# Functional Programming for Dynamic and Large Data with Self-Adjusting Computation

Yan Chen
Max Planck Institute for Software Systems
chenyan@mpi-sws.org

Umut A. Acar
Carnegie Mellon University
and INRIA
umut@cs.cmu.edu

Kanat Tangwongsan
Mahidol University International College
kanat.tan@mahidol.ac.th

## Abstract

Combining type theory, language design, and empirical work, we present techniques for computing with large and dynamically changing datasets. Based on lambda calculus, our techniques are suitable for expressing a diverse set of algorithms on large datasets and, via self-adjusting computation, enable computations to respond automatically to changes in their data. To improve the scalability of self-adjusting computation, we present a type system for precise dependency tracking that minimizes the time and space for storing dependency metadata. The type system eliminates an important assumption of prior work that can lead to recording spurious dependencies. We present a type-directed translation algorithm that generates correct self-adjusting programs without relying on this assumption. We then show a probabilistic-chunking technique to further decrease space usage by controlling the fundamental space-time tradeoff in self-adjusting computation. We implement and evaluate these techniques, showing promising results on challenging benchmarks involving large graphs.

**Categories and Subject Descriptors** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

**Keywords** Self-adjusting computation; information-flow type system; granularity control; incremental graph algorithms; performance

# 1. Introduction

Recent advances in the ability to collect, store, and process large amounts of information, often represented in the form of graphs, have led to a plethora of research on "big data." In addition to being large, such datasets are *diverse*, arising in many domains ranging from scientific applications to social networks, and *dynamic*, meaning they change gradually over time. For example, a social-network graph changes as users join and leave, or as they change their set of friends. Prior research on languages and programming systems for big-data applications has two important limitations:

- Due to their diversity, big-data applications benefit from expressive programming languages. Yet existing work offers domain-specific languages and systems such as "MapReduce" [20] with limited expressiveness that not only restrict the set of problems



*ICFP '14*, September 1–6, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2628136.2628150

that can be solved but also how efficiently they can be solved

(by limiting the algorithms that can be implemented).

- Even though big data applications often require operating on dynamically changing datasets, many existing languages and systems provide for a batch model of computation, where the data is assumed to be static or unchanging.

In this paper, we show that, when combined with the right set of techniques, functional programming can help overcome both of these limitations. First, as an expressive, general purpose programming model, functional programming enables efficient implementations of a broad range of algorithms for big data. Second, since functional programming is consistent with self-adjusting computation [1, 6, 15–17], it can also enable programs to respond efficiently to changing data provided that a major limitation of self-adjusting computation—space usage—can be overcome.

Self-adjusting computation [1, 6, 15–17] refers to a technique for compiling batch programs into programs that can automatically respond to changes to their data. The idea behind self-adjusting computation is to establish a space-time tradeoff so that the results of prior computations can be reused when computing the result for a different but similar input. Self-adjusting computation achieves this by representing the execution of a program as a higher-order graph data structure called a *dynamic dependency graph*, which records certain dependencies in the computation, and by using a *change-propagation algorithm* to update this graph and the computation. In a nutshell, change propagation identifies and rebuilds (via re-execution) only the parts of the computation that are affected by the changes. Unfortunately, existing approaches to self-adjusting computation require a significant amount of memory to store the dynamic dependency graph. For example, on a modest input of $10^7$

integers, a self-adjusting version of merge sort uses approximately 100x more space than its batch counterpart. Such demands for space can limit its applicability to relatively small problem sizes.

This paper presents two techniques for improving space usage of self-adjusting computation. The first technique reduces space usage by improving the precision of dependency tracking that self-adjusting computation relies on. The second technique enables programmers to control the space-time tradeoff fundamental to self-adjusting computation. Our first technique relies on a type system for precisely tracking dependencies and a type-directed translation algorithm that can generate correct and efficient self-adjusting programs. Our second technique is a probabilistic chunking scheme for coarsening the granularity at which dependencies are tracked without disproportionately degrading the update performance.

Our starting point is the recent work on type-based automatic incrementalization [15–17]. That work enables translating batch programs into self-adjusting programs that can efficiently respond to incremental changes. The idea behind the approach is to use a type inference algorithm to infer all *changeable data*, which

change over time, and track only their dependencies. Unfortunately, the type inference algorithm can identify non-changeable data as changeable, causing redundant dependencies to be recorded. The reason for this is the modal type system that all previous work on self-adjusting computation relies on [6]. That modal type system ensures a crucial property, that all relevant dependencies are tracked, but at the cost of being conservative and disallowing changeable data from being nested inside non-changeable data, which leads to redundant dependencies.

We solve this problem by designing a more refined type systems (Section 3) and a translation algorithm (Section 6) that can correctly translate source programs (Section 4) into a lower-level target language (Section 5). Our source-level type system is an information-flow type system that enables precise dependency tracking. This type system break an important assumption of prior work, allowing changeable data to be nested inside non-changeable data. We present a translation algorithm that nevertheless produces correct self-adjusting executables by emitting target code written in a destination-passing style. To provide the flexibility needed for operation on changeables without creating redundant dependencies, the target language is imperative but relies on a type and effect system [52] for correctness, guaranteeing that all dependencies are tracked. We prove that the translation generates well-typed and sound target code, consistent with the source typing, and thus guarantees the correctness of the resulting self-adjusting code in the target language.

When combined with an important facility in self-adjusting computation—the ability to control the granularity of dependency

tracking by selectively tracking dependencies—precise dependency tracking offers a powerful mechanism to control the space-time tradeoff fundamental to self-adjusting computation. By tracking dependencies at the level of (large) blocks of data, rather than individual data items, the programmer can further reduce space consumption. As we describe, however, this straightforward idea can lead to disproportionately slow updates (Section 7), because it can cause a small change to propagate to many blocks. We overcome this problem by presenting a *probabilistic blocking* technique. This technique divides the data into blocks in a probabilistic way, ensuring that small changes affect a small number of blocks.

We implement the proposed techniques in Standard ML and present an empirical evaluation by considering several list primitives, sorting algorithms, and more challenging algorithms for large graphs such as PageRank, graph connectivity, and approximate social-circle size. These problems, which are highly unstructured, put our techniques through a serious test. Our empirical evaluation leads to the following conclusions.

- Expressive languages such as lambda calculus augmented with simple type annotations, instead of domain-specific languages such as MapReduce, can lead to large (e.g. 50-100x) improvements in time and space efficiency.
- The type system for precise dependency tracking can significantly reduce space and time requirements (e.g., approximately by 2x and 6x respectively for MapReduce applications).
- Our techniques for controlling the space-time tradeoff for list data structures can reduce memory consumption effectively while only proportionally slowing down updates.
- Our techniques can enable responding significantly faster (e.g.,

several orders of magnitude or more) to both small and aggregate changes while moderately increasing memory usage compared to the familiar batch model of computation.

## 2. Background and Overview

Using a simple list-partitioning function, we illustrate the self-adjusting computation framework, outline two limitations of previous approaches, and describe how we resolve them.

### 2.1 Background and List Partition

Figure 1 shows SML code for a list-partition function `partition f l`, which applies $f$ to each element $x$ of $l$, from left to right, and returns a pair (pos, neg) where pos is the list of elements for which $f$ evaluated to true, and neg is the list of those for which $f\ x$ evaluated to false. The elements of pos and neg retain the same relative order from $l$. Ignoring the annotation $\mathbb{C}$, this is the same function from the SML basis library, which takes $\Theta(n)$ time for a list of size $n$.

Self-adjusting computation enables the programmer to develop efficient incremental programs by annotating the code for the non-incremental or batch programs. The key language construct is a *modifiable (reference)*, which stores a *changeable* value that may change over time [6]. The runtime system of a self-adjusting language track dependencies on modifiables in a dynamic dependency graph, enabling efficient *change propagation* when the data changes in small amounts.

Developing a self-adjusting program can involve significant changes to the batch program. Recent work [15–17] proposes a type-

directed approach for automatically deriving self-adjusting programs via simple type annotations. For example, given the code in the leftmost column of Figure 1 and the annotation $\mathbb{C}$ (the second line) that marks the tail of the list changeable, the compiler automatically derives the code in the middle figure.

These type annotations, broadly referred to as *level types*, partition all data types into stable and changeable *levels*. Programmers only need to annotate the types of changeable data with $\mathbb{C}$; all other types remain stable, meaning they cannot be changed later on. For example, $\mathbf{int}^{\mathbb{S}}$ is a stable integer, $\mathbf{int}^{\mathbb{C}}$ is a changeable integer and $\mathbf{int}^{\mathbb{S}}\ \mathbf{list}^{\mathbb{C}}$ is a changeable list of stable integers. This list allows insertion and deletion but each individual element cannot be altered.

In the translated code (Figure 1, middle), changeable data are stored in modifiables: a changeable `int` becomes an `int` **`mod`**. Given the self-adjusting list-partition function, we can run it in much the same way as running the batch version. After a complete first run, we can change any or all of the changeable data and update the output by performing change propagation. As an example, consider inserting an element into the input list and performing change propagation. This will trigger the execution of computation on the newly inserted elements without recomputing the whole list. It is straightforward to show that change propagation takes $\Theta(1)$ time for a single insertion.

## 2.2 Limitation 1: Redundant Dependencies

**The problem.** As with all other prior work on self-adjusting computation (e.g. [1]) that relies on a type system to eliminate difficult correctness problem (in change propagation), recent work [15–17] uses a modal type system to guarantee properties important

to the correctness of self-adjusting computation—all changeables are initialized and all their dependencies are tracked. This type system can be conservative and disallow changeable data to be nested inside changeable data. For example, in list partition, the type system forces the return the type to be changeable, i.e., the type ($\alpha$ `list` **mod** `*` $\alpha$ `list` **mod**) **mod**. This type is conservative; the outer modifiable (**mod**) is unnecessary as any observable change can be performed without it. By requiring the outer modifiable, the type system causes redundant dependencies to be recorded. In this simple example, this can nearly double the space usage while also degrading performance (likely as much as an order of magnitude).

```
1  fun partition f l
2    : (α list^C * α list^C) =
3
4    case l of
5      nil ⇒ (nil, nil)
6
7    | h::t ⇒
8        let val (a,b) = partition f t
9
10
11
12       in if f h then (h::a, b)
13
14           else (a, h::b)
15
16       end
```

```
fun partition f l
  : (α list mod * α list mod) mod =
 read l as l' in
   case l' of
     nil ⇒ write (mod (write nil),
                  mod (write nil))
   | h::t ⇒
     let val pair = mod (partition f t)


     in if f h then read pair as (a,b) in
          write (mod (write h::a), b)
        else read pair as (a,b) in
          write (a, mod (write h::b))
      end
```

```
fun partition f l (l00,l01)
  : (α list mod * α list mod) =
 let val () = read l as l' in
     case l' of
       nil ⇒ (write (l00,nil);
              write (l01,nil))
     | h::t ⇒
       let val (a,b) = let
           val (l00,l01) = (mod nil, mod nil)
           in partition f t (l00,l01)
           end
       in if f h then (write (l00, h::a);
```

```
        read b as b' in write (l01, b'))
      else (read a as a' in write (l00, a');
            write (l01, h::b))
    end
in (l00,l01) end
```

**Figure 1.** The list partition function: ordinary (left), self-adjusting (center), and with destination passing (right).

***Our solution.*** We can circumvent this problem by using unsafe, imperative operations. For our running example, `partition` can be rewritten as shown in Figure 1(right), in a destination passing style. The code takes an input list and two destinations, which are recorded separately. Without restrictions of the modal type system, it can return ($\alpha$ `list` **mod** `*` $\alpha$ `list` **mod**), as desired.

A major problem with this approach, however, is correctness: a simple mistake in using the imperative constructs can lead to errors in change propagation that are extremely difficult to identify. We therefore would like to derive the efficient, imperative version automatically from its purely functional version. There are three main challenges to such translation. (1) The source language has to identify which data is written to which part of the aggregate data types. (2) All changeable data should be placed into modifiables and all their dependencies should be tracked. (3) The target language must verify that self-adjusting constructs are used correctly to ensure correctness of change propagation.

To address the first challenge, we enrich an information-flow type to check dependencies among different components of the changeable pairs. We introduce *label*s $\rho$ into the changeable level annotations, denoted as $\mathbb{C}_\rho$. The label serves as an identifier for

modifiables. For each function of type $\tau_1 \to \tau_2$, we give labels for the return type $\tau_2$. The information flow type system then infers the dependencies for each label in the function body. These labels decide which data goes into which modifiable in the translated code.

To address the second challenge, the translation algorithm takes the inferred labels from the source program, and conducts a type directed translation to generate self-adjusting programs in *destination passing style*. Specifically, the labels in the function return type are translated into destinations (modifiables) in the target language, and expressions that have labeled level types are translated into explicit write into their corresponding modifiables. Finally, we wrap the destinations into the appropriate type and return the value.

As an example, consider how we derive the imperative self-adjusting program for list partition, starting from the purely functional implementation on the leftmost column of Figure 1. First, we mark the return type of the partition function as $(\alpha\ \texttt{list}^{\mathbb{C}_{00}} * \alpha\ \texttt{list}^{\mathbb{C}_{01}})^{\mathbb{S}}$, which indicates the return has two destinations $l_{00}$ and $l_{01}$, and the translated function will take, besides the original arguments $f$ and $l$, two modifiables $l_{00}$ and $l_{01}$ as arguments. Then an information flow type system infers that the expression `(h::a,b)` on line 12 of Figure 1 (left) has type $(\alpha\ \texttt{list}^{\mathbb{C}_{00}} * \alpha\ \texttt{list}^{\mathbb{C}_{01}})^{\mathbb{S}}$. Using these label information, the compiler generates a target expression **write** `(`$l_{00}$`,h::a);` **write** `(`$l_{01}$`,b)`. Finally, the translated function returns the destination as a pair `(`$l_{00}$`,`$l_{01}$`)`. Figure 1 (right) shows the translated code for list partition using our translation.

To address the third challenge, we design a new type system for the imperative target language. The type system distinguishes the modifiable as fresh modifiables and finalized modifiables. The typing rules enforce that all modifiables are finalized before reading, and the function fills in all the destinations, no matter which control

branch the program is taken. We further prove that following the translation rules, we generate target programs that are of the appropriate type, and are type safe.

### 2.3 Limitation 2: Dependency Metadata.

**The problem.** Even with precise dependency tracking, self-adjusting programs can require large amounts of memory, making them difficult to scale to large inputs. One culprit is the dynamic dependency graph that stores operations on modifiables. For example, the list partition function contains about $n$ **read** operations. Our experiments show, for example, that self-adjusting list partition requires 41x more memory than its batch counterpart. In principle, there is a way around this: simply treat blocks of data as a changeable unit instead of treating each unit as a changeable. However, it turns out to be difficult to make this work because doing so can disproportionately degrade performance.

At a very high level, self-adjusting computation may be seen as a technique for establishing a trade-off between space and time. By storing the dependency metadata, the technique enables responding to small changes to data significantly faster by identifying and recomputing only the parts of the computation affected by the changes. It is natural to wonder whether it would be possible to control this trade-off so that, for example, a $1/B$-th fraction (for some $B$) of the dependency metadata is stored at the expense of an increased update time, hopefully by no more than a factor of $B$.

**Our solution.** To see how we might solve this problem, consider the following simple idea: partition the data into equal-sized blocks and treat each of these blocks as a unit of changeable computation at which dependencies are tracked. This intuitive idea is indeed simple

and natural to implement. But there is a fundamental problem: fixed-size chunking is highly sensitive to small changes to the input. As a simple example, consider inserting or deleting a single element to a list of blocks. Such a change will cascade to all blocks in the list, preventing much of the prior computation from being reused. Even "in-place" changes, which the reader may feel would not cause this problem, are in fact unacceptable because they do not compose. Consider, for example, the output to the `filter` function, which takes an input list and outputs only elements for which a certain predicate evaluates to true. Modifying an input element in-place may drop or add an element to the output list, which can create a

$$
\begin{array}{lll}
\textit{Levels} & \delta & ::= \mathbb{S} \mid \mathbb{C}_\rho \mid \alpha \\
\textit{Types} & \tau & ::= \mathbf{int}^\delta \mid (\tau_1 \times \tau_2)^\delta \mid (\tau_1 + \tau_2)^\delta \mid (\tau_1 \to \tau_2)^\delta \\
\textit{Constraints} & C, D & ::= \mathbf{true} \mid \mathbf{false} \mid \alpha = \beta \mid \alpha \leq \beta \mid \\
 & & \quad \delta \lhd \tau \mid \rho_1 = \rho_2
\end{array}
$$

**Figure 2.** Levels, types and constraints

ripple effect to all the blocks. The main challenge in these examples lies in making sure the blocks remain stable under changes.

We solve these problems by eliminating the intrinsic dependency between block boundaries and the data itself. More precisely, we propose a *probabilistic chunking scheme* that decides block boundaries using a (random) hash function independently of the structure of the data rather than deterministically. Using this technique, we are able to reduce size of the dependency metadata by a factor $B$ in expectation by chunking the data into blocks of expected size $B$ while taking only about a factor of $B$ hit in the update time.

## 3. Fine-grained Information Flow Types

In this section, we derive a type system for self-adjusting computation that can identify precisely which part of the data, down to individual attributes of a record or tuple, is changeable. In particular, we extend the surface type system from previous work to track fine-grained dependencies in the surface language.

The formalism rests on a simple insight that *data that depends on changeable data must itself be changeable*, similar to situations in information-flow type systems, where "secret" (high-security)

data is infectious; therefore, any data that depends on secret data itself must be secret.

To track dependency precisely, we distinguish different changeable data further by giving them unique labels. Our types include a lattice of *(security) levels*: stable and changeable with labels. We generally follow the approach and notation of Chen et al. [15, 17] except that we need not have a mode on function types.

**Levels.** *Levels* $\mathbb{S}$ (*stable*) and $\mathbb{C}_\rho$ (*changeable*) have a partial order:

$$\overline{\mathbb{S} \leq \mathbb{S}} \qquad \overline{\mathbb{C}_\rho \leq \mathbb{C}_\rho} \qquad \overline{\mathbb{S} \leq \mathbb{C}_\rho} \qquad \overline{\mathbb{C}_{1\rho} \leq \mathbb{C}_{0\rho}}$$

Stable levels are lower than changeable; changeable levels with different labels are generally incomparable. Here, labels are used to distinguish different changeable data in the program. We also assume that labels with prefix 1 are lower than labels with prefix 0. This allows changeable data to flow into their corresponding destinations (labeled with prefix 0). We will discuss the subsumption in Section 4.

**Types.** Types consist of integers tagged with their levels, products, sums and arrow (function) types with an associated level, as shown in Figure 2. The label $\rho$ associated with each changeable level denotes fine-grained dependencies among changeables: two changeables with the same label have a dependency between them.

**Labels.** Labels are identifiers for changeable data. To facilitate translation into a destination-passing style, we use particular binary-encoded labels that identify each label with its destination. This binary encoding works in concert with the relation $\tau \downarrow_\rho \mathcal{D}; \mathcal{L}$, in Figure 3, which recursively determines the labels with respect to a prefix $\rho$, where the type of the destinations and the destination names

are stored in $\mathcal{D}$ and $\mathcal{L}$, respectively. For stable product, rule (#prodS), we label it based on the structure of the product. Specifically, we append 0 if the changeable level is on the left part of a product, and we append 1 if the changeable level is on the right part of a product. For changeable level types, we require that the outer level label is $\rho$. The relation does not restrict the inner labels. For stable level integers, sums and arrows, we do not look into the type structure, the inner changeable types can be labeled arbitrarily. As an example,

$$\frac{}{\mathbf{int}^{\mathbb{S}} \downarrow_{\rho} \emptyset; \emptyset} \text{ (\#intS)} \qquad \frac{}{\mathbf{int}^{\mathbb{C}_{\rho}} \downarrow_{\rho} \{\mathbf{int}\}; \{l_{\rho}\}} \text{ (\#intC)}$$

$$\frac{}{(\tau_1 + \tau_2)^{\mathbb{S}} \downarrow_{\rho} \emptyset; \emptyset} \text{ (\#sumS)} \qquad \frac{}{(\tau_1 \to \tau_2)^{\mathbb{S}} \downarrow_{\rho} \emptyset; \emptyset} \text{ (\#funS)}$$

$$\frac{\tau_1 \downarrow_{\rho 0} \mathcal{D}; \mathcal{L} \qquad \tau_2 \downarrow_{\rho 1} \mathcal{D}'; \mathcal{L}'}{(\tau_1 \times \tau_2)^{\mathbb{S}} \downarrow_{\rho} \mathcal{D} \cup \mathcal{D}'; \mathcal{L} \cup \mathcal{L}'} \text{ (\#prodS)}$$

$$\frac{}{(\tau_1 \times \tau_2)^{\mathbb{C}_{\rho}} \downarrow_{\rho} \{(\tau_1 \times \tau_2)\}; \{l_{\rho}\}} \text{ (\#prodC)}$$

$$\frac{}{(\tau_1 + \tau_2)^{\mathbb{C}_{\rho}} \downarrow_{\rho} \{(\tau_1 + \tau_2)\}; \{l_{\rho}\}} \text{ (\#sumC)}$$

$$\frac{}{(\tau_1 \to \tau_2)^{\mathbb{C}_{\rho}} \downarrow_{\rho} \{(\tau_1 \to \tau_2)\}; \{l_{\rho}\}} \text{ (\#funC)}$$

**Figure 3.** Labeling changeable types

$$\left[\!\left[\mathbf{int}^{\delta}\right]\!\right] = \delta \qquad \left[\!\left[(\tau_1 + \tau_2)^{\delta}\right]\!\right] = \delta$$

$$\left[\!\left[(\tau_1 \times \tau_2)^{\delta}\right]\!\right] = \delta \qquad \left[\!\left[(\tau_1 \to \tau_2)^{\delta}\right]\!\right] = \delta$$

$$\mathbf{int}^{\delta_1} \doteq \mathbf{int}^{\delta_2} \qquad (\tau_1 + \tau_2)^{\delta_1} \doteq (\tau_1 + \tau_2)^{\delta_2}$$

$$(\tau_1 \times \tau_2)^{\delta_1} \doteq (\tau_1 \times \tau_2)^{\delta_2} \qquad (\tau_1 \to \tau_2)^{\delta_1} \doteq (\tau_1 \to \tau_2)^{\delta_2}$$

**Figure 6.** Outer level of types, and equality up to outer levels

$$\begin{array}{lll} \textit{Values} & v & ::= n \mid x \mid (v_1, v_2) \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v \mid \mathbf{fun}\ f(x) = e \\ \textit{Expr.'s} & e & ::= v \mid \oplus(x_1, x_2) \mid \mathbf{fst}\ x \mid \mathbf{snd}\ x \mid \\ & & \quad \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \mid \\ & & \quad \mathbf{apply}(x_1, x_2) \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \end{array}$$

**Figure 7.** Abstract syntax of the source language

$\tau = \left(\mathbf{int}^{\mathbb{C}_{00}} \times \left(\mathbf{int}^{\mathbb{C}_{111}} + \mathbf{int}^{\mathbb{S}}\right)^{\mathbb{C}_{01}}\right)^{\mathbb{S}}$ is a valid label for $\tau \downarrow_0 \mathcal{D}; \mathcal{L}$. The type for the destinations are $\mathcal{D} = \{\mathbf{int}, \left(\mathbf{int}^{\mathbb{C}_{111}} + \mathbf{int}^{\mathbb{S}}\right)\}$, and the destination names are $\mathcal{L} = \{l_{00}, l_{01}\}$.

**Subtyping.** Figure 4 shows the subtyping relation $\tau <: \tau'$, which is standard except for the levels. It requires that the outer level of the subtype is smaller than the outer level of the supertype.

**Levels and types.** We need relations between levels and types to ensure certain invariants. A type $\tau$ is *higher than* $\delta$, written $\delta \lhd \tau$, if the outer level of the type is at least $\delta$. In other words, $\delta$ is a lower bound of the outer level of $\tau$. For products with outer stable levels, we check if each component is higher than $\delta$. Note that we do not

check the component of a stable sum type. Figure 5 defines this relation.

We define an outer-level operation $[\![\tau]\!]$ that derives the outer level of a type in Figure 6). Finally, two types $\tau_1$ and $\tau_2$ are *equal up to their outer levels*, written $\tau_1 \stackrel{\circ}{=} \tau_2$, if $\tau_1 = \tau_2$ or they differ only in their outer levels.

## 4. Source Language

**Abstract syntax.** Figure 7 shows the syntax for our source language, a purely functional language with integers (as base types), products, and sums. The expressions consist of values (integers, pairs, tagged values, and recursive functions), projections, case expressions, function applications, and let bindings. For convenience, we consider only expressions in A-normal form, which names in-

$$\frac{\delta \le \delta'}{\mathbf{int}^{\delta} <: \mathbf{int}^{\delta'}}\ (\text{subInt}) \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2' \quad \delta \le \delta'}{(\tau_1 \times \tau_2)^{\delta} <: (\tau_1' \times \tau_2')^{\delta'}}\ (\text{subProd}) \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2' \quad \delta \le \delta'}{(\tau_1 + \tau_2)^{\delta} <: (\tau_1' + \tau_2')^{\delta'}}\ (\text{subSum}) \qquad \frac{\delta \le \delta' \quad \tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2)^{\delta} <: (\tau_1' \to \tau_2')^{\delta'}}\ (\text{subArr})$$

**Figure 4.** Subtyping

$$\frac{\delta \le \delta'}{\delta \lhd \mathbf{int}^{\delta'}}\ (\lhd\text{-Int}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 \times \tau_2)^{\delta'}}\ (\lhd\text{-Prod}) \qquad \frac{\delta \lhd \tau_1 \qquad \delta \lhd \tau_2}{\delta \lhd (\tau_1 \times \tau_2)^{\mathbb{S}}}\ (\lhd\text{-InnerProd}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 \to \tau_2)^{\delta'}}\ (\lhd\text{-Arrow}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 + \tau_2)^{\delta'}}\ (\lhd\text{-Sum})$$

**Figure 5.** Lower bound of a type

$C; \mathcal{P}; \Gamma \vdash e : \tau$

Under constraint $C$, label set $\mathcal{P}$ and source typing environment $\Gamma$, source expression $e$ has type $\tau$

$$\frac{}{C; \mathcal{P}; \Gamma \vdash n : \mathbf{int}^{\mathbb{S}}}\ (\text{SInt}) \qquad \frac{\Gamma(x) = \tau}{C; \mathcal{P}; \Gamma \vdash x : \tau}\ (\text{SVar})$$

$$\frac{C; \mathcal{P}; \Gamma \vdash v_1 : \tau_1 \qquad C; \mathcal{P}; \Gamma \vdash v_2 : \tau_2}{C; \mathcal{P}; \Gamma \vdash (v_1, v_2) : (\tau_1 \times \tau_2)^{\mathbb{S}}}\ (\text{SPair})$$

$$\frac{C; \mathcal{P}; \Gamma \vdash v : \tau_1}{C; \mathcal{P}; \Gamma \vdash \mathbf{inl}\ v : (\tau_1 + \tau_2)^{\mathbb{S}}}\ (\text{SSum}) \qquad \frac{C; \mathcal{P}; \Gamma \vdash x : (\tau_1 \times \tau_2)^{\delta}}{C; \mathcal{P}; \Gamma \vdash \mathbf{fst}\ x : \tau_1}\ (\text{SFst})$$

$$\frac{C; \{1\rho\}; \Gamma, x : \tau_1, f : (\tau_1 \to \tau_2)^{\mathbb{S}} \vdash e : \tau_2 \qquad [\![\tau_1]\!] = \mathbb{C}_{1\rho} \qquad C \Vdash \tau_2 \downarrow_0 \mathcal{D}; \mathcal{L}}{C; \mathcal{P}'; \Gamma \vdash (\mathbf{fun}\ f(x) = e) : (\tau_1 \to \tau_2)^{\mathbb{S}}}\ (\text{SFun})$$

$$\frac{C; \mathcal{P}; \Gamma \vdash x_1 : \mathbf{int}^{\delta_1} \quad C; \mathcal{P}; \Gamma \vdash x_2 : \mathbf{int}^{\delta_2} \quad C \Vdash \delta_1 = \delta_2 \quad \oplus : \mathbf{int} \times \mathbf{int} \to \mathbf{int}}{C; \mathcal{P}; \Gamma \vdash \oplus(x_1, x_2) : \mathbf{int}^{\delta_1}}\ (\text{SPrim})$$

$$C; \mathcal{P}; \Gamma \vdash e_1 : \tau' \qquad C \Vdash \tau' <: \tau'' \quad [\![\tau'']\!] = \mathbb{C}_{\rho}$$

$$\frac{C; \mathcal{P} \cup \{\rho\}; \Gamma, x : \tau'' \vdash e_2 : \tau \quad C \Vdash \tau' \doteq \tau'' \quad C \Vdash \rho \notin \mathcal{P}}{C; \mathcal{P}; \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau} \text{ (SLet)}$$

$$\frac{C; \mathcal{P}; \Gamma \vdash x_1 : (\tau_1 \to \tau_2)^{\delta} \quad C; \mathcal{P}; \Gamma \vdash x_2 : \tau_1 \quad C \Vdash \delta \lhd \tau_2}{C; \mathcal{P}; \Gamma \vdash \textbf{apply}(x_1, x_2) : \tau_2} \text{ (SApp)}$$

$$\frac{\begin{array}{ll} C; \mathcal{P}; \Gamma \vdash x : (\tau_1 + \tau_2)^{\delta} & C; \mathcal{P}; \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \\ C \Vdash \delta \lhd \tau & C; \mathcal{P}; \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \end{array}}{C; \mathcal{P}; \Gamma \vdash \textbf{case } x \textbf{ of } \{x_1 \Rightarrow e_1 \ , \ x_2 \Rightarrow e_2\} : \tau} \text{ (SCase)}$$

**Figure 8.** Typing rules for source language

termediate results. A-normal form simplifies some technical issues, while maintaining expressiveness.

**Constraint-based type system.** The type system has the fine-grained level-decorated types and constraints (Figure 2) as was described in Section 3. After discussing the rules themselves, we will look at type inference.

The typing judgment $C; \mathcal{P}; \Gamma \vdash e : \tau$ has a constraint $C$, a label set $\mathcal{P}$ (storing used label names) and typing environment $\Gamma$, and infers type $\tau$ for expression $e$. Our work extends the type system in Chen et al. [15, 17] with labels. Although most of the typing rules remain the same, there are two major differences: (1) The source typing judgment no longer has a mode; (2) Our generalization has a label set in the typing rules to make sure the labels inside a function are unique. Furthermore, our generalization of changeable levels with labels does not affect inferring level polymorphic types. To simplify the presentation, we assume the source language presented

here is level monomorphic.

The typing rules for variables (SVar), integers (SInt), pairs (SPair), sums (SSum), primitive operations (SPrim), and projections (SFst) are standard. (We omit the symmetric rules for **inr** $v$ and **snd** $x$.) To type a function (SFun), we type the body specified by the function type $(\tau_1 \to \tau_2)^\delta$. The changeable types in the return type will translate to destinations when translating in the target language. To facilitate the translation, we need to fix the destination labels in the return type via $\tau_2 \downarrow_0 \mathcal{D}; \mathcal{L}$, where we assume destination labels all have prefix 0. We also assume that non-destination labels, e.g. labels for changeable input, have prefix 1. Note that these labels are only in a function scope, labels in different functions do not need to be unique. We omit the simpler rule for $[\![\tau_1]\!] = \mathbb{S}$.

Like in previous work, we allow subsumption only at let binding (SLet), e.g. from a bound expression $e_1$ of subtype $\mathbf{int}^{\mathbb{S}}$ to an assumption $x : \mathbf{int}^{\mathbb{C}_\rho}$. Note that when binding an expression into a variable with a changeable level, the label $\rho$ must be either unique or one of the labels from the destination. The subtype allows changeable labels with prefix 1 to be "promoted" as labels with prefix 0. This restriction makes sure the input data can flow to destinations, and the information flow type system tracks dependency correctly. We omit the simpler rule for $[\![\tau'']\!] = \mathbb{S}$. As in previous work, we restrict that we subsume only when the subtype and supertype are equal up to their outer levels. This simplifies the translation, with no loss of expressiveness: to handle "deep" subsumption, such as $(\mathbf{int}^{\mathbb{S}} \to \mathbf{int}^{\mathbb{S}})^{\mathbb{S}} <: (\mathbf{int}^{\mathbb{S}} \to \mathbf{int}^{\mathbb{C}_\rho})^{\mathbb{C}_{\rho'}}$, we can insert *coercions* into the source program before typing it with these rules. (This process could easily be automated.)

A function application (SApp) requires that the result of the function must be higher than the function's level: if a function is itself changeable $(\tau_1 \to \tau_2)^{\mathbb{C}_\rho}$, then it could be replaced by another function and thus the result of this application must be

changeable. Due to **let**-subsumption, checking this in (SFun) alone is not enough. Similarly, in rule (SCase) for typing a case expression, we ensure that the level of the result $\tau$ must also be higher than $\delta$: if the scrutinee changes, we may take the other branch, requiring a changeable result.

**Constraints and type inference.** Our rules and constraints fall within the HM(X) framework [44], permitting inference of principal types via constraint solving. Although our type system requires explicit labels for changeable levels, these labels can be inferred automatically. The user does not need to provide explicit labels when programming in the surface language. In all, we extend the type system with fine-grained dependency tracking without any burden on the programmer.

## 5. Target Language

**Abstract syntax.** The target language (Figure 9) is an imperative self-adjusting language with modifiables. In addition to integers,



| | | | |
|---|---|---|---|
| *Types* | $\underline{\tau}$ | ::= | $\mathbf{unit} \mid \mathbf{int} \mid \underline{\tau}\ \mathbf{mod} \mid \Box\ \underline{\tau} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \underline{\tau}_1 + \underline{\tau}_2 \mid \underline{\tau}_1 \xrightarrow[\mathcal{D}]{} \underline{\tau}_2$ |
| *Dest. Types* | $\mathcal{D}$ | ::= | $\left\{\underline{\tau}_1, \cdots, \underline{\tau}_n\right\}$ |
| *Labels* | $\mathcal{L}$ | ::= | $\{l_1, \cdots, l_n\}$ |
| *Variables* | $x$ | ::= | $y \mid l_i$ |
| *Typing Env.* | $\Gamma$ | ::= | $\cdot \mid \Gamma, x : \underline{\tau}$ |
| *Values* | $v$ | ::= | $n \mid x \mid \ell \mid (v_1, v_2) \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v \mid$ |

$$
\begin{array}{lrl}
 & & \mathbf{fun}^{\mathcal{L}}\ f(x) = e \\
\textit{Expressions} & e & ::= v \mid \oplus(x_1, x_2) \mid \mathbf{fst}\ x \mid \mathbf{snd}\ x \mid \\
 & & \mathbf{apply}^{\mathcal{L}}(x_1, x_2) \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\
 & & \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \mid \\
 & & \mathbf{mod}\ v \mid \mathbf{read}\ x\ \mathbf{as}\ y\ \mathbf{in}\ e \mid \mathbf{write}(x_1, x_2)
\end{array}
$$

**Figure 9.** Types and expressions in the target language

units, products, sums, the target type system makes a distinction between fresh modifiable types $\square\ \mathbf{int}$ (modifiables that are freshly allocated) and finalized modifiable types $\mathbf{int\ mod}$ (modifiables that are written after the allocation). The function type $\underline{\tau}_1 \underset{\mathcal{D}}{\rightarrow} \underline{\tau}_2$ contains an ordered set of destination types $\mathcal{D}$, indicating the type of the destinations of the function.

The variables consist of labels $l_i$ and ordinary variables $y$, which are drawn from different syntactically categories. The label variable $l_i$ is used as bindings for destinations.

The values of the language consist of integers, variables, locations $\ell$ (which appear only at runtime), pairs, tagged values, and functions. Each function $\mathbf{fun}^{\mathcal{L}}\ f(x) = e$ takes an ordered label set $\mathcal{L}$, which contains a set of destination modifiables $l_i$ that should be filled in before the function returns. An empty $\mathcal{L}$ indicates the function returns all stable values, and therefore takes no destination.

The expression $\mathbf{apply}^{\mathcal{L}}(x_1, x_2)$ applies a function while supplying a set of destination modifiables $\mathcal{L}$. The **mod** $v$ construct creates a new fresh modifiable $\square\ \underline{\tau}$ with an initial value $v$. The **read** expression binds the contents of a modifiable $x$ to a variable $y$ and evaluates the body of the **read**. The **write** constructor imperatively updates a modifiable $x_1$ with value $x_2$. The **write** operator can update both

modifiables in destination labels $\mathcal{L}$ and modifiables created by **mod**.

**Static semantics.** The typing rules in Figure 10 follow the structure of the expressions. Rules (TLoc), (TInt), (TVar), (TPair), (TSum), (TFst), (TPrim) are standard. Given an initial value $x$ of type $\underline{\tau}$, rule (TAlloc) creates a fresh modifiable of type $\square\ \underline{\tau}$. Note that the type system guarantees that this initial value $x$ will never be read. The reason for providing the an initial value is to determine the type of the modifiable, and making the type system sound. Rule (TWrite) writes a value $x_2$ of type $\underline{\tau}$ into a modifiable $x_1$, when $x_1$ is a fresh modifiable of type $\square\ \underline{\tau}$, and produces a new typing environment substituting the type of $x_1$ into an finalized modifiable type $\underline{\tau}$ **mod**. Note that Rule (TWrite) only allows writing into a fresh modifiable, thus guarantees that each modifiable can be written only once. Intuitively, **mod** and **write** separates the process of creating a value in a purely functional language into two steps: the creation of location and initialization. This separation is critical for writing programs in destination passing style. Rule (TRead) enforces that the programmer can only read a modifiable when it has been already written, that is the type of the modifiable should be $\underline{\tau}$ **mod**.

Rule (TLet) takes the produced new typing environment from the let binding, and uses it to check $e_2$. This allows the type system to keep track of the effects of **write** in the let binding. To ensure the correct usage of self-adjusting constructs, rule (TCase) enforces a conservative restriction that both the result type and the produced typing environment for each branch should be the same. This means that each branch should write to the same set of modifiables. If a

$\boxed{\Lambda; \Gamma \vdash e : \underline{\tau} \dashv \Gamma'}$ and target typing environment $\Gamma$, target expression $e$ has target type $\underline{\tau}$, and produces a typing environment $\Gamma'$

$$\frac{\Lambda(\ell) = \underline{\tau}}{\Lambda; \Gamma \vdash \ell : \underline{\tau} \dashv \Gamma} \text{ (TLoc)} \qquad \frac{}{\Lambda; \Gamma \vdash n : \mathbf{int} \dashv \Gamma} \text{ (TInt)}$$

$$\frac{\Gamma(x) = \underline{\tau}}{\Lambda; \Gamma \vdash x : \underline{\tau} \dashv \Gamma} \text{ (TVar)} \qquad \frac{\Lambda; \Gamma \vdash v : \underline{\tau} \dashv \Gamma}{\Lambda; \Gamma \vdash \mathbf{mod}\ v : \Box\ \underline{\tau} \dashv \Gamma} \text{ (TAlloc)}$$

$$\frac{\Lambda; \Gamma \vdash x_2 : \underline{\tau} \dashv \Gamma}{\Lambda; \Gamma, x_1 : \Box\ \underline{\tau} \vdash \mathbf{write}(x_1, x_2) : \mathbf{unit} \dashv \Gamma, x_1 : \underline{\tau}\ \mathbf{mod}} \text{ (TWrite)}$$

$$\frac{\Lambda; \Gamma \vdash x_1 : \underline{\tau}_1\ \mathbf{mod} \dashv \Gamma \qquad \Lambda; \Gamma, x : \underline{\tau}_1 \vdash e_2 : \underline{\tau}_2 \dashv \Gamma'}{\Lambda; \Gamma \vdash \mathbf{read}\ x_1\ \mathbf{as}\ x\ \mathbf{in}\ e_2 : \mathbf{unit} \dashv \Gamma'} \text{ (TRead)}$$

$$\frac{\Lambda; \Gamma \vdash v_1 : \underline{\tau}_1 \dashv \Gamma \qquad \Lambda; \Gamma \vdash v_2 : \underline{\tau}_2 \dashv \Gamma}{\Lambda; \Gamma \vdash (v_1, v_2) : \underline{\tau}_1 \times \underline{\tau}_2 \dashv \Gamma} \text{ (TPair)}$$

$$\frac{\begin{array}{c}\mathcal{L} = \{l_1, \cdots, l_n\} \quad \mathcal{D} = \{\underline{\tau}'_1, \cdots, \underline{\tau}'_n\} \quad \underline{\tau}_1 \neq \Box\ \underline{\tau}' \\ \Gamma_d(l_i) = \cdot, l_1 : \Box\ \underline{\tau}'_1, \cdots, l_n : \Box\ \underline{\tau}'_n \\ For\ i = 1, \cdots, n \quad \Gamma'(l_i) = \underline{\tau}'_i\ \mathbf{mod} \\ \Lambda; \Gamma, x : \underline{\tau}_1, f : (\underline{\tau}_1 \underset{\mathcal{D}}{\rightarrow} \underline{\tau}_2), \Gamma_d \vdash e : \underline{\tau}_2 \dashv \Gamma'\end{array}}{\Lambda; \Gamma \vdash \mathbf{fun}^{\mathcal{L}}\ f(x) = e : (\underline{\tau}_1 \underset{\mathcal{D}}{\rightarrow} \underline{\tau}_2) \dashv \Gamma} \text{ (TFun)}$$

$$\frac{\Lambda; \Gamma \vdash v : \underline{\tau}_1 \dashv \Gamma}{\Lambda; \Gamma \vdash \mathbf{inl}\ v : \underline{\tau}_1 + \underline{\tau}_2 \dashv \Gamma} \text{ (TSum)} \qquad \frac{\Lambda; \Gamma \vdash x : \underline{\tau}_1 \times \underline{\tau}_2 \dashv \Gamma}{\Lambda; \Gamma \vdash \mathbf{fst}\ x : \underline{\tau}_1 \dashv \Gamma} \text{ (TFst)}$$

$$\Lambda; \Gamma \vdash x_1 : \mathbf{int} \dashv \Gamma$$

$$\frac{\Lambda;\Gamma \vdash x_2 : \mathbf{int} \dashv \Gamma \qquad \vdash \oplus : \mathbf{int} \times \mathbf{int} \to \mathbf{int}}{\Lambda;\Gamma \vdash \oplus(x_1, x_2) : \mathbf{int} \dashv \Gamma} \text{ (TPrim)}$$

$$\frac{\Lambda;\Gamma \vdash e_1 : \underline{\tau} \dashv \Gamma' \qquad \Lambda;\Gamma', x : \underline{\tau} \vdash e_2 : \underline{\tau}' \dashv \Gamma''}{\Lambda;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \underline{\tau}' \dashv \Gamma''} \text{ (TLet)}$$

$$\frac{\begin{array}{c}\mathcal{L} = \{l_1, \cdots, l_n\} \qquad \mathcal{D} = \{\underline{\tau}'_1, \cdots, \underline{\tau}'_n\} \\ For\ i = 1, \cdots, n \quad \Gamma(l_i) = \square\, \underline{\tau}'_i \quad \Gamma'(l_i) = \underline{\tau}'_i\ \mathbf{mod} \\ \Lambda;\Gamma \vdash x_1 : (\underline{\tau}_1 \underset{\mathcal{D}}{\to} \underline{\tau}_2) \dashv \Gamma \qquad \Lambda;\Gamma \vdash x_2 : \underline{\tau}_1 \dashv \Gamma\end{array}}{\Lambda;\Gamma \vdash \mathbf{apply}^{\mathcal{L}}(x_1, x_2) : \underline{\tau}_2 \dashv \Gamma'} \text{ (TApp)}$$

$$\frac{\Lambda;\Gamma \vdash x : \underline{\tau}_1 + \underline{\tau}_2 \dashv \Gamma \qquad \begin{array}{c}\Lambda;\Gamma, x_1 : \underline{\tau}_1 \vdash e_1 : \underline{\tau} \dashv \Gamma' \\ \Lambda;\Gamma, x_2 : \underline{\tau}_2 \vdash e_2 : \underline{\tau} \dashv \Gamma'\end{array}}{\Lambda;\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \underline{\tau} \dashv \Gamma'} \text{ (TCase)}$$

**Figure 10.** Typing rules of the target language

modifiable $x$ is finalized in one branch, the other branch should also finalize the same modifiable.

Rule (TFun) defines the typing requirement for a function: (1) the destination types $\mathcal{D}$ are fresh modifiables, and the argument type should not contains fresh modifiable. Intuitively, the function arguments are partitions into two parts: destinations and ordinary arguments; (2) the body of the function $e$ has to finalize all the destination modifiables presented in $\mathcal{L}$. This requirement can be achieved by either explicitly **write**'ing into modifiables in $\mathcal{L}$, or by passing these modifiables into another function that takes the responsibility to write an actual value to them. Although all the modifiables in $\mathcal{L}$ should be finalized, other modifiables created

inside the function body may be fresh, as long as there is no read of those modifiables in the function body.

Rule (TApp) applies a function with fresh modifiables $\mathcal{L}$. The type of these modifiables should be the same as the destination types $\mathcal{D}$ as presented in the function type. The typing rule produces a new

$$
\begin{aligned}
\|\mathbf{int}^{\mathbb{S}}\| &= \mathbf{int} \\
\|(\tau_1 \to \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| \underset{\|\mathcal{D}\|}{\to} \|\tau_2\| \quad (\tau_2 \downarrow_0 \mathcal{D}; \mathcal{L}) \\
\|(\tau_1 \times \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| \times \|\tau_2\| \\
\|(\tau_1 + \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| + \|\tau_2\| \\
\|\tau\| &= \|\left|\tau\right|^{\mathbb{S}}\| \, \mathbf{mod} \quad (\llbracket\tau\rrbracket = \mathbb{C}_\rho)
\end{aligned}
$$

$$
\begin{aligned}
\|\cdot\| &= \cdot & \|\tau\|_\phi &= \|[\phi]\tau\| \\
\|\Gamma, x : \tau\| &= \|\Gamma\|, x : \|\tau\| & \|\Gamma\|_\phi &= \|[\phi]\Gamma\|
\end{aligned}
$$

**Figure 11.** Translations $\|\tau\|$ of types and typing environments

typing environment that guarantees that all the supplied destination modifiables are finalized after the function application.

**Dynamic semantics.** The dynamic semantics of our target language matches that of Acar et al [4] after two syntactical changes: $\mathbf{fun}^{\mathcal{L}}\ f(x) = e$ is represented as $\mathbf{fun}\ f(x) = \lambda\mathcal{L}.e$, and $\mathbf{apply}^{\mathcal{L}}(x_1, x_2)$ is represented as $(x_1\ x_2)\ \mathcal{L}$.

# 6. Translation

This section gives a high-level overview of the translation from the source language to the target self-adjusting language. To ensure type safety, we translate types and expressions together using a type-directed translation. Since the source and the target languages have different type systems, an expression $e : \tau$ cannot be translated to a target expression $e'$ of type $\tau$, the type also has to be translated, producing some $e' : \underline{\tau}'$ where $\underline{\tau}'$ is a target type that *corresponds*

*to* $\tau$. We therefore developed the translation of expressions and types together, along with the proof that the desired property holds. To understand how to translate expressions, it is helpful to first understand how we translate types.

### 6.1 Translating types.

Figure 11 defines the translation of types from the source language's types into the target types. We also use it to translate the types in the typing environment $\Gamma$. We define $\|\tau\|$ as the translation of types from the source language into the target types. We also use it for translating the types in the typing environment $\Gamma$. For integers, sums, and products with stable levels, we simply erase the level notation $\mathbb{S}$, and apply the function recursively into the type structure. For arrow types, we need to derive the destination types. In the source typing, we fix the destination type labels by $\tau_2 \downarrow_0 \mathcal{D}; \mathcal{L}$, where $\mathcal{D}$ stores the source type for the destinations. Therefore, the destination types for the target arrow function will be $\|\mathcal{D}\|$.

For source types with changeable levels, the target type will be modifiables. Since the source language is purely functional, the final result will always be a finalized modifiable $\tau$ **mod**. Here, we define a *stabilization* function $\left|\tau\right|^{\mathbb{S}}$ for changeable source types, which changes the outer level of $\tau$ from changeable into stable. Formally, we define the function as,

$$\left|\tau\right|^{\mathbb{S}} = \tau', \text{where } [\![\tau]\!] = \mathbb{C}_\rho, [\![\tau']\!] = \mathbb{S} \text{ and } \tau \doteq \tau'$$

Then, the target type for a changeable level source type $\tau$ will be $\|\left|\tau\right|^{\mathbb{S}}$ **mod**$\|$.

### 6.2 Translating Expressions

We define the translation of expressions as a set of type-directed rules. Given (1) a derivation of $C; \mathcal{P}; \Gamma \vdash e : \tau$ in the constraint-based typing system and (2) a satisfying assignment $\phi$ for $C$, it is always possible to produce a correctly-typed target expression $e_t$ (see Theorem 6.1 below). The environment $\Gamma$ in the translation rules is a source-typing environment and must have no free level variables. Given an environment $\Gamma$ from the constraint typing, we apply the satisfying assignment $\phi$ to eliminate its free level variables before

$\boxed{\Gamma \vdash e : \tau \hookrightarrow e'}$ Under closed source typing environment $\Gamma$, source expression $e$ is translated at type $\tau$ to target expression $e'$

$$\frac{}{\Gamma \vdash n : \mathbf{int}^{\mathbb{S}} \hookrightarrow n}\ (\text{Int}) \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \hookrightarrow x}\ (\text{Var})$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \hookrightarrow v_1' \qquad \Gamma \vdash v_2 : \tau_2 \hookrightarrow v_2'}{\Gamma \vdash (v_1, v_2) : (\tau_1 \times \tau_2)^{\mathbb{S}} \hookrightarrow (v_1', v_2')}\ (\text{Pair})$$

$$\frac{\Gamma, x : \tau_1, f : (\tau_1 \to \tau_2)^{\mathbb{S}} \vdash e : \tau_2 \rightsquigarrow e' \qquad \tau_2 \downarrow_0 \mathcal{D}; \mathcal{L}}{\Gamma \vdash \mathbf{fun}\ f(x) = e : (\tau_1 \to \tau_2)^{\mathbb{S}} \hookrightarrow \mathbf{fun}^{\mathcal{L}}\ f(x) = e'}\ (\text{Fun})$$

$$\frac{\Gamma \vdash v : \tau_1 \hookrightarrow v'}{\Gamma \vdash \mathbf{inl}\ v : (\tau_1 + \tau_2)^{\mathbb{S}} \hookrightarrow \mathbf{inl}\ v'}\ (\text{Sum}) \qquad \frac{\Gamma \vdash x : (\tau_1 \times \tau_2)^{\mathbb{S}} \hookrightarrow \underline{x}}{\Gamma \vdash \mathbf{fst}\ x : \tau_1 \hookrightarrow \mathbf{fst}\ \underline{x}}\ (\text{Fst})$$

$$\frac{\Gamma \vdash x_1 : \mathbf{int}^{\mathbb{S}} \hookrightarrow \underline{x_1} \qquad \Gamma \vdash x_2 : \mathbf{int}^{\mathbb{S}} \hookrightarrow \underline{x_2}}{\Gamma \vdash \oplus(x_1, x_2) : \mathbf{int}^{\delta} \hookrightarrow \oplus(\underline{x_1}, \underline{x_2})}\ (\text{Prim})$$

$$\Gamma \vdash x_1 : (\tau_1 \to \tau_2)^{\mathbb{S}} \hookrightarrow x_1$$

$$\frac{\Gamma \vdash x_2 : \tau_1 \hookrightarrow \underline{x_2} \qquad \tau_2 \downarrow_0 \overline{\mathcal{D}}; \mathcal{L}}{\Gamma \vdash \quad \mathbf{apply}(x_1, x_2) : \tau_2 \\ \hookrightarrow \mathbf{let}\ \{l_i = \mathbf{mod}\ (\tau_i'|_v)\}_{l_i \in \mathcal{L}}^{\tau_i' \in \mathcal{D}}\ \mathbf{in}\ \mathbf{apply}^{\mathcal{L}}(\underline{x_1}, \underline{x_2})} \text{(App)}$$

$$\frac{\Gamma \vdash x : (\tau_1 + \tau_2)^{\mathbb{S}} \hookrightarrow \underline{x} \qquad \begin{array}{r}\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \hookrightarrow e_1' \\ \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \hookrightarrow e_2'\end{array}}{\begin{array}{l}\Gamma \vdash \quad \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \tau \\ \hookrightarrow \mathbf{case}\ \underline{x}\ \mathbf{of}\ \{x_1 \Rightarrow e_1'\ ,\ x_2 \Rightarrow e_2'\}\end{array}} \text{(Case)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \hookrightarrow e_1' \qquad \Gamma, x : \tau' \vdash e_2 : \tau \hookrightarrow e_2'}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau \hookrightarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2'} \text{(Let)}$$

$$\frac{\Gamma \vdash e : \tau \hookrightarrow e' \qquad [\![\tau]\!] = \mathbb{C}_{1_\rho} \qquad \tau|_v = v}{\Gamma \vdash e : \tau \hookrightarrow \mathbf{let}\ l_{1_\rho} = \mathbf{mod}\ v\ \mathbf{in}\ e'} \text{(Mod)}$$

$$\frac{\Gamma \vdash e : \tau' \hookrightarrow e' \qquad [\![\tau]\!] = \mathbb{C}_{1_\rho} \qquad \left|\tau\right|^{\mathbb{S}} = \tau' \qquad \tau|_v = v}{\Gamma \vdash e : \tau \hookrightarrow \mathbf{let}\ l_{1_\rho} = \mathbf{mod}\ v\ \mathbf{in}\ e'} \text{(Lift)}$$

$$\frac{\Gamma \vdash e : \tau \hookrightarrow e' \qquad [\![\tau]\!] = \mathbb{C}_\rho}{\Gamma \vdash e : \tau \hookrightarrow \mathbf{let}\ () = \mathbf{write}(l_\rho, e')\ \mathbf{in}\ l_\rho} \text{(Write)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau' \vdash e') & [\![\tau']\!] = \mathbb{C}_\rho \\ \Gamma, x' : \left|\tau'\right|^{\mathbb{S}} \vdash e' : \tau \hookrightarrow e'' & \Gamma \vdash x : \tau' \hookrightarrow \underline{x}\end{array}}{\Gamma \vdash e : \tau \hookrightarrow \mathbf{read}\ \underline{x}\ \mathbf{as}\ x'\ \mathbf{in}\ e''} \text{(Read)}$$

**Figure 12.** Translation for destination passing style

using it in the translation $[\phi]\Gamma$. With the environment closed, we need not refer to $C$.

Our rules are nondeterministic, avoiding the need to "decorate" them with context-sensitive details.

**Direct rules.** The rules (Int), (Var), (Pair), (Sum), (Fst) and (Prim) follows the structure of the expression, and directly translate the expressions.

**Changeable rules.** The rules (Lift), (Mod), and (Write) translate expressions with outer level changeable $\mathbb{C}_\rho$. Given a translation of $e$ to some pure expression $e'$, rule (Write) translates $e$ into an imperative **write** expression that writes $e'$ into modifiable $l_\rho$.

For expressions with non-destination changeable levels, that is the label $\rho$ has a 1 as the prefix, we need to create a modifiable first. Rules (Lift) and (Mod) achieves this goal. (Mod) is the simpler of the two: if $e$ translates to $e'$ at type $\tau$, then $e$ translates to the **mod** expression at type $\tau$. To get an initial value for the modifiable, we define a function $\tau|_v$ that takes a source type $\tau$ and returns any

$$\boxed{\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau \vdash e')}$$

Under source typing $\Gamma$, renaming the "head" $x$ in $e$ to $x' : \tau$ yields expression $e'$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x \rightsquigarrow (x \gg x' : \tau \vdash x')}\ (\text{LVar}) \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \mathbf{fst}\ x \rightsquigarrow (x \gg x' : \tau \vdash \mathbf{fst}\ x')}\ (\text{LFst})$$

$$\frac{\Gamma \vdash x_1 : \tau}{\Gamma \vdash \oplus(x_1, x_2) \rightsquigarrow (x_1 \gg x_1' : \tau \vdash \oplus(x_1', x_2))}\ (\text{LPrimop1})$$

$$\frac{\Gamma \vdash x_1 : \tau}{\Gamma \vdash \mathbf{apply}(x_1, x_2) \rightsquigarrow (x_1 \gg x' : \tau \vdash \mathbf{apply}(x', x_2))}\ (\text{LApply})$$

$$\frac{\Gamma \vdash x : \tau}{\begin{array}{l}\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \\ \quad \rightsquigarrow (x \gg x' : \tau \vdash \mathbf{case}\ x'\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\})\end{array}}\ (\text{LCase})$$

**Figure 13.** Renaming the variable to be read

value $v$ of that type. Note that the initial value is only a placeholder, and will never be read, so the choice of the value is not important. In (Lift), the expression is translated not at the given type $\tau$ but at its *stabilized* $\left|\tau\right|^{\mathbb{S}}$, capturing the "shallow subsumption" in the constraint typing rules (SLet): a bound expression of type $\tau_0^{\mathbb{S}}$ can be translated at type $\tau_0^{\mathbb{S}}$ to $e'$, and then "promoted" to type $\tau_0^{\mathbb{C}_\rho}$ by placing it inside a modifiable $l_\rho$.

**Reading from changeable data.** To use an expression of changeable type in a context where a stable value is needed—such as passing some $x : \mathbf{int}^{\mathbb{C}}$ to a function expecting $\mathbf{int}^{\mathbb{S}}$—the (Read) rule

generates a target expression that reads the value out of $x : \mathbf{int}^{\mathbb{C}}$ into a variable $x' : \mathbf{int}^{\mathbb{S}}$. The variable-renaming judgment $\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau \vdash e')$ takes the expression $e$, finds a variable $x$ about to be used, and yields an expression $e'$ with that occurrence replaced by $x'$. For example, $\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \ldots \rightsquigarrow (x \gg x' : \tau \vdash \mathbf{case}\ x'\ \mathbf{of}\ \ldots)$. This judgment is derivable only for variable, **apply**, **case**, **fst**, and $\oplus$. For $\oplus(x_1, x_2)$, we need to read both variables; we omit the symmetric rules for reading the second variable. The rules are given in Figure 13.

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow e'}$ Under closed source typing environment $\Gamma$, function body $e$ is translated at type $\tau$ to target expression $e'$ with destination returns.

$$\frac{\Gamma \vdash v_1 : \tau_1 \rightsquigarrow v_1' \qquad \Gamma \vdash v_2 : \tau_2 \rightsquigarrow v_2'}{\Gamma \vdash (v_1, v_2) : (\tau_1 \times \tau_2)^{\mathbb{S}} \rightsquigarrow (v_1', v_2')}\ (\text{RPair})$$

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \rightsquigarrow e_1'}{\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \tau \rightsquigarrow e_1'}\ (\text{RCase})$$

$$\frac{[\![\tau]\!] = \mathbb{C}_\rho}{\Gamma \vdash e : \tau \rightsquigarrow l_\rho}\ (\text{RMod}) \qquad \frac{\Gamma \vdash e : \tau \hookrightarrow e'}{\Gamma \vdash e : \tau \rightsquigarrow e'}\ (\text{RTrans})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau' \hookrightarrow e_1' \qquad \Gamma \vdash e_2 : \tau' \hookrightarrow e_2' \\ \Gamma, x : \tau' \vdash e_2 : \tau \rightsquigarrow ret \qquad e_2 \neq \mathbf{let}\ x' = e_1'\ \mathbf{in}\ e_2'\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ \_ = e_2'\ \mathbf{in}\ ret\end{array}}\ (\text{RLet})$$

**Figure 14.** Deriving destination return

**Function and application rules.** Since the self-adjusting primi-

tives are imperative, an expression with outer changeable levels will be translated into a target expression that returns unit. To recover the type of the function return for the target language, we need to wrap the destinations, so that the function returns the correct type. Figure 14 shows the rules for translating the function body and wrapping the destinations. For a tuple expression (RPair), the translation returns the destination for each component. For a case expression (RCase), it is enough to return destinations from one of the branches since the source typing rule (SCase) guarantees that both branches will write to the same destinations. When the expression has a outer changeable level $\mathbb{C}_\rho$, rule (RMod) returns its modifiable variable $l_\rho$. For let bindings, rule (RLet) translates all the bindings in the usual way and derive destinations for the expressions in the tail position. For all other expressions, the translation simply switches to the ordinary translation rules in Figure 12. For example, expression $(1, x) : \left(\mathbf{int}^{\mathbb{S}} \times \mathbf{int}^{\mathbb{C}_{01}}\right)^{\mathbb{S}}$ will be translated to $(1, l_{01})$ by applying rules (RProd) (RTrans) (Int) (RMod).

When applying functions $\mathbf{apply}(x_1, x_2)$, rule (App) first creates a set of fresh modifiable destinations using **mod** , then supply both the destination set $\mathcal{L}$ and argument $x_2$ to function $x_1$. Note that although the destination names $l_i$ may overlap with the current function destination names, these variables are only locally scoped, the application of the function will return a new value, which contains the supplied destinations $\mathcal{L}$, but they are never mentioned outside of the function application.

The translation rules 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, the translation rules can generate

expressions like **read** $x$ **as** $x'$ **in write**$(l_\rho, x')$, which is equivalent to $x$. We can easily apply rewriting rules to get rid of these redundant operations after the translation.

**Translation correctness.** Given a constraint-based source typing derivation and assignment $\phi$ for some term $e$, there are translations from $e$ to (1) a target expression $e_t$ and (2) a destination return expression $e_r$, with appropriate target types:

**Theorem 6.1.** *If* $C; \mathcal{P}; \Gamma \vdash e : \tau$*, and* $\phi$ *is a satisfying assignment for* $C$*, then*

*(1) there exists* $e_t$ *and* $\Gamma'$ *such that* $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow e_t$*, and* $\cdot; \|\Gamma\|_\phi \vdash e_t : \|\tau\|_\phi \dashv \Gamma'$*,*

*(2) there exists* $e_r$ *and* $\Gamma'$ *such that* $[\phi]\Gamma \vdash e : [\phi]\tau \rightsquigarrow e_r$*, and* $\cdot; \|\Gamma\|_\phi \vdash e_r : \|\tau\|_\phi \dashv \Gamma'$*.*

The proof is by induction on the height of the given derivation of $C; \mathcal{P}; \Gamma \vdash e : \tau$. The proof relies on a substitution lemma for (SLet) case. We present the full proof in the appendix [14].

# 7. Probabilistic Chunking

Precise dependency tracking saves space by eliminating redundant dependencies. But even then, the dependency metadata required can still be large, preventing scaling to large datasets. In this section, we show how to reduce the size of dependency metadata further by controlling the granularity of dependency tracking, crucially in a way that does not affect performance disproportionately.

The basic idea is to track dependencies at the granularity of a *block* of items. This idea is straightforward to implement: simply place blocks of data into modifiables (e.g., store an array of integers

as a block instead of just one number). As such, if any data in a block changes, the computation that depends on that block must be rerun. While this saves space, the key question for performance is therefore: *how to chunk data into blocks without disproportionately affecting the update time*?

For fast updates, our chunking strategy must ensure that a small change to the input remains small and local, without affecting many other blocks. The simple strategy of chunking into fixed-size blocks does not work. To see why, consider the example in Figure 15 (left half), where a list containing numbers 1 through 16, missing 2, is chunked into equal-sized blocks of 4. The trouble begins when we insert 2 into the list between 1 and 3. With fixed-size chunking, all the blocks will change because the insertion shifts the position of all block boundaries by one. As a result, when tracking dependencies

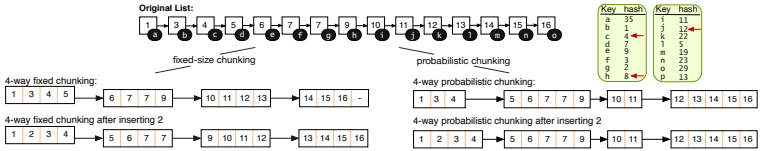


**Figure 15. Fixed-size chunking versus probabilistic chunking:** with block size $B = 4$. Next to each data cell in the original list (top) is a unique identifier (location). The hash values of these identifiers (used in probabilistic chunking) are shown in the table, with values divisible by $B = 4$ marked with an arrow.

at the level of blocks, we cannot reuse any prior computations and will essentially recompute the result anew.

We propose a *probabilistic chunking scheme* (PCS), which decouples locations of block boundaries from the data contents and absolute positions in the list while allowing users to control the block size probabilistically. Using randomization, we are able to prevent small (even adversarial) changes from spreading to the rest of the computation. Similar probabilistic chunking schemes have been proposed in other work but differently, they aim at discovering similarities across pieces of data (see, e.g., [43, 53] and the references therein) rather than creating independence between the data and how it is chunked as we do here.

PCS takes a target block size $B$ and determines block boundaries by hashing the location or the unique identifier of each data item and declaring it a block boundary if the hash is divisible by $B$. Figure 15 (right) illustrates how this works. Consider, again, a list holding numbers from 1 to 16, missing 2, with their location identifiers (a, b, ...) shown next to them. PCS chunks this into blocks of *expected* size $B = 4$ by applying a random hash function to each item. For this example, the hash values are given in a table on the right of the

figure; hash values divisible by 4 are marked with an arrow. PCS declares block boundaries where the hash value is 0 mod $B = 4$, thereby selecting 1 in 4 elements to be on the boundary. This means finishing the blocks at 4, 9, and 11, as shown.

To understand what happens when the input changes, consider inserting 2 (with location identifier $\mathsf{p}$) between 1 and 3. Because the hash value of $\mathsf{p}$ is 13, it is not on the boundary. This is the common case as there is only a $1/B$-th probability that a random hash value is divisible by $B$. As a result, only the block $[\![1, 3, 4]\!]$, where 2 is added, is affected. If, however, 2 happened to be a boundary element, we would only have two new blocks (inserting 2 splits an existing block into two). Either way, the rest of the list remains unaffected, enabling computation that depended on other blocks to be reused. Deletion is symmetric.

To conclude, by chunking a dataset into size-$B$ blocks, probabilistic chunking reduces the dependency metadata by a factor of $B$ in expectation. Furthermore, by keeping changes small and local, probabilistic chunking ensures maximum reuse of existing computations. Change propagation works analogously to the non-block version, except that if a block changes, work on the whole block must be redone, thus often increasing the update time by $B$ folds.

## 8. Evaluation

We performed extensive empirical evaluation on a range of benchmarks, including standard benchmarks from prior work, as well as new, more involved benchmarks on social network graphs. We report selected results in this section. All our experiments were performed on a 2GHz Intel Xeon with 1 TB memory running Linux.

Our implementation is single-threaded and therefore uses only one core. The code was compiled with MLton version 20100608 with flags to measure maximum live memory usage.

### 8.1 Benchmarks and Measurements

We have completed an implementation of the target language as a Standard ML (SML) library. The implementation follows the formalism except for the following: (1) it treats both fresh and finalized modifiable types as a single $\underline{\tau}$ **mod** type; (2) for function $\mathbf{fun}^{\mathcal{L}}\ f(x) = e$, it includes destination labels as part of the function argument, so the function is represented as `fun f(x) = fn` $\mathcal{L}$ `=> e`. Accordingly, the arrow type $(\underline{\tau}_1 \xrightarrow[\mathcal{D}]{} \underline{\tau}_2)$ is represented as $\underline{\tau}_1 \rightarrow \underline{\tau}' \rightarrow \underline{\tau}_2$, where $\underline{\tau}' = \underline{\tau}'_1$ **mod** $\times \cdots \times \underline{\tau}'_n$ **mod** and $\mathcal{D} = \{\underline{\tau}'_1, \cdots, \underline{\tau}'_n\}$.

Since our approach provides for an expressive language (any pure SML program can be made self-adjusting), we can implement a variety of domain-specific languages and algorithms. For the evaluation, we implemented the following:

- a *blocked list* abstract data type that uses our probabilistic chunking algorithm (Section 7),
- a sparse matrix abstract data type,
- as implementation of the MapReduce framework [20] that uses the blocked lists,
- several list operations and the merge sort algorithm,
- more sophisticated algorithms on graphs, which use the sparse-matrix data type to represent graphs, where a row of the matrix represents a vertex in the compressed sparse row format, including only the nonzero entries.

In our graph benchmarks, we control the space-time trade-off by

treating a block of 100 nonzero elements as a single changeable unit. For the graphs used, this block size is quite natural, as it corresponds roughly to the average degree of a node (the degree ranges between 20 and 200 depending on the graph).

For each benchmark, we implemented a *batch* version—an optimized implementation that operates on unchanging inputs—and a *self-adjusting* version by using techniques proposed in this paper. We compare these versions by considering a mix of synthetic and real-world data, and by considering different forms of changes ranging from small *unit changes* (e.g., insertion/deletion of one item) to *aggregate changes* consisting of many unit changes (e.g., insertion/deletion of 1000 items). We describe specific datasets employed and changes performed in the description of each experiment.

### 8.2 Block Lists and Sorting

Using our block list representation, we implemented batch and self-adjusting versions of several standard list primitives such as `map`, `partition`, and `reduce` as well as the merge sort algorithm `msort`. In the evaluation, all benchmarks operate on integers: `map` applies $f(i) = i \div 2$ to each element; `partition` partitions its input based on the parity of each element; `reduce` computes the sum of the list modular 100; and `msort` implements merge sort.

Table 1 reports our measurements at fixed input sizes $10^7$. For each benchmark, we consider three different versions: (1) a batch version (written with the `-batch` suffix); (2) a self-adjusting version *without* the chunking scheme (the first row below batch); (3) the self-adjusting version with different block sizes ($B = 3, 10, \ldots$). We report the block size used ($B$); the time to run from scratch (denoted by "Run") in seconds; the average time for a change propagation after one insertion/deletion from the input list (denoted by "Prop.") in milliseconds. Note that for batch versions, the propagation time (i.e., a rerun) is the same as a complete from-scratch run. We calculate the *speedup* as the ratio of the time for a run from-scratch to average propagation, i.e., the performance improvement obtained by the self-adjusting version with respect to the batch version of the same benchmark. "Memory" column shows the maximum memory footprint. The experiments show that as the block size increases, both the self-adjusting (from-scratch) run time and memory decreases, confirming that larger blocks generate fewer dependencies. As block size increases, time for change propagation does also, but in proportion with the block size. (From $B = 3$ to $B = 10$, propagation time decreases, because the benefit for processing more elements per block exceeds the overhead for accessing the blocks).

| Benchmark | $B$ | Run (s) | Prop. (ms) | Speedup | Memory |
|---|---|---|---|---|---|
| `map-batch` | 1 | 0.497 | 497 | 1 | 344M |
| | 1 | 11.21 | 0.001 | 497000 | 7G |
| | 3 | 16.86 | 0.012 | 41416 | 10G |
| | 10 | 5.726 | 0.009 | 55222 | 3G |

| | | | | | |
|---|---|---|---|---|---|
| map | 100 | 1.796 | 0.048 | 10354 | 1479M |
| | 1000 | 1.370 | 0.635 | 783 | 1192M |
| | 10000 | 1.347 | 9.498 | 52 | 1168M |
| partition-batch | 1 | 0.557 | 557 | 1 | 344M |
| partition | 1 | 10.42 | 0.015 | 37133 | 8G |
| | 3 | 20.06 | 0.033 | 16878 | 14G |
| | 10 | 6.736 | 0.028 | 19892 | 3G |
| | 100 | 1.920 | 0.049 | 11367 | 1508M |
| | 1000 | 1.420 | 0.823 | 677 | 1159M |
| | 10000 | 1.417 | 11.71 | 47 | 1124M |
| reduce-batch | 1 | 0.330 | 330 | 1 | 344M |
| reduce | 1 | 9.529 | 0.064 | 5156 | 5G |
| | 3 | 13.39 | 0.129 | 2558 | 6G |
| | 10 | 4.230 | 0.085 | 3882 | 1317M |
| | 100 | 0.990 | 0.083 | 3976 | 592M |
| | 1000 | 0.627 | 0.075 | 4400 | 420M |
| | 10000 | 0.593 | 0.244 | 1352 | 327M |
| msort-batch | 1 | 12.82 | 12820 | 1 | 1.3G |
| msort | 1 | 676.4 | 0.956 | 13410 | 121G |
| | 3 | 725.0 | 1.479 | 8668 | 157G |
| | 10 | 204.4 | 1.012 | 12668 | 44G |
| | 100 | 52.00 | 3.033 | 4227 | 10G |
| | 1000 | 43.80 | 22.36 | 573 | 9G |
| | 10000 | 35.35 | 119.7 | 107 | 8G |

**Table 1. Blocked lists and sorting:** time and space with varying block sizes on fixed input sizes of $10^7$.

In terms of memory usage, the version *without* block lists ($B = 1$) requires 15–100x more memory than the batch version. Block lists

significantly reduce the memory footprint. For example, with block size B = 100, the benchmarks require at most 7x more memory than the batch version, while still providing 4000–10000x speedup. In our experiments, we confirm that probabilistic chunking (Section 7) is essential for performance—when using fixed-size chunking, merge sort does not yield noticeable improvements.

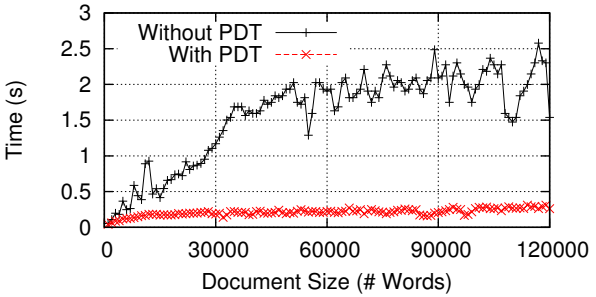


**Figure 16.** Run time (seconds) of incremental word count.

| Benchmark | Source | Input Size | Prop. (s) | Speedup | Memory |
|---|---|---|---|---|---|
| PR-Batch | Orkut | $3 \times 10^6$ vertices | 7 | 1 | 3G |
| PageRank | | $1 \times 10^8$ edges | 0.021 | 333 | 36G |
| PR-Batch | LiveJournal-1 | $4 \times 10^6$ vertices | 18 | 1 | 5G |
| PageRank | | $3 \times 10^7$ edges | 0.023 | 783 | 61G |
| PR-Batch | Twitter-1 | $3 \times 10^7$ vertices | 137 | 1 | 50G |
| PageRank | | $7 \times 10^8$ edges | 0.254 | 539 | 495G |
| Conn-Batch | LiveJournal-2 | $1 \times 10^6$ vertices | 105 | 1 | 4G |
| Connectivity | | $8 \times 10^6$ edges | 0.531 | 198 | 140G |

| | | | | | |
|---|---|---|---|---|---|
| SC-Batch | Twitter-2 | $1 \times 10^5$ vertices | 8 | 1 | 2G |
| Social Circle | | $2 \times 10^6$ edges | 0.079 | 101 | 34G |

**Table 2. Incremental sparse graphs:** time and space.

### 8.3 Word Count

A standard microbenchmark for big-data applications is word count, which maintains the frequency of each word in a document. Using our MapReduce library (run with block size $1,000$), we implemented a batch version and a self-adjusting version of this benchmark, which can update the frequencies as the document changes over time.

We use this benchmark to illustrate, in isolation, the impact of our precise dependency tracking mechanism. To this end, we implemented two versions of word count: one using prior art [16] (which contains redundant dependencies) and the other using the techniques presented in this paper. We use a publicly available Wikipedia dataset[1] and simulate evolution of the document by dividing it into blocks and incrementally adding these blocks to the existing text; the whole text has about $120,000$ words.

Figure 16 shows the time to insert $1,000$ words at a time into the existing corpus, where the horizontal axis shows the corpus size at the time of insertion. Note that the two curves differ only in whether the new precise dependency tracking is used. Overall, both incremental versions appear to have a logarithmic trend because in this case, both the shuffle and reduce phases require $\Theta(\log n)$ time for a single-entry update, where $n$ is the number of input words. Importantly, with precise dependency tracking (PDT), the update time is around 6x faster than without. In terms of memory consumption, PDT is 2.4x more space efficient. Compared to a batch run, PDT is $\sim$ 100x faster for a corpus of size 100K words or larger (since we change 1000 words/update, this is essentially optimal).

### 8.4 PageRank: Two Implementations

Another important big data benchmark is the PageRank algorithm, which computes the page rank of a vertex (site) in a graph (network). This algorithm can be implemented in several ways. For example, a domain specific language such as MapReduce can be (and often is) used even though it is known that for this algorithm, the shuffle step required by MapReduce is not needed. We implemented the

[1] Wikipedia dataset: `http://wiki.dbpedia.org/`

PageRank algorithm in two ways: once using our MapReduce library and once using a direct implementation, which takes advantage of the expressive power of our framework. Both implementations use the same block size of 100 for the underlying block-list data type. The second implementation is an iterative algorithm, which performs sparse matrix-vector multiplication at each step, until convergence.

In both implementations, we use floating-point numbers to represent PageRank values. Due to the imprecision in equality check for floating point numbers, we set three parameters to control the precision of our computation: 1) the iteration convergence threshold $\texttt{con}_\varepsilon$; 2) the equality threshold for page rank values $\texttt{eq}_\varepsilon$, i.e. if a page rank value does not change for more than $\texttt{eq}_\varepsilon$, we will not recompute the value; 3) the equality threshold for verifying the correctness of the result $\texttt{verify}_\varepsilon$. For all our experiments, we set $\texttt{con}_\varepsilon = 1 \times 10^{-6}$, and $\texttt{eq}_\varepsilon = 1 \times 10^{-8}$. For each change, we also perform a batch run to ensure the correctness of the result. All our experiments guarantee that $\texttt{verify}_\varepsilon \leq 1 \times 10^{-5}$.

Our experiments with PageRank show that MapReduce based implementation does not scale for incremental computation, because it requires massive amounts of memory, consuming 80GB of memory even for a small downsampled Twitter graph with $3 \times 10^3$ vertices and $10^4$ edges. After careful profiling, we found that this is due to the shuffle step performed by MapReduce, which is not needed for the PageRank algorithm. This is an example where a domain-specific approach such as MapReduce is too restrictive for an efficient implementation.

Our second implementation, which uses the expressive power of functional programming, performs well. Compared to the MapReduce-based version, it requires 0.88GB memory on the

same graph, nearly 100-fold less, and the update time is 50x faster on average.[2] We are thus able to use the second implementation on relatively large graphs. Table 2 shows a summary of our findings. For these experiments, we divide the edges into groups of $1,000$ edges starting with the first vertex and consider each of them in turn: for each group, we measure the time to complete the following steps: 1) delete all the edges from the group, 2) update the result, 3) reintroduce the edges, and 4) update the result. Since the average degree per vertex is approximately 100, each aggregate change affects approximately 10 vertices, which can then propagate to other vertices. (Since the vertices are ordered arbitrarily, this aggregate change can be viewed as inserting/deleting 10 arbitrarily chosen vertices).

Our PageRank implementation delivers significant speedups at the cost of approximately 10x more memory with different graphs including the datasets Orkut[3], LiveJournal[4] and Twitter graph[5]. For example on the Twitter datasets (labeled Twitter-1) with 30M vertices and 700M edges, our PageRank implementation reaches an average speedup of more than 500x compared to the batch version, at the cost of 10x more memory. Detailed measurements for the first 100 groups, as shown in Figure 17(left), show that for most trials, speedups usually approximate 4 orders of magnitude.

### 8.5 Incremental graph connectivity

Connectivity, which indicates the existence of a path between two vertices, is a central graph problem with many applications. Our incremental graph connectivity benchmark computes a label $\ell(v) \in \mathbb{Z}_+$ for every node $v$ of an undirected graph such that two nodes $u$ and $v$ have the same label (i.e. $\ell(u) = \ell(v)$) if and only

[2] This performance gap increases with the input size, so this is quite a conservative number.

[3] Orkut dataset: `http://snap.stanford.edu/data/com-Orkut.html`

[4] LiveJournal dataset:
`http://snap.stanford.edu/data/com-LiveJournal.html`

[5] Twitter dataset:`http://an.kaist.ac.kr/traces/WWW2010.html`

if $u$ and $v$ are connected. We use a randomized version of Kang et al.'s algorithm [36] that starts with random initial labels for improved incremental efficiency. The algorithm is iterative; in each iteration the label of each vertex is replaced with the minimum of its labels and those of its neighbors. We evaluate the efficiency of the algorithm under dynamic changes by for each vertex, deleting that vertex, updating the result, and reintroducing the vertex. We test the benchmark on an undirected graph from LiveJournal with 1M nodes and 8M edges. Our findings for 100 randomly selected vertices are shown in Figure 17(center); cumulative (average) measurements are shown in 2. Since deleting a vertex can cause widespread changes in connectivity, affecting many vertices, we expect this benchmark to be significantly more expensive than PageRank. Indeed, each change is more expensive than in PageRank but we still obtain speedups of as much as 200x.

### 8.6 Incremental social circles

An important quantity in social networks is the size of the circle of influence of a member of the network. Using advances in streaming algorithms, our final benchmark estimates for each vertex $v$, the number of vertices reachable from $v$ within 2 hops (i.e., how many friends and friends of friends a person has). Our implementation is similar to Kang et al.'s [35], which maintains for each node 10

Flajolet-Martin sketches (each a 32-bit word). The technique can be naturally extended to compute the number of nodes reachable from a starting point within $k$ hops ($k > 2$). To evaluate this benchmark, we use a down-sampled Twitter graph (Twitter-2) with 100K nodes and 2M edges. The experiment divides the edges into groups of 20 edges and considers each of these groups in turn: for each group, we measure the time to complete the following steps: delete the edges from the group, update social-circle sizes, reintroduce the edges, and update the social-circle sizes. The findings for 100 groups are shown in Figure 17(right); cumulative (average) measurements are shown in 2 in the last row. Our incremental version is approximately 100x faster than batch for most trials.

# 9. Related Work

Incremental computation techniques have been extensively studied in several areas of computer science. Much of this research focuses on time efficiency rather than space efficiency. In addition, there is relatively little (if any) work on providing control over the space-time tradeoff fundamental to essentially any incremental-computation technique. We discussed closely related work in the introduction (Section 1). In this section, we present a brief overview of some of the more remotely related work.

**Algorithmic Solutions.** Research in the algorithms community focuses primarily on devising *dynamic algorithms* or *dynamic data structures* for individual problems. There have been hundreds of papers with several excellent surveys reviewing the work (e.g., [23, 47]. Dynamic algorithms enable computing a desired property while allowing modifications to the input (e.g., inserting/deleting elements). These algorithms are often carefully designed to exploit

problem-specific structures and are therefore highly efficient. But they can be quite complex and difficult to design, analyze, and implement even for problems that are simple in the batch model where no changes to data are allowed. While dynamic algorithms can, in principle, be used with large datasets, space consumption is a major problem [22]. Bader et al. [48] present techniques for implementing certain dynamic graphs algorithms for large graphs.

**Language-Based Approaches.** Motivated by the difficulty in designing and implementing ad hoc dynamic algorithms, the programming languages community works on developing general-purpose, language-based solutions to incremental computation. This research has lead to the development of many approaches [21, 27, 46, 47],

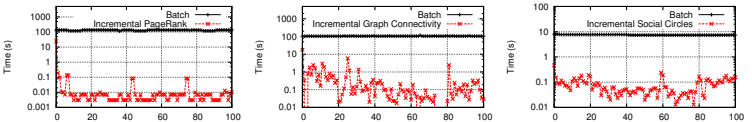


**Figure 17.** (left) PageRank: 100 trials (*x*-axis) of deleting 1,000 edges; (center) Connectivity: 100 trials of deleting a vertex; (right) Approximate social-circle size: 100 trials of deleting 20 edges. *Note: y-axis is in log-scale.*

including static dependency graphs [21], memoization [46], and partial evaluation [27]. Recent advances on self-adjusting computation [1, 6] builds on this prior work to offer techniques for efficient incremental computation expressed in a general-purpose purely functional and imperative languages. Variants of self-adjusting computation has been implemented in SML [1], Haskell [12], C [30], and OCaml [31]. The techniques have been applied to a number of problems in a relatively diverse set of domains including motion simulation [2, 5], dynamic computational geometry [7, 8], and machine learning [3, 51].

In more recent work, researchers proposed improvements on the power of underlying self-adjusting computation techniques. Hammer et al proposed techniques for demand-driven self-adjusting computation, where updates may be delayed until they are demanded [31]. Another line of research realized an interesting duality between incremental and parallel computation—both benefit from identifying independent computations—and proposed techniques for parallel self-adjusting computation. Some earlier work considered techniques for performing efficient parallel updates in the context of a lambda calculus extended with fork-join style parallelism [29]. Follow-up work considered the technique in the context of a more sophisticated problem showing both theoretical and empirical results

of its effectiveness [7]. Burckhardt et al consider a more powerful language based on concurrent-revisions, provide techniques for parallel change propagation for programs written in this language, and perform an experimental evaluation. Their evaluation shows relatively broad effectiveness in a challenging set of benchmarks [11].

**Systems.** There are several systems for big data computations such as MapReduce [20], Dryad [32], Pregel [40], GraphLab [39], and Dremel [41]. While these systems allow for computing with large datasets, they are primarily aimed at supporting the batch model of computation, where data does not change, and consider domain-specific languages such as flat data-parallel algorithms and certain graph algorithms.

Data flow systems like MapReduce and Dryad have been extended with support for incremental computation. MapReduce Online [18] can react efficiently to additional input records. Nectar [28] caches the intermediate results of DryadLINQ programs and generates programs that can re-use results from this cache. Prior work on Incoop applies the principles of self-adjusting computation to the big data setting but only in the context of MapReduce, a domain-specific language, by extending Hadoop to operate on dynamic datasets [10]. In addition, Incoop supports an asymptotically suboptimal change-propagation algorithm. Naiad [42] enables incremental computation on dynamic datasets in programs written with a specific set of data-flow primitives. In Naiad, dynamic updates cannot alter the dependency structure of the computation. Naiad is thus closely related to earlier work on incremental computation with static dependency graphs [21, 56]. Percolator [45] is Google's proprietary system that enables a more general programming model but requires programming in an event-based model with call-backs (notifica-

tions), a very low level of abstraction. While domain specific, these systems can all run in parallel and on multiple machines. The work that we presented here assumes sequential computation.

**Functional Reactive Programming.** More remotely related work includes functional reactive programming. Elliott and Hudak [26] introduced functional reactive programming (FRP) to provide primitives for operating on time-varying values. While greatly expressive, Elliott and Hudak's proposal turned out to be difficult to implement safely and efficiently, leading to much follow-up work on refinements such as real-time FRP [54], event-driven FRP [55], arrowized FRP [38], which restrict the set of acceptable FRP programs by using syntax and types to make possible efficient implementation. More recent approaches to FRP based on temporal logics include those of Sculthorpe and Nilsson [49], Jeffrey [33], Jeltsch [34], and Krishnaswami [37].

Much of the work on FRP can be viewed as a generalization of synchronous dataflow languages [9, 13] to handle richer computations where the dataflow graph can accept certain changes between steps. One limitation of the synchronous approach to reactive programming is that one step cannot be started before the previous one finishes. This leads to a range of other practical difficulties, such as the choice of the right frequency (or step size) for updates [19, 50]. Czaplicki and Chong propose techniques for asynchronous execution that allow certain computations to span multiple time steps [19].

While it appears likely that FRP programs would benefit from the efficiency improvements of incremental updates, much of the aforementioned work does not provide support for incremental updates. One exception is the recent work of Demetrescu et al, which provides the programmer with techniques for writing incremental

update functions in (imperative) reactive programs [24]. Another exception is Donham's Froc [25], which provides support for FRP based on a data-driven implementation using self-adjusting computation.

## 10. Conclusion

We present techniques for improving the scalability of automatic incrementalization techniques based on self-adjusting computation. These techniques enable expressing big-data applications in a functional language and rely on 1) an information-flow type systems and translation algorithm for tracking dependencies precisely, and 2) a probabilistic chunking technique for controlling the fundamental space-time trade-off that self-adjusting computation offers. Our results are encouraging, leading to important improvements over prior work, and delivering significant speedups over batch computation at the cost of moderate space overheads. Our results also show that functional programming can be significantly more effective than domain-specific languages such as MapReduce. In future work, we plan to parallelize these techniques, which would enable scaling to larger problems that require multiple computers. Parallelization seems fundamentally feasible because functional programming is inherently compatible with parallel computing.



## Acknowledgements

This research is partially supported by the National Science Foundation under grant number CCF-1320563 and by the European

Research Council under grant number ERC-2012-StG-308246.

# References

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

[2] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and J. L. Vittes. Kinetic algorithms via self-adjusting computation. In *Proceedings of the 14th Annual European Symposium on Algorithms*, pages 636–647, Sept. 2006.

[3] U. A. Acar, A. Ihler, R. Mettu, and O. Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

[4] 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.

[5] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, Sept. 2008.

[6] 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.

[7] 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.

[8] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. *Journal of Computational Geometry: Theory and Applications*, 2013.

[9] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, Nov. 1992. ISSN 0167-6423.

[10] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symposium*

*on Cloud Computing*, 2011.

[11] 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.

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

[13] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, 1987. ISBN 0-89791-215-2.

[14] Y. Chen, U. A. Acar, and K. Tangwongsan. Appendix to functional programming for dynamic and large data with self-adjusting computation. URL `http://www.mpi-sws.org/~chenyan/papers/icfp14-appendix.pdf`.

[15] 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.

[16] Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2012.

[17] Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Implicit self-adjusting computation for purely functional programs. *Journal of Functional Programming*, 24:56–112, 1 2014. ISSN 1469-7653.

[18] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proc. 7th Symposium on Networked systems design and implementation (NSDI'10)*.

[19] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 411–422, 2013. ISBN 978-1-4503-2014-6.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[21] 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.

[22] 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.

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

[24] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[25] J. Donham. Froc: a library for functional reactive programming in ocaml, 2010. URL `http://jaked.github.com/froc`.

[26] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

[27] J. Field and T. Teitelbaum. Incremental reduction in the lambda calculus. In *ACM Conference on LISP and Functional Programming*, pages 307–322, 1990.

[28] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in data centers. In *OSDI'10*.

[29] 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.

[30] 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.

[31] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, 2014. ISBN 978-1-4503-2784-8.

[32] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, Mar. 2007. ISSN 0163-5980.

[33] A. Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, pages 49–60, 2012. ISBN 978-1-4503-1125-0.

[34] W. Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*, PLPV '13, pages 69–78, 2013. ISBN 978-1-4503-1860-0.

[35] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011.

[36] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.

[37] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. *SIGPLAN Not.*, 48(9):221–232, Sept. 2013. ISSN 0362-1340.

[38] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, 2009. ISBN 978-1-60558-332-7.

[39] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB Endow.*, 5(8):716–727, Apr. 2012.

[40] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing.

In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, 2010.

[41] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011. ISSN 0001-0782.

[42] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proc. of SOSP*, pages 439–455, 2013.



[43] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, pages 174–187, 2001.

[44] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[45] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

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

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

[48] E. J. Riedy, H. Meyerhenke, D. A. Bader, D. Ediger, and T. G. Mattson. Analysis of streaming social networks and graphs on multicore architectures. In *ICASSP*, pages 5337–5340, 2012.

[49] N. Sculthorpe and H. Nilsson. Keeping calm in the face of change. *Higher Order Symbol. Comput.*, 23(2):227–271, June 2010. ISSN 1388-3690.

[50] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. *SIGPLAN Not.*, 44(9):23–34, Aug. 2009. ISSN 0362-1340.

[51] O. Sümer, U. A. Acar, A. Ihler, and R. Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 8:180–186, 2011.

[52] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, June 1994. ISSN 0890-5401.

[53] K. Tangwongsan, H. Pucha, D. G. Andersen, and M. Kaminsky. Efficient similarity estimation for systems exploiting data redundancy. In *INFOCOM*, pages 1487–1495, 2010.

[54] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. *SIGPLAN Not.*, 36 (10):146–156, 2001.

[55] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, 2002.

[56] D. M. Yellin and R. E. Strom. INC: a language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, Apr. 1991.