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

Yan Chen  
MPI-SWS  
chenyan@mpi-sws.org

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

Kanat Tangwongsan  
Mahidol University International College  
ktangwon@gmail.com

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. Compared to prior work, this work overcomes the main challenge of reducing the space usage of self-adjusting computation without disproportionately decreasing performance. To this end, 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 of spurious dependencies. We give a new 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 very promising results on challenging benchmarks and large graphs.

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: they change gradually over time. For example, a social-network graph from scientific applications to social networks, and dynamic: they change gradually over time. For example, a social-network graph changes as users join or leave the social network 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” [12] with limited expressiveness that not only restrict the set of problems 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 and tools, functional programming can overcome both of these limitations. First, as an expressive, general purpose programming model, functional programming can enable an efficient implementation of a broad range of algorithms for big data. Second, since functional programming is consistent with self-adjusting computation [1, 3, 9, 10], 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, 3, 9, 10] 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 affected by the changes.

Unfortunately, existing approaches to self-adjusting computation require a significant amount of memory to store the dynamic dependency graph, making them practically impossible to use with large datasets. 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 massive demands for space thus far have limited applicability of the technique to relatively small problem sizes.

This paper presents two techniques for overcoming this limitation of prior work. The first technique improves the space efficiency by improving the precision of dependency tracking that self-adjusting computation relies on. The second technique enables controlling the space-time tradeoff fundamental to self-adjusting computation. Our first technique relies on a type system for tracking dependencies precisely and a type-directed translation algorithm that can generate correct and efficient elf-adjusting programs. Our second technique is a probabilistic chunking scheme for coarse-graining 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 [9, 10]. 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 changes 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 $\Theta$. 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 5) 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 by breaking the assumption from prior work and allowing changeable data to be nested inside non-changeable data. We present a translation algorithm that can nevertheless produce 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 $\alpha$ 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 written 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 trade-off 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. The technique enables reducing the size of the dependency metadata by a desired factor $B$ (block size) under programmer control, while slowing down updates only by a factor of $\Theta(B)$.

We implement the proposed techniques in Standard ML and present an empirical evaluation by considering several list primitives, sorting algorithms, and 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 shows that

- Expressive languages such as lambda calculus instead of domain-specific languages such as MapReduce can lead to large (e.g., 50-100 fold) improvements in time and space efficiency.
- The type system for precise dependency tracking can significantly reduce space and time requirements (e.g., approximately by 2 and 10 folds respectively).
- Our techniques for controlling the space-time trade-off can reduce memory consumption effectively under programmer control, while 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.

Specific contributions of the paper include the type system, the translation algorithm from the source to the target language, the probabilistic chunking scheme for controlling the space-time tradeoff, the incremental algorithms for graphs, the implementation, and the evaluation.

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 $\mathcal{f}$, which applies $\mathcal{f}$ 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 $f$ evaluated to false. The elements of pos and neg retain the same relative order from $f$. Omitting the annotation $\mathcal{C}$, 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 $\alpha$. The runtime system of a self-adjusting language track dependencies on modificables 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 $\alpha$ 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 $\mathcal{C}$ (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 $\mathcal{C}$; all other types remain stable, meaning they cannot be changed later on. For example, int is a stable integer, int is a changeable integer and int list 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 modificables: 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 $\alpha$ $\beta$ 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 type to be changeable, i.e., the type $\alpha$ list mod $\star$ $\alpha$ list 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).
We therefore would like to derive the expression:

\[ \text{fun partition f l} = \begin{cases} \text{case l of} \\
\text{nil \Rightarrow (nil, nil)} \\
\text{h::t \Rightarrow let val (a,b) = partition f t then read pair as (a,b) in} \\
\text{write (mod (write h:a), b)} \\
\text{else (a, h:b) in} \\
\text{write (a, mod (write h:b))} \\
\text{end} \end{cases} \]

\[ \text{fun partition f l} = \begin{cases} \text{case l of} \\
\text{nil \Rightarrow (nil, nil)} \\
\text{h::t \Rightarrow let val (a,b) = partition f t then read pair as (a,b) in} \\
\text{write (mod (write h:a), b)} \\
\text{else (a, h:b) in} \\
\text{write (a, mod (write h:b))} \\
\text{end} \end{cases} \]

\[ \text{fun partition f l (l₀, l₁)} = \begin{cases} \text{case l of} \\
\text{nil \Rightarrow (write (l₀, nil)); \hspace{1em} write (l₁, nil))} \\
\text{h::t \Rightarrow let val (a,b) = let} \\
\text{val (l₀, l₁) = (mod nil, mod nil) in} \\
\text{partition f t (l₀, l₁) end} \end{cases} \]

\[ \text{fun partition f l (l₀, l₁)} = \begin{cases} \text{case l of} \\
\text{nil \Rightarrow (write (l₀, nil)); \hspace{1em} write (l₁, nil))} \\
\text{h::t \Rightarrow let val (a,b) = let} \\
\text{val (l₀, l₁) = (mod nil, mod nil) in} \\
\text{partition f t (l₀, l₁) end} \end{cases} \]

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), using 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 (\( \alpha \text{ list} \text{ mod} * \alpha \text{ list} \text{ mod} \)), 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 a translation. (1) The source language has 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.

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 \( 4x \) 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 dependences 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 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.
we label it based on the structure of the product. Specifically, we will discuss the reason in Section 4.

As an example, we consider only expressions in A-normal form, which names in-...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. [9] 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:

$$\mathbb{S} \preceq \mathbb{C}_{\rho} \preceq \mathbb{C}_{\rho} \preceq \mathbb{C}_{\rho}$$

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 is lower than labels with prefix 0, we will discuss the reason 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 $\tau \ Down \ \Delta; L$, 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 $\Delta$ and $L$, 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, $\tau = \left(\text{int}^{C_{\rho}} \times (\text{int}^{C_{\rho}} + \text{int}^{C_{\rho}})\right)^{\delta}$ is a valid label for $\tau \ Down \ \Delta; L$.

Subtyping. Figure 4 shows the subtyping relation $\tau \prec \tau'$, 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 $\delta \prec \tau$, 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 $\tau_1 \equiv \tau_2$, if $\tau_1 \equiv \tau_2$ 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 intermediate 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
The typing judgment $C; P; \Gamma \vdash e : \tau$ and source typing environment $\Gamma$, source expression $e$ has type $\tau$

$$C; P; \Gamma \vdash e : \tau \quad \text{(SVar)}$$

$C; P; \Gamma \vdash x : \tau \quad \text{(SInt)}$

$C; P; \Gamma \vdash \text{fst} x : \tau_1 \quad \text{(SFst)}$

$C; P; \Gamma \vdash \text{snd} x : \tau_2 \quad \text{(SFst)}$

$C; P; \Gamma \vdash (x_1, x_2) : \text{int}^2 \quad \text{(SPair)}$

$C; P; \Gamma \vdash \text{int} x : \tau_1 \quad \text{(SSum)}$

$C; P; \Gamma \vdash \text{int} \rightarrow \tau_2 : \text{int}^\tau \quad \text{(SPrim)}$

$C; P; \Gamma \vdash \text{let} x = e_1 : \tau \quad \text{(SLet)}$

$C; P; \Gamma \vdash \text{let} x = e_1 \in e_2 : \tau \quad \text{(SPrim)}$

$C; P; \Gamma \vdash \text{apply}(x_1, x_2) : \tau \quad \text{(SApp)}$

$C; P; \Gamma \vdash \text{case} x \ of \ \{ x_1 \Rightarrow e_1, \ x_2 \Rightarrow e_2 \} : \tau \quad \text{(SCase)}$

Figure 8. Typing rules for source language

---

Figure 4. Subtyping

$$\frac{\delta \leq \delta'}{\delta \preceq \text{int}^\delta : \text{int}^\delta} \quad \text{(SInt)}$$

$$\frac{\delta \preceq \tau_1 \quad \delta \preceq \tau_2 \quad \delta \preceq \delta'}{\tau_1 \times \tau_2 : \text{int}^\delta} \quad \text{(Prod)}$$

$$\frac{\delta \preceq \tau_1 \quad \delta \preceq \tau_2 \quad \delta \preceq \delta'}{\tau_1 \rightarrow \tau_2 : \text{int}^\delta} \quad \text{(Arrow)}$$

$$\frac{\delta \preceq \tau_1 \quad \delta \preceq \tau_2 \quad \delta \preceq \delta'}{\text{int}^\tau \rightarrow \tau_2 : \text{int}^\delta} \quad \text{(Let)}$$

$$\frac{\delta \preceq \tau_1 \quad \delta \preceq \tau_2 \quad \delta \preceq \delta'}{\text{int}^\delta \rightarrow \tau_2 : \text{int}^\delta} \quad \text{(Case)}$$

Figure 5. Lower bound of a type

---

Figure 3. Target Language

Abstract syntax. The target language (Figure 3) is an imperative self-adjusting language with modiftables. In addition to integers, units, products, sums, the target type system makes a distinction between fresh modifiable types $\text{int}^\tau$ (modifiables that are freshly allocated) and finalized modifiable types $\text{int}^\mod$ (modifiables that are written after the allocation). The function type $\tau_1 \rightarrow \tau_2$ contains
Types \[ \Sigma ::= \text{unit} \mid \text{int} \mid \tau \text{mod} \mid \tau \sqcup \tau \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau \vec{\circ} \tau \mid \tau_1 > \tau_2 \]

Dest. Types \[ \mathcal{D} ::= [x_1, \ldots, x_n] \]

Labels \[ \mathcal{L} ::= [l_1, \ldots, l_n] \]

Variables \[ x ::= y \mid l_i \]

Typing Env. \[ \Gamma ::= \emptyset \mid \Gamma, x : \tau \]

Values \[ v ::= n \mid x \mid f (v_1, v_2) \mid \text{int} v \mid \text{intr} v \mid \text{fun}^L f (x) = e \]

Expressions \[ e ::= v \mid (\theta(x_1, x_2)) \mid \text{fst} x \mid \text{snd} x \mid \text{apply}^L (x_1, x_2) \mid \text{let} x = e_1 \text{ in } e_2 \mid \text{case} x \text{ of } (x_1 \Rightarrow e_1, x_2 \Rightarrow e_2) \mid \text{mod} v \mid \text{read} x \text{ as } y \text{ in } e \mid \text{write} (x_1, x_2) \]

Figure 9. Types and expressions in the target language

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 \( f \) (which appear only at runtime), pairs, tagged values, and functions. Each function \( \text{fun}^L f (x) = e \) 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 \( \text{apply}^L (x_1, x_2) \) applies a function while supplying a set of destination modifiables \( \mathcal{L} \). The \text{mod} \( v \) constructor creates a new fresh modifiable \( \sqcup \tau \) with an initial value \( v \). The \text{read} expression binds the contents of a modifiable \( x \) to a variable \( y \) and evaluates the body of the \text{read}. The \text{write} constructor imperatively updates the modifiable \( x_1 \) with value \( x_2 \). The \text{write} operator can update both modifiables in destination labels \( \mathcal{L} \) and modifiables created by \text{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 \( \tau \), rule (TAlloc) creates a fresh modifiable of type \( \sqcup \tau \). 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 \( \tau \) into a modifiable \( x_1 \), when \( x_2 \) is a fresh modifiable of type \( \sqcup \tau \), and produces a new typing environment substituting the type of \( x_1 \) into an finalized modifiable type \( \tau \text{mod} \). Note that Rule (TWrite) only allows writing into a fresh modifiable, thus guarantees that each modifiable can be rewritten only once. Intuitively, \text{mod} and \text{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 \( \tau \text{mod} \).

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 \text{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 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 \text{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 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. [22] after two syntactical changes: \( \text{fun}^L f (x) = e \) is represented as \( \text{fun}^L f (x) = \Delta \mathcal{L} e \), and \( \text{apply}^L (x_1, x_2) \) is represented as \( (x_1, x_2) \mathcal{L} \).
Under closed source typing environment of we define the function as,

\[ \tau \Gamma \]

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 typedirected translation. Since the source and the target languages have different type systems, an expression \( e : \tau \) cannot be translated to a target expression \( \tau' \) where \( \tau' \) 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 from the source language into the target types. We also use it to translate the types in \( \Gamma \).

\[ \| \| = \cdot \| \|_0 = \| \phi \| \]

\[ \| \| \Gamma, x : \tau \| = \| \|_1 \| \|_1 (\| \|_1 = \| \phi \| ) \]

Figure 11. Translations of types and typing environments

6. Translation

For source types with changeable levels, the target type will be changeable as well. Since the source language is purely functional, the final result will always be a finalized modifiable \( \tau \) mod. Here, we define a stabilization function \( \| \tau \| \) for changeable source types, which changes the outer level of \( \tau \) from changeable into stable. Formally, we define the function as,

\[ \| \tau \| = \tau' \text{, where } \| \| = \| \|_1 \text{ and } \tau = \tau' \]

Then, the target type for a changeable level source type \( \tau \) will be \( \| \| \tau \| \mu \).

6.2 Translating Expressions

We define the translation of expressions as a set of type-directed rules. Given (1) a derivation of \( \Gamma \), \( \| e : \tau \| \) 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 \( e_1 \) (see Theorem A 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 using it in the translation \( \| \phi \| \). 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.

\[ \Gamma \vdash e : \tau \rightsquigarrow e' \]

Under closed source typing environment \( \Gamma \), source expression \( e \) is translated at type \( \tau \) to target expression \( e' \).

\[ \begin{align*}
\Gamma \vdash n : \text{int}^\mu & \Rightarrow n \\
\Gamma \vdash v_1 : \text{int}^\mu & \Rightarrow v_1' \\
\Gamma \vdash v_2 : \text{int}^\mu & \Rightarrow v_2' \\
\Gamma \vdash (v_1, v_2) : \text{int}^\mu & \Rightarrow (v_1', v_2') \\
\Gamma \vdash \text{fun} f(x) = e : \text{int}^\mu & \Rightarrow \text{fun} e f(x) = e' \\
\end{align*} \]

Figure 12. Translation for destination passing style

Changeable rules. The rules (Lift), (Mod), and (Write) translate expressions with outer level changeable \( \| \|_1 \). Given a translation of \( e \) to some pure expression \( e' \), rule (Write) translates \( e \) into an imperative write expression that writes \( e' \) into modifiable \( l_p \).

For expressions with non-destination changeable levels, that is the label \( l \) 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 \( \tau l \) that takes a source type \( \tau \) and returns any 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 \( \| \tau \| \), capturing the “shallow subsumption” in the constraint typing rules (SLet): a bound expression of type \( \| \tau \| \) can be translated at type \( \tau_0 \) to \( \tau' \), and then “promoted” to type \( \tau_0 \) by placing it inside a modifiable \( l_p \).
This judgment is derivable only for variable, function and application rules. Since the self-adjusting primitive switches to the ordinary translation rules in Figure 12. For example, in the tail position. For all other expressions, the translation simply binds in the usual way and derive destinations for the expressions to be translated. For a case type, type. Figure 14 shows the rules for translating the function body into a target expression that returns unit. To recover tives are imperative, an expression with outer changeable levels 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., [27][35] 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 blocks at 4, 9, and 11, as shown.

To understand what happens when the input changes, consider inserting 2 (with location identifier p) between 1 and 3. Because the hash value of 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 \([1, 3, 4, 5]\), 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 data set 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 re-done, thus often increasing the update time by \( B \) fold.

### 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 1TB 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 \( \tau \mod \) type; (2) for function \( \text{fun}^L f(x) = e \), it includes destination labels as part of the function argument, so the function is represented as \( \text{fun} f(x) = fn \ L \Rightarrow e \). Accordingly, the arrow type \( \tau_1 \rightarrow \tau_2 \) is represented as \( \tau_1 \Rightarrow \tau_2 \rightarrow \tau_2 \), where \( \tau' = \tau_1 \mod \times \cdots \times \tau_n \mod \) and \( D = \{ \tau'_1, \ldots, \tau'_n \} \). 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 [12] 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 unchanged 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 ssort. In the evaluation, all benchmarks operate on integers: map applies \( f(i) = i+2 \) to each element; partition partitions its input based on the parity of each element; reduce computes the sum of the list \( \text{D} \mod 100 \); and ssort 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).

### 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 $10000–100000x$ 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.

### 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 [10] (which contains redundant dependencies) and the other using the techniques presented in this paper. We use a publicly available Wikipedia data set3 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 3. 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 9x 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 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

---

1000 vertices are changed at a time, we don't expect the speedups 100-fold less, and the update time is 50x faster on average.

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 $3 \times 10^5$ vertices and $10^8$ 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 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. 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 there 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), shows that for most trials, speedups usually approximate 4 orders of magnitude. Since 1000 vertices are changed at a time, we don’t expect the speedups to exceed $10^4$, which is difficult to match on average because in PageRank a small change can affect the result for many vertices.

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 $f(v)$ in $Z_2$ for every node $v$ of an undirected graph such that two nodes $u$ and $v$ have the same label (i.e. $f(u) = f(v)$) if and only if $u$ and $v$ are connected. We use a randomized version of Kang et al.’s algorithm [22] 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 Figure 17 (right); cumulative (average) measurements are shown in Figure 17 (right); cumulative (average) measurements are shown in Figure 17 (right); cumulative (average) measurements are shown in Figure 17 (right). 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 [21], which maintains for each node 10 Edgelet-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 Figure 17 (right) 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 but much of this research focuses on time efficiency rather than space efficiency or the 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., [13, 31]. 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 [14]. Bader et al. [32] 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 led to the development of many approaches [13, 16, 30, 31], including static dependency graphs [13], memoization [30], and partial evaluation [16]. Recent advances on self-adjusting computation [13] 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 [11], Haskell [17], C [19], and applied to a number of problems (e.g., [8, 13]).

Systems. There are several systems for big data computations such as MapReduce [12], Dryad [20], Pregel [24], GraphLab [23].

---

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
6 PageRank a small change can a
7 efficiency rather than space e
and Dremel [25]. 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 [11] can react efficiently to additional input records. Nectar [17] 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 [5]. In addition, Incoop supports an asymptotically suboptimal change propagation algorithm. Naiad [26] enables incremental computation on dynamic datasets in programs written with a specific set of data-flow primitives. In Naiad dynamic changes cannot alter the dependency structure of the computation, which makes it closely related to earlier work on incremental computation with static dependency graphs [13][36]. Percolator [29] is Google’s proprietary system that enables a more general programming model but requires programming in an event-based model with call-backs (notifications), a very low level of abstraction.

While domain specific, these systems can all run in parallel and on multiple machines. In principle our approach can also be parallelized, especially because purely functional programming is naturally amenable to parallelism. Such a parallelization would require parallelizing the underlying self-adjusting computation techniques. There has been some research on this problem, but existing solutions work in certain domains and/or use a sub-optimal algorithms for parallel change propagation [5][6][18].

10. Conclusion

We present techniques for scaling automatic incrementalization techniques based on self-adjusting computation to large data sets. These techniques enable expressing big-data applications in a functional language and rely on 1) a new 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 very encouraging, leading to important improvements over prior work, and delivering significant speedups over batch computation at the cost of moderate and programmable space overhead. 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.

References


Appendix

In this section, we show that translation maps source types to corresponding target types.

A. Translation Type Soundness

Lemma A.1 (Substitution). Suppose \( \phi \) is a satisfying assignment for \( C \), and \( \phi(\bar{a}) = \bar{\delta} \), where \( \bar{a} \subseteq \text{FV}(C) \).

1. If \( D \) derives \( C; P; \Gamma \vdash e : \tau \), then there exists \( D' \) deriving \( C; \bar{\delta}/\bar{a}; \Gamma \vdash e : [\bar{\delta}/\bar{a}]\tau \), where \( D' \) has the same height as \( D \).
2. If \( C \vdash \delta' \triangleleft \tau \), then \( C \vdash [\delta'/\bar{a}]\delta \triangleleft [\delta/\bar{a}]\tau \).
3. If \( C \vdash \tau' \triangleleft \tau'' \), then \( C \vdash [\delta'/\bar{a}]\tau' \triangleleft [\delta/\bar{a}]\tau'' \).
4. If \( C \vdash \tau' \equiv \tau'' \), then \( C \vdash [\delta'/\bar{a}]\tau' \equiv [\delta/\bar{a}]\tau'' \).

Proof. By induction on the given derivation.

We define \( \tau \text{ O.S. if } \|\tau\| = S \), and \( \tau \text{ O.C. if } \|\tau\| = C_p \).

Lemma A.2. Given \( \tau' \triangleleft \tau'' \) and \( \tau' \equiv \tau'' \):
1. If \( \tau'' \text{ O.S. then } \tau' = \tau'' \).
2. If \( \tau'' \text{ O.C. then either } \tau' = \tau'' \text{ or } \tau' = [\tau'']^\phi \).

Proof. By induction on the derivation of \( \tau' \triangleleft \tau'' \).

- **Case** (subInt): \( \tau' = \text{int}^\phi \) and \( \tau'' = \text{int}^{\phi''} \), where \( \delta' \leq \delta'' \).
  - (1) If \( \tau'' \text{ O.S. then } \delta'' = S \). So \( \tau' = \tau'' \).
  - (1) If \( \tau'' \text{ O.C. then } \delta'' = C_p \). If \( \delta = S \) then \( [\tau'']^\phi = \text{int}^\phi = \tau' \); if \( \delta' \leq C_p \) then \( \tau'' = \text{int}^{\phi''} = \text{int}^{\phi} = \tau' \).
- **Case** (subProd):
  - (1) By definition of \( \equiv \), \( \tau' = \tau'' \).
  - (1) \( \tau'' \text{ O.C. is impossible.} \)
- **Case** (subSum):
  - (1) If \( \tau'' \text{ O.S. then } \tau'' = (\tau''_1 + \tau''_2)^\phi \). By inversion on (subSum), \( \tau' = (\tau''_1 + \tau''_2)^\phi \). By definition of \( \equiv \), \( \tau''_1 = \tau''_1'' \text{ and } \tau''_2 = \tau''_2'' \). Therefore \( \tau' = \tau'' \).
  - (1) If \( \tau'' \text{ O.C. then } \tau'' = (\tau''_1 + \tau''_2)^\phi \). By inversion on (subSum), \( \tau' = (\tau''_1 + \tau''_2)^\phi \). By definition of \( \equiv \), \( \tau''_1 = \tau''_1'' \text{ and } \tau''_2 = \tau''_2'' \). If \( \delta' \leq S \) then \( [\tau''_1]^\phi = (\tau''_1 + \tau''_2)^\phi \), which is equal to \( \tau' \). If \( \delta' = C_p \) then \( \tau'''' = (\tau''_1 + \tau''_2)^\phi = (\tau''_1 + \tau''_2)^\phi = (\tau''_1 + \tau''_2)^\phi = \tau' \).
- **Case** (subArrow): Similar to the (subSum) case.

Lemma A.3 (Translation of Outer Levels).

\[ [\phi] \Gamma \vdash \tau \text{ O.C. if and only if } \|\tau\| = [\|\tau\|]_c \text{ mod; } \]
\[ [\phi] \Gamma \vdash \tau \text{ O.S. if and only if } \|\tau\| = [\|\tau\|]_c. \]

Proof. Case analysis on \([\phi] \tau\), using the definitions of \( - \text{ O.S.}, - \text{ O.C.}, \|\|\tau\|_c \) and \( [\|\tau\|]_c \).

Lemma A.4. If \( \Lambda; \Gamma \vdash \nu : \tau \rightarrow \Gamma' \) then \( \Gamma = \Gamma' \).

Proof. By induction on the given derivation.

Theorem. If

(a) \( C; P; \Gamma \vdash e : \tau \), and
(b) \( \phi \) is a satisfying assignment for \( C \),

then

1. there exists \( e' \) and \( \Gamma' \) such that \([\phi] \Gamma \vdash e : [\phi] \Gamma \sim e'\), and \( :[\|\|\Gamma\|]_c \vdash e' : \|\|\tau\|_c + \Gamma' \).
2. there exists \( e' \) and \( \Gamma' \) such that \([\phi] \Gamma \vdash e : [\phi] \Gamma' \sim e'\), and \( :[\|\|\Gamma\|]_c \vdash e' : \|\|\tau\|_c + \Gamma' \).

Proof. By induction on the height of the derivation of \( C; P; \Gamma \vdash e : \tau \).

We present the proof in a line-by-line style, with the justification for each step on the right. Since we need to show that four different judgments are derivable (translation and typing for target program, and translation and typing for return expression), and often arrive at some of them early, we indicate them with \( \#\tau \).
• Case

\[ C; \mathcal{P}; \Gamma \vdash n : \text{int} \] *(SInt)*

Part (1): Let \( e' \) be \( n \).

- \([\phi] \Gamma \vdash n : \text{int} \simeq n \) \(\uparrow \) By (Int)
- \([\phi] \Gamma \vdash e : [\phi] \text{int} \simeq e' \) \(\uparrow \) By \( n = e \) and defn. of substitution
- \(-[\Gamma] \vdash n : \text{int} + [\Gamma] \) \(\uparrow \) By (TInt)
- \(-[\Gamma] \vdash e' : [\Gamma] \text{int} \ominus [\Gamma] \) \(\uparrow \) By \( \text{int} = [\Gamma] \text{int} \ominus [\Gamma] \) \(\uparrow \) \( e' \in \text{int} \)

Part (2): Let \( e' \) be \( n \).

- \([\phi] \Gamma \vdash n : \text{int} \simeq n \) Above
- \([\phi] \Gamma \vdash e : [\phi] \text{int} \simeq e' \) \(\uparrow \) By (RTrans)
- \(-[\Gamma] \vdash e' : [\Gamma] \text{int} \ominus [\Gamma] \) Above

• Case

\[ C; \mathcal{P}; \Gamma \vdash v_1 : \tau_1 \quad \quad C; \mathcal{P}; \Gamma \vdash v_2 : \tau_2 \] *(SPair)*

- Part (1)

\[ C; \mathcal{P}; \Gamma \vdash v_1 : \tau_1 \]

- \([\phi] \Gamma \vdash v_1 : [\phi] \tau_1 \simeq v_1 \) \(\uparrow \) By i.h.
- \(-[\Gamma] \vdash v_1 : [\Gamma] \text{int} \ominus [\Gamma] \) \(\uparrow \) By i.h. and Lemma \[ A,4 \]

\[ C; \mathcal{P}; \Gamma \vdash v_2 : \tau_2 \]

- \([\phi] \Gamma \vdash v_2 : [\phi] \tau_2 \simeq v_2 \) \(\uparrow \) By i.h.
- \(-[\Gamma] \vdash v_2 : [\Gamma] \text{int} \ominus [\Gamma] \) \(\uparrow \) By i.h. and Lemma \[ A,4 \]

Let \( e' = (v_1, v_2) \).

- \([\phi] \Gamma \vdash e : [\phi] (\tau_1 \times \tau_2) \simeq e' \) \(\uparrow \) By def. of substitution and \( e' = (v_1, v_2) \)
- \(-[\Gamma] \vdash (v_1, v_2) : [\Gamma] \tau_1 \times [\Gamma] \tau_2 \ominus [\Gamma] \) \(\uparrow \) By defn. of \([\Gamma] \ominus [\Gamma] \)

- Part (2):

\[ C; \mathcal{P}; \Gamma \vdash v_1 : \tau_1 \]

- \([\phi] \Gamma \vdash v_1 : [\phi] \tau_1 \simeq v'_1 \) \(\uparrow \) By i.h.
- \(-[\Gamma] \vdash v'_1 : [\Gamma] \text{int} \ominus [\Gamma] \) \(\uparrow \) By i.h. and Lemma \[ A,4 \]

\[ C; \mathcal{P}; \Gamma \vdash v_2 : \tau_2 \]

- \([\phi] \Gamma \vdash v_2 : [\phi] \tau_2 \simeq v'_2 \) \(\uparrow \) By i.h.
- \(-[\Gamma] \vdash v'_2 : [\Gamma] \text{int} \ominus [\Gamma] \) \(\uparrow \) By i.h. and Lemma \[ A,4 \]

Let \( e' = (v'_1, v'_2) \).

- \([\phi] \Gamma \vdash e : [\phi] (\tau_1 \times \tau_2) \simeq e' \) \(\uparrow \) By def. of substitution and \( e' = (v'_1, v'_2) \)
- \(-[\Gamma] \vdash (v'_1, v'_2) : [\Gamma] \tau_1 \times [\Gamma] \tau_2 \ominus [\Gamma] \) \(\uparrow \) By defn. of \([\Gamma] \ominus [\Gamma] \)

• Case

\[ C; \mathcal{P}; \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2) \simeq e' : \tau_2 \downarrow \scriptstyle{D} \mathcal{L} \] *(SFun)*
(a) Suppose \( [[\phi]_{\tau_1}] = S \), that is \( \mathcal{P} = \emptyset \).
\[
\begin{align*}
C; \mathcal{P} ; \Gamma \vdash t_1 : \tau_1 & \quad by i.h. \\
\vdash [[\Gamma; \mathcal{P} ; x : \tau_1] \vdash t_1 : \tau_1]^\mathcal{P} \vdash e' : \tau_2 & \quad by (Frst) \\
\vdash [[\Gamma; \mathcal{P} ; f : \tau_1 \rightarrow \tau_2]^\mathcal{P} \vdash e' : [[\phi]_{\tau_2}] \rightarrow e' & \quad by \text{TVar} \\
\vdash [[\Gamma; \mathcal{P} ; x : \tau_1, f : \tau_1 \rightarrow \tau_2]^\mathcal{P} \vdash e' : [[\tau_2]]^\mathcal{P} \vdash e' : [[\Gamma]]^\mathcal{P} & \quad by \text{Premise}
\end{align*}
\]

Let \( e' \) be \( \text{fun}^C f(x) = e' \).

(1) By \text{defn. of } [[\tau]]^\mathcal{P}: 
\[
\begin{align*}
\vdash [[\Gamma]]^\mathcal{P}, x : [[\tau_1]]^\mathcal{P}, f : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} \vdash e' : [[\tau_2]]^\mathcal{P} & \quad by (TFun) \\
\vdash [[\Gamma]]^\mathcal{P}, \text{fun}^C f(x) = e' : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} & \quad by (TFun)
\end{align*}
\]

Let \( e' \) be \( \text{let } r = \text{inr } e' \).

(2) By \text{defn. of } [[\tau]]^\mathcal{P}: 
\[
\begin{align*}
\vdash [[\Gamma]]^\mathcal{P}, x : [[\tau_1]]^\mathcal{P}, f : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} \vdash e' : [[\tau_2]]^\mathcal{P} & \quad by (TFun) \\
\vdash [[\Gamma]]^\mathcal{P}, \text{let } r = \text{inr } e' : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} & \quad Analogous to (a)
\end{align*}
\]

(b) Suppose \( [[\phi]_{\tau_1}] = C_{\mathcal{P}_1} \), that is \( \mathcal{P} = \{1\} \).
\[
\begin{align*}
\vdash [[\Gamma]]^\mathcal{P}, x : [[\tau_1]]^\mathcal{P}, f : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} \vdash e' : [[\phi]_{\tau_2}] \rightarrow e' & \quad by i.h. and \([\phi] = C_{\mathcal{P}_1} \) \\
\vdash [[\Gamma]]^\mathcal{P}, \text{fun}^C f(x) = e' : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} & \quad by (FSup)
\end{align*}
\]

Let \( e' \) be \( \text{fun}^C f(x) = e' \).

(1) By \text{defn. of } [[\tau]]^\mathcal{P}: 
\[
\begin{align*}
\vdash [[\Gamma]]^\mathcal{P}, x : [[\tau_1]]^\mathcal{P}, f : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} \vdash e' : [[\tau_2]]^\mathcal{P} \vdash e' : [[\Gamma]]^\mathcal{P} & \quad by (TFun) \\
\vdash [[\Gamma]]^\mathcal{P}, \text{fun}^C f(x) = e' : [[\tau_1]]^\mathcal{P} \rightarrow [[\tau_2]]^\mathcal{P} & \quad by (TFun)
\end{align*}
\]

Part (1): Similar to (SPair), using \((\tau_1 + \tau_2)^\mathcal{P}\) O.S.

Part (2): Similar to (SPair), using \((\tau_1 + \tau_2)^\mathcal{P}\) O.S.
Part (2): Similar to (SVar):

- If \( \tau_1 \).O.S., let \( e' \) be let \( r = \text{fst } x \) in write(\( r \)) and apply rule (Write).
- If \( \tau_1 \).O.C., let \( e' \) be let \( r = \text{fst } x \) in read \( r \) as \( r' \) in write(\( r' \)) and apply rule (ReadWrite).

• Suppose \( [\phi][\delta] = C_{\rho} \). We have the premise \( C \vdash \delta \leq \), so \( [\phi] = C_{\rho} \); we only need to show part (2).

Part (2):

- If \( \tau_1 \).O.S., let \( e' \) be read \( x \) as \( x' \) in let \( r = \text{fst } x \) in write(\( r \)) and apply rule (Read) with (LFst).

\[
\begin{align*}
\ldots : r : [\tau_1] & \vdash [\tau_1] \quad \text{By (TPVar)} \\
\ldots : r : [\tau_1] & \vdash \text{write}(r) : [\tau_1] \quad \text{By (TWrite)} \\
\ldots : r : [\tau_1] & \vdash \text{write}(r) : [\tau_1] \subseteq C_{\rho} \\
\end{align*}
\]

By (TPVar) then (TFst).

\[
\begin{align*}
[\Gamma], x' : [\tau_1] \times \tau_2 & \vdash \text{fst } x' : [\tau_1] \quad \text{By (TPVar)} \\
\end{align*}
\]

By (TLet)

\[
\begin{align*}
[\Gamma] & \vdash x : (\tau_1 \times \tau_2)^\subseteq [\tau_1] \\
\end{align*}
\]

By i.h.

\[
\begin{align*}
[\Gamma] & \vdash x : [\tau_1] \times [\tau_2] \mod \\
[\Gamma] & \vdash x : [\tau_1] \times [\tau_2] \mod \\
[\Gamma] & \vdash \text{read } x \text{ as } x' \text{ in let } r = \text{fst } x' \text{ in write}(r) : [\tau_1] \subseteq C_{\rho} \\
\end{align*}
\]

By (TRed)

The remaining steps are similar to the \( \tau_1 \).O.S. subcase immediately above.

• Case

(a) Subcase for \([\phi]r''\).O.C.

\[
\begin{array}{c}
C; P; \Gamma \vdash e_1 : \tau' \\
\vdash [\phi] \Gamma \vdash e_1 : [\phi] \tau' \equiv e' \quad \text{By i.h.} \\
\vdash [\phi][\Gamma] \vdash e' : [\tau_1] \subseteq C_{\rho} \\
\Gamma \vdash \tau' \equiv \tau'' \quad \text{Premise} \\
[\phi] \tau'' \equiv [\phi] \tau'' \quad \text{By Lemma A.1} \\
\Gamma \vdash \tau'' \equiv \tau'' \quad \text{Premise} \\
[\phi] \tau'' \equiv [\phi] \tau'' \quad \text{O.C.} \\
\Gamma \vdash [\phi] \tau'' \equiv [\phi] \tau'' \quad \text{Subcase (a) assumption} \\
[\phi] \tau'' = [\phi] \tau'' \quad \text{or} \quad [\phi] \tau'' = [\phi] \tau'' \\
\end{array}
\]

(b) Subcase for \([\phi]r''\)

Now we have the same judgment no matter which equation Lemma A.2 gave us.

\[
\begin{align*}
\vdash [\phi][\Gamma] \vdash e' : [\tau'' \times \tau''] \subseteq C_{\rho} \\
\vdash [\phi][\Gamma] \vdash \text{mod } e' : [\tau'' \times \tau''] \subseteq C_{\rho} \\
\vdash [\phi][\Gamma] \vdash \text{mod } e' : [\tau'' \times \tau''] \subseteq C_{\rho} \\
\vdash [\phi][\Gamma] \vdash [\phi] \tau'' \quad \text{O.C.} \\
[\phi] \tau'' = [\phi] \tau'' \\
\vdash [\phi][\Gamma] \vdash \text{mod } e' : [\tau'' \times \tau''] \\
\end{align*}
\]

Above

By (TMod)

By def. of \(-\) or copying

Subcase (a) assumption

By Lemma A.3

By above equation

4
Proof of Theorem A

Appendix, ICFP 2014

Case

We distinguish four subcases "S-S", "C_\varphi-S", "C_\varphi-C_\varphi", "S-C_\varphi" according to \([\phi]\) and \([\phi] \delta\) respectively.

- **Subcase \("S-S\) for \([\phi] \delta = S\).**
  Part (1):
  
  \[
  C; \Gamma \vdash x_1 : \tau_1 \quad [\phi] \Gamma \vdash x_1 : [\phi] \tau_1 \hookrightarrow x_1 \quad \text{By \(\Gamma\).}
  \]
  
  \[
  \vdash [\Gamma]_0 \vdash x_1 : [\Gamma] \tau_1 = \tau_0 \quad \text{Subderivation}
  \]
  
  \[
  C; \Gamma \vdash x_1 : (\tau_1 \rightarrow \tau) \quad C; \Gamma \vdash x_2 : \tau_1 \quad C \vdash \delta \triangleleft \tau \quad \text{(SApp)}
  \]

  Let \(e' = \text{apply}^\varphi(x_1, x_2)\).

  \[
  [\phi] \Gamma \vdash e : [\phi] \tau \hookrightarrow e' \quad \text{apply}^\varphi(x_1, x_2) \quad \text{By (App)}
  \]

  \[
  [\phi] \Gamma \vdash e : [\phi] \tau \hookrightarrow e' \quad \text{apply}^\varphi(x_1, x_2) \quad \text{By (TApp)}
  \]

  Part (2):
  (a) Suppose \([\phi] \tau\) O.S.

  \[
  [\phi] \Gamma \vdash e : [\phi] \tau \hookrightarrow \text{let } r = e' \in \text{write}(r) \quad \text{By (Write)}
  \]

  \[
  \vdash [\Gamma]_0 \vdash r : [\Gamma] \tau = \tau_0 \quad \text{By (TPVar)}
  \]

  \[
  [\phi] \Gamma \vdash \text{write}(r) : [\Gamma] \tau = \tau_0 \quad \text{By (TWrite)}
  \]

  \[
  [\phi] \Gamma \vdash \text{let } r = e' \in \text{write}(r) : [\Gamma] \tau = \tau_0 \quad \text{By (TLet)}
  \]

  \[
  [\phi] \Gamma \vdash e : [\phi] \tau \hookrightarrow e' \quad \text{O.S.} \quad \text{Subcase (a) assumption}
  \]

  (b) Suppose \([\phi] \tau\) O.C.
Subcase “C⁺⁻S” where \( \phi = C_p \) and \( \phi \delta = \mathbb{S} \).

Part (2):

\[
[\phi] \Gamma \vdash_\phi (x_1 : \phi \rho) \leftrightarrow \phi \ 	ext{read } r \text{ as } r' \text{ in write}(r') \tag{Part (2)}
\]

By (ReadWrite)

Part (1):

\[
[\phi] \Gamma \vdash_\phi e : [\phi] \rho \leftrightarrow e'
\]

Above

\[
[\phi] \Gamma \vdash_\phi e : [\phi] \rho \leftrightarrow \text{mod } e'
\]

By (Mod)

\[
[\phi] \Gamma \vdash_\phi \text{mod } e' : [[\tau]]^C_\rho \mod
\]

By (TMod)

\[
[\phi] \Gamma \vdash_\phi \text{mod } e' : [[\tau]]^C_\rho \mod
\]

\[
[\phi] \Gamma \vdash_\phi \text{mod } e' : [[\tau]]^C_\rho
\]

By Lemma A.3

Subcase “C⁺⁻C⁺” where \( \phi = C_p \) and \( \phi \delta = C_p \).

Part (2):

\[
(\Gamma, \tau' : (\tau_1 \to \tau)^\rho)(x') = \forall \delta \text{true}. (\tau_1 \to \tau)^\rho
\]

By defn. of \( \Gamma \)

\[
C \vdash \exists \delta \text{true}
\]

By defn. of \( \vdash \)

\[
[\phi] \Gamma, \tau' \vdash (\delta | \tau_1 \to \delta | \tau)^\rho \leftrightarrow x' : (\tau_1 \to \tau)^\rho
\]

By (SVar)

\[
[\phi] \Gamma, \tau' \vdash (\delta | \tau_1 \to \delta | \tau)^\rho \leftrightarrow x' : (\delta | \tau_1 \to \delta | \tau)^\rho
\]

By (Var)

\[
[\phi] \Gamma, \tau' \vdash (\delta | \tau_1 \to \delta | \tau)^\rho \leftrightarrow x_1 : (\delta | \tau_1 \to \delta | \tau)^\rho
\]

By extending \( \Gamma \)

\[
[\phi] \Gamma, \tau' \vdash (\delta | \tau_1 \to \delta | \tau)^\rho \leftrightarrow \text{apply}(x', x_2) : [\phi] \rho \leftrightarrow \text{apply}(x', x_2)
\]

By (App)

\[
[\phi] \Gamma, \tau' \vdash (\text{apply}(x', x_2)) : [\phi] \rho \leftrightarrow \text{apply}(x', x_2)
\]

By defn. of substitution and \( \rho \)

\[
[\phi] \Gamma \vdash e \leftrightarrow (x_1 : \phi \tau' \vdash \text{apply}(x', x_2))
\]

By (LApply)

\[
(\delta | \tau_1 \to \delta \rho \leftrightarrow \text{apply}(x', x_2))
\]

By defn. of \( \vdash \)

\[
C \vdash \exists \delta \text{true}
\]

Subderivation

\[
[\phi] \Gamma, \tau_1 \vdash (\delta | \tau_1 \to \delta \rho \leftrightarrow \text{apply}(x', x_2))
\]

By i.h.

\[
[\phi] \Gamma, \tau_1 \vdash (\delta | \tau_1 \to \delta \rho \leftrightarrow \text{apply}(x', x_2))
\]

\[
[\phi] \Gamma \vdash e \leftrightarrow \text{read } x_1 \text{ as } x' \text{ in } \text{apply}(x', x_2)
\]

By (Read)
Proof of Theorem A

Appendix, ICFP 2014

Let \( e' \) be read \( x_1 \) as \( x' \) in \( \text{apply}^C(x', x_2) \)

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \llbracket (x', x_2) \rrbracket^C \quad \text{by (TPVar)} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \llbracket x_1 \rrbracket^C \quad \text{by extending } \Gamma \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \text{apply}^C(x', x_2) : \llbracket \Gamma \rrbracket^C \quad \text{by (TApp)} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \llbracket x_1 \rrbracket^C \quad \text{mod} \quad \text{Above} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \text{read}_x as x' in apply^C(x', x_2) : \llbracket \Gamma \rrbracket^C \quad \text{by (TRead) (*)} \]

Part (1):

\[ C \vdash \delta \triangleleft \tau \quad \text{Premise} \]

\[ [\delta] \Gamma \vdash e : [\delta] \Gamma \triangleleft e' \quad \text{By (Mod)} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow e' : \llbracket \Gamma \rrbracket^C \quad \text{above (**)} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \text{mod} e' : \llbracket \Gamma \rrbracket^C \quad \text{By reasoning in subcase } C_p \text{-} S \text{ for Part (1); note that } [\delta] \Gamma \text{ O.C.} \]

**Subcase “-Cp,”** where \( [\delta] = S \) and \( [\delta] C_p = C_p^p \):

Part (2):

\[ [\delta] \Gamma, x' : ([\delta] \tau_1) \rightarrow ([\delta] \tau_2) \rightarrow \text{apply}(x', x_2) : [\delta] \tau \triangleleft e'_0 \quad \text{Above and substitute } x_1 \text{ with } x' \]

\[ [\delta] \Gamma, x' : ([\delta] \tau_1) \rightarrow x'/x_1 e : [\delta] \tau \triangleleft e'_0 \quad \text{By defn. of } [\delta] \text{ and substitution} \]

\[ [\delta] \Gamma, x_1 : ([\delta] \tau_1) \rightarrow [\delta] \tau \triangleleft x_1 \quad \text{By i.h.} \]

\[ [\delta] \Gamma \vdash e : [\delta] \rightarrow \text{read}_x as x' in e'_0 \quad \text{by (Read)} \]

Let \( e' = \text{read}_x \text{ as } x' \text{ in } e'_0 \),

\[ \vdash \llbracket \Gamma \rrbracket_x : e'_0 : \llbracket \Gamma \rrbracket^C \quad \text{Above and substitute } x_1 \text{ with } x' \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \llbracket x_1 \rrbracket^C \quad \text{by extending } \Gamma \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \llbracket \Gamma \rrbracket_x \rightarrow \text{mod} e'_0 : \llbracket \Gamma \rrbracket^C \quad \text{by i.h.} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \text{read}_x as x' in e'_0 : \llbracket \Gamma \rrbracket^C \quad \text{by (TRead)} \]

Part (1): Similar to Part (1) of the subcase for \( C_p \).

**Case**

\[ C ; P ; \Gamma \vdash x_1 : \text{int}^d_1 \]

\[ C ; P ; \Gamma \vdash x_2 : \text{int}^d_2 \]

\[ C ; P ; \Gamma \vdash \oplus : \text{int} \times \text{int} \rightarrow \text{int} \quad \text{(SPrim)} \]

If \( [\delta]_1 = [\delta]_2 = S \) then:

\[ C ; \Gamma \vdash x_1 : \text{int}^d_1 \quad \text{Subderivation} \]

\[ [\delta] \Gamma \vdash x_1 : \text{int}^d_1 \triangleleft x_1 \quad \text{By i.h.} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : x_1 \triangleleft x_1 \quad \text{"} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : x_1 \triangleleft \text{int} \quad \text{By def. of } \llbracket - \rrbracket \]

\[ [\delta] \Gamma \vdash x_2 : \text{int}^d_2 \triangleleft x_2 \quad \text{Similar to above} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : x_2 \triangleleft \text{int} \quad \text{Similar to above} \]

\[ [\delta] \Gamma \vdash e : \text{int}^d \triangleleft \oplus(x_1, x_2) \quad \text{by (Prim)} \]

\[ [\delta] \Gamma \vdash e : \text{int}^d \triangleleft \oplus(x_1, x_2) \quad \text{by (Prim)} \]

Let \( e' = \oplus(x_1, x_2) \),

\[ \vdash \oplus : \text{int} \rightarrow \text{int} \quad \text{Premise} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \oplus(x_1, x_2) \triangleleft \text{int} \quad \text{by (TPrim)} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \oplus(x_1, x_2) : \llbracket \text{int}^d \rrbracket_x \quad \text{by def. of } \llbracket - \rrbracket \]

Part (2): Similar to (SPair); note that \( \tau \text{ O.S. holds.} \)

If \([\delta]_1 = [\delta]_2 = C_p^p \) then:

Part (2):

\[ [\delta] \Gamma, y_1 : \text{int}^d_1, y_2 : \text{int}^d \rightarrow \oplus(y_1, y_2) : \text{int}^d \triangleleft \oplus(y_1, y_2) \quad \text{by (Var), (Var), (Prim)} \]

\[ [\delta] \Gamma, y_1 : \text{int}^d_1, y_2 : \text{int}^d \rightarrow \oplus(y_1, y_2) : \text{int}^d \rightarrow \text{let } r = \oplus(y_1, y_2) \text{ in write}(r) \quad \text{by (Write)} \]

\[ [\delta] \Gamma, y_1 : \text{int}^d_1, y_2 : \text{int}^d \rightarrow \oplus(y_1, y_2) : \text{int}^d \rightarrow \text{read} \text{ as } y_2 \text{ in let } r = \oplus(y_1, y_2) \text{ in write}(r) \quad \text{by (LPrimop2) then (Read)} \]

\[ [\delta] \Gamma \vdash \oplus(y_1, y_2) : \text{int}^d \rightarrow \text{read} \text{ as } y_1 \text{ in read } \text{ as } y_2 \text{ as } y_2 \text{ in write}(r) \quad \text{by (LPrimop1) then (Read)} \]

\[ \vdash \llbracket \Gamma \rrbracket_x : \text{read} \text{ as } y_1 \text{ in read } \text{ as } y_2 \text{ as } y_2 \text{ in write}(r) \quad \text{by (LPrimop1) then (Read)} \]
\[ \text{Appendix, ICFP 2014} \]

\[ \begin{array}{ll}
\text{By (TVar)} & \text{then (TWrite)} \\
\text{(TVar) and (TVar), then (TPrim)} \\
\text{(TLet)} \\
\text{(TRead)} \\
\text{(TRead)} \\
\text{By def. of } \| \cdot \| \text{ and } [\phi] \delta_1 = C_p
\end{array} \]

Part (1): As the immediately preceding Part (2), but then using rule (Mod).

\[ \begin{array}{ll}
\text{Case} \\
\text{(SCase)} & \text{By i.h.}
\end{array} \]

(a) Suppose \([\phi] \delta = \mathbb{S}\).

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{Subderivation} & \text{By i.h.}
\end{array} \]

\[ \begin{array}{ll}
\text{By i.h.} & \text{Subderivation}
\end{array} \]

\[ \begin{array}{ll}
\text{By (Case)} & \text{By (Case)}
\end{array} \]

\[ \begin{array}{ll}
\text{By (TCase)} & \text{By (TCase)}
\end{array} \]

(b) Suppose \([\phi] \delta = C_p\).

\[ \begin{array}{ll}
\text{By (LCase)} & \text{By (Case)}
\end{array} \]

\[ \begin{array}{ll}
\text{By (Read)} & \text{By (TCase)}
\end{array} \]

\[ \begin{array}{ll}
\text{By (TCase)} & \text{By (Case)}
\end{array} \]

\[ \begin{array}{ll}
\text{By Lemma A.3} & \text{By above equation}
\end{array} \]