change propagation for `vec-mult`

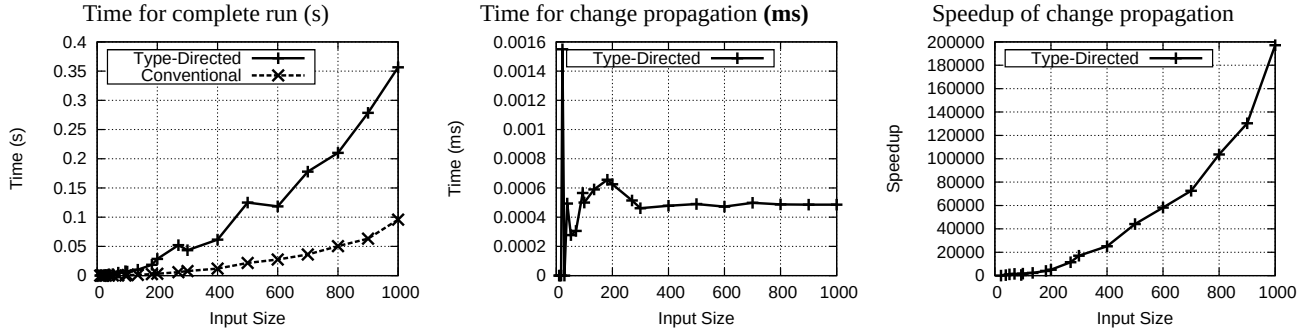

Figure 14. Time for complete run; time and speedup for change propagation for `mat-add`