Implicit Self-Adjusting Computation for Purely Functional Programs

Thesis approved by
the Department of Computer Science of the University of Kaiserslautern
for the award of Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Yan Chen

Date of Defense:	21 September 2017
Dean:	Prof. Dr. Klaus Schneider
Reviewer:	Prof. Dr. Umut A. Acar
Reviewer:	Prof. Dr. Arnd Poetzsch-Heffter

D 386

Abstract

Computational problems that involve dynamic data, such as physics simulations and program development environments, have been an important subject of study in programming languages. Recent advances in self-adjusting computation made progress towards achieving efficient incremental computation by providing algorithmic language abstractions to express computations that respond automatically to dynamic changes in their inputs. Self-adjusting programs have been shown to be efficient for a broad range of problems via an explicit programming style, where the programmer uses specific primitives to identify, create and operate on data that can change over time.

This dissertation presents *implicit self-adjusting computation*, a type directed technique for translating purely functional programs into self-adjusting programs. In this implicit approach, the programmer annotates the (top-level) input types of the programs to be translated. Type inference finds all other types, and a type-directed translation rewrites the source program into an explicitly self-adjusting target program. The type system is related to information-flow type systems and enjoys decidable type inference via constraint solving. We prove that the translation outputs well-typed self-adjusting programs and preserves the source program's input-output behavior, guaranteeing that translated programs respond correctly to all changes to their data. Using a cost semantics, we also prove that the translation preserves the asymptotic complexity of the source program.

As a second contribution, we present two techniques to facilitate the processing of large and dynamic data in self-adjusting computation. First, we present a type system for precise dependency tracking that minimizes the time and space for storing dependency metadata. The type system improves the scalability of self-adjusting computation by eliminating 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. Second, we show a probabilistic-chunking technique to further decrease space usage by controlling the fundamental space-time tradeoff in self-adjusting computation.

We implement implicit self-adjusting computation as an extension to Standard ML with compiler and runtime support. Using the compiler, we are able to incrementalize an interesting set of applications, including standard list and matrix benchmarks, ray tracer, PageRank, sparse graph connectivity, and social circle counts. Our experiments show that our compiler incrementalizes existing code with only trivial amounts of annotation, and the resulting programs bring asymptotic improvements to large datasets from real-world applications, leading to orders of magnitude speedups in practice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis

Dynamic changes are pervasive in computational problems: physics simulations often involve moving objects; robots interact with dynamic environments; compilers must respond to slight modifications in their input programs. Such dynamic changes are often small, or *incremental*, and result in only slightly different output, so computations can often respond to them asymptotically faster than performing a complete re-computation. Such asymptotic improvements can lead to massive speedup in practice but traditionally require careful algorithm design and analysis (Chiang and Tamassia 1992; Guibas 2004; Demetrescu et al. 2005), which can be challenging even for seemingly simple problems.

Motivated by this problem, researchers have developed language-based techniques that enable computations to respond to dynamic data changes automatically and efficiently (see Ramalingam and Reps (1993) for a survey). This line of research, traditionally known as *incremental computation*, aims to reduce dynamic problems to static (conventional or batch) problems by developing compilers that automatically generate code for dynamic responses. This is challenging, because the compiler-generated code aims to handle changes asymptotically faster than the source code. Early proposals (Demers et al. 1981; Pugh and Teitelbaum 1989; Field and Teitelbaum 1990) were limited to certain classes of applications (e.g., attribute grammars), allowed limited forms of data changes, and/or yielded suboptimal efficiency. Some of these approaches, however, had the important advantage of being *implicit*: they required little or no change to the program code to support dynamic change—conventional programs could be compiled to executables that respond automatically to dynamic changes.

Recent work based on *self-adjusting computation* made progress towards achieving efficient incremental computation by providing algorithmic language abstractions to express computations that respond automatically to changes to their data (Acar 2005; Acar et al. 2006c, 2009a). Self-adjusting computation can deliver asymptotically efficient updates in a reasonably broad range of problem domains, including dynamic trees (Acar et al. 2004, 2005), kinetic motion simulation (Acar et al. 2006d, 2008b), dynamic computational geometry (Acar et al. 2007a, 2010b, 2011; Türkoğlu 2012; Acar et al. 2013b), machine learning (Acar et al. 2007c, 2008c, 2009b; Sümer et al. 2011; Sümer 2012), and big-data systems (Bhatotia et al. 2011a,b, 2014; Bhatotia 2015; Bhatotia et al. 2015).

Existing self-adjusting computation techniques, however, require the programmer to program *explicitly* by using a certain set of primitives (Acar 2005; Acar et al. 2006c; Ley-Wild et al. 2008; Acar et al. 2009a). Specifically the programmer must manually distinguish *stable data*, which remains the same, from *changeable data*, which can change over time, and operate on changeable data via a special set of primitives.

This dissertation builds upon the preceding foundations to present techniques for *implicit* self-adjusting computation that allow conventional programs to be translated automatically into efficient self-adjusting programs.

> **Thesis Statement.** Implicit self-adjusting computation improves the experience of designing and implementing incremental programs. Combining type theory, language design and empirical work, implicit self-adjusting computation provides a model of computation that is general-purpose, transparent, sound and efficient.

We substantiate this claim with the following work.

Type systems. We show that an information-flow type system (Denning and Denning 1977; Pottier and Simonet 2003; Sabelfeld and Myers 2003) enables self-adjusting computation via tracking data dependencies (of sensitive data and changeable data, respectively) as well as dependencies between expressions and data. Specifically, we show that a type system that encodes the changeability of data and expressions in self-adjusting computation as secrecy of information suffices to statically enforce the invariants needed by self-adjusting computation. The type system uses polymorphism to capture stable and changeable uses of the same data or expression. We present a constraint-based formulation of our type system where the constraints are a strict subset of those needed by traditional information-flow systems. Consequently, as with traditional information flow, our type system admits an HM(X) inference algorithm (Odersky et al. 1999) that can infer all type annotations from top-level type specifications on the input of a program.

We also define a refinement to the type system that allows changeable data to be nested inside stable data. The refined type system eliminates the spurious redundancies caused by the modal type system, enabling *p*recise dependency tracking of self-adjusting programs, which significantly reduces the runtime overhead both in time and space.

Translation. We show that based on types, we can translate conventional purely functional programs into self-adjusting programs. Types provide crucial information that enables transformation. First, we present a set of compositional, non-deterministic translation rules. Guided by the types, these rules identify the set of all changeable expressions that operate on changeable data and rewrite them into the self-adjusting target language. We then present a deterministic translation algorithm that applies the compositional rules judiciously, considering the type and context (enclosing expressions) of each translated subexpression, to generate a well-typed self-adjusting target program.

Taken together, the type system, its inference algorithm, and the translation algorithm enable translating purely functional source programs to self-adjusting target programs using top-level type annotations on the input type of the source program. These top-level type annotations simply mark what part of the input data is subject to change. Type inference assigns types to the rest of the program and the translation algorithm translates the program into self-adjusting target code.

$$\begin{tikzcd}
\text{Level ML} & e \arrow[r, "\text{Type inference}"] & e : \tau \arrow[r, "\text{Evaluation}", "\text{in k steps}"'] \arrow[d, hook, shift right, "\text{Type-Directed Translation}"'] \arrow[d, equal, shift left, "\text{Type Soundness}"] & v \arrow[d, equal, "\text{Observational Equivalence}"] \\
\text{AFL} & & e' : \tau' \arrow[r, "\text{Evaluation}", "\text{in } \Theta(\text{k}) \text{ steps}"'] & w
\end{tikzcd}$$

Figure 1.1: Visualizing the translation between the source language Level ML and the target language AFL, and related properties.

Theoretical results. Figure 1.1 illustrates how source programs written in Level ML, a purely functional subset of ML with *level types*, can be translated to self-adjusting programs in the target language AFL, a language for self-adjusting computation with explicit primitives (Acar et al. 2006c). We prove three critical properties of the approach.

- **Type soundness.** On source code of a given type, the translation algorithm produces well-typed self-adjusting code of a corresponding target type (Theorem 6.4.1, Theorem 6.7.1).
- **Observational equivalence.** The translated self-adjusting program, when evaluated, produces the same value as the source program (Theorem 6.5.4).
- **Asymptotic complexity.** The time to evaluate the translated program is asymptotically the same as the time to evaluate the source program (Theorem 6.6.9).

Type soundness and observational equivalence together imply a critical consistency property: that self-adjusting programs respond correctly to changing data (via the consistency of the target self-adjusting language (Acar et al. 2006c)). The third property shows that the translated program takes asymptotically as long to evaluate (from scratch) as the corresponding source program. The time for incremental updates via change propagation is usually asymptotically more efficient than running from scratch. We also present experimental evidence that the target programs respond to dynamic changes efficiently.

Compiler and runtime design. We have implemented our approach as an extension of Standard ML with the HaMLet and MLton compiler (HaMLet; MLton). The implementation takes SML code annotated with level types at the top-level, conducts type inference that assigns level types to the rest of the programs, and generates self-adjusting code by inserting the self-adjusting primitives via type-directed, local rewrites. Such local rewrites, however, can lead to globally suboptimal code by inserting redundant calls to self-adjusting primitives. We therefore formulate a global rewriting system for eliminating such redundant code, and prove that the rewriting rules are terminating and confluent.

We implement the runtime system for self-adjusting primitives directly in SML. To facilitate the processing of large and dynamic data in self-adjusting computation, we add an important facility to the runtime library—the ability to control the granularity of dependency tracking by selectively tracking dependencies—that 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. To avoid disproportionately degrading the update performance, we present a *p*robabilistic chunking scheme. This technique divides the data into blocks in a probabilistic way, ensuring that small changes affect a small number of blocks.

Applications. We evaluate our implementation by considering a wide range of benchmarks including various primitives on lists, sorting functions, vector operations, matrix operations, ray tracer, PageRank, sparse graph connectivity, and approximate social circle counting. For each of these, we only need to insert some keywords into the program to specify the desired behavior. Specifically, most benchmarks require trivial decorations, often amounting to inserting type qualifiers in one or two lines of code. No changes to the structure of the types, or any part of the code itself, are necessary. Our techniques for controlling the space-time tradeoff for list data structures can reduce memory consumption effectively while only proportionally slowing down updates. The executables generated by the compilers respond automatically and 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.

1.2 Chapter outline

In Chapter 2, we motivate the need for an implicit style for self-adjusting computation. We consider previous language proposals for self-adjusting computation and the need for compilation support. We give a high-level description of implicit self-adjusting computation, including its level types and translation. Finally, we describe two techniques for scaling self-adjusting computation to large and dynamic data: precise dependency tracking and probabilistic chunking scheme.

In Chapter 3, we present level types for self-adjusting computation, and how it relates to information flow. For precise dependency tracking, we introduce labeled level types that can identify precisely which part of the data is changeable via labels.

In Chapter 4, we present our surface language that extends λ-calculus with level types, its static and dynamic semantics, and the constraint based type inference. We present the semantics for both level types and labeled level types, specifying the type systems track the dependencies of the surface language.

In Chapter 5, we present two self-adjusting target languages: the monadic language as the target language for level types; and the imperative language as the target language for labeled level types that enable precise dependency tracking. We present their perspective static and dynamic semantics.

In Chapter 6, we give a type-directed translation from source language to target language for both the monadic language and the imperative language. We proved the translation soundness and the cost preservation of translation.

In Chapter 7, we discuss our Standard ML compiler as an implementation of implicit self-adjusting computation, including the design for syntax extension, type inference, translation, optimization, runtime library, and memoization. Finally, we present probabilistic chunking scheme as a library extension for controlling the granularity of self-adjusting computation.

In Chapter 8, we present experimental evaluation of our compiler with wide range of benchmarks, including list primitives, vector primitives, sorting, ray tracer, MapReduce, PageRank, sparse graph connectivity, and approximate social circle counting.

In Chapter 9, we conclude with related and future work.

Chapter 2

Overview

We present an informal overview of our approach via examples in an extension of SML with features for implicit self-adjusting computation. We start with a brief description of our target language, explicit self-adjusting computation, as laid out in previous work. After this description, we outline our proposed approach.

2.1 Explicit self-adjusting computation

2.1.1 Core language

The key concept behind explicit approaches is the notion of a *modifiable (reference)*, which stores *changeable* values that can change over time (Acar et al. 2006c). The programmer operates on modifiables with **mod**, **read**, and **write** constructs to create, read from, and write into modifiables. The run-time system of a self-adjusting language uses these constructs to represent the execution as a (directed, acyclic) dependency graph, enabling efficient *change propagation* when the data changes in small amounts. There is a modal type system that enforces an important correctness property: any computation that depends on a modifiable itself must be written in a modifiable.

As an example, consider a trivial program that computes $x^2 + y$:

```
squareplus: int * int → int
fun squareplus (x, y) =
  let x2 = x * x in
    let r = x2 + y in
      r
```

To make this program self-adjusting with respect to changes in y, while leaving x unchanging or *stable*, we assign y the type int **mod** (of modifiables containing integers) and **read** the contents of the modifiable. The body of the **read** is a *changeable expression* ending with a **write**. This function has a changeable arrow type $\underset{\mathbb{C}}{\rightarrow}$:

```
squareplus_SC: int * int mod $\underset{\mathbb{C}}{\rightarrow}$ int
fun squareplus_SC (x, y) =
  let x2 = x * x in
    read y as y' in
      let r = x2 + y' in
        write(r)
```

The read operation delimits the code that can directly inspect the changeable value y, and the changeable arrow type ensures an important consistency property: $\underset{\mathbb{C}}{\rightarrow}$-functions can only be called within the context of a changeable expression. If we change the value of y, change propagation can update the result, re-executing only the **read** and its body, and thus reusing the computation of the square x2.

Note that the result type of squareplus_SC is int, not int **mod**; squareplus_SC does not itself create a modifiable, it just writes to the modifiable created by the caller of the function in the context of a (dynamically) enclosing **mod** expression.

Now suppose we wish to make x changeable while leaving y stable. We can read x and place x2 into a modifiable (because we can only read within the context of a changeable expression), and immediately read back x2 and finish by writing the sum.[1]

```
squareplus_CS: int mod * int $\underset{\mathbb{C}}{\rightarrow}$ int
fun squareplus_CS (x, y) =
  let x2 = mod (read x as x' in write(x' * x')) in
    read x2 as x2' in
      let r = x2' + y in
        write(r)
```

As this example shows, rewriting even a trivial program can require modifications to the code, and different choices about what is or is not changeable lead to different code. Moreover, if we need squareplus_SC *and* squareplus_CS—for instance, if we want to pass squareplus to various higher-order functions—we must write, and maintain, both versions. If we conservatively treat all data as modifiable, we would only need to write one version of each function, but this would introduce unacceptably high overhead. It is also possible to take the other extreme and treat all data as stable, but this would yield a non-self-adjusting program. Our approach treats data as modifiable only where necessary.

2.1.2 Meta language

The run-time system of a self-adjusting language also supplies *meta operations*: **change** for inspecting and changing the values stored in modifiables and **propagate** for perform-

[1] This is not the only way to express the computation. For instance, one could bind x' * x' to x2' and do the addition within the body of **read** x. The code shown here is the same as the code produced by our translation, and has the property that the scope of each read is as small as possible, which leads to more efficient updates during change propagation.

ing change propagation. The **change** function is similar to the **write** construct: it assigns a new value to the modifiable to a new value. The **propagate** function runs the change-propagation algorithm, which updates a computation based on the changes made since the last execution or the last change propagation. The meta operations can only be used at the top level—the run-time system guarantees correct behavior only if meta operations are not used inside the core self-adjusting program. Interested readers can refer to Acar et al. (2006a) for a more detailed discussion of the meta operations, and the change propagation algorithms used in self-adjusting computation.

As an example, consider calling the `squareplus_SC` function in a Standard ML implementation of self-adjusting runtime:

```
let
  val x = 1
  val y = mod 2
  val z = mod (squareplus_SC (x, y))
  val () = change (y, 3)
  val () = propagate ()
in () end
```

When calling the `squareplus_SC` function, `z` will be a modifiable containing 3. The **change** function updates modifiable `y` to be 3. The **propagate** function triggers reevaluation of the plus operation (while the square computation is reused), and stores the result 4 into modifiable `z`.

Implicit self-adjusting computation, described below, is an alternative approach for writing the self-adjusting computation itself; the interface to the meta operations remains the same.

2.2 Implicit self-adjusting computation

To make self-adjusting computation implicit, we use type information to insert **read**s, **write**s, and **mod**s automatically. The user annotates the input type, as well as the corresponding data declarations, of the program; we infer types for all expressions, and use this information to guide a translation algorithm. The translation algorithm returns well-typed self-adjusting target programs. The translation requires no expression-level annotations. For the example function `squareplus` above, we can automatically derive `squareplus_SC` and `squareplus_CS` from just the type of the function (expressed in a slightly different form, as we discuss next).

2.2.1 Level types

To uniformly describe source functions (more generally, expressions) that differ only in their "changeability", we need a more general type system than that of the target language. This type system refines types with *levels* $\mathbb{S}$ (stable) and $\mathbb{C}$ (changeable). The type $\mathbf{int}^{\delta}$ is an integer whose level is δ; for example, to get `squaresum_CS` we can annotate `squaresum`'s argument with the type $\mathbf{int}^{\mathbb{C}} \times \mathbf{int}^{\mathbb{S}}$.

Level types are an important connection between information-flow types (Denning and Denning 1977; Pottier and Simonet 2003) and those needed for our translation: high-security secret data (level H) behaves like changeable data (level $\mathbb{C}$), and low-security public data (level L) behaves like stable data (level $\mathbb{S}$). In information flow, data that depends on secret data must be secret; in self-adjusting computation, data that depends on changeable data must be changeable. Building on this connection, we develop a type system with several features and mechanisms similar to information flow. Among these is level polymorphism; our type system assigns level-polymorphic types to expressions that accommodate various "changeabilities". (As with ML's polymorphism over types, our level polymorphism is prenex.) Another similarity is evident in our constraint-based type inference system, where the constraints are a strict subset of those in Pottier and Simonet (2003). As a corollary, our system admits a constraint-based type inference algorithm (Odersky et al. 1999).

2.2.2 Translation

The main purpose of our type system is to support translation. Given a source expression and its type, translation inserts the appropriate `mod`, `read`, and `write` primitives and restructures the code to produce an expression that is well-typed in the target language.

The implicitly self-adjusting source language is polymorphic over levels. The type system of the target language, which is explicitly self-adjusting, is also polymorphic but *explicitly* so: polymorphic values are given as lists of values (within a **select** construct), with each value in the list being the translation of the source value at specific levels. Moreover, polymorphic values are explicitly instantiated by a syntactic construct in the target language; in the source language, instantiation is implicit.

Our translation generates code that is well-typed, has the same input-output behavior as the source program, and is, at worst, a constant factor slower than the source program. Since the source and target languages differ, proving these properties is nontrivial; in fact, the proofs critically guided our formulation of the type system and translation algorithm.

A more detailed example: `mapPair`**.** To illustrate how our translation works, consider a function `mapPair` that takes two integer lists and increments the elements in both lists. This function can be written by applying the standard higher-order `map` over lists. Figure 2.1 shows the purely functional code in an ML-like language for an implementation of `mapPair`, with a datatype α `list`, an increment function `inc`, and a polymorphic `map` function. Type signatures give the types of functions.

To obtain a self-adjusting `mapPair`, we first decide how we wish to allow the input to change. Suppose that we want to allow insertion and deletion of elements in the first list, but we expect the length of the second list to remain constant, with only its elements changing. We can express this with the versions of the list type with different

```
datatype α list = nil | cons of α * α list

inc : int → int
fun inc (x) = x+1

map : (α → β) → α list → β list
fun map f l =
  case l of
    nil ⇒ nil
  | cons(h,t) ⇒ cons(f h, map f t)

mapPair : (int list * int list) → (int list * int list)
fun mapPair (l1,l2) = (map inc l1, map inc l2)
```

Figure 2.1: Function `mapPair` in ML.

$$\begin{array}{l}
\textbf{datatype}\ \alpha\ \texttt{list}^{\delta}\ \texttt{= nil | cons}\ \textbf{of}\ \alpha * (\alpha\ \texttt{list}^{\delta}) \\
\\
\texttt{mapPair : ((int}^{\mathbb{S}}\ \texttt{list}^{\mathbb{C}}\texttt{)} * \texttt{(int}^{\mathbb{C}}\ \texttt{list}^{\mathbb{S}}\texttt{))} \\
\qquad\qquad \xrightarrow[\mathbb{S}]{}\ \texttt{((int}^{\mathbb{S}}\ \texttt{list}^{\mathbb{C}}\texttt{)} * \texttt{(int}^{\mathbb{C}}\ \texttt{list}^{\mathbb{S}}\texttt{))} \\
\\
\texttt{...}\quad \textit{(* inc, map, mapPair same as in Figure 2.1. *)}
\end{array}$$

Figure 2.2: Function `mapPair` in Level ML, with level types.

changeability:

- $\alpha\ \texttt{list}^{\mathbb{C}}$ for lists of α with changeable tails;
- $\alpha\ \texttt{list}^{\mathbb{S}}$ for lists of α with stable tails.

Then a list of integers allowing insertion and deletion has type $\mathbf{int}^{\mathbb{S}}\ \texttt{list}^{\mathbb{C}}$, and one with unchanging length has type $\mathbf{int}^{\mathbb{C}}\ \texttt{list}^{\mathbb{S}}$. Now we can write the type annotation on `mapPair` shown in Figure 2.2. Given only that annotation, type inference can find appropriate types for `inc` and `map` and our translation algorithm generates self-adjusting code from these annotations. Note that to obtain a self-adjusting program, we only had to provide types for the function. We call this language with level types Level ML.

Target code for `mapPair`. Translating the code in Figure 2.2 produces the self-adjusting target code in Figure 2.3. Note that `inc` and `map` have *level-polymorphic* types. In `map inc l1` we increment stable integers, and in `map inc l2` we increment changeable integers, so the type inferred for `inc` must be generic: $\forall\delta.\ \mathbf{int}^{\delta} \xrightarrow[\delta]{} \mathbf{int}^{\delta}$. Our translation produces two implementations of `inc`, one per instantiation ($\delta{=}\mathbb{S}$ and $\delta{=}\mathbb{C}$): `inc_S` and `inc_C` (in Figure 2.3). Since we want to use `inc` with the higher-order function `map`, we need to generate a “selector” function that takes an instantiation and picks out the appropriate implementation:

$$\begin{array}{l} \texttt{inc : } \forall\delta.\ \texttt{int}^{\delta} \xrightarrow[\delta]{} \texttt{int}^{\delta} \\ \textbf{val}\ \texttt{inc = }\ \textbf{select}\ \{\delta{=}\mathbb{S} \Rightarrow \texttt{inc\_S} \\ \qquad\qquad\qquad\ \mid\ \delta{=}\mathbb{C} \Rightarrow \texttt{inc\_C}\} \end{array}$$

In `mapPair` itself, we pass a level instantiation to the selector: `inc`$[\delta{=}\mathbb{S}]$. (This instantiation is known statically, so it could be replaced with `inc_S` at compile time.) The types of `inc_S` and `inc_C` are produced by a type-level translation that, very roughly, replaces changeable types with **mod** types (Section 6.1).

Observe how the single annotation on `mapPair` led to duplication of the two functions it uses. While `inc_S` is the same as the original `inc`, the changeable version `inc_C` adds a **read** and a **write**. Note also that the two generated versions of `map` are both different from the original.

The interplay of type inference and translation. Given user annotations on the input, type inference finds a satisfying type assignment, which then guides our translation algorithm to produce self-adjusting code. In many cases, multiple type assignments could satisfy the annotations; for example, subsumption allows any stable type to be promoted to a changeable type. Translation yields target code that satisfies the crucial type soundness, operational equivalence, and complexity properties under any satisfying assignment. But some type assignments are preferable, especially when one considers constant factors. Choosing $\mathbb{C}$ levels whenever possible is always a viable strategy, but treating all data as changeable results in more overhead. As in information flow, where we want to consider data secret only when absolutely necessary, inference yields principal typings that are minimally changeable, always preferring $\mathbb{S}$ over $\mathbb{C}$.

```
datatype α list_S = nil | cons of α * α list_S
datatype α list_C = nil | cons of α * (α list_C) mod

inc_S : int →_𝕊 int      (* ‘inc’ specialized for stable data *)
fun^𝕊 inc_S (x) = x+1

inc_C : int mod →_ℂ int     (* ‘inc’ specialized for changeable data *)
fun^ℂ inc_C (x) = read x as x’ in write (x’+1)

inc : ∀δ. int^δ →_δ int^δ
val inc = select {δ=𝕊 ⇒ inc_S
                | δ=ℂ ⇒ inc_C}

map_SC : (α →_𝕊 β) →_𝕊 (α list_C) mod →_𝕊 (β list_C) mod
fun^𝕊 map_SC f l =     (* ‘map’ for stable heads, changeable tails *)
  mod (read l as x in
          case x of
            nil ⇒ write nil
          | cons(h,t) ⇒ write (cons(f h, map_SC f t)))

map_CS : (α →_ℂ β) →_𝕊 (α list_S) →_𝕊 ((β mod) list_S)
fun^𝕊 map_CS f l =     (* ‘map’ for changeable heads, stable tails *)
  case l of
    nil ⇒ nil
  | cons(h,t) ⇒ let val h’ = mod (f h)
                in cons(h’, map_CS f t)

map : ∀δ_H,δ_T. (α →_δH β) →_𝕊 α list^δT →_𝕊 β list^δT
val map = select {δ_H=𝕊, δ_T=ℂ ⇒ map_SC
                | δ_H=ℂ, δ_T=𝕊 ⇒ map_CS}

mapPair :    ((int list_C) mod * (int mod) list_S)
          →_𝕊 ((int list_C) mod * (int mod) list_S)
fun^𝕊 mapPair (l1, l2) = (map[δ_H=𝕊,δ_T=ℂ] inc[δ=𝕊] l1,
                          map[δ_H=ℂ,δ_T=𝕊] inc[δ=ℂ] l2)
```

Figure 2.3: Translated `mapPair` with **mod** types and explicit level polymorphism.

```
fun partition f l : (α list^ℂ * α list^ℂ) =
  case l of
    nil ⇒ (nil, nil)
  | h::t ⇒
     let val (a,b) = partition f t
     in if f h then (h::a, b)
        else (a, h::b)
     end
```

Figure 2.4: The list partition function in Level ML.

A combinatorial explosion? A type scheme quantifying over n level variables has up to 2^n instances. However, our experience suggests that n is usually small: level variables tend to recur in types, as in the type of `inc` above. Even if n turns out to be large for some practical applications, the number of *used* instantiations will surely be much less than 2^n, suggesting that generating instances lazily would suffice.

2.3 Scaling to large and dynamic data

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. Self-adjusting computation provides a natural abstraction for enabling programs to respond efficiently to dynamic data provided that the space usage remains relatively small. In this section, we provide the overview of two techniques for scaling self-adjusting computation to process large and dynamic data: precise dependency tracking and probabilistic chunking scheme. The first technique reduces time and 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.

2.3.1 Precise dependency tracking

Implicit self-adjusting computation relies on a modal type system to guarantee properties important to the correctness of change propagation—all changeables are initialized and all their dependencies are tracked. This type system can be conservative and can generate self-adjusting programs that contain redundant dependencies. Using a simple list-partitioning function, we outline the limitations of the type system, and describe how we resolve them to improve the time and space usage of self-adjusting computation.

```
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
```

Figure 2.5: The self-adjusting list partition function

List Partition in Level ML. Figure 2.4 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 List Partition. With the type annotation on the first line in Figure 2.4, the compiler derives a self-adjusting version of list partition in Figure 2.5. Given the self-adjusting 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.

To ensure the correctness, the type system conservatively disallows 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 (α list **mod** * α 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).

We can circumvent this problem by using unsafe, imperative operations. For our running example, `partition` can be rewritten as shown in Figure 2.6, 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 (α list **mod** * α list **mod**), as desired.

```
fun partition f l (l_00,l_01) : (α list mod * α list mod) =
 let val () = read l as l' in
    case l' of
      nil ⇒ (write (l_00,nil);
              write (l_01,nil))
    | h::t ⇒
      let val (a,b) = let
        val (l_00,l_01) = (mod nil, mod nil)
        in partition f t (l_00,l_01)
        end
      in if f h then (write (l_00, h::a);
         read b as b' in write (l_01, b'))
      else (read a as a' in write (l_00, a');
            write (l_01, h::b))
      end
 in (l_00,l_01) end
```

Figure 2.6: The self-adjusting list partition function with destination passing.

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 *labels* ρ into the changeable level annotations, denoted as $\mathbb{C}_\rho$. The label serves as an identifier for modifiables. For each function of type $\tau_1 \rightarrow \tau_2$, we give labels for the return type τ_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 in Figure 2.4. 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 6 of Figure 2.4 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 2.6 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.

2.3.2 Probabilistic chunking scheme

An important facility in processing large and dynamic data in self-adjusting computation is the ability to control the space-time tradeoff by controlling the granularity of dependency tracking. By tracking dependencies at the level of (large) blocks of data, rather than individual data items, the programmer can further reduce space consumption. In principle, there is a straightforward way: 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.

```
fun bpar L =
  case L
    of NIL ⇒ (NIL, NIL)
     | BLOCK(b, t) ⇒
     let val (p, q) = bpar t
            val (p’, q’) = partition b
     in (mkCons(p’, p), mkCons(q’, q)) end
```

Figure 2.7: List partition using a block sequence abstraction.

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. For instance, the list partition routine can be adapted with little changes to use a block list abstraction, as shown in Figure 2.7.

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. It may seem that restricting ourselves to in-place changes would resolve this issue but this is not the case because such changes 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 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.

We also provide a block sequence abstraction to help programmers write code that utilizes blocks. Block sequences can be embedded inside other data structures or nested to form more advanced data structures, for example, as a data structure for sparse graphs.

Chapter 3

Level Types

This chapter is based on work on the theoretical formulation of implicit self-adjusting computation (Chen et al. 2011, 2014b), and a type system extension for precise dependency tracking (Chen et al. 2014a).

3.1 From information flow types to SAC

Self-adjusting computation separates the computation and data into two parts: stable and changeable. Changeable data refers to data that can change over time; all non-changeable data is stable. Similarly, changeable expressions refers to expressions that operate (via elimination forms) on changeable data; all non-changeable expressions are stable. Evaluation of changeable expressions (that is, changeable computations) can change as the data that they operate on changes: changes in data cause changes in control flow. These distinctions are critical to effective self-adjustment: previous work on explicit self-adjusting computation (Acar 2005; Acar et al. 2006c; Ley-Wild et al. 2008; Acar et al. 2009a) shows that it suffices to track and remember changeable data and evaluations of changeable expressions because stable data and evaluations of stable expressions remain invariant over time. This previous work developed languages that enable the programmer to separate stable and changeable data, and type systems that enforce correct usage of these constructs.

In this section, we describe the self-adjusting computation types that we infer for purely functional programs. A key insight behind our approach is that in information-flow type systems, secret (high-security) data is infectious: any data that depends on secret data itself must be secret. This corresponds to self-adjusting computation: data that depends on changeable data must itself be changeable. In addition, self-adjusting computation requires expressions that inspect changeable data—elimination forms—to be changeable. To encode this invariant, we extend function types with a *mode*, which is either stable or changeable; only changeable functions can inspect changeable data. This additional structure preserves the spirit of information flow-based type systems, and, moreover, supports constraint-based type inference in a similar style.

The starting point for our formulation is Pottier and Simonet (2003). Types in Level

$$\begin{array}{lrcl} \textit{Levels} & \delta, \varepsilon & ::= & \mathbb{S} \mid \mathbb{C} \mid \alpha \\ \textit{Types} & \tau & ::= & \mathbf{int}^{\delta} \mid (\tau_1 \times \tau_2)^{\delta} \mid (\tau_1 + \tau_2)^{\delta} \mid (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta} \\ \textit{Constraints} & C, D & ::= & \mathbf{true} \mid \mathbf{false} \mid \exists \vec{\alpha}.C \mid C \wedge D \mid \\ & & & \alpha = \beta \mid \alpha \leq \beta \mid \delta \lhd \tau \\ \textit{Type schemes} & \sigma & ::= & \tau \mid \forall \vec{\alpha}[D].\,\tau \end{array}$$

Figure 3.1: Levels, constraints, types, and type schemes in Level ML.

ML (Figure 3.1) include two *(security) levels*, stable and changeable. We generally follow their approach and notation. The two key differences are that (1) since Level ML is purely functional, we need no "program counter" level "*pc*"; (2) we need a mode ε on function types.

Levels. The *levels* $\mathbb{S}$ (*stable*) and $\mathbb{C}$ (*changeable*) have a total order:

$$\frac{}{\mathbb{S} \leq \mathbb{S}} \qquad\qquad \frac{}{\mathbb{C} \leq \mathbb{C}} \qquad\qquad \frac{}{\mathbb{S} \leq \mathbb{C}}$$

To support polymorphism and enable type inference, we allow *level variables* α, β to appear in types.

Types. Types consist of integers tagged with their level, products[1] and sums with an associated level, and arrow (function) types. Function types $(\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta}$ carry two level annotations ε and δ. The *mode* ε is the level of the computation encapsulated by the function. This mode determines how a function can manipulate changeable values: a function in stable mode cannot directly manipulate changeable values; it can only pass them around. By contrast, a changeable-mode function can directly manipulate changeable values. The outer level δ is the level of the function itself, as a value. We say that a type is *ground* if it contains no level variables.

In practice, types in source programs can omit levels, which will be derived through type inference. For example, if the user writes `int`, the system will add a level variable δ and do type inference with $\mathbf{int}^{\delta}$.

Subtyping. Figure 3.2 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 and that the modes match in the case of functions: a stable-mode function is never a subtype or supertype of a changeable-mode function. (It would be sound to make stable-mode functions subtypes of changeable-mode functions, but changeable mode functions are more expensive; silent coercion would make performance less predictable.)

[1] In Pottier and Simonet (2003), product types are low-security (stable) because pairing adds no extra information. In our setting, changeable products give more control over the granularity of change propagation.

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

$$\frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2' \qquad \delta \le \delta'}{(\tau_1 + \tau_2)^{\delta} <: (\tau_1' + \tau_2')^{\delta'}}\ (\text{subSum})$$

$$\frac{\varepsilon = \varepsilon' \qquad \delta \le \delta' \qquad \tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{(\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta} <: (\tau_1' \underset{\varepsilon'}{\rightarrow} \tau_2')^{\delta'}}\ (\text{subArrow})$$

Figure 3.2: Subtyping.

$$\frac{\delta \le \delta'}{\delta \lhd \mathbf{int}^{\delta'}}\ (\lhd\text{-Int})$$

$$\frac{\delta \le \delta'}{\delta \lhd (\tau_1 \times \tau_2)^{\delta'}}\ (\lhd\text{-Prod}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta'}}\ (\lhd\text{-Arrow}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 + \tau_2)^{\delta'}}\ (\lhd\text{-Sum})$$

Figure 3.3: Lower bound of a type.

$$\frac{}{\mathbf{int}^{\mathbb{S}}\ \mathsf{O.S.}} \quad \frac{}{(\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{S}}\ \mathsf{O.S.}} \quad \frac{}{(\tau_1 \times \tau_2)^{\mathbb{S}}\ \mathsf{O.S.}} \quad \frac{}{(\tau_1 + \tau_2)^{\mathbb{S}}\ \mathsf{O.S.}}$$

$$\frac{}{\mathbf{int}^{\mathbb{C}}\ \mathsf{O.C.}} \quad \frac{}{(\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{C}}\ \mathsf{O.C.}} \quad \frac{}{(\tau_1 \times \tau_2)^{\mathbb{C}}\ \mathsf{O.C.}} \quad \frac{}{(\tau_1 + \tau_2)^{\mathbb{C}}\ \mathsf{O.C.}}$$

$$\begin{array}{rclcrcl}
\mathbf{int}^{\delta_1} & \circeq & \mathbf{int}^{\delta_2} & \qquad & (\tau_1 + \tau_2)^{\delta_1} & \circeq & (\tau_1 + \tau_2)^{\delta_2} \\
(\tau_1 \times \tau_2)^{\delta_1} & \circeq & (\tau_1 \times \tau_2)^{\delta_2} & \qquad & (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta_1} & \circeq & (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta_2}
\end{array}$$

Figure 3.4: Outer-stable and outer-changeable types, and equality up to outer levels.

For simplicity, our type system will support only a weaker form of subtyping where only the outer levels can differ. In practice, the more powerful subtyping system could be used; see the discussion of let-expressions in Section 4.2.

Levels and types. We rely on several relations between levels and types to ascertain various invariants. A type τ is *higher than* δ, written $\delta \lhd \tau$, if the outer level of the type is at least δ. Figure 3.3 defines this relation. We distinguish between outer-stable and outer-changeable types (Figure 3.4). We write τ O.S. if the outer level of τ is $\mathbb{S}$. Similarly, we write τ O.C. if the outer level of τ is $\mathbb{C}$. Finally, two types τ_1 and τ_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.

Constraints. To perform type inference, we extend levels with level variables α and β, and use a constraint solver to find solutions for the variables. Our constraints C, D include level-variable comparisons $\leq$ and level-type comparisons $\delta \lhd \tau$, which type inference composes into conjunctions of satisfiability predicates $\exists\vec{\alpha}.C$.

The subtyping and lower bound relations defined in Figures 3.2 and 3.3 consider closed types only. For type inference, we can extend these with a constraint to allow non-closed types.

A *(ground) assignment*, written ϕ, substitutes concrete levels $\mathbb{S}$ and $\mathbb{C}$ for level variables. An assignment ϕ satisfies a constraint C, written $\phi \vdash C$, if and only if C holds true after the substitution of variables to ground types as specified by ϕ. We say that C *entails* D, written $C \Vdash D$, if and only if every assignment ϕ that satisfies C also satisfies D. We write $\phi(\alpha)$ for the solution (instantiation) of α in ϕ, and $[\phi]\tau$ for the usual substitution operation on types. For example, if $\phi(\alpha) = \mathbb{S}$ then $[\phi]\left(\mathbf{int}^{\alpha} + \mathbf{int}^{\mathbb{C}}\right)^{\alpha} = \left(\mathbf{int}^{\mathbb{S}} + \mathbf{int}^{\mathbb{C}}\right)^{\mathbb{S}}$.

Type schemes. A *type scheme* σ is a type with universally quantified level variables: $\sigma = \forall\vec{\alpha}[D].\,\tau$. We say that the variables $\vec{\alpha}$ are bound by σ. The type scheme is bounded by the constraint D, which specifies the conditions that must hold on the variables. As usual, we consider type schemes equivalent under capture-avoiding renaming of their bound variables. Ground types can be written as type schemes, e.g. $\mathbf{int}^{\mathbb{C}}$ as $\forall\emptyset[\mathbf{true}].\,\mathbf{int}^{\mathbb{C}}$.

3.2 Labeled level 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 section 3.1 to track fine-grained dependencies in the surface language.

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.

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

Figure 3.5: Labeled levels, types and constraints

$$\frac{}{\mathbf{int}^{\mathbb{S}} \downarrow_\rho \emptyset; \emptyset}\ (\#\text{intS}) \qquad \frac{}{\mathbf{int}^{\mathbb{C}_\rho} \downarrow_\rho \{\mathbf{int}\}; \{l_\rho\}}\ (\#\text{intC}) \qquad \frac{}{(\tau_1 + \tau_2)^{\mathbb{S}} \downarrow_\rho \emptyset; \emptyset}\ (\#\text{sumS})$$

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

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

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

Figure 3.6: Labeling changeable types

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

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

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 4.2.2.

Types. Types consist of integers tagged with their levels, products, sums and arrow (function) types with an associated level, as shown in Figure 3.5. The label ρ 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 3.6, which recursively determines the labels with respect to a prefix ρ, 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

$$\frac{\delta \leq \delta'}{\delta \ \lhd \ (\tau_1 \times \tau_2)^{\delta'}} \ (\lhd\text{-Prod}) \qquad\qquad \frac{\delta \lhd \tau_1 \qquad \delta \lhd \tau_2}{\delta \ \lhd \ (\tau_1 \times \tau_2)^{\mathbb{S}}} \ (\lhd\text{-InnerProd})$$

Figure 3.7: Lower bound of a product type with labels

$$\begin{array}{rcl@{\qquad}rcl} \llbracket \mathbf{int}^{\delta} \rrbracket & = & \delta & \left\llbracket (\tau_1 + \tau_2)^{\delta} \right\rrbracket & = & \delta \\ \left\llbracket (\tau_1 \times \tau_2)^{\delta} \right\rrbracket & = & \delta & \left\llbracket (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\delta} \right\rrbracket & = & \delta \end{array}$$

Figure 3.8: Outer level of types

that the outer level label is ρ. 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, $\tau = \left(\mathbf{int}^{\mathbb{C}_{00}} \times \left(\mathbf{int}^{\mathbb{C}_{111}} + \mathbf{int}^{\mathbb{S}}\right)^{\mathbb{C}_{01}}\right)^{\mathbb{S}}$ 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 $\mathcal{L} = \{l_{00}, l_{01}\}$.

Levels and types. We need to extend the higher than relation for product type to accommodate the extension of labels. Specifically, for products with outer stable levels, we check if each component is higher than δ. Figure 3.7 defines this relation.

We define an outer-level operation $\llbracket \tau \rrbracket$ that derives the outer level of a type in Figure 3.8.

Chapter 4

Source Language

This chapter is based on work on the theoretical formulation of implicit self-adjusting computation (Chen et al. 2011, 2014b), and a type system extension for precise dependency tracking (Chen et al. 2014a).

4.1 Syntax

Figure 4.1 shows the syntax for our source language Level ML, a purely functional language with integers (as base types), products, and sums. The expressions consist of values (integers, pairs, tagged values, 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.

4.2 Static semantics

4.2.1 Constraint-based type system

Consider the types defined by the grammar

$$\tau \ ::= \ \mathbf{int} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2$$

We augment this type system with features that allow us to directly translate Level ML programs into self-adjusting programs in AFL. This constraint-based type system has the

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

Figure 4.1: Abstract syntax of the source language Level ML.

$\boxed{C; \Gamma \vdash_{\varepsilon} e : \tau}$ Under constraint C and source typing environment Γ, source expression e has type τ

$$\frac{}{C; \Gamma \vdash_{\varepsilon} n : \mathbf{int}^{\mathbb{S}}}\ \text{(SInt)} \qquad \frac{\Gamma(x) = \forall\vec{\alpha}[D].\,\tau \quad C \Vdash \exists\vec{\beta}.[\vec{\beta}/\vec{\alpha}]D}{C \wedge [\vec{\beta}/\vec{\alpha}]D; \Gamma \vdash_{\varepsilon} x : [\vec{\beta}/\vec{\alpha}]\tau}\ \text{(SVar)}$$

$$\frac{C; \Gamma \vdash_{\varepsilon} v_1 : \tau_1 \qquad C; \Gamma \vdash_{\varepsilon} v_2 : \tau_2}{C; \Gamma \vdash_{\varepsilon} (v_1, v_2) : (\tau_1 \times \tau_2)^{\mathbb{S}}}\ \text{(SPair)} \qquad \frac{C; \Gamma \vdash_{\varepsilon} v : \tau_1}{C; \Gamma \vdash_{\varepsilon} \mathbf{inl}\ v : (\tau_1 + \tau_2)^{\mathbb{S}}}\ \text{(SSumLeft)}$$

$$\frac{C; \Gamma, x : \tau_1, f : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}} \vdash_{\varepsilon} e : \tau_2 \qquad C \Vdash \varepsilon \lhd \tau_2}{C; \Gamma \vdash_{\varepsilon'} (\mathbf{fun}\ f(x) = e) : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}}\ \text{(SFun)}$$

$$\frac{\begin{array}{ll} C; \Gamma \vdash_{\mathbb{S}} x_1 : \mathbf{int}^{\delta_1} & C \Vdash \delta_1 = \delta_2 \\ C; \Gamma \vdash_{\mathbb{S}} x_2 : \mathbf{int}^{\delta_2} & C \Vdash \delta_1 \leq \varepsilon \quad \oplus : \mathbf{int} \times \mathbf{int} \rightarrow \mathbf{int} \end{array}}{C; \Gamma \vdash_{\varepsilon} \oplus(x_1, x_2) : \mathbf{int}^{\delta_1}}\ \text{(SPrim)}$$

$$\frac{C; \Gamma \vdash_{\mathbb{S}} x : (\tau_1 \times \tau_2)^{\delta} \qquad C \Vdash \delta \leq \varepsilon}{C; \Gamma \vdash_{\varepsilon} \mathbf{fst}\ x : \tau_1}\ \text{(SFst)} \qquad \frac{\begin{array}{ll} C; \Gamma \vdash_{\varepsilon'} e_1 : \tau' & C; \Gamma, x : \tau'' \vdash_{\varepsilon} e_2 : \tau \\ C \Vdash \tau' <: \tau'' & C \Vdash \tau' \circeq \tau'' \end{array}}{C; \Gamma \vdash_{\varepsilon} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}\ \text{(SLetE)}$$

$$\frac{\begin{array}{c} C \wedge D; \Gamma \vdash_{\mathbb{S}} v_1 : \tau' \qquad C; \Gamma, x : \forall\vec{\alpha}[D].\,\tau'' \vdash_{\varepsilon} e_2 : \tau \\ \vec{\alpha} \cap FV(C, \Gamma) = \emptyset \qquad C \Vdash \tau' <: \tau'' \qquad C \Vdash \tau' \circeq \tau'' \end{array}}{C \wedge \exists\vec{\alpha}.D; \Gamma \vdash_{\varepsilon} \mathbf{let}\ x = v_1\ \mathbf{in}\ e_2 : \tau}\ \text{(SLetV)}$$

$$\frac{\begin{array}{ll} C; \Gamma \vdash_{\mathbb{S}} x_1 : (\tau_1 \xrightarrow[\varepsilon']{} \tau_2)^{\delta} & C \Vdash \varepsilon' = \varepsilon \\ C; \Gamma \vdash_{\mathbb{S}} x_2 : \tau_1 & C \Vdash \delta \lhd \tau_2 \end{array}}{C; \Gamma \vdash_{\varepsilon} \mathbf{apply}(x_1, x_2) : \tau_2}\ \text{(SApp)}$$

$$\frac{\begin{array}{ll} C; \Gamma \vdash_{\mathbb{S}} x : (\tau_1 + \tau_2)^{\delta} & C; \Gamma, x_1 : \tau_1 \vdash_{\varepsilon} e_1 : \tau \\ C \Vdash \delta \leq \varepsilon \quad C \Vdash \delta \lhd \tau & C; \Gamma, x_2 : \tau_2 \vdash_{\varepsilon} e_2 : \tau \end{array}}{C; \Gamma \vdash_{\varepsilon} \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \tau}\ \text{(SCase)}$$

Figure 4.2: Typing rules for Level ML.

level-decorated types, constraints, and type schemes in Figure 3.1 and described in Section 3.1. After discussing the rules themselves, we will look at type inference (Section 4.3).

Typing takes place in the context of a constraint formula C and a typing environment Γ that maps variables to type schemes: $\Gamma ::= \cdot \mid \Gamma, x : \sigma$. The typing judgment $C; \Gamma \vdash_{\varepsilon} e : \tau$ has a constraint C and typing environment Γ, and infers type τ for expression e in mode ε. Beyond the usual typing concerns, there are three important aspects of the typing rules: the determination of modes and levels, level polymorphism, and constraints. To help separate concerns, we discuss constraints later in the section—at this time, the reader can ignore the constraints in the rules and read $C; \Gamma \vdash_{\varepsilon} e : \tau$ as $\Gamma \vdash_{\varepsilon} e : \tau$, read $C \Vdash \delta \lhd \tau_2$ as $\delta \lhd \tau_2$, and so on.

The mode of each typing judgment affects the types that can be used "directly" by the expression being typed. Specifically, the mode discipline prevents the elimination forms from being applied to changeable values in the stable mode. This is a key principle of the type system.

Typing rules for values. No computation happens in values, so they can be typed in either mode. The typing rules for variables (SVar), integers (SInt), pairs (SPair), and sums (SSumLeft) are otherwise standard (we omit the symmetric rule typing **inr** v). Rule (SVar) instantiates a variable's polymorphic type. For clarity, we also make explicit the renaming of the quantified type variables $\vec{\alpha}$ to some fresh $\vec{\beta}$ (which will be instantiated later by constraint solving).

To type a function (SFun), we type the body in the mode ε specified by the function type $(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\delta}$, and require the result type τ_2 to be higher than the mode, $\varepsilon \lhd \tau_2$. That is, a changeable-mode function must have a changeable return type. This captures the idea that a changeable-mode function is a computation that depends on changeable data, and thus its result must accommodate changes to that data. (We could instead do this check in rule (SApp), where functions are applied, but then we would have functions that are well-typed but can never be applied.)

Typing primitive operators. Rule (SPrim) allows primitive operators $\oplus$ to be applied to two stable integers, returning a stable integer, or to two changeable integers, returning a changeable integer. Allowing a mix of stable and changeable arguments in this rule would be sound, but is already handled by outer-level subsumption (discussed below).

Typing let-expressions. As is common in Damas-Milner-style systems, when typing **let** we can generalize variables in types (in our system, level variables) to yield a polymorphic value only when the bound expression is a value. This *value restriction* is not essential because Level ML is pure, but its presence facilitates support for side effects in extensions of the language (such as the extension of full Standard ML supported by our implementation).

- **(SLetE):** In the first let-rule (SLetE), the expression bound may be a non-value, so we do not generalize and simply type the body in the same mode as the whole **let**, assuming that the bound expression has the specified type in any mode ε'.[1]
- **Subsumption on outer levels:** We allow subsumption only when the subtype and supertype are equal up to their outer levels, e.g. from a bound expression e_1 of subtype $\mathbf{int}^{\mathbb{S}}$ to an assumption $x : \mathbf{int}^{\mathbb{C}}$. This simplifies the translation, with no loss of expressiveness: to handle "deep" subsumption, such as

  $$(\mathbf{int}^{\mathbb{S}} \underset{\mathbb{S}}{\rightarrow} \mathbf{int}^{\mathbb{S}})^{\mathbb{S}} <: (\mathbf{int}^{\mathbb{S}} \underset{\mathbb{S}}{\rightarrow} \mathbf{int}^{\mathbb{C}})^{\mathbb{C}}$$

  we can insert *coercions* (essentially, eta-expanded identity functions) into the source program before typing it with these rules. This technique of eta-expanding terms to eliminate the need for nontrivial subsumption goes back to (at least) Barendregt et al. (1983), and could easily be automated.
- **(SLetV):** In the second let-rule (SLetV), when the expression bound is a value, we type the let expression in mode ε by typing the body in the same mode ε, assuming that the value bound is typed in the stable mode (the mode is ignored in the rules typing values). As in (SLetE), we allow subsumption on the bound value only when the types are equal up to their outer level. Because we are binding a value, we generalize its type by quantifying over the type's free level variables.

Typing elimination forms. Function application, $\oplus$ (discussed above), **fst**, and **case** are the forms that eliminate values of changeable type.

Rule (SApp) types applications. Two additional constraints are needed, beyond the one enforced in (SFun) (that changeable-mode functions have changeable result types: $\varepsilon \lhd \tau_2$):

- The mode ε' of the function being called must match the current mode ε (the caller's mode): $\varepsilon' = \varepsilon$.

  To see why, first consider the case where we are in stable mode and try to apply a changeable-mode function ($\varepsilon = \mathbb{S}$ and $\varepsilon' = \mathbb{C}$). Changeable data can be directly inspected only in changeable mode; since changeable-mode functions can directly inspect changeable data, the call would allow us to inspect changeable data from stable mode, breaking the property that stable data depends only on stable data.

  Now consider the case where we are in changeable mode, and try to call a stable-mode function ($\varepsilon = \mathbb{C}$ and $\varepsilon' = \mathbb{S}$). This call would not directly violate the same property; we forbid it to simplify translation to a target language that distinguishes stable and changeable modes. Since the rules (SLetV) and (SLetE) can switch from changeable mode to stable mode, we lose no expressive power.
- The outer level of the result of the function, τ_2, must be higher than δ, the function's level: $\delta \lhd \tau_2$.

[1] In the target language, bound expressions must be stable-mode, but the translation puts changeable bound expressions inside a **mod**, yielding a stable-mode bound expression.

The situation we disallow is when $\delta = \mathbb{C}$ and τ_2 is outer-stable, that is, when the called function has a type like $(\tau_1 \underset{\varepsilon}{\rightarrow} \mathbf{int}^{\mathbb{S}})^{\mathbb{C}}$. Here, the result type $\mathbf{int}^{\mathbb{S}}$ is stable and therefore must not depend on changeable data. But the type $(\tau_1 \underset{\varepsilon}{\rightarrow} \mathbf{int}^{\mathbb{S}})^{\mathbb{C}}$ is changeable: a change in program input could cause it to be entirely replaced by another function, which could of course return a different result!

(Assuming "deep" subsumption, we lose no expressive power: we can coerce a function of type $(\tau_1 \underset{\varepsilon}{\rightarrow} \mathbf{int}^{\mathbb{S}})^{\mathbb{C}}$ to type $(\tau_1 \underset{\varepsilon}{\rightarrow} \mathbf{int}^{\mathbb{C}})^{\mathbb{C}}$, which satisfies the constraint.)

Note that neither of these constraints could be enforced via (SFun). The first depends on the current (caller's) mode, so it must be checked at the call site. The second depends on the outer level δ, which might have been originally declared as $\mathbb{S}$, but can rise to $\mathbb{C}$ via subsumption.

The rule (SCase) types a case expression, in either mode ε, by typing each branch in ε. The mode ε must be higher than the level δ of the scrutinee to ensure that a changeable sum type is not inspected at the stable mode. Furthermore, the level of the result τ must also be higher than δ: if the scrutinee changes, we may take the other branch, requiring a changeable result.

Rule (SFst) enforces a condition, similar to (SCase), that we can project out of a changeable tuple of type $(\tau_1 \times \tau_2)^{\mathbb{C}}$ only in changeable mode. We omit the symmetric rule for **snd**.

Our premises on variables, such as the scrutinee of (SCase), are stable-mode ($\vdash_{\mathbb{S}}$), but this was an arbitrary decision; since (SVar) is the only rule that can derive such premises, their mode is irrelevant.

4.2.2 Extending types with precise dependency tracking

In this section, we extend the constraint-based type system to accommodate the fine-grained level-decorated types and constraints (Figure 3.5) as was described in Section 3.2. Figure 4.3 shows the extended type system with precise dependency tracking.

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 Γ, and infers type τ for expression e. 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 $\mathcal{P}$ in the typing rules to make sure the labels inside a function are unique. 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 (SSFun) and (SCFun), we type the body specified by the function type $(\tau_1 \rightarrow \tau_2)^{\delta}$. 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 la-

$\boxed{C; \mathcal{P}; \Gamma \vdash e : \tau}$ Under constraint C, label set $\mathcal{P}$ and source typing environment Γ, source expression e has type τ

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

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

$$\frac{C; \mathcal{P}; \Gamma \vdash x : (\tau_1 \times \tau_2)^{\delta}}{C; \mathcal{P}; \Gamma \vdash \mathbf{fst}\, x : \tau_1}\;(\text{SFst})$$

$$\frac{C; \emptyset; \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^{\mathbb{S}} \vdash e : \tau_2 \qquad \llbracket \tau_1 \rrbracket = \mathbb{S} \qquad C \Vdash \tau_2 \downarrow_0 \mathcal{D}; \mathcal{L}}{C; \mathcal{P}'; \Gamma \vdash (\mathbf{fun}\; f(x) = e) : (\tau_1 \rightarrow \tau_2)^{\mathbb{S}}}\;(\text{SSFun})$$

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

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

$$\frac{\begin{array}{ll} C; \mathcal{P}; \Gamma \vdash e_1 : \tau' & C \Vdash \tau' <: \tau'' \\ C; \mathcal{P}; \Gamma, x : \tau'' \vdash e_2 : \tau & C \Vdash \tau' \doteq \tau'' \quad \llbracket \tau'' \rrbracket = \mathbb{S} \end{array}}{C; \mathcal{P}; \Gamma \vdash \mathbf{let}\; x = e_1\; \mathbf{in}\; e_2 : \tau}\;(\text{SSLet})$$

$$\frac{\begin{array}{ll} C; \mathcal{P}; \Gamma \vdash e_1 : \tau' & C \Vdash \tau' <: \tau'' \quad \llbracket \tau'' \rrbracket = \mathbb{C}_{\rho} \\ C; \mathcal{P} \cup \{\rho\}; \Gamma, x : \tau'' \vdash e_2 : \tau & C \Vdash \tau' \doteq \tau'' \quad C \Vdash \rho \notin \mathcal{P} \end{array}}{C; \mathcal{P}; \Gamma \vdash \mathbf{let}\; x = e_1\; \mathbf{in}\; e_2 : \tau}\;(\text{SCLet})$$

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

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

Figure 4.3: Typing rules for level-monomorphic source language with precise dependency tracking

bels, e.g. labels for changeable input, have prefix 1. To ensure unique labeling in the function scope, we insert the non-destination label 1ρ into the label set $\mathcal{P}$ in (SCFun).

In Section 4.2.1, we require that a changeable-mode function must have a changeable return type. Since our type system eliminates the notion of function mode, we do not have this restriction. This allows us to express more flexible functions, such as returning a stable product with changeable components. This was not possible in the previous type system because a changeable-mode function would require the whole product to be changeable, a root cause for redundant dependencies when translating into self-adjusting programs.

We allow subsumption only at let binding (SSLet) and (SCLet), 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 ρ 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. To distinguish destinations for modifiables created in the let bindings, we assume the labels have prefix 1.

4.3 Constraints and type inference

Both type systems in Section 4.2.1 and Section 4.2.2 are constraint-based type systems. Many of the rules simply pass around the constraint C. An implementation of rules with constraint-based premises, such as (SFun), implicitly adds those premises to the constraint, so that $C = \ldots \wedge (\varepsilon \lhd \tau_2)$. Rule (SLetV) generalizes level variables instead of type variables, with the "occurs check" $\vec{\alpha} \cap FV(C, \Gamma) = \emptyset$.

Standard techniques in the tradition of Damas and Milner (1982) can infer types for Level ML. In particular, our rules and constraints fall within the HM(X) framework (Odersky et al. 1999), permitting inference of principal types via constraint solving. As always, we cannot infer the types of polymorphically recursive functions.

Using a constraint solver that, given the choice between assigning $\mathbb{S}$ or $\mathbb{C}$ to some level variable, prefers $\mathbb{S}$, inference finds principal typings that are *minimally* changeable. Thus, data and computations will only be made changeable—and incur tracking overhead—where necessary to satisfy the programmer's annotation. This corresponds to preferring a lower security level in information flow (Pottier and Simonet 2003). Interestingly, while preferring higher security in information flow is not useful, a constraint solver that prefers $\mathbb{C}$ over $\mathbb{S}$ yields a maximally changeable program, allowing completely automatic (though high-overhead) self-adjusting computation.

Our formulation of the constraint-based rules follows a standard presentation style (Odersky et al. 1999). That style, while relatively concise, obscures how constraints are manipulated in practice: It is tempting to read the typing rules in Figure 4.2 as taking in a constraint C as input. But in an actual constraint-based typechecker, C cannot be input, because C is not known until the program has been traversed! In practice, C should be thought of as both input and output: at the start of typechecking, C is empty (equiva-

$\boxed{e \Downarrow v}$ Source expression e evaluates to v

$$\frac{}{v \Downarrow v}\ \text{(SEvValue)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}\ \text{(SEvPair)}$$

$$\frac{e \Downarrow v}{\textbf{inl}\ e \Downarrow \textbf{inl}\ v}\ \text{(SEvSumLeft)} \qquad \frac{\begin{array}{l} e_1 \Downarrow v_1 \\ e_2 \Downarrow v_2 \quad \oplus(v_1, v_2) = v' \end{array}}{\oplus(e_1, e_2) \Downarrow v'}\ \text{(SEvPrimop)} \qquad \frac{e \Downarrow (v_1, v_2)}{\textbf{fst}\ e \Downarrow v_1}\ \text{(SEvFst)}$$

$$\frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v_2}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v_2}\ \text{(SEvLet)} \qquad \frac{e \Downarrow \textbf{inl}\ v_1 \qquad [v_1/x_1]e_1 \Downarrow v}{\textbf{case}\ e\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \Downarrow v}\ \text{(SEvCaseLeft)}$$

$$\frac{e_1 \Downarrow \textbf{fun}\ f(x) = e \qquad e_2 \Downarrow v_2 \qquad [(\textbf{fun}\ f(x) = e)/f][v_2/x]e \Downarrow v}{\textbf{apply}(e_1, e_2) \Downarrow v}\ \text{(SEvApply)}$$

Figure 4.4: Dynamic semantics of source Level ML programs.

lently, is true); as the typechecker traverses the program, C is extended with additional constraints. For example, the premise $C \Vdash \delta \leq \varepsilon$ in (SFst) really corresponds to adding $\delta \leq \varepsilon$ to the "current" C, not to checking $\delta \leq \varepsilon$ under a known constraint.

An alternative would be to use a judgment with both an input constraint and an output constraint. For a typing of the entire program, the input constraint would be true (at the beginning of typechecking) and the output constraint would correspond to the "final" C in the current formulation. Such a formulation would be closer to an algorithm, but would require explicitly threading the constraint through the rules. Moreover, our meta-theoretical development would become more complicated; in the meta-theory, we care about the *result* of type inference, not internal details of the algorithm.

4.4 Dynamic semantics

The call-by-value semantics of source programs is defined by a big-step judgment $e \Downarrow v$, read "e evaluates to value v". Our rules in Figure 4.4 are standard; we write $[v/x]e$ for capture-avoiding substitution of v for the variable x in e. To simplify the presentation, we omit the symmetric rules (SEvSumRight), (SEvSnd) and (SEvCaseRight).

Chapter 5

Target Language

This chapter is based on work on the theoretical formulation of implicit self-adjusting computation (Chen et al. 2011, 2014b), and a type system extension for precise dependency tracking (Chen et al. 2014a).

5.1 Monadic language

5.1.1 Static semantics

The target language AFL (Figure 5.1) is a self-adjusting language with modifiables. In addition to integers, products, and sums, the target type system makes a modal distinction between ordinary types (e.g. **int**) and modifiable types (e.g. **int mod**). It also distinguishes stable-mode and changeable-mode functions.

Level polymorphism is supported through an explicit **select** construct and an explicit polymorphic instantiation. In Section 6, we describe how polymorphic source expressions become **select**s in AFL. The type schemes used in the target are identical to those in the source language; $\underline{\sigma} = \Pi\vec{\alpha}[D].\,\tau$ quantifies over *source* types τ (from Figure 3.1), not target types $\underline{\tau}$. We cannot quantify over target types here, because no single type scheme over target types can represent exactly the set of types corresponding to the instances of a source type scheme. For example, the source type scheme $\forall\alpha[\mathsf{true}].\,\mathbf{int}^{\alpha}$ corresponds to **int** if α is instantiated with $\mathbb{S}$, and to **int mod** if α is instantiated with $\mathbb{C}$, but the set of types $\{\mathbf{int}, (\mathbf{int\ mod})\}$ does not correspond to the instances of any type scheme.

The values w of the language are integers, variables, polymorphic variable instantiation $x[\vec{\alpha} = \vec{\delta}]$, locations ℓ (which appear only at runtime), pairs, tagged values, stable and changeable functions, and the **select** construct, which acts as a function and case expression on levels: if x is bound to $\mathbf{select}\,\{(\alpha = \mathbb{S}) \Rightarrow e_1 \mid (\alpha = \mathbb{C}) \Rightarrow e_2\}$ then $x[\alpha = \mathbb{S}]$ yields e_1. The symbol $\underline{x}$ stands for a bare variable x or an instantiation $x[\vec{\alpha} = \vec{\delta}]$.

We distinguish stable expressions $e^{\mathbb{S}}$ from changeable expressions $e^{\mathbb{C}}$. Stable expressions create purely functional values; $\mathbf{apply}^{\mathbb{S}}$ applies a stable-mode function. The **mod** construct evaluates a changeable expression and writes the output value to a modifiable, yielding a location, which is a stable expression. Changeable expressions are computa-

$$\begin{array}{lrcl}
\textit{Levels} & \delta, \varepsilon & ::= & \mathbb{S} \mid \mathbb{C} \\
\textit{Types} & \underline{\tau} & ::= & \textbf{int} \mid \underline{\tau}\ \textbf{mod} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \underline{\tau}_1 + \underline{\tau}_2 \mid \underline{\tau}_1 \underset{\varepsilon}{\rightarrow} \underline{\tau}_2 \\
\textit{Type schemes} & \underline{\sigma} & ::= & \Pi\vec{\alpha}[D].\,\tau \\
\textit{Typing environments} & \Gamma & ::= & \cdot \mid \Gamma, x : \underline{\sigma} \mid \Gamma, x : \underline{\tau} \\
\textit{Variables} & \underline{x} & ::= & x \mid x[\vec{\alpha} = \vec{\delta}] \\
\textit{Values} & w & ::= & n \mid \underline{x} \mid \ell \mid (w_1, w_2) \mid \textbf{inl}\ w \mid \textbf{inr}\ w \mid \\
 & & & \textbf{fun}^{\mathbb{S}}\ f(x) = e^{\mathbb{S}} \mid \textbf{fun}^{\mathbb{C}}\ f(x) = e^{\mathbb{C}} \mid \textbf{select}\,\{(\vec{\alpha_i} = \vec{\delta_i}) \Rightarrow e_i\}_i \\
\textit{Expressions} & e & ::= & e^{\mathbb{S}} \mid e^{\mathbb{C}} \\
\textit{Stable expressions} & e^{\mathbb{S}} & ::= & w \mid \oplus(e^{\mathbb{S}}, e^{\mathbb{S}}) \mid \textbf{fst}\ e^{\mathbb{S}} \mid \textbf{snd}\ e^{\mathbb{S}} \mid \\
 & & & \textbf{apply}^{\mathbb{S}}(e^{\mathbb{S}}, e^{\mathbb{S}}) \mid \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e^{\mathbb{S}} \mid \\
 & & & \textbf{case}\ e^{\mathbb{S}}\ \textbf{of}\,\{x_1 \Rightarrow e^{\mathbb{S}}\ ,\ x_2 \Rightarrow e^{\mathbb{S}}\} \mid \\
 & & & \textbf{mod}\ e^{\mathbb{C}} \\
\textit{Changeable expressions} & e^{\mathbb{C}} & ::= & \textbf{apply}^{\mathbb{C}}(e^{\mathbb{S}}, e^{\mathbb{S}}) \mid \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e^{\mathbb{C}} \mid \\
 & & & \textbf{case}\ e^{\mathbb{S}}\ \textbf{of}\,\{x_1 \Rightarrow e^{\mathbb{C}}\ ,\ x_2 \Rightarrow e^{\mathbb{C}}\} \mid \\
 & & & \textbf{read}\ e^{\mathbb{S}}\ \textbf{as}\ y\ \textbf{in}\ e^{\mathbb{C}} \mid \textbf{write}(e^{\mathbb{S}})
\end{array}$$

Figure 5.1: Types and expressions in the target language AFL.

tions that end in a **write** of a pure value. Changeable-mode application **apply**$^{\mathbb{C}}$ applies a changeable-mode function.

The **let** construct is either stable or changeable according to its body. When the body is a changeable expression, **let** enables a changeable computation to evaluate a stable expression and bind its result to a variable. The **case** expression is likewise stable or changeable, according to its case arms. The **read** expression binds the contents of a modifiable $\underline{x}$ to a variable y and evaluates the body of the **read**.

The typing rules in Figure 5.2 follow the structure of the expressions. Rule (TSelect) checks that each monomorphized expression e_i within a **select** has type $\|[\vec{\delta}/\vec{\alpha}]\tau\|$, where $[\vec{\delta}/\vec{\alpha}]\tau$ is a source-level polymorphic type with the levels $\vec{\delta}$ substituted for the variables $\vec{\alpha}$, and $\|-\|$ translates source types to target types (see Section 6.1). Rule (TPVar) is a standard rule for variables of monomorphic type, but rule (TVar) gives the instantiation $x[\vec{\alpha} = \vec{\delta}]$, of a variable x of polymorphic type, the type $\|[\vec{\delta}/\vec{\alpha}]\tau\|$—matching the monomorphic expression from the **select** to which x is bound.

5.1.2 Dynamic semantics

For the source language, our big-step evaluation rules (Figure 4.4) are standard. In the target language AFL, our rules (Figure 5.3) model the evaluation of a first run of the program: modifiables are created, written to (once), and read from (any number of times), but never updated to reflect changes to the program input. Again, we omit symmetric rules such as (SEvSumRight).

According to the grammar in Figure 5.1, $x[\vec{\alpha} = \vec{\delta}]$ is a value. It might seem that

$\boxed{\Lambda;\Gamma \vdash_{\varepsilon} w : \underline{\sigma}}$ Under store typing Λ and target typing environment Γ, target value w has type scheme $\underline{\sigma}$

$$\frac{\begin{array}{c}\text{for all } \vec{\delta_i} \text{ such that } \vec{\alpha} = \vec{\delta_i} \Vdash D \\ \Lambda;\Gamma \vdash_{\mathbb{S}} e_i : \|[\vec{\delta_i}/\vec{\alpha}]\tau\|\end{array}}{\Lambda;\Gamma \vdash_{\mathbb{S}} \textbf{select}\, \{\vec{\delta_i} \Rightarrow e_i\}_i : \Pi\vec{\alpha}[D].\,\tau}\ \text{(TSelect)}$$

$\boxed{\Lambda;\Gamma \vdash_{\varepsilon} e^{\varepsilon} : \underline{\tau}}$ Under store typing Λ and target typing environment Γ, target expression e^{ε} has target type $\underline{\tau}$

$$\frac{\Lambda(\ell) = \underline{\tau}}{\Lambda;\Gamma \vdash_{\mathbb{S}} \ell : \underline{\tau}}\ \text{(TLoc)} \qquad \frac{}{\Lambda;\Gamma \vdash_{\mathbb{S}} n : \textbf{int}}\ \text{(TInt)}$$

$$\frac{\Gamma(x) = \underline{\tau}}{\Lambda;\Gamma \vdash_{\mathbb{S}} x : \underline{\tau}}\ \text{(TPVar)} \qquad \frac{\Gamma(x) = \Pi\vec{\alpha}[D].\,\tau}{\Lambda;\Gamma \vdash_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}] : \|[\vec{\delta}/\vec{\alpha}]\tau\|}\ \text{(TVar)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : \underline{\tau}_1 \qquad \Lambda;\Gamma \vdash_{\mathbb{S}} e_2^{\mathbb{S}} : \underline{\tau}_2}{\Lambda;\Gamma \vdash_{\mathbb{S}} (e_1^{\mathbb{S}}, e_2^{\mathbb{S}}) : \underline{\tau}_1 \times \underline{\tau}_2}\ \text{(TPair)} \qquad \frac{\Lambda;\Gamma, x : \underline{\tau}_1, f : (\underline{\tau}_1 \underset{\varepsilon}{\rightarrow} \underline{\tau}_2) \vdash_{\varepsilon} e : \underline{\tau}_2}{\Lambda;\Gamma \vdash_{\mathbb{S}} \textbf{fun}^{\varepsilon}\ f(x) = e : (\underline{\tau}_1 \underset{\varepsilon}{\rightarrow} \underline{\tau}_2)}\ \text{(TFun)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e^{\mathbb{S}} : \underline{\tau}_1}{\Lambda;\Gamma \vdash_{\mathbb{S}} \textbf{inl}\ e^{\mathbb{S}} : \underline{\tau}_1 + \underline{\tau}_2}\ \text{(TSumLeft)} \qquad \frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e^{\mathbb{S}} : \underline{\tau}_1 \times \underline{\tau}_2}{\Lambda;\Gamma \vdash_{\mathbb{S}} \textbf{fst}\ e^{\mathbb{S}} : \underline{\tau}_1}\ \text{(TFst)}$$

$$\frac{\begin{array}{l}\Lambda;\Gamma \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : \textbf{int} \\ \Lambda;\Gamma \vdash_{\mathbb{S}} e_2^{\mathbb{S}} : \textbf{int} \qquad \vdash \oplus : \textbf{int} \times \textbf{int} \to \textbf{int}\end{array}}{\Lambda;\Gamma \vdash_{\mathbb{S}} \oplus(e_1^{\mathbb{S}}, e_2^{\mathbb{S}}) : \textbf{int}}\ \text{(TPrim)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : \underline{\sigma} \qquad \Lambda;\Gamma, x : \underline{\sigma} \vdash_{\varepsilon} e_2 : \underline{\tau}'}{\Lambda;\Gamma \vdash_{\varepsilon} \textbf{let}\ x = e_1^{\mathbb{S}}\ \textbf{in}\ e_2 : \underline{\tau}'}\ \text{(TLet)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : (\underline{\tau}_1 \underset{\varepsilon}{\rightarrow} \underline{\tau}_2) \qquad \Lambda;\Gamma \vdash_{\mathbb{S}} e_2^{\mathbb{S}} : \underline{\tau}_1}{\Lambda;\Gamma \vdash_{\varepsilon} \textbf{apply}^{\varepsilon}(e_1^{\mathbb{S}}, e_2^{\mathbb{S}}) : \underline{\tau}_2}\ \text{(TApp)}$$

$$\frac{\begin{array}{ll} & \Lambda;\Gamma, x_1 : \underline{\tau}_1 \vdash_{\varepsilon} e_1 : \underline{\tau} \\ \Lambda;\Gamma \vdash_{\mathbb{S}} e^{\mathbb{S}} : \underline{\tau}_1 + \underline{\tau}_2 & \Lambda;\Gamma, x_2 : \underline{\tau}_2 \vdash_{\varepsilon} e_2 : \underline{\tau}\end{array}}{\Lambda;\Gamma \vdash_{\varepsilon} \textbf{case}\ e^{\mathbb{S}}\ \textbf{of}\, \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \underline{\tau}}\ \text{(TCase)} \qquad \frac{\Lambda;\Gamma \vdash_{\mathbb{C}} e^{\mathbb{C}} : \underline{\tau}}{\Lambda;\Gamma \vdash_{\mathbb{S}} \textbf{mod}\ e^{\mathbb{C}} : \underline{\tau}\ \textbf{mod}}\ \text{(TMod)}$$

$$\frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e^{\mathbb{S}} : \underline{\tau}}{\Lambda;\Gamma \vdash_{\mathbb{C}} \textbf{write}(e^{\mathbb{S}}) : \underline{\tau}}\ \text{(TWrite)} \qquad \frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : \underline{\tau}_1\ \textbf{mod} \qquad \Lambda;\Gamma, x : \underline{\tau}_1 \vdash_{\mathbb{C}} e_2^{\mathbb{C}} : \underline{\tau}_2}{\Lambda;\Gamma \vdash_{\mathbb{C}} \textbf{read}\ e_1^{\mathbb{S}}\ \textbf{as}\ x\ \textbf{in}\ e_2^{\mathbb{C}} : \underline{\tau}_2}\ \text{(TRead)}$$

Figure 5.2: Typing rules of the target language AFL.

$\boxed{\rho \vdash e \Downarrow (\rho' \vdash w)}$ In the store ρ, target expression e evaluates to w with updated store ρ'

$$\rho \vdash w \Downarrow (\rho \vdash w) \;\text{(TEvValue)}$$

$$\frac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash w_1) \qquad \rho_1 \vdash e_2 \Downarrow (\rho' \vdash w_2)}{\rho \vdash (e_1, e_2) \Downarrow (\rho' \vdash (w_1, w_2))} \;\text{(TEvPair)} \qquad \frac{\rho \vdash e \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{inl}\; e \Downarrow (\rho' \vdash \mathbf{inl}\; w)} \;\text{(TEvSumLeft)}$$

$$\frac{\begin{array}{l}\rho \vdash e_1 \Downarrow (\rho_1 \vdash w_1) \\ \rho_1 \vdash e_2 \Downarrow (\rho' \vdash w_2) \qquad \oplus(w_1, w_2) = w'\end{array}}{\rho \vdash \oplus(e_1, e_2) \Downarrow (\rho' \vdash w')} \;\text{(TEvPrimop)} \qquad \frac{\rho \vdash e \Downarrow (\rho' \vdash (w_1, w_2))}{\rho \vdash \mathbf{fst}\; e \Downarrow (\rho' \vdash w_1)} \;\text{(TEvFst)}$$

$$\frac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash w_1) \qquad \rho_1 \vdash [w_1/x]e_2 \Downarrow (\rho' \vdash w_2)}{\rho \vdash \mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2 \Downarrow (\rho' \vdash w_2)} \;\text{(TEvLet)}$$

$$\frac{\rho \vdash e \Downarrow (\rho_1 \vdash \mathbf{inl}\; w_1) \qquad \rho_1 \vdash [w_1/x_1]e_1 \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{case}\; e \;\mathbf{of}\; \{x_1 \Rightarrow e_1 \;,\; x_2 \Rightarrow e_2\} \Downarrow (\rho' \vdash w)} \;\text{(TEvCaseLeft)}$$

$$\frac{\begin{array}{l}\rho \vdash e_1^{\varepsilon} \Downarrow (\rho_1 \vdash \mathbf{fun}^{\varepsilon}\; f(x) = e^{\varepsilon}) \\ \rho_1 \vdash e_2^{\varepsilon} \Downarrow (\rho_2 \vdash w_2) \\ \rho_2 \vdash [(\mathbf{fun}^{\varepsilon}\; f(x) = e^{\varepsilon})/f][w_2/x]e^{\varepsilon} \Downarrow (\rho' \vdash w)\end{array}}{\rho \vdash \mathbf{apply}^{\varepsilon}(e_1^{\varepsilon}, e_2^{\varepsilon}) \Downarrow (\rho' \vdash w)} \;\text{(TEvApply)}$$

$$\frac{\rho \vdash e \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{write}(e) \Downarrow (\rho' \vdash w)} \;\text{(TEvWrite)}$$

$$\frac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash \ell) \qquad \rho_1 \vdash [\rho_1(\ell)/x']e^{\mathbb{C}} \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{read}\; e_1 \;\mathbf{as}\; x' \;\mathbf{in}\; e^{\mathbb{C}} \Downarrow (\rho' \vdash w)} \;\text{(TEvRead)}$$

$$\frac{\rho \vdash e \Downarrow (\rho' \vdash w)}{\rho \vdash (\mathbf{select}\,\{\ldots, \vec{\delta} \Rightarrow e, \ldots\})[\vec{\alpha} = \vec{\delta}] \Downarrow (\rho' \vdash w)} \;\text{(TEvSelect)}$$

$$\frac{\rho \vdash e^{\mathbb{C}} \Downarrow (\rho' \vdash w) \qquad \ell \notin \mathsf{dom}(\rho')}{\rho \vdash \mathbf{mod}\; e^{\mathbb{C}} \Downarrow ((\rho', \ell \mapsto w) \vdash \ell)} \;\text{(TEvMod)}$$

Figure 5.3: Dynamic semantics for first runs of AFL programs.

$$\begin{array}{lcll}
\textit{Types} & \underline{\tau} & ::= & \textbf{unit} \mid \textbf{int} \mid \underline{\tau}\ \textbf{mod} \mid \Box\ \underline{\tau} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \\
 & & & \underline{\tau}_1 + \underline{\tau}_2 \mid \underline{\tau}_1 \xrightarrow[\mathcal{D}]{} \underline{\tau}_2 \\
\textit{Dest. Types} & \mathcal{D} & ::= & \{\underline{\tau}_1, \cdots, \underline{\tau}_n\} \\
\textit{Labels} & \mathcal{L} & ::= & \{l_1, \cdots, l_n\} \\
\textit{Variables} & x & ::= & y \mid l_i \\
\textit{Typing Env.} & \Gamma & ::= & \cdot \mid \Gamma, x : \underline{\tau} \\
\textit{Values} & v & ::= & n \mid x \mid \ell \mid (v_1, v_2) \mid \textbf{inl}\ v \mid \textbf{inr}\ v \mid \\
 & & & \textbf{fun}^{\mathcal{L}}\ f(x) = e \\
\textit{Expressions} & e & ::= & v \mid \oplus(x_1, x_2) \mid \textbf{fst}\ x \mid \textbf{snd}\ x \mid \\
 & & & \textbf{apply}^{\mathcal{L}}(x_1, x_2) \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \\
 & & & \textbf{case}\ x\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \mid \\
 & & & \textbf{mod}\ v \mid \textbf{read}\ x\ \textbf{as}\ y\ \textbf{in}\ e \mid \textbf{write}(x_1, x_2)
\end{array}$$

Figure 5.4: Types and expressions in the imperative target language

evaluation (Figure 5.3) could replace the variable x by a **select** expression, yielding **select** $\{\ldots\}[\vec{\alpha} = \vec{\delta}]$, which does not evaluate to itself. However, $x[\vec{\alpha} = \vec{\delta}]$ is not closed, and we only evaluate closed target expressions.

5.2 Imperative language

5.2.1 Static semantics

To facilitate labels from the source language (Section 4.2.2), the target language (Figure 5.4) has to be an imperative self-adjusting language with modifiables. In addition to integers, units, products, sums, the target type system makes a distinction between fresh modifiable types $\Box$ **int** (modifiables that are freshly allocated) and finalized modifiable types **int mod** (modifiables that are written after the allocation). The function type $\underline{\tau}_1 \xrightarrow[\mathcal{D}]{} \underline{\tau}_2$ contains an ordered set of destination types $\mathcal{D}$, indicating the type of the destinations of the function. Compared with the previous monadic self-adjusting language, a function with an empty set $\mathcal{D}$ means a stable-mode function, and a non-empty set $\mathcal{D}$ means a changeable-mode 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 ℓ (which appear only at runtime), pairs, tagged values, and functions. Each function $\textbf{fun}^{\mathcal{L}}\ f(x) = e$ 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 $\textbf{apply}^{\mathcal{L}}(x_1, x_2)$ applies a function while supplying a set of destination modifiables $\mathcal{L}$. The **mod** v construct creates a new fresh modifiable $\Box\ \underline{\tau}$ 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**.

The typing rules in Figure 5.5 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 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 typing environment that guarantees that all the supplied destination modifiables are finalized after the function application.

5.2.2 Dynamic semantics

The dynamic semantics of our target language matches that of Acar et al. (Acar et al. 2008a) after two syntactical changes: $\textbf{fun}^{\mathcal{L}}\ f(x) = e$ is represented as $\textbf{fun}\ f(x) = \lambda\mathcal{L}.e$,

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

$$\dfrac{\Lambda(\ell) = \underline{\tau}}{\Lambda; \Gamma \vdash \ell : \underline{\tau} \dashv \Gamma}\ (\text{TLoc}) \qquad \dfrac{}{\Lambda; \Gamma \vdash n : \textbf{int} \dashv \Gamma}\ (\text{TInt}) \qquad \dfrac{\Gamma(x) = \underline{\tau}}{\Lambda; \Gamma \vdash x : \underline{\tau} \dashv \Gamma}\ (\text{TVar})$$

$$\dfrac{\Lambda; \Gamma \vdash v : \underline{\tau} \dashv \Gamma}{\Lambda; \Gamma \vdash \textbf{mod}\ v : \Box\ \underline{\tau} \dashv \Gamma}\ (\text{TAlloc})$$

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

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

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

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

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

$$\dfrac{\begin{array}{l} \Lambda; \Gamma \vdash x_1 : \textbf{int} \dashv \Gamma \\ \Lambda; \Gamma \vdash x_2 : \textbf{int} \dashv \Gamma \qquad \vdash \oplus : \textbf{int} \times \textbf{int} \rightarrow \textbf{int} \end{array}}{\Lambda; \Gamma \vdash \oplus(x_1, x_2) : \textbf{int} \dashv \Gamma}\ (\text{TPrim})$$

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

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

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

Figure 5.5: Typing rules of the imperative target language

and $\mathbf{apply}^{\mathcal{L}}(x_1, x_2)$ is represented as $(x_1\ x_2)\ \mathcal{L}$.

Chapter 6

Translation

This chapter is based on work on the theoretical formulation of implicit self-adjusting computation (Chen et al. 2011, 2014b), and a type system extension for precise dependency tracking (Chen et al. 2014a).

We specify the translation from Level ML to the target language AFL by a set of a rules. Because AFL is a modal language that distinguishes stable and changeable expressions, with a corresponding type system (Section 5.1), the translation is also modal: the translation in the stable mode $\underset{\mathbb{S}}{\hookrightarrow}$ produces a stable AFL expression $e^{\mathbb{S}}$, and the translation in the changeable mode $\underset{\mathbb{C}}{\hookrightarrow}$ produces a changeable expression $e^{\mathbb{C}}$.

It is not enough to generate AFL expressions of the right syntactic form; they must also have the right type. To achieve this, the rules are type-directed: we translate a source expression e *at type* τ. But we are transforming expressions from one language to another, where each language has its own type system; translating some $e : \tau$ cannot produce some $e' : \tau$, but some $e' : \underline{\tau}'$ where $\underline{\tau}'$ is a target type that *corresponds to* τ. To express this vital property, we need to translate types, as well as expressions. We developed the translation of expressions and types together (along with the proof that the property holds); the translation of types was instrumental in getting the translation of expressions right. To understand how to translate expressions, it is helpful to first understand how we translate types.

6.1 Translating types

Figure 6.1 defines the translation of types via two mutually recursive functions from Level ML types to AFL types. The first function, $\|\tau\|$, tells us what type the target expression $e^{\mathbb{S}}$ should have when we translate e in the stable mode, $e : \tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$. We also use it to translate the types in the environment Γ. The second function, $\|\tau\|^{\to\mathbb{C}}$, makes sense in two related situations: translating the type τ of an expression e in the changeable mode ($e : \tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$) and translating the codomain of changeable functions.

In the stable mode, values of stable type can be used and created directly, so the "stable" translation $\|\mathbf{int}^{\mathbb{S}}\|$ of a stable integer is just **int**. In contrast, a changeable in-

$\left|\tau\right|^{\mathbb{S}}$: Stabilization of types

$$\begin{aligned}
\left|\mathbf{int}^{\mathbb{C}}\right|^{\mathbb{S}} &= \mathbf{int}^{\mathbb{S}} \\
\left|(\tau_1 \times \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} &= (\tau_1 \times \tau_2)^{\mathbb{S}} \\
\left|(\tau_1 + \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} &= (\tau_1 + \tau_2)^{\mathbb{S}} \\
\left|(\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} &= (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{S}}
\end{aligned}$$

$\|\tau\|$: (Stable) translation of types

$$\begin{aligned}
\|\mathbf{int}^{\mathbb{S}}\| &= \mathbf{int} \\
\|\mathbf{int}^{\mathbb{C}}\| &= \mathbf{int\ mod} \\
\|(\tau_1 \underset{\mathbb{S}}{\rightarrow} \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| \underset{\mathbb{S}}{\rightarrow} \|\tau_2\| \\
\|(\tau_1 \underset{\mathbb{S}}{\rightarrow} \tau_2)^{\mathbb{C}}\| &= \Big(\|\tau_1\| \underset{\mathbb{S}}{\rightarrow} \|\tau_2\|\Big)\ \mathbf{mod} \\
\|(\tau_1 \underset{\mathbb{C}}{\rightarrow} \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| \underset{\mathbb{C}}{\rightarrow} \|\tau_2\|^{\to\mathbb{C}} \\
\|(\tau_1 \underset{\mathbb{C}}{\rightarrow} \tau_2)^{\mathbb{C}}\| &= \Big(\|\tau_1\| \underset{\mathbb{C}}{\rightarrow} \|\tau_2\|^{\to\mathbb{C}}\Big)\ \mathbf{mod} \\
\|(\tau_1 \times \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| \times \|\tau_2\| \\
\|(\tau_1 \times \tau_2)^{\mathbb{C}}\| &= (\|\tau_1\| \times \|\tau_2\|)\ \mathbf{mod} \\
\|(\tau_1 + \tau_2)^{\mathbb{S}}\| &= \|\tau_1\| + \|\tau_2\| \\
\|(\tau_1 + \tau_2)^{\mathbb{C}}\| &= (\|\tau_1\| + \|\tau_2\|)\ \mathbf{mod}
\end{aligned}$$

$\|\tau\|^{\to\mathbb{C}}$: Output-changeable translation of types

$$\|\tau\|^{\to\mathbb{C}} = \begin{cases} \|\left|\tau\right|^{\mathbb{S}}\| & \text{if } \tau \text{ O.C.} \\ \|\tau\| & \text{if } \tau \text{ O.S.} \end{cases}$$

$\|\Gamma\|$: Translation of contexts

$$\begin{aligned}
\|\cdot\| &= \cdot \\
\|\Gamma, x : \forall\emptyset[\mathsf{true}].\,\tau\| &= \|\Gamma\|, x : \|\tau\| \\
\|\Gamma, x : \forall\vec{\alpha}[D].\,\tau\| &= \|\Gamma\|, x : \Pi\vec{\alpha}[D].\,\tau
\end{aligned}$$

Translations under substitution ϕ

$$\begin{aligned}
\|\tau\|_{\phi} &= \|[\phi]\tau\| \\
\|\tau\|_{\phi}^{\to\mathbb{C}} &= \|[\phi]\tau\|^{\to\mathbb{C}} \\
\|\Gamma\|_{\phi} &= \|[\phi]\Gamma\|
\end{aligned}$$

Figure 6.1: Stabilization of types $\left|\tau\right|^{\mathbb{S}}$; translations $\|\tau\|$ and $\|\tau\|^{\to\mathbb{C}}$ of types translation of typing environments $\|\Gamma\|$; translations under substitution.

teger cannot be inspected or directly created in stable mode, but must be placed into a modifiable: $\|\mathbf{int}^{\mathbb{C}}\| = \mathbf{int\ mod}$. The remaining parts of the definition follow this pattern: the target type is wrapped with **mod** if and only if the outer level of the source type is $\mathbb{C}$. When we translate a changeable-mode function type (with $\mathbb{C}$ below the arrow), its codomain is translated "output-changeable": $\|(\tau_1 \underset{\mathbb{C}}{\to} \tau_2)^{\mathbb{S}}\| = \|\tau_1\| \underset{\mathbb{C}}{\to} \|\tau_2\|^{\to\mathbb{C}}$. The reason is that a changeable-mode function can only be applied in the changeable mode; the function result is not placed into a modifiable until we return to the stable mode, so putting a **mod** on the codomain would not match the dynamic semantics of AFL.

The second function $\|\tau\|^{\to\mathbb{C}}$ defines the type of a changeable expression e that writes to a modifiable containing τ, yielding a changeable target expression $e^{\mathbb{C}}$. The source type has an outer $\mathbb{C}$, so when the value is written, it will be placed into a modifiable and have **mod** type. But *inside* the evaluation of $e^{\mathbb{C}}$, there is no outer **mod** on the type.[1] Thus the translation $\|\tau\|^{\to\mathbb{C}}$ ignores the outer level (using the function $|-|^{\mathbb{S}}$, which replaces an outer level $\mathbb{C}$ with $\mathbb{S}$), and never returns a type of the form $(\cdots\ \mathbf{mod})$. However, since the value being returned may contain subexpressions that will be placed into modifiables, we use $\|-\|$ for the inner types. For instance, $\|(\tau_1 + \tau_2)^{\delta}\|^{\to\mathbb{C}} = \|\tau_1\| + \|\tau_2\|$.

These functions are defined on closed types—types with no free level variables. Before applying one of these functions to a type found by the constraint typing rules, we always need to apply the satisfying assignment ϕ to the type, so for convenience we write $\|\tau\|_{\phi}$ for $\|[\phi]\tau\|$, and so on.

Because the translation only makes sense for closed types, type schemes $\forall\vec{\alpha}[D].\,\tau$ cannot be translated before instantiation. Consider the type $\forall\alpha[\mathsf{true}].\,\mathbf{int}^{\alpha}$, and the translations of its instantiations:

$$\begin{aligned}\|\mathbf{int}^{\mathbb{S}}\| &= \mathbf{int}\\ \|\mathbf{int}^{\mathbb{C}}\| &= \mathbf{int\ mod}\end{aligned}$$

No single type scheme over target types can represent exactly the set of types $\{\mathbf{int}, \mathbf{int\ mod}\}$. The translation $\|\Gamma\|$, therefore, translates only monomorphic types τ; type schemes are left alone until instantiation. Once instantiated, the type scheme is an ordinary closed source type, and can be translated by rule (Var) in Figure 6.2.

6.2 Translating expressions

We define the translation of expressions as a set of type-directed rules. Given (1) a derivation of $C; \Gamma \vdash_{\varepsilon} e : \tau$ in the constraint-based typing system and (2) a satisfying assignment ϕ for C, it is always possible to produce a correctly typed stable target expression $e^{\mathbb{S}}$ *and* a correctly typed changeable target expression $e^{\mathbb{C}}$ (see Theorem 6.4.1 below). The environment Γ in the translation rules is a source typing environment, but must have no free level variables. Given an environment Γ from the constraint typing,

[1] In this respect, **mod** behaves like a monadic or computational type constructor, like the $\bigcirc$ modality of lax logic (Pfenning and Davies 2001); *inside* a computation-level (changeable) expression, the result type is τ, but outside of the computation/monad, the result has type $\bigcirc\tau$.

$\boxed{\Gamma \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e^{\varepsilon}}$ Under closed source typing environment Γ,
source expression e is translated at type τ in mode ε to target expression e^{ε}

$$\frac{}{\Gamma \vdash n : \mathbf{int}^{\delta} \underset{\mathbb{S}}{\hookrightarrow} n}\ \text{(Int)} \qquad \frac{\Gamma(x) = \forall\vec{\alpha}[D].\,\tau}{\Gamma \vdash x : [\vec{\delta}/\vec{\alpha}]\tau \underset{\mathbb{S}}{\hookrightarrow} x[\vec{\alpha} = \vec{\delta}]}\ \text{(Var)}$$

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

$$\frac{\Gamma, x : \tau_1, f : (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{S}} \vdash e : \tau_2 \underset{\varepsilon}{\hookrightarrow} e^{\varepsilon}}{\Gamma \vdash \mathbf{fun}\ f(x) = e : (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} \mathbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon}}\ \text{(Fun)}$$

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

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

$$\frac{\Gamma \vdash x_1 : (\tau_1 \underset{\varepsilon}{\rightarrow} \tau_2)^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} \underline{x_1} \qquad \Gamma \vdash x_2 : \tau_1 \underset{\mathbb{S}}{\hookrightarrow} \underline{x_2}}{\Gamma \vdash \mathbf{apply}(x_1, x_2) : \tau_2 \underset{\varepsilon}{\hookrightarrow} \mathbf{apply}^{\varepsilon}(\underline{x_1}, \underline{x_2})}\ \text{(App)}$$

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

$$\frac{\Gamma \vdash e : \tau' \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}} \qquad \left|\tau\right|^{\mathbb{S}} = \tau'}{\Gamma \vdash e : \tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}}}\ \text{(Lift)} \qquad \frac{\Gamma \vdash e : \tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}} \qquad \tau\ \mathsf{O.C.}}{\Gamma \vdash e : \tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}}}\ \text{(Mod)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}} \qquad \Gamma, x : \tau' \vdash e_2 : \tau \underset{\varepsilon}{\hookrightarrow} e_2'}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau \underset{\varepsilon}{\hookrightarrow} \mathbf{let}\ x = e^{\mathbb{S}}\ \mathbf{in}\ e_2'}\ \text{(LetE)}$$

$$\frac{\Gamma, x : \forall\vec{\alpha}[D].\,\tau' \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e' \qquad \begin{array}{r} \text{For all } \vec{\delta_i} \text{ s.t. } \vec{\alpha} = \vec{\delta_i} \Vdash D, \\ \Gamma \vdash v : [\vec{\delta_i}/\vec{\alpha}]\tau' \underset{\mathbb{S}}{\hookrightarrow} e_i' \end{array}}{\Gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ e : \tau \underset{\varepsilon}{\hookrightarrow} \mathbf{let}\ x = \mathbf{select}\,\{(\vec{\alpha}{=}\vec{\delta_i}) \Rightarrow e_i'\}_i\ \mathbf{in}\ e'}\ \text{(LetV)}$$

$$\frac{\begin{array}{ll} \Gamma \vdash e \rightsquigarrow (x \gg x' : \tau' \vdash e') & \tau'\ \mathsf{O.C.} \\ \Gamma, x' : \left|\tau'\right|^{\mathbb{S}} \vdash e' : \tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}} & \Gamma \vdash x : \tau' \underset{\mathbb{S}}{\hookrightarrow} \underline{x} \end{array}}{\Gamma \vdash e : \tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{read}\ \underline{x}\ \mathbf{as}\ x'\ \mathbf{in}\ e^{\mathbb{C}}}\ \text{(Read)}$$

$$\frac{\Gamma \vdash e : \tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}} \qquad \tau\ \mathsf{O.S.}}{\Gamma \vdash e : \tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r)}\ \text{(Write)}$$

$$\frac{\Gamma \vdash e : \tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}} \qquad \tau\ \mathsf{O.C.}}{\Gamma \vdash e : \tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r')}\ \text{(ReadWrite)}$$

Figure 6.2: Monomorphizing translation.

$\boxed{\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau \vdash e')}$ Under source typing Γ,
renaming the "head" x in e to $x' : \tau$ yields expression e'

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

$$\frac{\Gamma \vdash x_2 : \tau}{\Gamma \vdash \oplus(x_1, x_2) \rightsquigarrow (x_2 \gg x_2' : \tau \vdash \oplus(x_1, x_2'))} \text{ (LPrimop2)}$$

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

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

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

Figure 6.3: Renaming the variable to be read (elimination forms).

we apply the satisfying assignment ϕ to eliminate its free level variables before using it for the translation: $[\phi]\Gamma$. With the environment closed, we need not refer to C.

Many of the rules in Figure 6.2 are purely syntax-directed and are similar to the constraint-based rules. One exception is the (Var) rule, which needs the source type to know how to instantiate the level variables in the type scheme. For example, given the polymorphic $x : \forall\alpha[\mathsf{true}].\, (\mathbf{int}^{\alpha} \underset{\alpha}{\rightarrow} \mathbf{int}^{\alpha})^{\mathbb{S}}$, we need the type from $C; \Gamma \vdash_{\varepsilon} x : (\mathbf{int}^{\mathbb{C}} \underset{\mathbb{C}}{\rightarrow} \mathbf{int}^{\mathbb{C}})^{\mathbb{S}}$ so we can instantiate α in the translated term $x[\alpha = \mathbb{C}]$.

Our rules are nondeterministic, avoiding the need to "decorate" them with context-sensitive details. Our algorithm in Section 6.3 resolves the nondeterminism through type information.

Stable rules. The rules (Int), (Var), (Pair), (Fun), (SumLeft), (Fst) and (Prim) can only translate in the stable mode. To translate to a changeable expression, use a rule that shifts to changeable mode.

Shifting to changeable mode. Given a translation of e in the stable mode to some $e^{\mathbb{S}}$, the rules (Write) and (ReadWrite) at the bottom of Figure 6.2 translate e in the changeable mode, producing an $e^{\mathbb{C}}$. If the expression's type τ is outer stable (say, $\mathbf{int}^{\mathbb{S}}$), the (Write) rule simply binds it to a variable and then writes that variable. If τ is outer changeable (say, $\mathbf{int}^{\mathbb{C}}$) it will be in a modifiable at runtime, so we read it into r' and then write it. (The let-bindings merely satisfy the requirements of A-normal form.)

Shifting to stable mode. To generate a stable expression $e^{\mathbb{S}}$ based on a changeable expression $e^{\mathbb{C}}$, we have the (Lift) and (Mod) rules. These rules require the source type τ to be outer changeable: in (Lift), the premise $\left|\tau\right|^{\mathbb{S}} = \underline{\tau}'$ requires that $\left|\tau\right|^{\mathbb{S}}$ is defined, and it is defined only for outer changeable τ; in (Mod), the requirement is explicit: $\vdash \tau$ O.C.

(Mod) is the simpler of the two: if e translates to $e^{\mathbb{C}}$ at type τ, then e translates to the stable expression **mod** $e^{\mathbb{C}}$ at type τ. In (Lift), the expression is translated not at the given type τ but at its *stabilized* $\left|\tau\right|^{\mathbb{S}}$, capturing the "shallow subsumption" in the constraint typing rules (SLetE) and (SLetV): a bound expression of type $\tau_0^{\mathbb{S}}$ can be translated at type $\tau_0^{\mathbb{S}}$ to $e^{\mathbb{S}}$, and then "promoted" to type $\tau_0^{\mathbb{C}}$ by placing it inside a **mod**.

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, $\Gamma \vdash$ **case** x **of** $\ldots \rightsquigarrow (x \gg x' : \tau \vdash$ **case** x' **of** $\ldots)$. This judgment is derivable only for **apply**, **case**, **fst**, and $\oplus$, because these are the elimination forms for outer-changeable data. For $\oplus(x_1, x_2)$, we need to read both variables, so we have one rule for each. The rules are given in Figure 6.3.

Monomorphization. A polymorphic source expression has no directly corresponding target expression: the `map` function from Section 2 corresponds to the two functions `map_SC` and `map_CS`. Given a polymorphic source value $v : \forall\vec{\alpha}[D].\,\tau'$, the (LetV) rule translates v once for each instantiation $\vec{\delta_i}$ that satisfies the constraint D (each $\vec{\delta_i}$ such that $\vec{\alpha} = \vec{\delta_i} \Vdash D$). That is, we translate the value at source type $[\vec{\delta_i}/\vec{\alpha}]\tau'$. This yields a sequence of source expressions $e_1, \ldots, e_n$ for the n possible instances. For example, given $\forall\alpha[\mathsf{true}].\,\tau'$, we translate the value at type $[\mathbb{S}/\alpha]\tau'$ yielding e_1 and at type $[\mathbb{C}/\alpha]\tau'$ yielding e_2. Finally, the rule produces a **select** expression, which acts as a function that takes the desired instance $\vec{\delta_i}$ and returns the appropriate e_i.

Since (LetV) generates one function for each satisfying $\vec{\delta_i}$, it can create up to 2^n instances for n variables. However, dead-code elimination can remove functions that are not used. Moreover, the functions that *are* used would have been handwritten in an explicit setting, so while the code size is exponential in the worst case, the saved effort is as well.

6.3 Algorithm

The system of translation rules in Figure 6.2 is not deterministic. In fact, if the wrong choices are made it can produce painfully inefficient code. Suppose we have $2 : \mathbf{int}^{\mathbb{C}}$, and want to translate it to a stable target expression. Choosing rule (Int) yields the target expression 2. But we could use (Int), then (ReadWrite)—which generates an $e^{\mathbb{C}}$ with a

```
function trans ($e$, $\varepsilon$) = case ($e$, $\varepsilon$) of
 | ($n$, $\mathbb{S}$) $\Rightarrow$ Int
 | ($x$, $\mathbb{S}$) $\Rightarrow$ Var
 | (($v_1$,$v_2$), $\mathbb{S}$) $\Rightarrow$ Pair(trans($v_1$, $\mathbb{S}$), trans($v_2$, $\mathbb{S}$))
 | ($\mathbf{fun}\ f(x) = e' : (\tau_1 \xrightarrow[\varepsilon']{} \tau_2)^{\mathbb{S}}$, $\mathbb{S}$) $\Rightarrow$ Fun(trans($e'$, $\varepsilon'$))
 | ($\mathbf{inl}\ v$, $\mathbb{S}$) $\Rightarrow$ SumLeft(trans($v$, $\mathbb{S}$))
 | ($\mathbf{fst}\,(x : (\tau_1 \times \tau_2)^{\delta}$, $\varepsilon$) $\Rightarrow$ case ($\delta$, $\varepsilon$) of
    | ($\mathbb{S}$,$\mathbb{S}$) $\Rightarrow$ Fst(trans($x$, $\mathbb{S}$))
    | ($\mathbb{S}$,$\mathbb{C}$) $\Rightarrow$ if $\tau_1$ O.S. then Write(trans($e$, $\mathbb{S}$))
                else ReadWrite(trans($e$, $\mathbb{S}$)))
    | ($\mathbb{C}$,$\mathbb{C}$) $\Rightarrow$ Read(LFst, trans($\mathbf{fst}\,(x'{:}(\tau_1 \times \tau_2)^{\mathbb{S}})$, $\mathbb{C}$), trans($x$, $\mathbb{S}$))
 | ($\oplus(x_1 : \mathbf{int}^{\mathbb{S}}, x_2 : \mathbf{int}^{\mathbb{S}})$, $\mathbb{S}$) $\Rightarrow$ Prim(trans($x_1$, $\mathbb{S}$), trans($x_2$, $\mathbb{S}$))
 | ($\oplus(x_1 : \mathbf{int}^{\mathbb{S}}, x_2 : \mathbf{int}^{\mathbb{S}})$, $\mathbb{C}$) $\Rightarrow$ Write(trans($e$, $\mathbb{S}$))
 | ($\oplus(x_1 : \mathbf{int}^{\mathbb{C}}, x_2 : \mathbf{int}^{\mathbb{C}})$, $\mathbb{S}$) $\Rightarrow$ Mod(trans($e$, $\mathbb{C}$))
 | ($\oplus(x_1 : \mathbf{int}^{\mathbb{C}}, x_2 : \mathbf{int}^{\mathbb{C}})$, $\mathbb{C}$) $\Rightarrow$ Read(LPrimop1,
                                     Read(LPrimop2,
                                          Write(trans($\oplus(x_1', x_2')$, $\mathbb{S}$)),
                                          trans($x_2$,$\mathbb{S}$)),
                                     trans($x_1$,$\mathbb{S}$))
 | ($\mathbf{let}\ x : \tau'' = e_1 : \tau'\ \mathbf{in}\ e_2$, $\varepsilon$) $\Rightarrow$
     LetE(if $\tau''$ O.S. then trans($e_1$, $\mathbb{S}$)
          else (if $\tau' = \tau''$ then Mod(trans($e_1$, $\mathbb{C}$))
                else Lift(trans($e_1$, $\mathbb{C}$))),
          trans($e_2$, $\varepsilon$))
 | ($\mathbf{let}\ x : \forall\vec{\alpha}[D].\,\tau'' = v_1 : \tau'\ \mathbf{in}\ e_2$, $\varepsilon$) $\Rightarrow$
     let variants = all $\vec{\delta_i}$ such that $\vec{\alpha} = \vec{\delta_i} \Vdash D$ in
     let f = $\lambda\vec{\delta}$. if $[\vec{\delta}/\vec{\alpha}]\tau''$ O.S. then trans($v_1$, $\mathbb{S}$)
                 else (if $\tau' = [\vec{\delta}/\vec{\alpha}]\tau''$ then Mod(trans($v_1$, $\mathbb{C}$))
                       else Lift(trans($v_1$, $\mathbb{C}$))) in
       LetV(map f variants, trans($e_2$, $\varepsilon$))
 | ($\mathbf{apply}(x_1 : (\tau_1 \xrightarrow[\varepsilon']{} \tau_2)^{\delta}, x_2)$, $\varepsilon$) $\Rightarrow$ case ($\varepsilon'$, $\delta$, $\varepsilon$) of
    | ($\mathbb{S}$,$\mathbb{S}$,$\mathbb{S}$) $\Rightarrow$ App(trans($x_1$, $\mathbb{S}$), trans($x_2$, $\mathbb{S}$))
    | ($\mathbb{C}$,$\mathbb{S}$,$\mathbb{C}$) $\Rightarrow$ App(trans($x_1$, $\mathbb{S}$), trans($x_2$, $\mathbb{S}$))
    | ($\mathbb{S}$,$\mathbb{S}$,$\mathbb{C}$) $\Rightarrow$ if $\tau_2$ O.S. then Write(trans($e$, $\mathbb{S}$))
                    else ReadWrite(trans($e$, $\mathbb{S}$))
    | ($\varepsilon'$,$\mathbb{C}$,$\mathbb{C}$) $\Rightarrow$ Read(LApply, trans($\mathbf{apply}(x'{:}(\tau_1 \xrightarrow[\varepsilon']{} \tau_2)^{\mathbb{S}}, x_2)$, $\mathbb{C}$),
                          trans($x_1$, $\mathbb{S}$))
    | ($\mathbb{C}$,$\mathbb{S}$,$\mathbb{S}$) $\Rightarrow$ Mod(trans($e$, $\mathbb{C}$))
    | ($\varepsilon'$,$\mathbb{C}$,$\mathbb{S}$) $\Rightarrow$ Mod(trans($e$, $\mathbb{C}$))
 | ($\mathbf{case}\ x : \tau\ \mathbf{of}\ \{x_1 \Rightarrow e_1,\ x_2 \Rightarrow e_2\}$, $\varepsilon$) $\Rightarrow$
     if $\tau$ O.S. then
       Case(trans($x$, $\mathbb{S}$), trans($e_1$, $\varepsilon$), trans($e_2$, $\varepsilon$))
     else Read(LCase, trans($\mathbf{case}\ x' : |\tau|^{\mathbb{S}}\ \mathbf{of}\ \{x_1 \Rightarrow e_1,\ x_2 \Rightarrow e_2\}$, $\mathbb{C}$),
               trans($x$, $\mathbb{S}$))
 | ($x : \tau$, $\mathbb{C}$) $\Rightarrow$ if $\tau$ O.S. then Write(trans($e$, $\mathbb{S}$))
                else ReadWrite(trans($e$, $\mathbb{S}$))
 | ($\mathbf{fun}\ f(x) = e'$, $\mathbb{C}$) | ($\mathbf{inl}\ v$, $\mathbb{C}$) | ($n$, $\mathbb{C}$) | (($v_1$, $v_2$), $\mathbb{C}$) $\Rightarrow$ Write(trans($e$,$\mathbb{S}$))
```

Figure 6.4: Translation algorithm.

let, a **read** and a **write**—then (Mod), which wraps that $e^{\mathbb{C}}$ in a **mod**. Clearly, we should have stopped with (Int).

To resolve this nondeterminism in the rules would complicate them further. Instead, we give the algorithm in Figure 6.4, which examines the source expression e and, using type information, applies the rules necessary to produce an expression of mode ε.

6.4 Translation type soundness

Given a constraint-based source typing derivation and assignment ϕ for some term e, it is possible to translate e to (1) a stable $e^{\mathbb{S}}$ and (2) a changeable $e^{\mathbb{C}}$, with appropriate target types:

Theorem 6.4.1 (Translation Type Soundness).
If $C; \Gamma \vdash_{\varepsilon} e : \tau$ *and* ϕ *is a satisfying assignment for* C *then*

(1) there exists $e^{\mathbb{S}}$ *such that* $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$ *and* $\cdot; \|\Gamma\|_{\phi} \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_{\phi}$ *and, if* e *is a value, then* $e^{\mathbb{S}}$ *is a value;*

(2) there exists $e^{\mathbb{C}}$ *such that* $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$ *and* $\cdot; \|\Gamma\|_{\phi} \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_{\phi}^{\rightarrow\mathbb{C}}$.

The proof (Appendix A.1) is by induction on the height of the given derivation of $C; \Gamma \vdash_{\varepsilon} e : \tau$. If the concluding rule was (SLetE), we use a substitution property (Lemma A.1.2) for each $\vec{\delta_i}$ to get a monomorphic constraint typing derivation; that derivation is not larger than the input derivation, so we can apply the induction hypothesis to get a translated e'_i. The proof constructs the same translation derivations as the algorithm in Figure 6.4 (in fact, we extracted the algorithm from the proof).

When applying the theorem "from the outside", it suffices to get an expression of the same mode as the the typing derivation: given $C; \Gamma \vdash_{\mathbb{S}} e : \tau$, use part (1) to get $e^{\mathbb{S}}$; given $C; \Gamma \vdash_{\mathbb{C}} e : \tau$, use part (2) to get $e^{\mathbb{C}}$. However, inside the proof, we need both parts (1) and (2). For example, in the (SLetE) case of the proof, we apply the induction hypothesis to the typing derivation for the let-bound subexpression; in one subcase, the subexpression e_1 is typed in stable mode, but we need a changeable-mode translation of e_1.

6.5 Translation soundness

Having shown that the translated programs have appropriate types, we now prove that running a translated program gives the same result as running the source program.

Theorem 6.5.4 states that if evaluating the translated program e' (in an initially-empty store) yields a (target-language) value w under a new store ρ', then the source program e evaluates to v where v corresponds to $[\rho']w$ (the result of substituting values in the store ρ' for locations appearing in w).

We define this correspondence via a *back-translation* $[\![e']\!]$ which, given some e' which resulted from translating e, essentially yields e. The modifier "essentially" is necessary because, to facilitate the proof of translation soundness, the result of back-translation is not literally e—it may add some **let** expressions. The result of back-translation is, however, *equivalent* to e: if $\llbracket e' \rrbracket \Downarrow v$ then $e \Downarrow v$.

Concretely, the back-translation removes all the constructs related to the store: $\textbf{write}(x)$ becomes x; **read** expressions are replaced with **let**s; **mod** expressions are replaced with their bodies. The back-translation also removes level superscripts: $\textbf{apply}^{\varepsilon}$ becomes **apply**, etc. Finally, the back-translation drops instantiations of polymorphic variables, replacing $x[\vec{\alpha} = \vec{\delta}]$ with x, and replaces **select** expressions with the back-translation of a branch. (The translation guarantees that all branches of a **select** will have semantically equivalent back-translations, a property we call **select**-uniformity.) We give the full definition in Figure 6.5.

$$
\begin{array}{rcl}
\llbracket x \rrbracket & = & x \\
\llbracket e[\vec{\alpha} = \vec{\delta}] \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \textbf{apply}^{\varepsilon}(e_1, e_2) \rrbracket & = & \textbf{apply}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
\llbracket \textbf{fun}^{\varepsilon}\ f(x) = e \rrbracket & = & \textbf{fun}\ f(x) = \llbracket e \rrbracket \\
\llbracket \textbf{select}\ \{(\vec{\alpha_i}{=}\vec{\delta_i}) \Rightarrow e_i\}_i \rrbracket & = & \llbracket e_1 \rrbracket \\
\llbracket \textbf{mod}\ e \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \textbf{write}(e) \rrbracket & = & \llbracket e \rrbracket \\
\llbracket \textbf{read}\ e_1\ \textbf{as}\ y\ \textbf{in}\ e_2 \rrbracket & = & \textbf{let}\ y = \llbracket e_1 \rrbracket\ \textbf{in}\ \llbracket e_2 \rrbracket
\end{array}
$$

$$
\begin{array}{rcl}
\llbracket n \rrbracket & = & n \\
\llbracket (e_1, e_2) \rrbracket & = & (\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
\llbracket \textbf{fst}\ e \rrbracket & = & \textbf{fst}\ \llbracket e \rrbracket \\
\llbracket \textbf{snd}\ e \rrbracket & = & \textbf{snd}\ \llbracket e \rrbracket \\
\llbracket \textbf{inl}\ e \rrbracket & = & \textbf{inl}\ \llbracket e \rrbracket \\
\llbracket \textbf{inr}\ e \rrbracket & = & \textbf{inr}\ \llbracket e \rrbracket \\
\llbracket \textbf{case}\ e\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \rrbracket & = & \textbf{case}\ \llbracket e \rrbracket\ \textbf{of}\ \{x_1 \Rightarrow \llbracket e_1 \rrbracket\ ,\ x_2 \Rightarrow \llbracket e_2 \rrbracket\} \\
\llbracket \oplus(e_1, e_2) \rrbracket & = & \oplus(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \\
\llbracket \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rrbracket & = & \textbf{let}\ x = \llbracket e_1 \rrbracket\ \textbf{in}\ \llbracket e_2 \rrbracket
\end{array}
$$

Figure 6.5: Back-translation, used for correspondence between target and source dynamic semantics

The details of soundness depend on a simple notion of source equivalence (source terms are equivalent if they either evaluate to the same value, or diverge), and on an ordinary substitution $[\rho]e$ on target terms.

Definition 6.5.1. Source expressions e_1, e_2 are *equivalent*, $e_1 \sim e_2$, iff both evaluate to the same value, or both diverge: if there exists v_1 such that $e_1 \Downarrow v_1$ then $e_2 \Downarrow v_1$, and if there exists v_2 such that $e_2 \Downarrow v_2$ then $e_1 \Downarrow v_2$.

Definition 6.5.2. Given a store ρ, which maps locations ℓ to target terms $\rho(\ell)$, and a target term e, the *store substitution* operation $[\rho]e$, for all $\ell \in \mathsf{dom}(\rho)$, replaces each occurrence of ℓ in e with $[\rho](\rho(\ell))$.

For example, $[\ell_1 \mapsto 1, \ell_2 \mapsto (\ell_1, 2)](\ell_1, \ell_2) = (1, (1, 2))$.

Theorem 6.5.3 (Evaluation Soundness).
If $\rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FLV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select**-uniform*
then $\llbracket[\rho]e\rrbracket \Downarrow \llbracket[\rho']w\rrbracket$.

Theorem 6.5.4 (Translation Soundness).
If $\cdot \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$ *and* $\cdot \vdash e' \Downarrow (\rho' \vdash w)$ *then* $e \Downarrow \llbracket[\rho']w\rrbracket$.

Acar et al. (2006c) proved that given a well-typed AFL program, change propagation updates the output consistently with an initial run. Using Theorems 6.4.1 and 6.5.4, this implies that change propagation is consistent with an initial run of the source program.

6.6 Cost of translated code

Our last main result extends Theorem 6.5.4, showing that the size $W(\mathcal{D})$ of the derivation of the target-language evaluation$\cdot \vdash e' \Downarrow (\rho \vdash w)$ is asymptotically the same as the size $W(\mathcal{D}')$ of the derivation of the source-language evaluation[2], $\llbracket e'\rrbracket \Downarrow \llbracket[\rho]w\rrbracket$. To prove Theorem 6.6.9, the extended version of Theorem 6.5.4, we need a few definitions and intermediate results. The proof hinges on classifying keywords added by the translation, such as **write**, as "dirty": a dirty keyword leads to applications of the dirty rule (TEvWrite) in the evaluation derivation; such applications have no equivalent in the source-language evaluation.

We then define the "head cost" *HC* of terms and derivations, which counts the number of dirty rules applied near the root of the term, or the root of the derivation, without passing through clean parts of the term or derivation. Just counting *all* the dirty keywords in a term would not rule out a β-reduction duplicating a particularly dirty part of the term. By defining head cost and proving that the translation generates terms with bounded head cost—including for all subterms—we ensure that *no* part of the term is too dirty; consequently, substituting a subterm during evaluation yields terms that are not too dirty.

The omitted proofs can be found in Appendix A.3.

To extend the evaluation soundness result above (Theorem 6.5.3) with a guarantee that the evaluation derivation $\mathcal{D}$ is not too large—within a constant factor of the source evaluation derivation $\mathcal{D}'$—we need several definitions:

Definition 6.6.1. The *weight* $W(\mathcal{D})$ of a derivation $\mathcal{D}$ is the number of rule applications (that is, the number of horizontal lines) in $\mathcal{D}$.

Next, we define the "head cost" of a derivation. This measures the overhead introduced by translation, in the part of the derivation that is *near its conclusion* (the root of the derivation tree). To measure the overhead, we count the number of "dirty" rules applied near the root.

[2] As we mentioned in Section 6.5, the back-translation $\llbracket e'\rrbracket$ is not exactly the same as the source program e: it may be **let**-expanded. We are, therefore, relying on the property that **let**-expansion preserves asymptotic complexity (since the resulting evaluation will be larger by, at worst, a constant factor). Since we assume source programs are in A-normal form, however, we already need that property.

Definition 6.6.2. Rules (TEvValue), (TEvPair), (TEvSumLeft), (TEvPrimop), (TEvCase), (TEvFst), and (TEvApply) are clean. Rule (TEvLet) is clean, since each **let** in the target expression becomes a **let** in the back-translation. The rules (TEvWrite), (TEvMod), (TEvRead) and (TEvSelect) are dirty.

Definition 6.6.3. The *head cost* $HC(\mathcal{D})$ of a derivation $\mathcal{D}$ is the number of dirty rule applications reachable from the root of $\mathcal{D}$ without passing through any clean rule applications.

Definition 6.6.4. The *head cost* $HC(e)$ of a term e is defined in Figure 6.6.

Definition 6.6.5. A term e is shallowly k-bounded if $HC(e) \leq k$.

A term e is deeply k-bounded if every subterm of e (including e itself) is shallowly k-bounded.

Similarly, a derivation $\mathcal{D}$ is shallowly k-bounded if $HC(\mathcal{D}) \leq k$, and deeply k-bounded if all its subderivations are shallowly k-bounded.

$$
\begin{array}{rcl}
HC(x) & = & 0 \\
HC(x[\vec{\alpha} = \vec{\delta}]) & = & 1 \\
HC((\mathbf{select}\,\{\vec{\alpha_i} = \vec{\delta_i} \Rightarrow e_i\}_i)[\vec{\alpha} = \vec{\delta}]) & = & 1 + \max_i(HC(e_i)) \\
HC(\mathbf{select}\,\{\vec{\alpha_i} = \vec{\delta_i} \Rightarrow e_i\}_i) & = & 0 \\
HC(n) & = & 0 \\
HC((e_1, e_2)) & = & 0 \\
HC(\mathbf{inl}\ e) & = & 0 \\
HC(\mathbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon}) & = & 0 \\
HC(\oplus(e_1, e_2)) & = & 0 \\
HC(\mathbf{fst}\ e) & = & 0 \\
HC(\mathbf{apply}^{\varepsilon}(e_1, e_2)) & = & 0 \\
HC(\mathbf{case}\ e\ \mathbf{of}\,\{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}) & = & 0 \\
HC(\mathbf{let}\ x = e_1^{\mathbb{S}}\ \mathbf{in}\ e_2) & = & 0 \\
HC(\mathbf{mod}\ e^{\mathbb{C}}) & = & 1 + HC(e^{\mathbb{C}}) \\
HC(\mathbf{write}(e)) & = & 1 + HC(e) \\
HC(\mathbf{read}\ e_1\ \mathbf{as}\ y\ \mathbf{in}\ e_2^{\mathbb{C}}) & = & \begin{cases} 1 + HC(e_1) + HC(e_2^{\mathbb{C}}) & \text{if (for } y \text{ not free in } e_3,\, e_4\text{):} \\ & \quad e_2^{\mathbb{C}} \text{ has the form } \mathbf{apply}^{\epsilon}(y, e_3) \\ & \quad \text{or } \mathbf{case}\ y\ \mathbf{of}\,\{x_1 \Rightarrow e_3\ ,\ x_2 \Rightarrow e_4\} \\ & \quad \text{or } \mathbf{let}\ r = \oplus(e_3, y)\ \mathbf{in}\ \mathbf{write}(r) \\ & \quad \text{or } \mathbf{read}\ e_2'\ \mathbf{as}\ y_2\ \mathbf{in} \\ & \qquad \mathbf{let}\ r = \oplus(y, y_2)\ \mathbf{in}\ \mathbf{write}(r) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6.6: Definition of the "head cost" $HC(e)$ of a target expression e.

Theorem 6.6.6. *If* `trans` $(e, \epsilon) = e'$ *then* e' *is deeply* 1*-bounded.*

Theorem 6.6.7 (Cost Result)**.** *Given* $\mathcal{D} :: \rho \vdash e' \Downarrow (\rho' \vdash w)$ *where for every subderivation* $\mathcal{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathcal{D}$ *(including* $\mathcal{D}$*),* $HC(\mathcal{D}^*) \leq k$*, then the number of dirty rule applications in* $\mathcal{D}$ *is at most* $\frac{k}{k+1} W(\mathcal{D})$.

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

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

Figure 6.7: Translations $\|\tau\|$ of labeled types and typing environments

The extended soundness result, Theorem 6.6.9 below, will follow from Theorem 6.6.8, which generalizes Theorem 6.5.3, and Theorem 6.6.7. The parts that differ from the uncosted result (Theorem 6.5.3) are shaded.

Theorem 6.6.8 (Costed Evaluation Soundness)**.**
If $\mathcal{D} :: \rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FLV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select**-uniform and* $[\rho]e$ *is deeply* k*-bounded*
then $\mathcal{D}' :: \llbracket[\rho]e\rrbracket \Downarrow \llbracket[\rho']w\rrbracket$
and $[\rho']w$ *is deeply* k*-bounded*
and for every subderivation $\mathcal{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathcal{D}$ *(including* $\mathcal{D}$*),*
$\quad HC(\mathcal{D}^*) \leq HC(e^*) \leq k$*,*
and the number of clean rule applications in $\mathcal{D}$ *equals* $W(\mathcal{D}')$*.*

Theorem 6.6.9. *If* `trans` $(e, \varepsilon) = e'$ *and* $\mathcal{D}' :: \cdot \vdash e' \Downarrow (\rho' \vdash w)$*, then* $\mathcal{D} :: \llbracket e'\rrbracket \Downarrow v$ *where* $W(\mathcal{D}') = \Theta(W(\mathcal{D}))$*.*

6.7 Translation for destination passing style

This section gives an overview of the translation from the source language with precise dependency tracking (Section 4.2.2) to the imperative target self-adjusting language (Section 5.2). To ensure type safety, we translate types and expressions together using a type-directed translation.

Translating types. Figure 6.7 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 Γ. 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 Γ. 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 τ **mod**. Here, we define a *stabilization* function $\left|\tau\right|^{\mathbb{S}}$ for changeable source types, which changes the outer level of τ from changeable into stable. Formally, we define the function as,

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

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

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 ϕ for C, it is always possible to produce a correctly-typed target expression e_t (see Theorem 6.4.1 below). The environment Γ in the translation rules is a source-typing environment and must have no free level variables. Given an environment Γ from the constraint typing, we apply the satisfying assignment ϕ to eliminate its free level variables before using it in the translation $[\phi]\Gamma$. With the environment closed, we need not refer to C.

Figure 6.8 shows the translation rules for destination passing style. Most rules are the same as the discussed in Section 6.2. Here we highlight the rules that are different from the previous sections.

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_ρ.

For expressions with non-destination changeable levels, that is the label ρ has a 1 as the prefix, we need to create a modifiable first. Rules (Lift) and (Mod) achieves this goal. In (Mod), if e translates to e' at type τ, then e translates to the **mod** expression at type τ. To get an initial value for the modifiable, we define a function $\tau|_v$ that takes a source type τ 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 τ but at its *stabilized* $\left|\tau\right|^{\mathbb{S}}$, capturing the "shallow subsumption" in the constraint typing rules (SCLet).

Reading from changeable data. The (Read) rule generates a target expression that reads the value out of $x : \textbf{int}^{\mathbb{C}}$ into a variable $x' : \textbf{int}^{\mathbb{S}}$. The variable-renaming judgment $\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau \vdash e')$ is shown in Figure 6.3, extended with one more rule for variables.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6.8: Translation for destination passing style

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

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

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \rightsquigarrow e_1'}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \tau \rightsquigarrow e_1'} \ \text{(RCase)} \qquad \frac{[\![\tau]\!] = \mathbb{C}_\rho}{\Gamma \vdash e : \tau \rightsquigarrow l_\rho} \ \text{(RMod)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau' \underset{0}{\hookrightarrow} e_1' \\ \Gamma \vdash e_2 : \tau' \underset{0}{\hookrightarrow} e_2' \qquad \Gamma, x : \tau' \vdash e_2 : \tau \rightsquigarrow \mathrm{ret} \qquad e_2 \neq \textbf{let}\ x' = e_1'\ \textbf{in}\ e_2'\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau \rightsquigarrow \textbf{let}\ x = e_1'\ \textbf{in} \\ \qquad\qquad\qquad\qquad\qquad\quad\ \textbf{let}\ \_ = e_2'\ \textbf{in}\ \mathrm{ret}\end{array}} \ \text{(RLet)}$$

Figure 6.9: Deriving destination return

Function and application rules. Since the self-adjusting primitives 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 6.9 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_ρ. 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 6.2. For example, expression $(1, x) : \left(\textbf{int}^{\mathbb{S}} \times \textbf{int}^{\mathbb{C}_{01}}\right)^{\mathbb{S}}$ will be translated to $(1, l_{01})$ by applying rules (RProd) (RTrans) (Int) (RMod).

When applying functions $\textbf{apply}(x_1, x_2)$, 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** $\textbf{write}(l_\rho, x')$, 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 ϕ 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 6.7.1. *If* $C;\mathcal{P};\Gamma \vdash e : \tau$*, and* ϕ *is a satisfying assignment for* C*, then*

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

Chapter 7

Compiler Design and Implementation

This chapter is based on work on the empirical evaluation of implicit self-adjusting computation (Chen et al. 2012), and a runtime extension for granularity control for self-adjusting computation (Chen et al. 2014a).

To support type-directed, automatic incrementalization, we extended Standard ML with a single keyword `$C`, a type qualifier, and extended the compiler for Standard ML to generate self-adjusting executables. This extension to SML does not restrict the language in any way, allowing all its features to be used freely, including the module language. In this chapter, we describe the structure of our system, and discuss some of its key components in more detail.

7.1 Structure of the compiler

The overall structure of the compiler is in Figure 7.1. The front-end of the compiler is implemented as a source-to-source translation based on HaMLet (HaMLet). The front-end takes the annotated ML program, performs type inference to track the dependency, and generates self-adjusting programs based on the type information. The code generation for the front-end follows a deterministic translation algorithm extracted from the proof of the type soundness theorem (Figure 6.4), thus guaranteeing the correctness of our implementation.

The back-end of the compiler extends MLton (MLton) to compile and optimize the generated self-adjusting programs together with a self-adjusting run-time library. This library implements the self-adjusting primitive (`Mod`, `Read`, `Write`) used by the translated code. When executed, the library constructs a dependency graph during the initial run. The library also provides facilities for changing the input and invoking change propagation to reflect changes to the output.

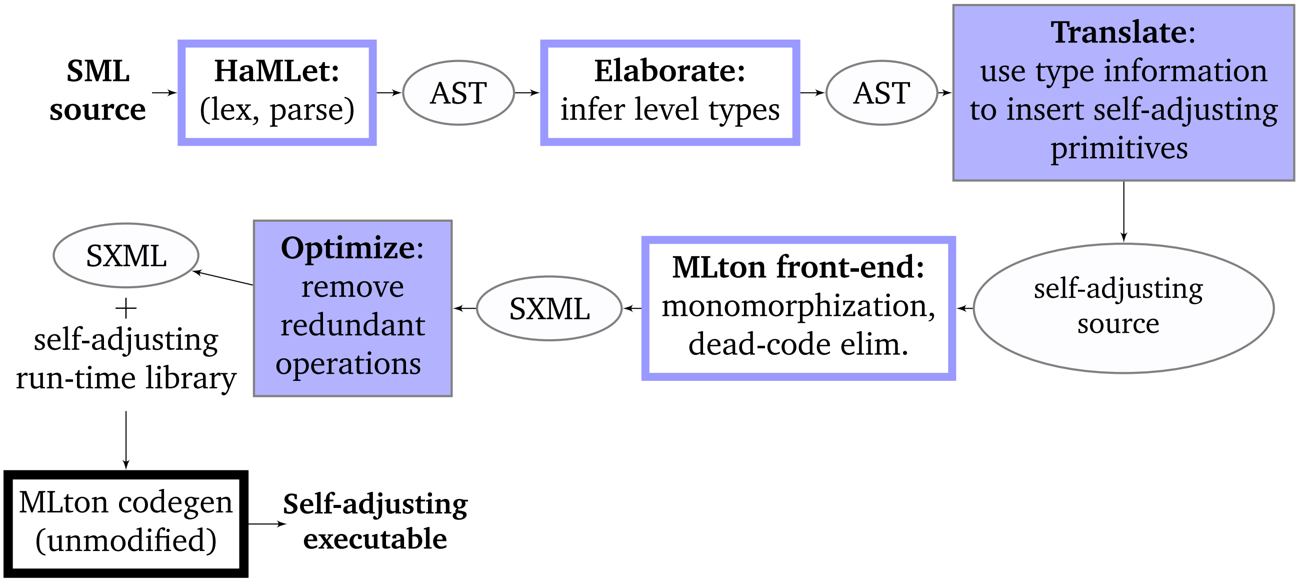

SML source
HaMLet: (lex, parse)
AST
Elaborate: infer level types
AST
Translate: use type information to insert self-adjusting primitives
self-adjusting source
MLton front-end: monomorphization, dead-code elim.
SXML
Optimize: remove redundant operations
SXML
+
self-adjusting run-time library
MLton codegen (unmodified)
Self-adjusting executable

Figure 7.1: The structure of our compiler: new phases, modified phases, unmodified phases, ILs

```
  level := . | $S | $C
     ty := tyvar
         | tycon level
         | ty1 → level ty2
         | { label : ty, ... } level
         | ty1 * ty2 * ... * tyk # level
         | ( ty )
datbind := tycon = level conbind
```

Figure 7.2: Syntax extension for Standard ML

7.2 Front end

The front end of the compiler is implemented based on HaMLet 1.3.1. One nice feature about HaMLet is that it is a direct and faithful implementation of Standard ML. Type inference is performed directly on the abstract syntax tree (AST). This simplicity makes type system extension relatively easy to implement than other ML systems.

7.2.1 Syntax

We extended Standard ML's lexer and parser to handle types with `$C` annotations, producing abstract syntax in which types include level information. The concrete extension of the syntax is shown in Figure 7.2. For function types, record types and product types, we extend the types with level annotations $C or $S, indicating whether the types contain changeable values or not. When declaring datatypes, each type constructor is annotated with a level. For example, we can define a list with changeable tails as,

```
datatype α list = $C
   Nil | Cons of α * α list $C
```

The $C annotation on the second line specifies that the tail of the list is changeable, and the $C annotation on the first line indicates that the whole list is changeable, and it will be put into a modifiable when translated into self-adjusting programs.

7.2.2 Type inference

We use a two-phase approach in the type inference. The first phase conducts the original ML type inference, ignoring all the level annotations; and the second phase infers levels based on the result of the first phase.

There are several changes we need to make to the original type inference algorithm in order to make the two-phase approach work.

- **Storing type information.** To correctly infer levels in the second phase, we need to remember the type information for every AST node. However, HaMLet uses environment to record type information during type inference. This means that when an AST node is out of scope, the type information for that AST node is also lost. So the first change we made is to keep type information for every AST, by attaching a property list for each AST node. We also need to store the def-use information, as in the second phase, for polymorphic types, we need to instantiate concrete types from their definitions. This is achieved by adding a property list in the value binding environment (VE) and the type constructor environment (TE).
- **Node sharing.** Certain AST nodes in HaMLet are shared in the abstract syntax tree. Sharing is fine when the type inference is traversing the syntax tree only once. For the level inference to perform correctly, we need to deep copy certain AST nodes. There are two kinds of sharing in HaMLet. First, the type inference uses an imperative unification algorithm. This means some of the type information is shared among different AST nodes. This can generate incorrect result in the second phase of the type inference. That is, when we assign a level to a shared type node, we are actually assigning one level to multiple places, which is not desired. To solve this problem, after the first phase of type inference, we perform a deep copy for every type; Second, in the rewriting of derived forms, some AST nodes are getting reused. Due to polymorphism, these shared nodes may have different level types. So we need to copy the property list after rewriting of derived forms. Third, type nodes are shared for the same type variable. This is fine, because when instantiating types, we do need the type nodes to share, when they corresponds to the same type variable.
- **Generalization.** HaMLet generalizes type variables in the binding environment (closures). When a type gets generalized in the environment, the type stored in the

AST node is still the original one. This becomes a problem especially for recursive functions, as we may lose some of the typing constraints. The current approach to solve this problem is to simply elaborate the value binding again after the type gets generalized to make sure the new type information is updated in AST node.

The second phase of the type inference follows the constraint-based typing rules in Figure 4.2. The constraints C are collected during the type checking. At the start of the type checking, C is empty (equivalently, is true); as the typechecker traverses the program, C is extended with additional constraints. For example, the premise $C \Vdash \delta \leq \varepsilon$ in (SFst) really corresponds to adding $\delta \leq \varepsilon$ to the "current" C, not to checking $\delta \leq \varepsilon$ under a known constraint. At the end of the type checking, the constraints C are feed into a constraint solver that inferences principal typings that are "minimally" changeable, that is given the choice between assigning $\mathbb{S}$ or $\mathbb{C}$ to some level variables, the solver will prefer $\mathbb{S}$. Thus, data and computations will only be made changeable—and incur tracking overhead—where necessary to satisfy the programmer's annotation.

7.2.3 Self-adjusting translation

Our translation algorithm extends the algorithm in Section 6.3 to support full SML, including (recursive) data types and imperative references. The translation algorithm takes SML code and transforms it into SML code containing self-adjusting computation primitives, whose implementations will be supplied by the run-time system. The self-adjusting primitives include **mod**, **read**, and **write** functions for creating, reading from, and writing to modifiable references. At a high level, the translation rules inspect the code locally, insert **read**s where changeable data is used (according to type information), and ensure that each **read** takes place within the dynamic scope of a call to **mod**. To ensure this and other correctness properties, the rules distinguish stable and changeable modes.

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

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

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

Figure 7.3: Translation rules for mutable references

Figure 7.3 shows the translation rules for mutable references, which we translate to modifiables. The translation judgment $\Gamma \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$ is read "in environment Γ and

$$\begin{array}{rcll}
\textbf{read}\ (\textbf{mod}\ (\textbf{let}\ r = e_1\ \textbf{in}\ \textbf{write}(r))) & & & \\
\textbf{as}\ x'\ \textbf{in}\ e_2 & \longrightarrow & \textbf{let}\ x' = e_1\ \textbf{in}\ e_2 & (1) \\
\textbf{read}\ (\textbf{mod}\ e)\ \textbf{as}\ x'\ \textbf{in}\ \textbf{write}(x') & \longrightarrow & e & (2) \\
\textbf{mod}\ (\textbf{read}\ e\ \textbf{as}\ x'\ \textbf{in}\ \textbf{write}(x')) & \longrightarrow & e & (3)
\end{array}$$

Figure 7.4: Optimization rules

mode ε, source expression e at type τ translates to e'”. In stable mode $\mathbb{S}$, the translation produces stable code that cannot inspect changeable data or directly use changeable code; in changeable mode $\mathbb{C}$, the translation produces changeable code that can appear within the body of a **read** and can manipulate references. For translation of imperative references, we add another primitive **impwrite** that updates the value of a modifiable directly.

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

7.3 Back end

The back end of the compiler is implemented based on MLton. It takes the SML code generated from the front end, conducts various optimizations, and links the code with a self-adjusting run-time library to generate a self-adjusting executable. MLton's whole program compilation strategy makes optimization relatively easy to implement, producing efficient self-adjusting programs. Besides, MLton's support for equality testing, including polymorphic equality and pointer equality, is critical for building the self-adjusting runtime.

7.3.1 Optimization

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

Such redundancies turn out to be common, because of the local nature of the translation algorithm. We therefore developed a post-translation optimization phase to eliminate redundant operations. Figure 7.4 illustrates the rules that drive the optimization phase. Each rule eliminates three operations: reading from a modifiable, writing to a modifiable, and creating a modifiable. As we show in Section 8.1.7, this optimization phase reduces the execution time for self-adjusting programs by up to 60%.

- **Eliminating write-create-read.** The left-hand side of rewrite rule (1) evaluates an expression e_1 into a new modifiable, then immediately reads the contents of the modifiable into x'. The right-hand side evaluates e_1 and binds the result to x' with no extra modifiable.
- **Eliminating create-read-write.** The left-hand side of (2) evaluates e (which, since it is the body of a **mod**, must end in a **write**), creates a modifiable, reads the just-written value into x', and writes it again. The right-hand side just evaluates e.
- **Eliminating read-write-create.** Rule (3) is similar to rule (2): the left-hand side reads some modifiable e into a variable x', and immediately writes x' back to a new modifiable; the right-hand side only evaluates e.

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

Theorem 7.3.1. *Rules (1)–(3) are locally confluent.*

Proof

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

$$\begin{array}{ll} & \underline{\mathbf{read}\ (\mathbf{mod}\ (\mathbf{read}\ e_3\ \mathbf{as}\ x'_3\ \mathbf{in}\ x'_3))\ \mathbf{as}\ x'_2\ \mathbf{in}\ \mathbf{write}(x'_2)} \\ \xrightarrow{(2)} & \underline{\mathbf{read}\ e_3\ \mathbf{as}\ x'_3\ \mathbf{in}\ \mathbf{write}(x'_3)} \\ & \mathbf{read}\ (\underline{\mathbf{mod}\ (\mathbf{read}\ e_3\ \mathbf{as}\ x'_3\ \mathbf{in}\ x'_3)})\ \mathbf{as}\ x'_2\ \mathbf{in}\ \mathbf{write}(x'_2) \\ \xrightarrow{(3)} & \mathbf{read}\ \underline{e_3}\ \mathbf{as}\ x'_2\ \mathbf{in}\ \mathbf{write}(x'_2) \end{array}$$

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

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

```
signature COMBINATORS = sig
  eqtype α modref
  type α cc
  (* Core *)
  val mod : α cc → α modref
  val write : (α * α → bool) → α → α cc
  val read : β modref * (β → α cc) → α cc
  (* Meta *)
  val change : (α * α → bool) → α modref → α → unit
  val deref : α modref → α
  val propagate : unit → unit
end
```

Figure 7.5: Signatures for AFL runtime library

We apply our optimization on MLton's SXML intermediate language, a monomorphic subset of SML in A-normal form. Before SXML, MLton has conducted various code transformations to simplify the program, including elaboration, defunctorization, linearization (conversion to A-normal form), dead code elimination, and monomorphization, etc. We mark the self-adjusting primitives as special keywords, so that self-adjusting primitives becomes intact during these code transformations, and we can apply optimization rules in SXML as a phase of optimization.

7.4 Runtime environment

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

7.4.1 Library interface

We implement the target language AFL presented in Section 5.1 as combinator library. Figure 7.5 shows the interface to the library. The `COMBINATORS` module defines modifiable references and changeable computations. Every execution of a changeable computation of type α cc starts with the creation of a fresh modifiable of type α modref. The modifiable is written at the end of the computation. For the duration of the execution, the reference never becomes explicit. Instead, it is carried "behind the scenes" in a way that is strongly reminiscent of a monadic computation. Any non-trivial changeable computation reads one or more other modifiables and performs calculations based on the values read. Values of type α cc representing changeable computations are constructed using `write`, `read`. The `mod` function executes a given computation on a freshly generated modifiable before returning that modifiable as its result. The `write` function creates a trivial computation which merely writes the given value into the underlying modifiable. To avoid unnecessary propagation, old and new values are compared for equality at the time of write using the equality function provided. The `read` combinator takes an existing modifiable reference together with a receiver for the value read. The result of the read combinator is a computation that encompasses the process of reading from the modifiable, a calculation that is based on the resulting value, and a continuation represented by another changeable computation. Calculation and continuation are given by body and result of the receiver.

The library also supplies meta operations for inspecting and changing the values stored in modifiables and performing change propagation. The `change` function is similar to `write` function. It changes the underlying value of the modifiable to a new value—this is implemented as a side effect. The `propagate` function runs the change-propagation algorithm. Change propagation updates a computation based on the changes issued since the last execution or the last change propagation. The meta operations should only be used at the top level—the library guarantees correct behavior only in the cases that meta operations are not used inside the program. Our compiler generates programs using the core operations, such as `mod`, `read`, and `write`. The programmer has to write the meta operations at the top level to specify how the changes are made to the data.

7.4.2 Memoization

Change propagation detects changes to data so that the computations that process those data can be re-executed. But re-executing the entire remainder of a changeable computation starting at an affected read can be very inefficient as significant portions of it might not truly depend on the changed value. We use *memoization* of computations to detect such situations dynamically: when a function is called with the exact same arguments as during the previous execution, then the part of the computation corresponding to this function call can be re-used. However, since a memoized computation might contain read operations that are affected by pending changes, one must propagate these changes into the result of a successful memo lookup.

```
val mkLift : (α * α → bool) → (index list * α) →
             (α mod → β) → β
```

In the library, we provide function `mkLift` to handle adaptive memoization, i.e., memoization based on partially matching function arguments. With ordinary memoization, a memo table lookup for a function call will fail whenever there is no precise match for the entire argument list. The idea behind adaptive memoization is to distinguish between strict and non-strict arguments and base memoization on strict arguments only. By storing computations (as opposed to mere return values) in the memo table, a successful lookup can then be adjusted to any changes in non-strict arguments using the change-propagation machinery. Since change propagation relies on read operations on modifiables, the memoized function has to access its non-strict arguments via such modifiables. The memo table, indexed by just the strict part of the original argument list, remembers the modifiables set aside for non-strict arguments as well as the memoized computation.

Given the strict part of the argument list, a *lift operation* maps a function of type $\alpha\ \textbf{mod} \rightarrow \beta$ to a function of type $\alpha \rightarrow \beta$ where α is the type of the non-strict argument. Our `mkLift` combinators create lift operations for ordinary and changeable computations from appropriately chosen equality predicates for the types involved. The strict part of the argument list is represented by an index list, assuming a one-to-one mapping between values and indices.

When its memo table lookup fails, a lifted function creates fresh modifiables containing its non-strict arguments, executes its body, and stores both the computation and the modifiables into the memo table. Computations are memoized by recording their return value and a representation of their dynamic dependence graph. A successful memo lookup finds modifiables and a computation. The lifted function then writes its current non-strict arguments into their respective modifiables, and lets change propagation adjust the computation to the resulting changes.

To support adaptive memoization at the source language, we extend the language with `mfun` and `mfn` keywords to represent memoized functions in the following syntax.

```
mfun f [strict_args] args = e
```

Semantically, **mfun** `f [strict_args] args = e` is equivalent to **fun** `f args = e`, with the additional feature that the function will return the memoized result when the arguments `strict_args` match the memo table. Our compiler translates `mfun` function into the following code,

```
fun f args =
  let
    val lift = mkLift (MLton.eq)
    fun f_memo args =
      lift (strict_args, args \ strict_args) (fn margs ⇒ e’)
  in
    f_memo args
  end
```

where e' is derived by

$$\frac{\Gamma(f) = (\tau \underset{\varepsilon}{\rightarrow} \tau')^{\mathbb{S}} \qquad \forall arg_i \in args \setminus strict_{args}, \Gamma(arg_i) = \tau_i \qquad \|\beta_i\| = \|\tau_i\|\ \textbf{mod}}{\Gamma, f_{memo} : (\tau \underset{\varepsilon}{\rightarrow} \tau')^{\mathbb{S}}, marg_i : \beta_i \vdash [f_{memo}/f, marg_i/arg_i]e : \tau' \underset{\mathbb{S}}{\hookrightarrow} e'}$$

It is the responsibility of the programmer to ensure that all memoized functions specify the strict and non-strict variables accurately. The classification of a variable as strict or non-strict does not affect correctness but just performance.

7.5 Probabilistic chunking

Given the self-adjusting runtime library, we can write various data structures that can respond to data changes efficiently. But the dependency metadata the runtime maintains can be very large, preventing scaling to large datasets. In this section, we show how to reduce the size of dependency metadata 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*?

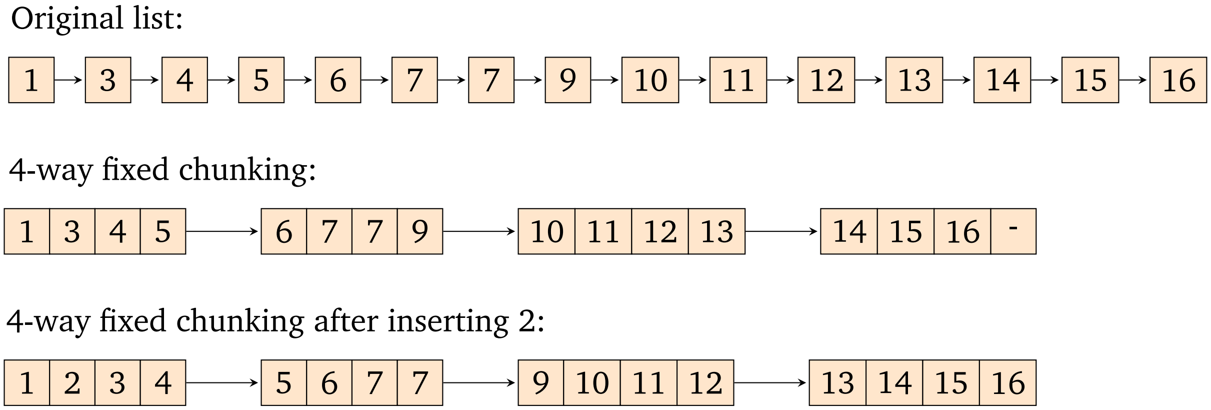
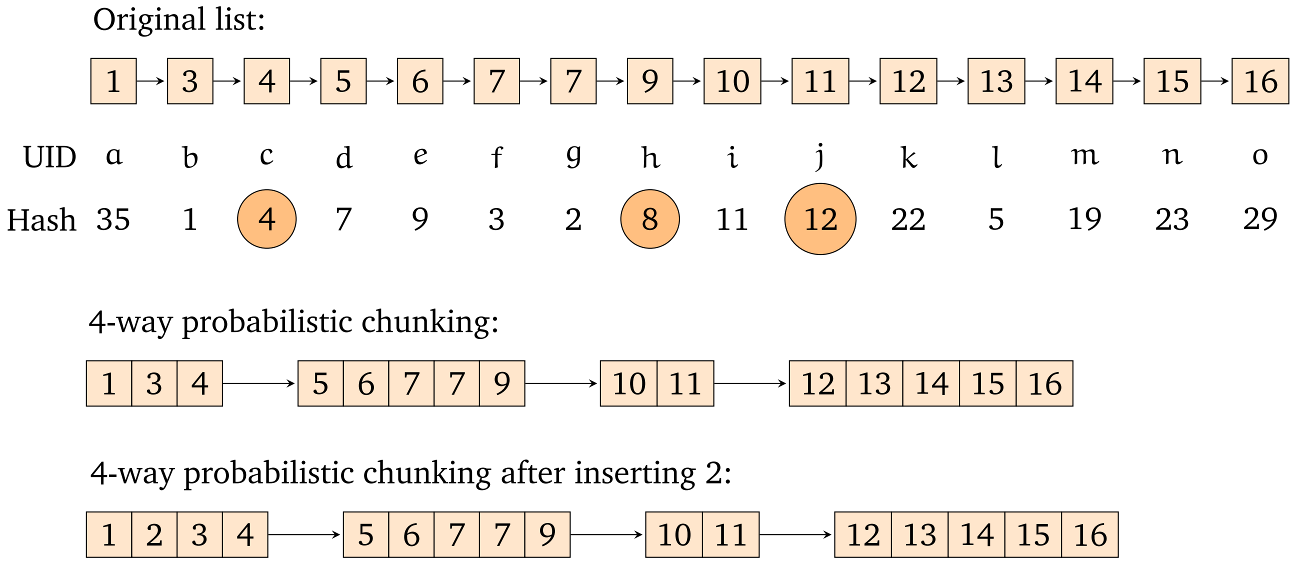


Figure 7.6: Fixed-size chunking with block size $B = 4$.

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 7.6, 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 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., (Muthitacharoen et al. 2001; Tangwongsan et al. 2010) 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 7.7 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 in the figure; hash values divisible by 4 are marked with a circle. PCS declares block boundaries where the hash value is $0 \bmod B = 4$, thereby selecting 1 in 4 elements to be on the boundary. This means finishing the blocks at 4, 9, and 11, as shown.



UID	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Hash	35	1	4	7	9	3	2	8	11	12	22	5	19	23	29

Figure 7.7: Probabilistic chunking with block size $B = 4$. Each data cell in the original list (top) has a unique identifier (location). The hash values of these identifiers are shown in the table, with values divisible by $B = 4$ marked with a circle.

To understand what happens when the input changes, consider inserting 2 (with a unique location identifier p) between 1 and 3. When the hash value of p is not divisible by B, 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 $\llbracket 1, 3, 4 \rrbracket$, 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.

This scheme has good block stability because adding or deleting an element does not change the block structure unless it is at the chunk boundary, which happens rather infrequently (with probability $1/B$). Besides block stability, PCS has other desirable properties: to further analyze this, we make a standard assumption that the hash values are independent. Hence, each element can be seen as flipping an independent coin with bias $p = 1/B$. Using standard results in probability theory, we have the following theorem:

Theorem 7.5.1. *Let* B *be a target block size. On a sequence of* n *elements, the probabilistic chunking scheme (PCS) yields expected* n/B *blocks, each with size* $\Theta(\mathrm{B})$ *in expectation.*

Furthermore, the target block size B can be set for each sequence, allowing programmers to tune the tradeoffs between time and space.

Understanding performance implications. Because the basic changeable unit is now a block, any change to a block—no matter how small or large—will cause the computation that depends on that block to rerun, increasing the amount of recomputation in return for space reduction. *How does this affect the update time performance?*

We pick list partition and merge sort as examples to study the effects of PCS and the block sequence abstraction. In concert with our intuition, we show that using a target block size B will cost $\Theta(\mathrm{B})$ more work in expectation. Like before, the update performance of an algorithm can be understood by analyzing the static algorithm's trace stability—how much the trace changes as a result of an update. The trace model remains unchanged except that the trace only remembers blocks of elements instead of individual elements.

List partitioning. We analyze the algorithm in Figure 2.7, which implements the list partition routine using the block list abstraction. We consider the case of inserting a new element; deletion is symmetric. To analyze the stability, we focus on the trace difference on `partition` (the static partition code); other trace differences can be charged to them, increasing the total cost by at most a constant factor. Because a single insertion can affect at most two input blocks (the worst case is when it splits an existing block), the trace difference due to `partition` is $\Theta(1)$. Therefore, the trace difference is bounded by a constant, which translates to $\Theta(\mathrm{B})$ expected time because each static function takes expected $\Theta(\mathrm{B})$ time and the priority queue size is constant. We have the following theorem:

Theorem 7.5.2. *The sequence operation* `partition` *is* $\Theta(1)$*-stable with respect to a single insertion/deletion. Thus, an update costs expected* $\Theta(\mathrm{B})$ *time.*

Merge sort. We sketch an analysis for merge sort (`msort`). We consider inserting a new element; again, deletion is symmetric. The crux of a stable `msort` is a stable `merge` routine: as long as the split is reasonably balanced and the merge is $\Theta(1)$-stable per level, the recursion tree's depth is $\Theta(\log n)$ and the overall trace difference is $\Theta(\log n)$. We therefore focus our discussion on `merge`: Whereas a traditional merge performs comparisons on the elements itself, a block merge algorithm builds on a static routine, which we call `smerge(a, b)` that takes two sorted (ordinary) list (`a` and `b`) and produces a combined sorted list, as well as the remainders from one of the input lists. The dominant cost in `merge`'s trace difference comes from `smerge`.

In general, `merge` is unstable: an element may be compared against a large number of elements. In merge sort, however, we can keep this number small: when splitting a sequence, we flip an unbiased coin for each element (e.g., using a hash function on the box id) to decide which sublist it goes into. By an argument similar to that of Acar (Acar 2005), we know that an element is compared against at most 2 blocks in expectation.

```
signature BLOCK_SEQUENCE =
sig
  datatype α bseq = NIL
                 | BLOCK of (α * α bseq mod)
  type α block = α Box.t list
  type α t = α block bseq mod
  val mkCons : α block * α t → α t
end
```

Figure 7.8: Blocked sequence signature.

As a result, the trace difference due to smerge is $\Theta(1)$—and the trace difference for a single merge is $\Theta(1)$ (assuming the random split we use in our msort). This means the propagation queue is $\Theta(1)$ in expectation throughout. Hence, msort is $\Theta(\log n)$-stable overall. Because each smerge costs expected $\Theta(B)$ time, we have the following theorem:

Theorem 7.5.3. *On input of size n, the merge sort algorithm is $\Theta(\log n)$-stable with respect to a single insertion/deletion. Thus, an update costs expected $\Theta(B \log n)$ time.*

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.

Working with block sequences. The block sequence abstraction is similar to the list abstraction in Standard ML, except that each unit of a sequence is a block, which is internally represented as a boxed (ordinary) list. There is a function mkCons that takes an (ordinary) list of elements and inserts them into a block sequence, ensuring that they are chunked according to PCS.

To satisfy correctness guarantees, deep equality test is needed to perform memoization on block sequences. Since testing blocks for equality takes linear time, we can optimize for the common case: create a “fingerprint” of the block using a hash function and compare the blocks when the fingerprints match.

There are a number of common patterns that help with programming in this abstraction, which essentially alternates between stable and changeable computations. First, within a stable computation block, memory allocation should be made deterministic to avoid unnecessary recomputation

Figure 7.8 shows the signature for blocked sequence abstraction. Similar to the modifiable list data structure, the tail of the block sequence is a modifiable, which allows us to insert and delete the sequence at the block level. A block is an ordinary list with each elements boxed with a unique id. The function mkCons takes a block and inserts it into a block sequence, ensuring that each block ends with hash value 0 with probability $1/B$.

Programming in this interface requires some care to derive efficient results. We need to determinize the memory allocation within the stable block computation. Otherwise, the change can propagate to the rest of the block sequence even if the result is the same. We discover two ways that are effective for solving this problem: 1) programming in destination-passing style preallocates the modifiables, making the program more stable; 2) When memoizing the self-adjusting block sequence computation, we need to use the deep equality test for the stable block, instead of pointer equality. The deep equality takes $\Theta(B)$ time in expectation.[1] Note that we can decide different block sizes for different input sizes to adjust the time and space trade-off. As long as the hash function stays the same for each function, we can derive efficient update time.

[1] An alternative way is to memoize at the level of each element in the block, but then the memo table will dominate the space, thus ruined the whole purpose of blocking.

Chapter 8

Empirical Evaluation

This chapter is based on work on the empirical evaluation of implicit self-adjusting computation (Chen et al. 2012), and a language extension for computing with large and dynamic data in self-adjusting computation (Chen et al. 2014a).

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. 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 Experiments with automatic incrementalization

8.1.1 Benchmarks

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

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

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

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

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

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

8.1.2 Experimental setup

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

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

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

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

8.1.3 Timings summary

Table 8.1 shows a summary of the timings that we collected for our benchmarks at fixed input sizes (written in parentheses after the benchmark's name). All times are reported in seconds. The first column ("Conv. Run") shows the run time of the conventional (reference) implementation with an input of specified size. The conventional version cannot self-adjust, but does not incur the overhead of trace construction as self-adjusting versions do. The second column ("Self-Adj. Run") shows the run time of the self-adjusting version with an input of specified size. Such a self-adjusting run constructs a trace as it executes, which can then be used to respond automatically to incremental changes to data via change propagation. The third column ("Self-Adj. Avg. Prop.") shows the average time for a change propagation after a small change to the input (as described in Section 8.1.1, the specific nature of the changes depend on the application).

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

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

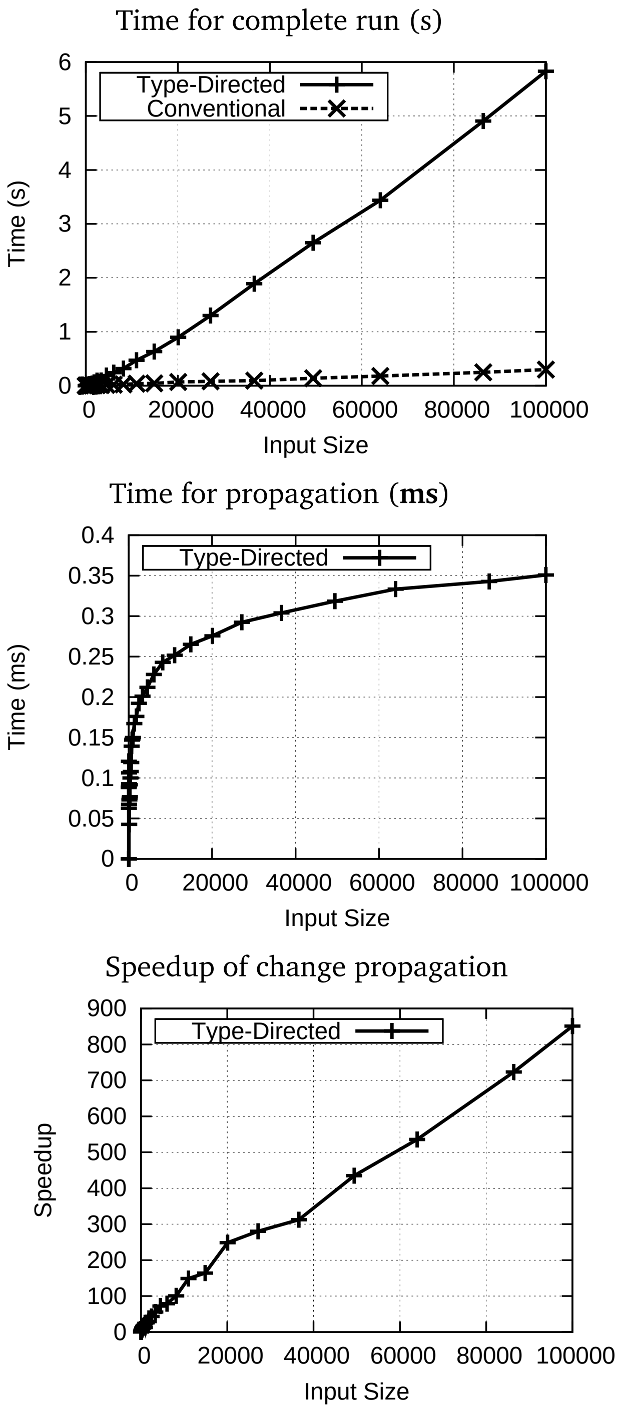
Table 8.1: Summary of benchmark timings.

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

8.1.4 Merge sort

Although it is not apparent from the summary in Table 8.1, our experiments show that for all our benchmarks, the overheads of self-adjusting versions are constant and do not depend on the input size, whereas speedups, being asymptotically significant, increase with the input size. To illustrate this property, we examine our merge sort benchmark. In Appendix B, we show the corresponding data for six more representative benchmarks.

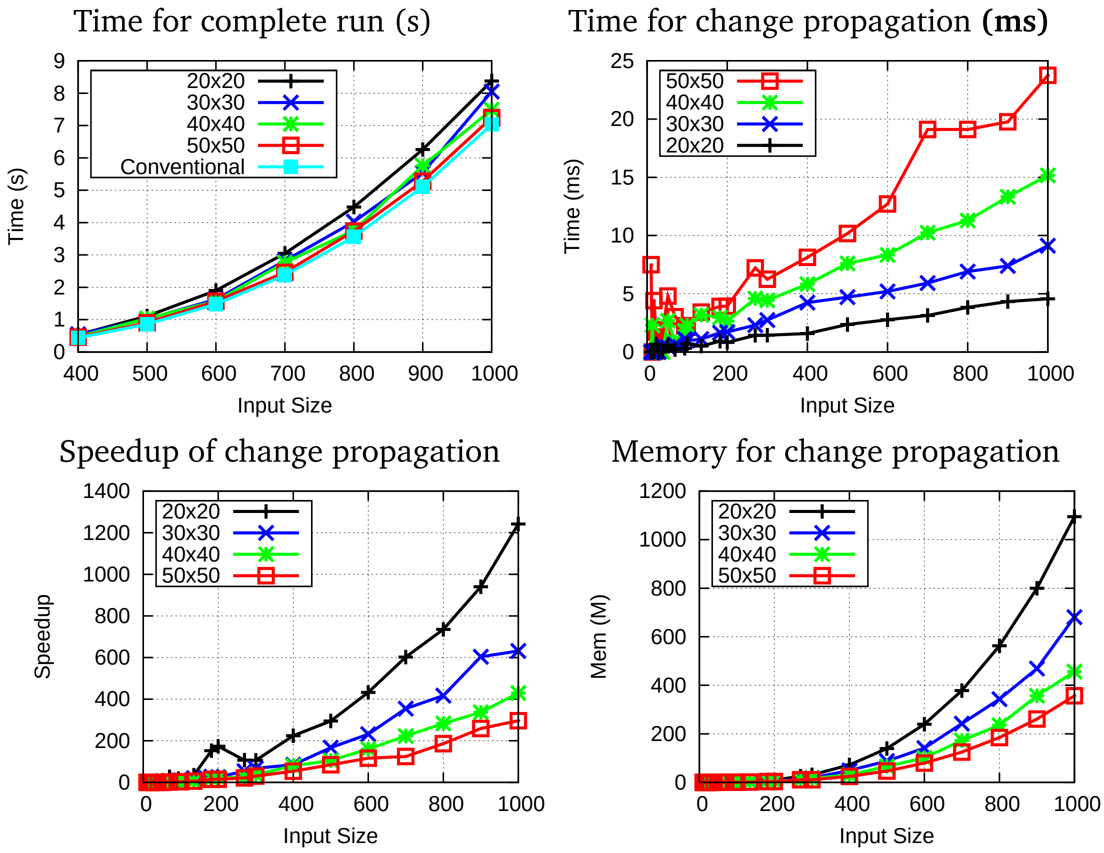
The plot on the left in Figure 8.1 shows the time (in seconds) for the complete run of self-adjusting merge sort compared to the conventional version, for a range of in-


Time for complete run (s)

Time for propagation (**ms**)

Speedup of change propagation

Figure 8.1: Time for complete run; time and speedup for change propagation for `msort`


Time for complete run (s)

Time for change propagation **(ms)**

Speedup of change propagation

Memory for change propagation

Figure 8.2: Time for complete run; time, speedup and memory for change propagation for blocked matrix multiply

put sizes (x axis). The figure suggests that, in both the conventional and self-adjusting versions, the complete-run time grows almost linearly, and they exhibit the same asymptotic complexity, $O(n \log n)$. The plot in the middle of Figure 8.1 shows the time—in milliseconds—for change propagation after inserting/deleting one element for each input size (x axis). As can be seen, the time taken by change propagation grows sublinearly as the input size increases. This is consistent with the $O(\log n)$ bound that we can show analytically. The plot on the right in Figure 8.1 shows the speedups for different input sizes (x axis): the time for a run of the conventional algorithm divided by the time for change propagation. The plots show that speedups increase linearly with the input size, consistent with the theoretical bound. To obtain this asymptotic improvement, we only insert one keyword in the code and use our compiler to generate the self-adjusting version.

8.1.5 Matrix multiplication

Our type-based approach gives the programmer great flexibility in specifying changeable elements in different granularity. The difference in the data representation can lead to dramatically different time and space performance. For example, `mat-mult` performs matrix multiplication using the standard matrix representation where each element of the matrix is changeable, while `block-mat-mult` considers the blocked representation where elements are blocked into groups of 20×20 submatrices. As we can see from Table 8.1, although the blocked version has a smaller speedup, as changing one element requires recomputing the whole submatrix, it has much less overhead compared to the standard representation. To further explore this trade-off, Figure 8.2 shows the time and space of blocked matrix multiplication with block sizes from 20×20 to 50×50.

Run time. In Figure 8.2, the leftmost plot shows the time for the complete run of self-adjusting blocked-matrix multiply with different block sizes, as well as the conventional version. The figure suggests that all benchmarks exhibit the same asymptotic complexity, $O(n^3)$. We also observe that as the block size increases, the overhead becomes smaller. This is because we treat each block as a single modifiable, reducing the number of modifiables tracked at run time. The second plot in Figure 8.2 shows the time for change propagation after changing a block for each input size (x axis) with different block sizes. The time taken by change propagation grows almost linearly as the input size increases, which is consistent with the $O(n \log n)$ bound that we can compute analytically. Changing any part of a block requires recomputing the whole block, so propagation is faster with smaller block sizes. The third plot in Figure 8.2 shows speedups for different input sizes. Speedups increase asymptotically with input size, consistent with the theoretical bound of $O(n^2/\log n)$. Smaller blocks have higher speedups. For example, for a 1000×1000 matrix, the 20×20 block enables 1200× speedup, while the 50×50 block has 280× speedup.

Space. The rightmost plot in Figure 8.2 shows the memory used by change propagation with different block sizes (the complete run never uses more memory than change propagation). As with all other approaches to self-adjusting computation, our approach exploits a trade-off between memory (space) and time (we compare the space usage of other approaches in Section 8.1.8). We store computation traces in memory and use them to respond to incremental data changes, resulting in an increase in memory usage and a decrease in response time. Typically, self-adjusting programs use asymptotically as much memory as the run-time of the computation. The plot shows results consistent with the theoretical bound of $O(n^3)$. Although a smaller block size leads to larger speedups, it requires more memory, because the total number of modifiables created is proportional to the number of blocks in the matrix.

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

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

Table 8.2: Summary of ray tracer timings.

the space needed to store the two input matrices and the result matrix, but enables a 280× speedup when re-computing the output.

8.1.6 Ray tracer

Many applications are suitable for incremental computation, because making a small change to their input on average causes small changes to their output. But some applications are not. Arguably, incremental computation techniques should be avoided or used cautiously in such applications. To evaluate the effectiveness of our approach in such limiting cases, we considered ray tracing, where a small change to the input can require a large fraction of the output image to be updated. In our experiments, we rendered an input scene of 3 light sources and 19 objects with an output image size of 512×512 and then repeatedly changed the surface properties of a single surface (which may be shared by multiple objects in the scene). We considered three kinds of changes: a change to the color of a surface, a change from a diffuse (non-reflective) surface to a mirrored surface,

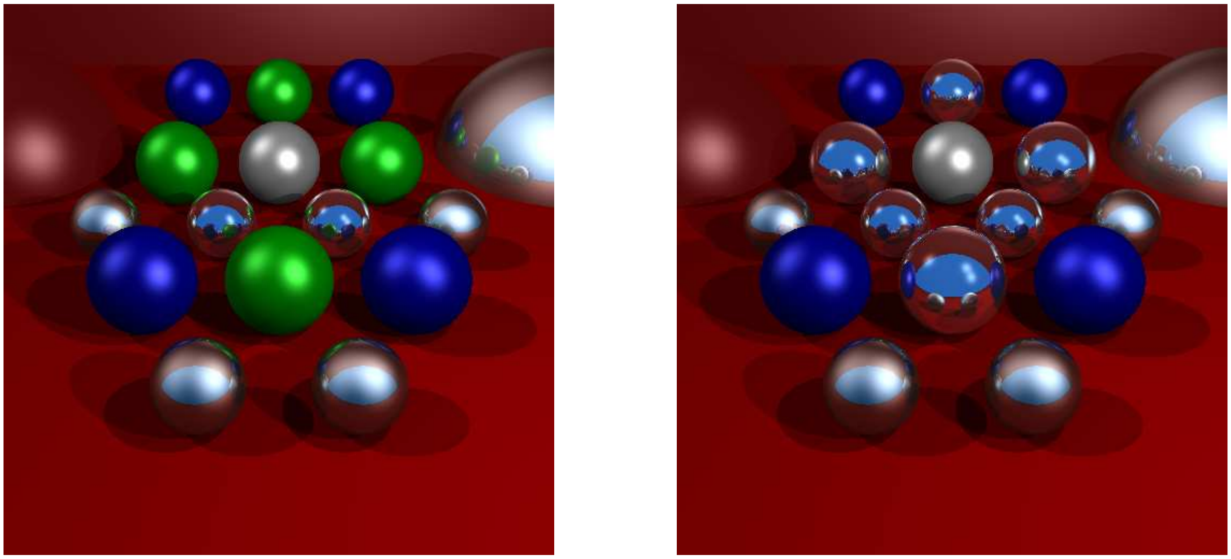


Figure 8.3: Two images produced by our ray tracer. To produce the right-hand image, we change the surfaces of the four green balls from diffused surfaces to mirrored surfaces. Change propagation yields about a 2× speedup over re-computing the output.

and a change from a mirrored to diffuse surface. We measured the time for a complete run of both the conventional and the self-adjusting versions, and the average propagation time for a single toggle of a surface property. For each change to the input, we also measured the change in the output image as a fraction of pixels. Figure 8.3 illustrates an example.

Table 8.2 shows the timings for various kinds of changes. The first column shows the percentage of pixels changed in the output. Each pair of rows corresponds to changing the surface properties of a set of objects (sets labeled A through G) to diffuse (non-reflective) and mirror surfaces in that order, as indicated by superscripts $\cdot^{D}$ and $\cdot^{M}$ respectively. Our measurements show that even when a large fraction of the output image must change, our approach can perform better than recomputing from scratch. We also observe that since mirrors reflect light, making a surface mirrored often requires performing more work during change propagation than making the surface diffuse. Indeed, we observe that the speedups obtained for mirror changes are consistently about half of the speedups for diffuse changes.

8.1.7 Compiler optimizations

In Section 7.3.1 we described some key optimizations that eliminate redundancies in the code. To measure the effectiveness of these optimizations, we measured the running time for our benchmarks compiled with and without these optimizations. Since the optimizations always eliminate redundant calls, we expected them to improve efficiency consistently, and also quite significantly. As can be seen in Figure 8.4 by comparing the bars labeled "Unopt." (green) and "Type-Directed" (black), our experiments indeed show

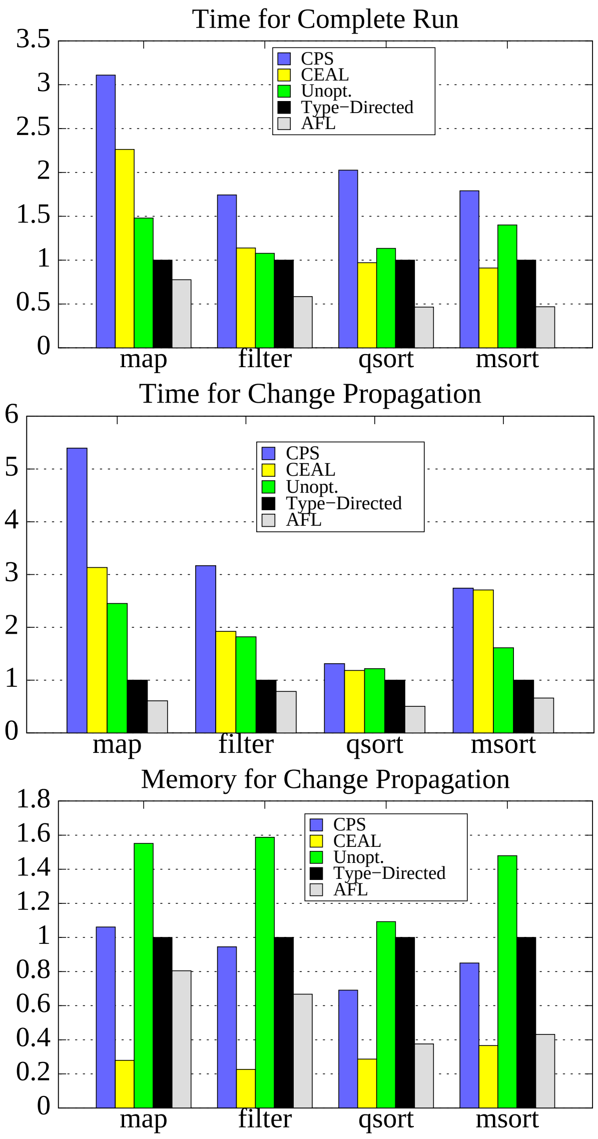


Figure 8.4: Time for initial run and change propagation at fixed input sizes, normalized to AFL = 1.

that the optimizations can improve the time of the complete run and the time and space for change propagation by as much as 60%. The complete run never uses more space than change propagation does. We will discuss the rest of Figure 8.4 in Section 8.1.8.

8.1.8 Comparison to previous work

We compare our results with previous work: the combinator library (AFL) in SML (Acar et al. 2009a), the continuation-passing style (CPS) approach in SML (Ley-Wild et al. 2008), and the C-based CEAL system (Hammer et al. 2011), which is a carefully engineered and highly optimized system that can be competitive with hand-crafted algorithms (Demetrescu et al. 2004). Figure 8.4 shows this comparison for the common benchmarks with fixed input sizes of 1 million keys for list operations and 100,000 keys for sorting, with results normalized to "Type-Directed" ($= 1.0$).

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

Compared to CPS, our approach is approximately twice as fast, even though the CPS approach requires widespread changes to the program code and ours does not. The primary reason for the performance gap is likely that the CPS-based transformation relies on coarse approximations of true dependencies (based on continuations); our compiler identifies dependencies more precisely by using a type-directed translation. Additionally, we compared our ray tracer with one based on CPS, where our approach ("Type-Dir.") is approximately twice as fast as CPS (Table 8.3). Our approach often uses slightly more space than the CPS based approach, probably because of redundancies in the automatically generated code that can be eliminated manually in the CPS approach.

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

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

Table 8.3: Comparison of ray tracer with CPS

decreases by a factor of two.

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

8.1.9 Effect of garbage collection

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

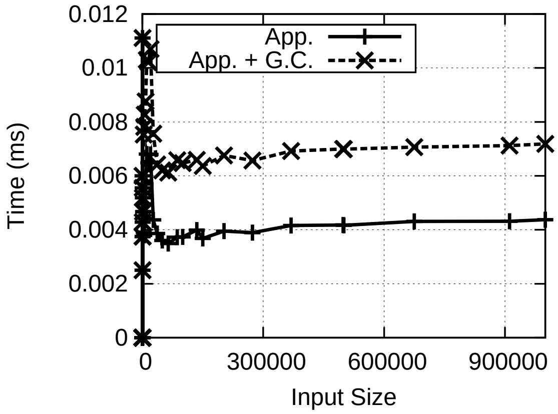


Figure 8.5: Propagation time for `vec-reduce` including GC time.

this setting, our timings show that garbage collection behaves differently in the complete run and change propagation. The time can vary from negligible to moderate during complete runs of self-adjusting executables. For example, in blocked matrix multiplication, garbage collection times are less than 10%, but in vector multiplication, garbage collection takes nearly half of the total running time. Previous work on self-adjusting computation shows similar tradeoffs (Hammer and Acar 2008; Acar et al. 2009a). During change propagation, however, we observe that garbage-collection times are relatively small even when not using a fixed heap. Figure 8.5 shows the garbage collection time for vector reduce, which is close to the worst-case typical behavior that we obtain in our benchmarks. In some applications such as ray-tracing and blocked matrix multiplication, garbage collection times are negligible.

8.2 Experiments with large and dynamic data

8.2.1 Benchmarks and measurements

To evaluate the effectiveness of our approach for large and dynamic data, we implement a variety of domain-specific languages and algorithms.

- a *blocked list* abstract data type that uses our probabilistic chunking algorithm (Section 7.5),
- a sparse matrix abstract data type,
- an implementation of the MapReduce framework (Dean and Ghemawat 2008) 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.

8.2.2 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 8.4 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).

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 7.5) is essential for performance—when using fixed-size chunking, merge sort does not yield noticeable improvements.

Benchmark	B	Run (s)	Prop. (ms)	Speedup	Memory
`map-batch`	1	0.497	497	1	344M
`map`	1	11.21	0.001	497000	7G
	3	16.86	0.012	41416	10G
	10	5.726	0.009	55222	3G
	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 8.4: Blocked lists and sorting: time and space with varying block sizes on fixed input sizes of 10^7.

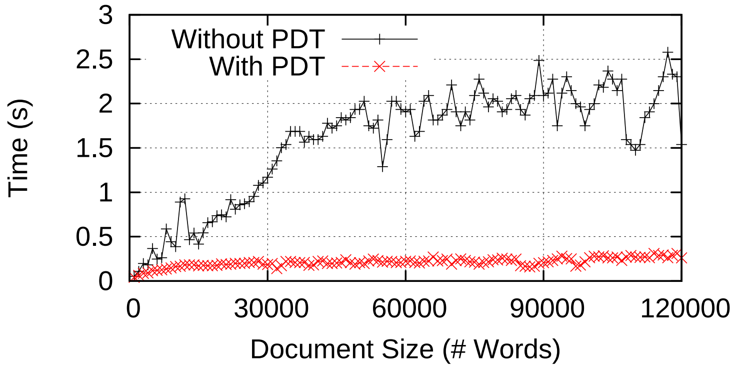


Figure 8.6: Run time (seconds) of incremental word count.

8.2.3 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 (Chen et al. 2012) (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 8.6 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).

8.2.4 Incremental PageRank

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

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

Benchmark	Source	Input Size	Prop. (s)	Speedup	Memory
PR-Batch	Orkut	3×10^6 vertices	7	1	3G
PageRank		1×10^8 edges	0.021	333	36G
PR-Batch	LiveJournal-1	4×10^6 vertices	18	1	5G
PageRank		3×10^7 edges	0.023	783	61G
PR-Batch	Twitter-1	3×10^7 vertices	137	1	50G
PageRank		7×10^8 edges	0.254	539	495G
Conn-Batch	LiveJournal-2	1×10^6 vertices	105	1	4G
Connectivity		8×10^6 edges	0.531	198	140G
SC-Batch	Twitter-2	1×10^5 vertices	8	1	2G
Social Circle		2×10^6 edges	0.079	101	34G

Table 8.5: Incremental sparse graphs: time and space.

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 $\texttt{con}_\varepsilon$; 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 3×10^3 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.

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

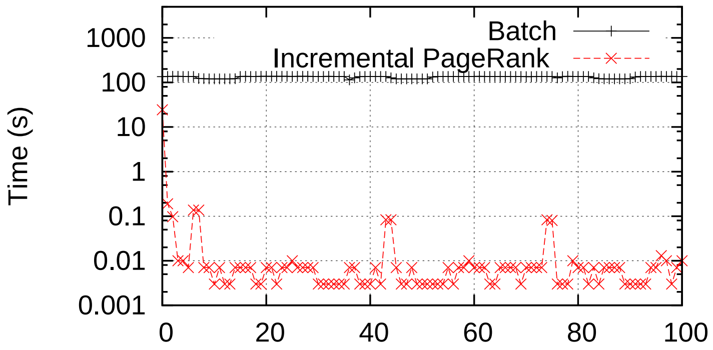


Figure 8.7: Incremental PageRank: 100 trials (x-axis) of deleting 1,000 edges

Table 8.5 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 8.7, show that for most trials, speedups usually approximate 4 orders of magnitude.

8.2.5 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 if u and v are connected. We use a randomized version of Kang et al.'s algorithm (Kang et al. 2011b) 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

[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`

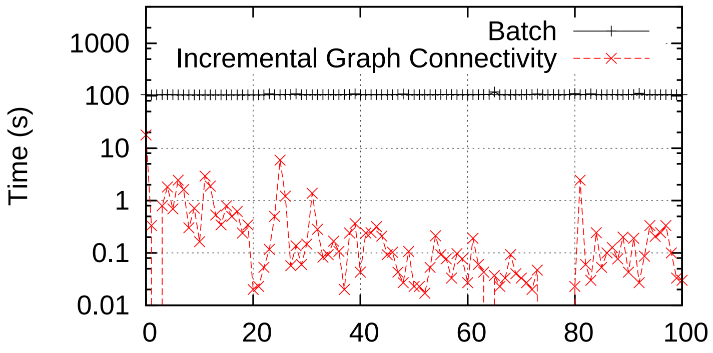


Figure 8.8: Incremental graph connectivity: 100 trials (x-axis) of deleting a vertex

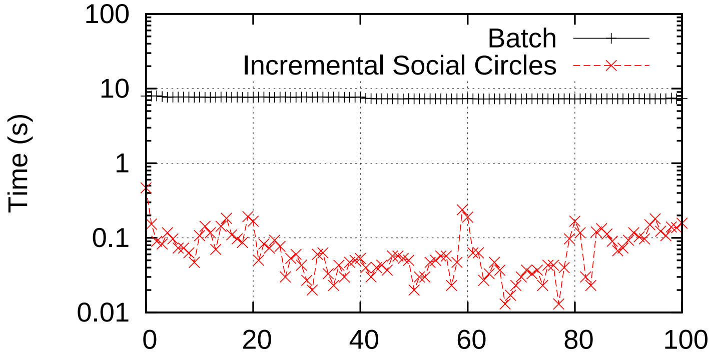


Figure 8.9: Incremental social-circle size: 100 trials (x-axis) of deleting 20 edges

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 8.8; cumulative (average) measurements are shown in Table 8.5. 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.

8.2.6 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 (Kang et al. 2011a), 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 8.9; cumulative (average) measurements are shown in Table 8.5 in the last row. Our incremental version is approximately 100x faster than batch for most trials.

Chapter 9

Conclusion

In this chapter, we discuss related work and directions for future work.

9.1 Related Work

The problem of enabling computation to respond efficiently to changes has been studied extensively. We briefly examine related techniques for incremental computation in algorithms, programming languages and systems. Detailed background can be found in several excellent surveys (Chiang and Tamassia 1992; Ramalingam and Reps 1993; Agarwal et al. 2002; Demetrescu et al. 2005).

9.1.1 Dynamic algorithms and data structures

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., (Ramalingam and Reps 1993; Demetrescu et al. 2005). 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 (Demetrescu et al. 2004). Riedy et al. (2012) present techniques for implementing certain dynamic graphs algorithms for large graphs.

9.1.2 Language-based incremental computation

Motivated by the difficulty in designing and implementing ad hoc dynamic algorithms, the programming languages community works on developing language-based solutions to incremental computation. Here we briefly review the domain-specific approaches to incremental computation.

Data dependence graph. Earlier work on incremental computation, which took place in the '80s and '90s, was primarily based on dependence graphs and memoization. Dependence graphs record the dependencies between data in a computation and use a change-propagation algorithm to update the computation when the input is modified (Demers et al. 1981; Hoover 1987). Dependence graphs have been effective in applications such as syntax-directed computations, but are not general-purpose because change propagation cannot update the dependence structure.

Memoization. Memoization, also called function caching (Pugh and Teitelbaum 1989; Abadi et al. 1996; Heydon et al. 2000), is a technique that can improve efficiency of any purely functional program, when executions of a program with similar inputs involve similar function calls. The idea dates back to the late 1950s (Bellman 1957; McCarthy 1963; Michie 1968), and was first used for incremental computation by Pugh and Teitelbaum (1989) and then by many others (e.g. Abadi et al. (1996)). Due to its strict reliance on argument equivalence, sometimes a small data modifications can prevent memoization-based reuse in general incremental computations. Self-adjusting computation overcomes these restrictions by using a form of memoization that allows reuse of computations that themselves can be incrementally updated.

Partial evaluation. Another approach to incremental computation is partial evaluation (Field and Teitelbaum 1990; Sundaresh and Hudak 1991). Similar to implicit self-adjusting computation, the input is statically partitioned into a fixed portion known at compile time, and a dynamic portion. The partial evaluator will specialize the program with the fixed input, so this part of the input can never change in the runtime. In contrast, in self-adjusting computation, the stable input, although it cannot be changed via change propagation, can still take different values for different initial runs. Although partial evaluation speeds up responses when the dynamic input is modified, the lack of runtime dependency tracking means that the update time may not be as efficient as self-adjusting computation.

Attribute grammars. Attribute grammars can be viewed as a simple declarative functional language, where programs specify sets of attributes that relate data items whose interdependencies are defined by a tree structure (e.g., the context-sensitive attributes, such as typing information, that decorate an abstract syntax tree). These attributes can be evaluated incrementally. When the attributed tree is changed, the system can sometimes update the affected attributes in optimal time (Reps 1982a,b; Reps and Teitelbaum 1989; Efremidis et al. 1993). The definition of optimality is input-sensitive: it is based on counting of the minimum number of attributes that must be updated after a tree edit occurs; in general this is not known until such an incremental reevaluation is completed.

In this sense, the change propagation algorithm used in these systems is similar to that of self-adjusting computation, but in a less general setting.

Invariant checking. DITTO (Shankar and Bodik 2007) offers support for incremental invariants-checking in Java. By customizing the computation graph data structures for invariant checking programs, they implemented a fully automatic incremental invariant checker that can speed up invariant checks by an order of magnitude, without programmer annotations. But it only supports a purely-functional subset of Java. DITTO also places further restrictions on the programs (e.g., functions cannot return arbitrary values) and is unsound in general.

Static differentiation. Static differentiation transforms programs to derive a second program that can handle input changes explicitly. The derived program takes the original input and a "delta" input that specifies how the original input is going to change, and produces the output change. The program derivation is performed statically. Compared to self-adjusting computation, the static approach does not have the extra time and space overhead. But being static in nature, they are not general-purpose approaches: it cannot handle programs with general recursions (Cai et al. 2014); Differential dataflow (McSherry et al. 2013; Abadi et al. 2015) only supports data flow programs with nested iterations. Also, they cannot take advantage of memoization, which is a purely dynamic approach.

9.1.3 Self-adjusting computation

Recent work based on self-adjusting computation has made progress towards achieving general-purpose efficient incremental computation by providing algorithmic language abstractions to express computations that respond automatically to changes to their data (Acar 2005; Acar et al. 2006c, 2009a). Variants of self-adjusting computation has been implemented as an ML library (Acar 2005; Acar et al. 2006c, 2009a), as a language extension to ML (Ley-Wild et al. 2008; Ley-Wild 2010), and as a language extension to C (Hammer et al. 2009, 2011; Hammer 2012), with similar systems also implemented in Haskell (Carlsson 2002), Java (Shankar and Bodik 2007), and OCaml (Hammer et al. 2014; JaneStreet 2015). Self-adjusting computation can deliver asymptotically efficient updates in a reasonably broad range of problem domains, including dynamic trees (Acar et al. 2004, 2005), kinetic motion simulation (Acar et al. 2006d, 2008b), dynamic computational geometry (Acar et al. 2007a, 2010b, 2011; Türkoğlu 2012; Acar et al. 2013b), machine learning (Acar et al. 2007c, 2008c, 2009b; Sümer et al. 2011; Sümer 2012), and big-data systems (Bhatotia et al. 2011a,b, 2014; Bhatotia 2015; Bhatotia et al. 2015). Here we briefly review the main techniques.

The foundations. Self-adjusting computation generalizes the data dependence graphs of earlier techniques (see above) by introducing dynamic dependency graphs (DDGs) (Acar et al. 2002, 2006c). Dynamic dependency graphs and the change propagation algorithm offer a general-purpose technique that enables any purely functional program to respond to changing data. In terms of effectiveness, Acar et al. (2003) showed that there is a duality between change propagation and memoization, and developed a form of memoization that allows reuse of computations that themselves can be incrementally updated (Acar et al. 2006b, 2007b, 2009a, 2013a). Acar et al. (2008a) extended the semantics, the language constructs, and the algorithms to include imperative programs, by using persistent data structures (Driscoll et al. 1989) to represent the dynamic dependency graphs. Acar et al. (2002) implemented a library for self-adjusting computation using monadic types in Standard ML. Soon afterward, Carlsson (2002) showed how to adapt this programming model for Haskell, by exploiting certain Haskell features such as monads and type class.

Traceable data structures. To achieve automatic and correct updates in response to changing data, self-adjusting computation techniques trace dependencies at the level of changeable data. Acar et al. (2010a) extended self-adjusting computation to support dependency tracking at the level of traceable data types, instead of changeable data. A traceable data structure can be used in a self-adjusting program much like its ordinary, non-traceable version. Since they enable tracking dependencies at the granularity of data structural operations, rather than all changeable data, traceable data types can lead to significant improvements in efficiency. Perhaps more importantly, traceable data structures enable the algorithm designer and the programmer to incorporate domain-specific knowledge into a self-adjusting program in a composable fashion, thereby making it possible for the approach to inter-operate with "hand-crafted" dynamic/incremental algorithms.

DeltaML. In contrast to Acar et al. (2006c), who use a library of modal self-adjusting primitives, Ley-Wild et al. (2008) give a more programmer-friendly approach where a compiler does the work to translate certain self-adjusting parts of a lightly annotated SML program into a self-adjusting program in continuation-passing style (CPS). This approach dramatically simplifies writing self-adjusting code by enabling the compiler to identify dependencies between code and data (Ley-Wild 2010). Hence, a programmer can more directly see the conventional semantics of the program, without having to be explicit about the higher-order plumbing required in the library-based approach. Though much easier to use than earlier approaches, the programmer must still be explicit about what is incrementally modifiable. In a follow-up work, Ley-Wild et al. (2009) gave a cost-semantics for this new source language and proved that the cost semantics predicts the asymptotic efficiency of dynamic updates. Ley-Wild et al. (2012) extends self-adjusting computation to non-monotonic reuses of subcomputations, and provided a new change propagation algorithm that realizes the semantics of trace reordering, which incurs a logarithmic overhead.

CEAL. Hammer et al. (2009) extends self-adjusting computation to work in low-level languages such as C, where higher-order functions and garbage collection are not available. Hammer et al. (2009) consider a low-level, self-adjusting language and design a normalization algorithm that transforms the code into closures in C to simulate the use of higher-order functions in functional languages. Hammer and Acar (2008) develop a technique that integrates garbage collection with change propagation, making it possible to find and reclaim memory that becomes inaccessible during propagation. An interesting property of this garbage collection algorithm is that it does not have to trace memory to identify inaccessible blocks, but instead takes advantage of the invariants of self-adjusting computation and change propagation to discover them directly. Hammer et al. (2011) implemented these techniques in CEAL, a language that extends that C language for self-adjusting computation, and formalized the semantics of CEAL (Hammer 2012).

Adapton. Hammer et al. (2014) developed Adapton, a technique for demand-driven self-adjusting computation in OCaml, where updates may be delayed until they are demanded. The technique presents a demand-driven semantics to incremental computation, tracking changes in a hierarchical fashion in an acyclic dependency graph. They also formalizes an explicit separation between inner, incremental computations and outer observers. This combination ensures programs only recompute computations as demanded by observers, and allows inner computations to be reused more liberally. The formulation enables change propagation for non-monotonic changes, such as sharing, switching and swapping.

Parallelism. 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 work considered techniques for performing efficient parallel updates in the context of a lambda calculus extended with fork-join style parallelism (Hammer et al. 2007), as well as POSIX thread-based parallel incremental computation (Bhatotia et al. 2015). Follow-up work considered the technique in the context of a more sophisticated problem showing both theoretical and empirical results of its effectiveness (Acar et al. 2011). Burckhardt et al. (2011) 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.

9.1.4 Type systems and semantics

Information flow. Information flows have been developed to check security properties. Detailed background can be found in the survey of Sabelfeld and Myers (2003). Denning and Denning (1977) introduce the concept of tainted values as lattices, and describe a static analysis on a simple imperative language. Heintze and Riecke (1998) developed a type system with security annotations in lambda calculus to ensure non-interference property (Goguen and Meseguer 1982). Myers (1999) created JFlow, an extension to

Java that allows programmers to annotate values and uses a type system with both static and dynamic enforcement. It does not guarantee noninterference. Pottier and Simonet (2003) present a type system that guarantees noninterference for an ML-like language.

Abadi et al. (1999) discover an important connection between secure information flow and three types of program analysis techniques: program slicing, binding-time analysis and call tracking, under the central notion of dependency. Motivated by this discovery, implicit self-adjusting computation uses information flow type system to infer program dependency from input type annotations. Another similar information flow application is EnerJ (Sampson et al. 2011), which uses information flow to isolate parts of the program that must be precise from those that can be approximated so that a program functions correctly even as quality of service degrades via lightweight type annotations.

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

Constraint-based type inference. Our type system uses many ideas from Pottier and Simonet (2003), including a form of constraint-based type inference (Odersky et al. 1999), and is also broadly similar to other systems that use subtyping constraints (Simonet 2003; Foster et al. 2006).

Cost semantics. To prove that our translation yields efficient self-adjusting target programs, we use a simple cost semantics. The idea of instrumenting evaluations with cost information goes back to the early ’90s (Sands 1990). Cost semantics is particularly important in lazy (Sands 1990; Sansom and Peyton Jones 1995)) and parallel languages (Spoonhower et al. 2008) where it is especially difficult to relate execution time to the source code, as well as in self-adjusting computation (Ley-Wild et al. 2009; Çiçek et al. 2015).

9.1.5 Functional reactive programming

Elliott and Hudak (1997) introduced functional reactive programming (FRP), which provides powerful primitives for operating on continuously changing values, called *behaviors*, and values that change at certain points in time, called discrete *events*. The approach proposed by Elliott and Hudak, which is known as *classical FRP*, turned out to be difficult to realize in practice, because it allows expressing computations that depend on future values and because its implementations were prone to so-called time and space leaks. Since its publication, the classical FRP paper started a lively line of research, leading to much work that continues to this day. Due to space limitations, we are able to discuss a relatively small subset of this work. The interested reader can find more details on the FRP literature in recent papers on this topic (Krishnaswami et al. 2012; Czaplicki and Chong 2013; Jeffrey 2013).

Real-time FRP (Wan et al. 2001) proposed techniques for eliminating time and space leaks by introducing *signals* as a uniform representation of behaviours and events and by presenting a restricted language for operating on signals. By further restricting behaviors to change only at events (Wan et al. 2002), event-driven FRP offered a way to reduce redundant recomputation by requiring updates only when an event takes place. The restrictions, however, led to a loss of expressiveness, which subsequent work on arrowized FRP tried to recover (Liu and Hudak 2007; Liu et al. 2009). The work on arrowized FRP shows that much of the expressiveness of classical FRP may be regained while avoiding time and space leaks and while also preserving causality.

Citing difficulties in using arrow combinators and the relatively complex semantics model, other authors pursued research in at least two separate directions. Some research gave up the idea of trying to match the classical FRP semantics. Some work (e.g. Cooper and Krishnamurthi (2006); Czaplicki and Chong (2013)) considered only discretely changing values, called signals, and restricts the use of signals to ensure efficient implementation. Other work developed semantics for classical FRP that guarantees causality (Krishnaswami and Benton 2011; Jeffrey 2012; Krishnaswami et al. 2012).

Even though functional reactive programming may be viewed as naturally amenable to incremental computation—it should be possible to incorporate time-varying values into a computation by performing an incremental update—most existing research does not employ incremental computation techniques, focusing instead on taming the time and space consumption of the “one-shot”, re-computation-based approach. Some recent work took steps in the direction of connecting functional programming and incremental computation (Donham 2010; Demetrescu et al. 2011). Recent work of Demetrescu et al.provides the programmer with techniques for writing incremental update functions in (imperative) reactive programs (Demetrescu et al. 2011). Another work is Donham’s Froc (Donham 2010), which provides support for FRP based on a data-driven implementation using self-adjusting computation.

9.1.6 Systems

There are several systems for big data computations such as MapReduce (Dean and Ghemawat 2008), Dryad (Isard et al. 2007), Pregel (Malewicz et al. 2010), GraphLab (Low et al. 2012), and Dremel (Melnik et al. 2011). 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.

Bhatotia et al.applies the principles of self-adjusting computation to the big data setting but only in the context of domain-specific languages, including incremental MapReduce programs (Bhatotia et al. 2011a,b), incremental sliding windows analytics (Bhatotia et al. 2014) and POSIX thread-based parallel incremental computation (Bhatotia et al. 2015).

Data flow systems like MapReduce and Dryad have been extended with support for incremental computation. MapReduce Online (Condie et al. 2010) can react efficiently to additional input records. Nectar (Gunda et al. 2010) caches the intermediate results of DryadLINQ programs and generates programs that can re-use results from this cache. Naiad (Murray et al. 2013) 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 (Demers et al. 1981; Yellin and Strom 1991). Percolator (Peng and Dabek 2010) 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. The work that we presented here assumes sequential computation.

9.2 Future work

Implicit monadic programming. The type-directed translation technique (Chapter 4–6) essentially translates purely functional programs with changeable ($\mathbb{C}$) notations into an explicit monadic style. This approach can be applied to other monadic constructs besides self-adjusting computation, such as approximate computation, probabilistic programming, and numerical computation. The information-flow type system ensures that the effectful/imprecise computations do not flow into the critical control path in the program. With the corresponding monadic runtime library, we can achieve the same correctness/efficiency guarantee as the monadic language, while still being able to program in a direct and purely functional style. Using a more powerful type system, such as a quantitative information-flow type system or dependent type, we can even infer performance or error bounds for the corresponding computation. It is also interesting to explore the language abstractions and compilation strategies for mixed use of monads, e.g. incremental approximate computation.

Memoization. The AFL target language uses a `lift` primitive for memoization that conflates keyed allocation and adaptive memoization. A better linguistic support is to separate the primitive into two different constructs (Ley-Wild et al. 2008). The choices of which computation to memoize and when a computation matches the previous result can significantly affect the efficiency of change propagation. The matching process is done via pointer equality for non-primitive data. One major problem is that pointer identity is fragile and non-deterministic. During change propagation, the address of a reference may change coincidentally, leading to missed opportunities for memoization match. These spurious changes can cascade, causing subsequent computations built on top of it to also change their identities. To give programmers better control over memoization, we need to build high-level language abstracts for memoization, and provide better support for debugging and profiling of self-adjusting programs. There has been some research that provides names as an abstract identity for the memoization matching process (Hammer et al. 2015). We are also building tools similar to QuickCheck (Claessen and Hughes 2000) to detect performance bugs in self-adjusting programs.

Parallel self-adjusting computation. In principle our approach can also be parallelized, especially because purely functional programming is naturally amenable to parallelism. The block sequence abstraction (Section 7.5) not only provides a way to coarsen the granularity of dependency, but also suggests how we can parallelize the computation. As each block sequence is independent, we can easily fork the computation of each block into different machines, and have a master server that keeps track of dependencies for each block. 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 (Hammer et al. 2007; Bhatotia et al. 2011b; Burckhardt et al. 2011).

Appendix A

Proofs

A.1 Proof of translation type soundness

First, we need a few simple lemmas.
Lemma A.1.1 (Translation of Outer Levels).

$[\phi]\tau$ O.C. *if and only if* $\|\tau\|_{\phi} = \|\tau\|_{\phi}^{\to\mathbb{C}}$ **mod**;
$[\phi]\tau$ O.S. *if and only if* $\|\tau\|_{\phi} = \|\tau\|_{\phi}^{\to\mathbb{C}}$.

Proof
Case analysis on $[\phi]\tau$, using the definitions of $-$ O.S., $-$ O.C., $\|-\|_{\phi}$ and $\|-\|_{\phi}^{\to\mathbb{C}}$. □

Lemma A.1.2 (Substitution). *Suppose* ϕ *is a satisfying assignment for* C*, and* $\phi(\vec{\alpha}) = \vec{\delta}$*, where* $\vec{\alpha} \subseteq FV(C)$.

1. *If* $\mathcal{D}$ *derives* $C; \Gamma \vdash_{\varepsilon} e : \tau$*, then there exists* $\mathcal{D}'$ *deriving* $C; [\vec{\delta}/\vec{\alpha}]\Gamma \vdash_{\varepsilon} e : [\vec{\delta}/\vec{\alpha}]\tau$*, where* $\mathcal{D}'$ *has the same height as* $\mathcal{D}$.
2. *If* $C \Vdash \delta' \lhd \tau$*, then* $C \Vdash [\vec{\delta}/\vec{\alpha}]\delta' \lhd [\vec{\delta}/\vec{\alpha}]\tau$.
3. *If* $C \Vdash \tau' <: \tau''$*, then* $C \Vdash [\vec{\delta}/\vec{\alpha}]\tau' <: [\vec{\delta}/\vec{\alpha}]\tau''$.
4. *If* $C \Vdash \tau' \mathrel{\hat{=}} \tau''$*, then* $C \Vdash [\vec{\delta}/\vec{\alpha}]\tau' \mathrel{\hat{=}} [\vec{\delta}/\vec{\alpha}]\tau''$.

Proof
By induction on the given derivation. □

Lemma A.1.3. *Given* $\tau' <: \tau''$ *and* $\tau' \mathrel{\mathring{=}} \tau''$*:*
(1) If τ'' O.S. *then* $\tau' = \tau''$.
(2) If τ'' O.C. *then either* $\tau' = \tau''$ *or* $\tau' = \left|\tau''\right|^{\mathbb{S}}$.

Proof
By induction on the derivation of $\tau' <: \tau''$.

- **Case** (subInt): $\tau' = \mathbf{int}^{\delta'}$ and $\tau'' = \mathbf{int}^{\delta''}$, where $\delta' \leq \delta''$.

  (1) If τ'' O.S. then $\delta'' = \mathbb{S}$. So $\tau' = \tau''$.

(2) If τ'' O.C. then $\delta'' = \mathbb{C}$. If $\delta' = \mathbb{S}$ then $\left|\tau''\right|^{\mathbb{S}} = \mathbf{int}^{\mathbb{S}} = \mathbf{int}^{\delta'} = \tau'$; if $\delta' = \mathbb{C}$ then $\tau'' = \mathbf{int}^{\mathbb{C}} = \mathbf{int}^{\delta'} = \tau'$.

- **Case** (subProd):

  (1) By definition of $\circeq$, $\tau' = \tau''$.

  (2) τ'' O.C. is impossible.

- **Case** (subSum):

  (1) If τ'' O.S. then $\tau'' = (\tau''_1 + \tau''_2)^{\mathbb{S}}$. By inversion on (subSum), $\tau' = (\tau'_1 + \tau'_2)^{\mathbb{S}}$. By definition of $\circeq$, $\tau'_1 = \tau''_1$ and $\tau'_2 = \tau''_2$. Therefore $\tau' = \tau''$.

  (2) If τ'' O.C. then $\tau'' = (\tau''_1 + \tau''_2)^{\mathbb{C}}$. By inversion on (subSum), $\tau' = (\tau'_1 + \tau'_2)^{\delta'}$. By definition of $\circeq$, $\tau'_1 = \tau''_1$ and $\tau'_2 = \tau''_2$. If $\delta' = \mathbb{S}$ then $\left|\tau''\right|^{\mathbb{S}} = (\tau''_1 + \tau''_2)^{\mathbb{S}}$, which is equal to τ'. If $\delta' = \mathbb{C}$ then $\tau'' = (\tau''_1 + \tau''_2)^{\mathbb{C}} = (\tau'_1 + \tau'_2)^{\mathbb{C}} = (\tau'_1 + \tau'_2)^{\delta'} = \tau'$.

- **Case** (subArrow): Similar to the (subSum) case. □

Theorem 6.4.1 (Translation Type Soundness).
If $C; \Gamma \vdash_\varepsilon e : \tau$ *and* ϕ *is a satisfying assignment for* C *then*
(1) there exists $e^{\mathbb{S}}$ *such that* $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$ *and* $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_\phi$,
and if e *is a value, then* $e^{\mathbb{S}}$ *is a value;*
(2) there exists $e^{\mathbb{C}}$ *such that* $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$ *and* $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$.

Proof
By induction on the height of the derivation of $C; \Gamma \vdash_\varepsilon e : \tau$.

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 in the $\mathbb{S}$ mode, typing in the $\mathbb{S}$ mode, translation in the $\mathbb{C}$ mode, and typing in the $\mathbb{C}$ mode), and often arrive at some of them early, we indicate them with "☞".

"Part (1)" and "Part (2)" refer to the two parts of the conclusion: (1) "there exists $e^{\mathbb{S}}$ ..." and (2) "there exists $e^{\mathbb{C}}$ ...". In some cases, it is convenient to prove these simultaneously, so we sometimes annotate the "☞" symbol to clarify which part is being proved:

(1)☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} (\mathbf{fun}^{\mathbb{C}}\ f(x) = \underline{e'}) : \|\tau\|_\phi$ By (TFun)

This information can also be read off from the turnstile: $\vdash_{\mathbb{S}}$ means part (1), and $\vdash_{\mathbb{C}}$ means part (2).

- **Case** $\boxed{\dfrac{}{C; \Gamma \vdash_\varepsilon n : \underbrace{\mathbf{int}^{\mathbb{S}}}_{\tau}}\ \text{(SInt)}}$

  Part (1): Let $e^{\mathbb{S}}$ be n.

	$[\phi]\Gamma \vdash n : \textbf{int}^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} n$	By (Int)
☞	$[\phi]\Gamma \vdash e : [\phi](\textbf{int}^{\mathbb{S}}) \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$ and $e^{\mathbb{S}}$ is a value	By $n = e$ and def. of substitution

	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} n : \textbf{int}$	By (TInt)
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} e^{\mathbb{S}} : \Vert\textbf{int}^{\mathbb{S}}\Vert_{\phi}$	By $\textbf{int} = \Vert\textbf{int}^{\mathbb{S}}\Vert = \Vert[\phi]\textbf{int}^{\mathbb{S}}\Vert = \Vert\textbf{int}^{\mathbb{S}}\Vert_{\phi}$ and $n = e^{\mathbb{S}}$
☞	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} e^{\mathbb{S}} : \Vert\tau\Vert_{\phi}$	By $\tau = \textbf{int}^{\mathbb{S}}$

Part (2): Let $e^{\mathbb{C}}$ be $\textbf{let}\ r = n\ \textbf{in}\ \textbf{write}(r)$.

	$[\phi]\Gamma \vdash n : \textbf{int}^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} n$	Above
	$[\phi]\Gamma \vdash n : \textbf{int}^{\mathbb{S}} \underset{\mathbb{C}}{\hookrightarrow} \textbf{let}\ r = n\ \textbf{in}\ \textbf{write}(r)$	By (Write)
☞	$[\phi]\Gamma \vdash e : [\phi](\textbf{int}^{\mathbb{S}}) \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$	$n = e$; def. of subst.; $e^{\mathbb{C}} = \ldots$

	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} n : \textbf{int}$	By (TInt)
	$\cdot; \Vert\Gamma\Vert_{\phi}, r : \textbf{int} \vdash_{\mathbb{S}} r : \textbf{int}$	By (TVar)
	$\cdot; \Vert\Gamma\Vert_{\phi}, r : \textbf{int} \vdash_{\mathbb{C}} \textbf{write}(r) : \textbf{int}$	By (TWrite)
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{C}} \textbf{let}\ r = n\ \textbf{in}\ \textbf{write}(r) : \textbf{int}$	By (TLet)
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{C}} e^{\mathbb{C}} : \textbf{int}$	By def. of $e^{\mathbb{C}}$
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{C}} e^{\mathbb{C}} : \Vert\textbf{int}^{\mathbb{S}}\Vert_{\phi}^{\to\mathbb{C}}$	By $\textbf{int} = \Vert\textbf{int}^{\mathbb{S}}\Vert_{\phi}^{\to\mathbb{C}}$
☞	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{C}} e^{\mathbb{C}} : \Vert\tau\Vert_{\phi}^{\to\mathbb{C}}$	By $\tau = \textbf{int}^{\mathbb{S}}$

- **Case** $\boxed{\dfrac{\Gamma(x) = \forall\vec{\alpha}[D].\,\tau_0 \qquad C \Vdash \exists\vec{\beta}.[\vec{\beta}/\vec{\alpha}]D}{C \wedge D; \Gamma \vdash_{\varepsilon} x : [\vec{\beta}/\vec{\alpha}]\tau_0}\ (\text{SVar})}$

Part (1): Let $e^{\mathbb{S}}$ be $x[\vec{\alpha} = \vec{\delta}]$.

	$\Gamma(x) = \forall\vec{\alpha}[D].\,\tau_0$	Premise
	$([\phi]\Gamma)(x) = [\phi](\forall\vec{\alpha}[D].\,\tau_0) = \forall\vec{\alpha}[[\phi]D].\,[\phi]\tau_0$	By def. of substitution
	$[\phi]\Gamma \vdash x : [\vec{\delta}/\vec{\alpha}]([\phi]\tau_0) \underset{\mathbb{S}}{\hookrightarrow} x[\vec{\alpha} = \vec{\delta}]$	By (Var)
	$[\phi]\Gamma \vdash x : [\phi][\vec{\delta}/\vec{\alpha}]\tau_0 \underset{\mathbb{S}}{\hookrightarrow} x[\vec{\alpha} = \vec{\delta}]$	$\vec{\delta}$ closed and $\vec{\alpha} \cap \mathsf{dom}(\phi) = \emptyset$
	$[\phi]\Gamma \vdash x : [\phi][\vec{\delta}/\vec{\beta}]([\vec{\beta}/\vec{\alpha}]\tau_0) \underset{\mathbb{S}}{\hookrightarrow} x[\vec{\alpha} = \vec{\delta}]$	Intermediate subst.
	$[\phi]\Gamma \vdash x : [\phi](\underbrace{[\vec{\beta}/\vec{\alpha}]\tau_0}_{\tau}) \underset{\mathbb{S}}{\hookrightarrow} x[\vec{\alpha} = \vec{\delta}]$	$\phi(\vec{\beta}) = \vec{\delta}$
☞	$[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$ and $e^{\mathbb{S}}$ is a value	By $e = x$; $\tau = [\vec{\beta}/\vec{\alpha}]\tau_0$; $e^{\mathbb{S}} = x[\vec{\alpha}{=}\vec{\delta}]$

	$(\Vert\Gamma\Vert_{\phi})(x) = \Pi\vec{\alpha}[[\phi]D].\,[\phi]\tau_0$	By def. of $\Vert{-}\Vert_{\phi}$ and def. of subst.
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}] : \Vert[\vec{\delta}/\vec{\alpha}]([\phi]\tau_0)\Vert$	By (TVar)
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} e^{\mathbb{S}} : \Vert[\phi][\vec{\delta}/\vec{\alpha}]\tau_0\Vert$	$\vec{\alpha} \cap \mathsf{dom}(\phi) = \emptyset$
	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} e^{\mathbb{S}} : \Vert[\vec{\beta}/\vec{\alpha}]\tau_0\Vert_{\phi}$	Intermed. subst., $\phi(\vec{\beta}) = \vec{\delta}$, def. of $\Vert{-}\Vert_{\phi}$
☞	$\cdot; \Vert\Gamma\Vert_{\phi} \vdash_{\mathbb{S}} e^{\mathbb{S}} : \Vert\tau\Vert_{\phi}$	$\tau = [\vec{\beta}/\vec{\alpha}]\tau_0$

Part (2), subcase (a) where $[\phi]\tau$ O.S.: Let $e^{\mathbb{C}}$ be $\textbf{let}\ r = x[\vec{\alpha} = \vec{\delta}]\ \textbf{in}\ \textbf{write}(r)$.

$[\phi]\Gamma \vdash x : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} x[\vec{\alpha} = \vec{\delta}]$ Above

$[\phi]\Gamma \vdash x : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = x[\vec{\alpha} = \vec{\delta}]\ \mathbf{in}\ \mathbf{write}(r)$ By (Write)

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$ By $e = x$ and def. of $e^{\mathbb{C}}$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}] : \|\tau\|_\phi$ Above

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi$ By (TVar)

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau\|_\phi$ By (TWrite)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = x[\vec{\alpha} = \vec{\delta}]\ \mathbf{in}\ \mathbf{write}(r) : \|\tau\|_\phi$ By (TLet)

$\vdash [\phi]\tau$ O.S. Subcase (a) assumption

$\|\tau\|_\phi = \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By Lemma A.1.1

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By above equality

Part (2), subcase (b) where $[\phi]\tau$ O.C.: Let $e^{\mathbb{C}} = \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r')$.

$[\phi]\Gamma \vdash x : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$ Above

$[\phi]\tau$ O.C. Subcase (b) assumption

☞ $[\phi]\Gamma \vdash x : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$ By (ReadWrite)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_\phi$ Above

$[\phi]\tau$ O.C. Subcase (b) assumption

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi^{\vec{\mathbb{C}}}\ \mathbf{mod}, r' : \|\tau\|_\phi^{\vec{\mathbb{C}}} \vdash_{\mathbb{S}} r' : \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By (TPVar)

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi^{\vec{\mathbb{C}}}\ \mathbf{mod}, r' : \|\tau\|_\phi^{\vec{\mathbb{C}}} \vdash_{\mathbb{C}} \mathbf{write}(r') : \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By (TWrite)

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi^{\vec{\mathbb{C}}}\ \mathbf{mod} \vdash_{\mathbb{S}} r : \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By (TVar)

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi^{\vec{\mathbb{C}}}\ \mathbf{mod} \vdash_{\mathbb{C}} \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r')$ By (TRead)

$\|\tau\|_\phi = \|\tau\|_\phi^{\vec{\mathbb{C}}}\ \mathbf{mod}$ By Lemma A.1.1

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_\phi^{\vec{\mathbb{C}}}\ \mathbf{mod}$ By above eqn.

☞ $\cdot; \|\Gamma\|_\phi, x' : \|\tau\|_\phi^{\vec{\mathbb{C}}} \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By (TLet)

- **Case** $\boxed{\dfrac{C; \Gamma \vdash_\varepsilon v_1 : \tau_1 \qquad C; \Gamma \vdash_\varepsilon v_2 : \tau_2}{C; \Gamma \vdash_\varepsilon \underbrace{(v_1, v_2)}_{e} : \underbrace{(\tau_1 \times \tau_2)^{\mathbb{S}}}_{\tau}}\ (\text{SPair})}$

  - Part (1), stable mode translation:

    $C; \Gamma \vdash_\varepsilon v_1 : \tau_1$ Subderivation

    $[\phi]\Gamma \vdash v_1 : [\phi]\tau_1 \underset{\mathbb{S}}{\hookrightarrow} \underline{v_1}$ By i.h.

    $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{v_1} : \|\tau_1\|_\phi$ ″

    $C; \Gamma \vdash_\varepsilon v_2 : \tau_2$ Subderivation

    $[\phi]\Gamma \vdash v_2 : [\phi]\tau_2 \underset{\mathbb{S}}{\hookrightarrow} \underline{v_2}$ By i.h.

    $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{v_2} : \|\tau_2\|_\phi$ ″

Let $e^{\mathbb{S}} = (\underline{v_1}, \underline{v_2})$.

$[\phi]\Gamma \vdash (v_1, v_2) : ([\phi]\tau_1 \times [\phi]\tau_2)^{\mathbb{S}} \xhookrightarrow[\mathbb{S}]{} (\underline{v_1}, \underline{v_2})$ By (Pair)

☞ $[\phi]\Gamma \vdash e : [\phi](\underbrace{(\tau_1 \times \tau_2)^{\mathbb{S}}}_{\tau}) \xhookrightarrow[\mathbb{S}]{} e^{\mathbb{S}}$ and $e^{\mathbb{S}}$ is a value By def. of subst. and $e^{\mathbb{S}} = (\underline{v_1}, \underline{v_2})$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} (\underline{v_1}, \underline{v_2}) : \|\tau_1\|_\phi \times \|\tau_2\|_\phi$ By (TPair)

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\underbrace{(\tau_1 \times \tau_2)^{\mathbb{S}}}_{\tau}\|_\phi$ By def. of $\|-\|_\phi$

- Part (2), changeable mode translation: Let $e^{\mathbb{C}}$ be **let** $r = e^{\mathbb{S}}$ **in write**(r).

$[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{S}]{} e^{\mathbb{S}}$ Above

$\underbrace{(\tau_1 \times \tau_2)^{\mathbb{S}}}_{\tau}$ O.S. By definition of O.S.

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{C}]{}$ **let** $r = e^{\mathbb{S}}$ **in write**(r) By (Write)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_\phi$ Above

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi$ By (TPVar)

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau\|_\phi$ By (TWrite)

$\|\tau\|_\phi = \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By Lemma A.1.1

$\cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By above equality

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By (TLet)

- **Case** $$\boxed{\dfrac{C; \Gamma, x : \tau_1, f : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}} \vdash_\varepsilon e' : \tau_2}{C; \Gamma \vdash_{\varepsilon'} \underbrace{\mathbf{fun}\ f(x) = e'}_{e} : \underbrace{(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}}_{\tau}}\ (\text{SFun})}$$

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

$C; \Gamma, x : \tau_1, f : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}} \vdash_\varepsilon e' : \tau_2$ Subderivation

$[\phi](\Gamma, x : \tau_1, f : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}) \vdash e' : [\phi]\tau_2 \xhookrightarrow[\mathbb{S}]{} \underline{e'}$ By i.h. and $[\phi]\varepsilon = \mathbb{S}$

$\cdot; \|\Gamma, x : \tau_1, f : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}\|_\phi \vdash_{\mathbb{S}} \underline{e'} : \|\tau_2\|_\phi$ ″

$[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{S}]{} \mathbf{fun}^{\mathbb{S}}\ f(x) = \underline{e'}$ By (Fun) and $([\phi]\tau_1 \xrightarrow[\mathbb{S}]{} [\phi]\tau_2)^{\mathbb{S}} = [\phi]\tau$

Let $e^{\mathbb{S}}$ be $\mathbf{fun}^{\mathbb{S}}\ f(x) = \underline{e'}$.

(1)☞ $[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{S}]{} e^{\mathbb{S}}$ and $e^{\mathbb{S}}$ is a value

$\cdot; \|\Gamma\|_\phi, x : \|\tau_1\|_\phi, f : \|\tau_1\|_\phi \xrightarrow[\mathbb{S}]{} \|\tau_2\|_\phi \vdash_{\mathbb{S}} \underline{e'} : \|\tau_2\|_\phi$ By def. of $\|-\|_\phi$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} (\mathbf{fun}^{\mathbb{S}}\ f(x) = \underline{e'}) : \|\tau_1\|_\phi \xrightarrow[\mathbb{S}]{} \|\tau_2\|_\phi$ By (TFun)

(1)☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underbrace{(\mathbf{fun}^{\mathbb{S}}\ f(x) = \underline{e'})}_{e^{\mathbb{S}}} : \|\underbrace{(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}}_{\tau}\|_\phi$ By def. of $\|-\|_\phi$

(2)☞ $[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{C}]{}$ **let** $r = e^{\mathbb{S}}$ **in write**(r) By (Write)

Let $e^{\mathbb{C}}$ be **let** $r = e^{\mathbb{S}}$ **in write**(r).

$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi$ By (TPVar)

$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau\|_\phi$ By (TWrite)

(2)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TLet) and Lemma A.1.1

(b) Suppose $[\phi]\varepsilon = \mathbb{C}$.

$[\phi](\Gamma, x : \tau_1, f : (\tau_1 \underset{\varepsilon}{\to} \tau_2)^{\mathbb{S}}) \vdash \ e' : [\phi]\tau_2 \underset{\mathbb{C}}{\hookrightarrow} \underline{e'}$ By i.h. and $[\phi]\varepsilon = \mathbb{C}$

$\cdot;\|\Gamma, x : \tau_1, f : (\tau_1 \underset{\varepsilon}{\to} \tau_2)^{\mathbb{S}}\|_\phi \vdash_{\mathbb{C}} \underline{e'} : \|\tau_2\|_\phi^{\to\mathbb{C}}$ ″

$[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{fun}^{\mathbb{C}}\ f(x) = \underline{e'}$ By (Fun) and $([\phi]\tau_1 \underset{\mathbb{C}}{\to} [\phi]\tau_2)^{\mathbb{S}} = [\phi]\tau$

Let $e^{\mathbb{S}}$ be $\mathbf{fun}^{\mathbb{C}}\ f(x) = \underline{e'}$.

(1)☞ $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}}$ and $e^{\mathbb{S}}$ is a value

$\cdot;\|\Gamma\|_\phi, x : \|\tau_1\|_\phi, f : \|\tau_1\|_\phi \underset{\mathbb{C}}{\to} \|\tau_2\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{C}} e' : \|\tau_2\|_\phi^{\to\mathbb{C}}$ By def. of $\|-\|_\phi$

(1)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} (\mathbf{fun}^{\mathbb{C}}\ f(x) = \underline{e'}) : \|\tau\|_\phi$ By (TFun)

(2)☞ $[\phi]\Gamma \vdash \ e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in\ write}(r)$ Analogous to (a)

(2)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in\ write}(r) : \|\tau\|_\phi^{\to\mathbb{C}}$ ″

- **Case** $\boxed{\dfrac{C;\Gamma \vdash_\varepsilon v : \tau_1}{C;\Gamma \vdash_\varepsilon \underbrace{\mathbf{inl}\ v}_{e} : \underbrace{(\tau_1 + \tau_2)^{\mathbb{S}}}_{\tau}}\ \text{(SSumLeft)}}$

Part (1):

$C;\Gamma \vdash_\varepsilon v : \tau_1$ Subderivation

$[\phi]\Gamma \vdash \ v : [\phi]\tau_1 \underset{\mathbb{S}}{\hookrightarrow} \underline{v}$ By i.h.

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{v} : \|\tau_1\|_\phi$ ″

☞ $[\phi]\Gamma \vdash \ e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{inl}\ \underline{v}$ By (SumLeft)

Let $e^{\mathbb{S}} = \ \mathbf{inl}\ \underline{v}$.

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{inl}\ \underline{v} : \|\tau_1\|_\phi + \|\tau_2\|_\phi$ By (TSumLeft)

☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underbrace{\mathbf{inl}\ \underline{v}}_{e^{\mathbb{S}}} : \|\underbrace{(\tau_1 + \tau_2)^{\mathbb{S}}}_{\tau}\|_\phi$ By (TSumLeft)

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

- **Case** $\boxed{\dfrac{C;\Gamma \vdash_\varepsilon x : (\tau_1 \times \tau_2)^{\delta} \qquad C \Vdash \delta \leq \varepsilon}{C;\Gamma \vdash_\varepsilon \underbrace{\mathbf{fst}\ x}_{e} : \tau_1}\ \text{(SFst)}}$

  - Suppose $[\phi]\delta = \mathbb{S}$.

    Part (1):

$C; \Gamma \vdash_{\varepsilon} x : (\tau_1 \times \tau_2)^{\delta}$ Subderivation

$[\phi]\Gamma \vdash x : ([\phi]\tau_1 \times [\phi]\tau_2)^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} \underline{x}$ By i.h.

$\cdot; \|\Gamma\|_{\phi} \vdash_{\mathbb{S}} \underline{x} : \|\tau_1\|_{\phi} \times \|\tau_2\|_{\phi}$ ″

☞ $[\phi]\Gamma \vdash e : [\phi]\tau_1 \underset{\mathbb{S}}{\hookrightarrow} \textbf{fst}\ \underline{x}$ By (Fst)

Let $e^{\mathbb{S}} = \textbf{fst}\ \underline{x}$.

☞ $\cdot; \|\Gamma\|_{\phi} \vdash_{\mathbb{S}} \textbf{fst}\ \underline{x} : \|\tau_1\|_{\phi}$ By (TFst)

Part (2): Similar to (SVar):

- If τ_1 O.S., let $e^{\mathbb{C}}$ be $\textbf{let}\ r = \textbf{fst}\ \underline{x}\ \textbf{in}\ \textbf{write}(r)$ and apply rule (Write).
- If τ_1 O.C., let $e^{\mathbb{C}}$ be $\textbf{let}\ r = \textbf{fst}\ \underline{x}\ \textbf{in}\ \textbf{read}\ r\ \textbf{as}\ r'\ \textbf{in}\ \textbf{write}(r')$ and apply rule (ReadWrite).

▪ Suppose $[\phi]\delta = \mathbb{C}$. We have the premise $C \Vdash \delta \leq \varepsilon$, so $[\phi]\varepsilon = \mathbb{C}$; we only need to show part (2).
Part (2):

- If τ_1 O.S., let $e^{\mathbb{C}}$ be $\textbf{read}\ \underline{x}\ \textbf{as}\ x'\ \textbf{in}\ \textbf{let}\ r = \textbf{fst}\ x'\ \textbf{in}\ \textbf{write}(r)$ and apply rule (Read) with (LFst).

  $\ldots, r : \|\tau_1\| \vdash_{\mathbb{S}} r : \|\tau_1\|$ By (TPVar)

  $\ldots, r : \|\tau_1\| \vdash_{\mathbb{C}} \textbf{write}(r) : \|\tau_1\|$ By (TWrite)

  $\ldots, r : \|\tau_1\| \vdash_{\mathbb{C}} \textbf{write}(r) : \|\tau_1\|^{\to\mathbb{C}}$ τ_1 O.S.

  $\|\Gamma\|, x' : \|\tau_1\| \times \|\tau_2\| \vdash_{\mathbb{S}} \textbf{fst}\ x' : \|\tau_1\|$ By (TPVar) then (TFst)

  $\|\Gamma\|, x' : \|\tau_1\| \times \|\tau_2\| \vdash_{\mathbb{C}} \textbf{let}\ r = \textbf{fst}\ x'\ \textbf{in}\ \textbf{write}(r) : \|\tau_1\|^{\to\mathbb{C}}$ By (TLet)

  $\|\Gamma\| \vdash_{\mathbb{S}} \underline{x} : \|(\tau_1 \times \tau_2)^{\mathbb{C}}\|$ By i.h.

  $\|\Gamma\| \vdash_{\mathbb{S}} \underline{x} : (\|\tau_1\| \times \|\tau_2\|)\ \textbf{mod}$ By def. of $\|-\|$

  $\|\Gamma\| \vdash_{\mathbb{C}} \textbf{read}\ \underline{x}\ \textbf{as}\ x'\ \textbf{in}\ \textbf{let}\ r = \textbf{fst}\ x'\ \textbf{in}\ \textbf{write}(r) : \|\tau_1\|^{\to\mathbb{C}}$ By (TRead)

- If τ_1 O.C., then $\|\tau_1\| = \underline{\tau}_1'\ \textbf{mod}$ for some $\underline{\tau}_1'$.
  Let $e^{\mathbb{C}}$ be $\textbf{read}\ \underline{x}\ \textbf{as}\ x'\ \textbf{in}\ \textbf{let}\ r = \textbf{fst}\ x'\ \textbf{in}\ \textbf{read}\ r\ \textbf{as}\ r'\ \textbf{in}\ \textbf{write}(r')$ and apply rule (Read) with (LFst).

  $\ldots, r' : \underline{\tau}_1' \vdash_{\mathbb{C}} \textbf{write}(r') : \underline{\tau}_1'$ By (TPVar), (TWrite)

  $\ldots, r' : \underline{\tau}_1' \vdash_{\mathbb{C}} \textbf{write}(r') : \|\tau_1\|^{\to\mathbb{C}}$ τ_1 O.C.

  $\ldots, r : \underline{\tau}_1'\ \textbf{mod} \vdash_{\mathbb{C}} r : \underline{\tau}_1'\ \textbf{mod}$ By (TPVar)

  $\ldots, r : \underline{\tau}_1'\ \textbf{mod} \vdash_{\mathbb{C}} \textbf{read}\ r\ \textbf{as}\ r'\ \textbf{in}\ \textbf{write}(r) : \|\tau_1\|^{\to\mathbb{C}}$ By (TRead)

  The remaining steps are similar to the τ_1 O.S. subcase immediately above.

- **Case**

$$\dfrac{C;\Gamma \vdash_{\varepsilon'} e_1 : \tau' \qquad C;\Gamma, x : \tau'' \vdash_{\varepsilon} e_2 : \tau \qquad \begin{array}{l} C \Vdash \tau' <: \tau'' \\ C \Vdash \tau' \mathrel{\overset{\circ}{=}} \tau'' \end{array}}{C;\Gamma \vdash_{\varepsilon} \underbrace{\textbf{let}\ x = e_1\ \textbf{in}\ e_2}_{e} : \tau}\ \text{(SLetE)}$$

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

$$\begin{array}{rll}
C;\Gamma \vdash_{\varepsilon'} & e_1 : \tau' & \text{Subderivation}\\
[\phi]\Gamma \vdash & e_1 : [\phi]\tau' \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}} & \text{By i.h.}\\
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} & e^{\mathbb{C}} : \|\tau'\|^{\to\mathbb{C}}_{\phi} & ''
\end{array}$$

$$\begin{array}{rll}
C \Vdash & \tau' <: \tau'' & \text{Premise}\\
 & [\phi]\tau' <: [\phi]\tau'' & \text{By Lemma A.1.2}\\
C \Vdash & \tau' \stackrel{\circ}{=} \tau'' & \text{Premise}\\
 & [\phi]\tau' \stackrel{\circ}{=} [\phi]\tau'' & \text{By Lemma A.1.2}\\
 & [\phi]\tau'' \text{ O.C.} & \text{Subcase (a) assumption}\\
 & [\phi]\tau' = [\phi]\tau'' \text{ or } [\phi]\tau' = \left|[\phi]\tau''\right|^{\mathbb{S}} & \text{By Lemma A.1.3 (2)}
\end{array}$$

If $[\phi]\tau' = [\phi]\tau''$ then:

$$\begin{array}{ll}
[\phi]\Gamma \vdash e_1 : [\phi]\tau' \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}} & \text{By (Mod)}\\
[\phi]\Gamma \vdash e_1 : [\phi]\tau'' \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}} & \text{By } [\phi]\tau' = [\phi]\tau''
\end{array}$$

Otherwise, $[\phi]\tau' = \left|[\phi]\tau''\right|^{\mathbb{S}}$.

$$\begin{array}{ll}
[\phi]\Gamma \vdash e_1 : [\phi]\tau'' \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}} & \text{By (Lift)}
\end{array}$$

Now we have the same judgment no matter which equation Lemma A.1.3 gave us.

$$\begin{array}{rll}
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} & e^{\mathbb{C}} : \|\tau'\|^{\to\mathbb{C}}_{\phi} & \text{Above}\\
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{mod}\ e^{\mathbb{C}} : \|\tau'\|^{\to\mathbb{C}}_{\phi}\ \mathbf{mod} & \text{By (TMod)}
\end{array}$$

$$\begin{array}{rll}
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{mod}\ e^{\mathbb{C}} : \|\left|\tau''\right|^{\mathbb{S}}\|^{\to\mathbb{C}}_{\phi}\ \mathbf{mod} & \\
\text{or } \cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{mod}\ e^{\mathbb{C}} : \|\tau''\|^{\to\mathbb{C}}_{\phi}\ \mathbf{mod} & \text{By } \tau' = \tau'' \text{ or } \left|\tau''\right|^{\mathbb{S}} = \tau'
\end{array}$$

$$\begin{array}{rll}
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{mod}\ e^{\mathbb{C}} : \|\tau''\|^{\to\mathbb{C}}_{\phi}\ \mathbf{mod} & \text{By def. of } \left|-\right|^{\mathbb{S}} \text{ or copying}\\
 & [\phi]\tau'' \text{ O.C.} & \text{Subcase (a) assumption}\\
 & \|\tau''\|^{\to\mathbb{C}}_{\phi} = \|\tau''\|_\phi\ \mathbf{mod} & \text{By Lemma A.1.1}\\
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & \underbrace{\mathbf{mod}\ e^{\mathbb{C}}}_{e^{\mathbb{S}}} : \|\tau''\|_\phi & \text{By above equation}
\end{array}$$

(b) Subcase for $[\phi]\tau''$ O.S.

$$\begin{array}{rll}
C;\Gamma \vdash_{\varepsilon'} & e_1 : \tau' & \text{Subderivation}\\
[\phi]\Gamma \vdash & e_1 : [\phi]\tau' \underset{\mathbb{S}}{\hookrightarrow} e^{\mathbb{S}} & \text{By i.h.}\\
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & e^{\mathbb{S}} : \|\tau'\|_\phi & ''\\
 & [\phi]\tau'' \text{ O.S.} & \text{Subcase (b) assumption}\\
 & [\phi]\tau'' = [\phi]\tau' & \text{By Lemma A.1.3 (1)}\\
\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} & e^{\mathbb{S}} : \|\tau''\|_\phi & \text{By above equation}
\end{array}$$

For both subcases, we have:

$$\begin{array}{rll}
C;\Gamma, x : \tau'' \vdash_{\varepsilon} & e_2 : \tau & \text{Subderivation}\\
[\phi]\Gamma, x : [\phi]\tau'' \vdash & e_2 : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e_2^{\mathbb{S}} & \text{By i.h. and def. of subst.}\\
\cdot;\|\Gamma\|_\phi, x : \|\tau''\|_\phi \vdash_{\mathbb{S}} & e_2^{\mathbb{S}} : \|\tau\|_\phi & \text{By i.h. and def. of } \|-\|_\phi
\end{array}$$

(1)☞ $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e_2^{\mathbb{S}}$ By (LetE)

(1)☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e_2^{\mathbb{S}} : \|\tau\|_\phi$ By (TLet)

$C; \Gamma, x : \tau'' \vdash_\varepsilon e_2 : \tau$ Subderivation

$[\phi]\Gamma, x : [\phi]\tau'' \vdash e_2 : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e_2^{\mathbb{C}}$ By i.h. and def. of subst.

$\cdot; \|\Gamma\|_\phi, x : \|\tau''\|_\phi \vdash_{\mathbb{C}} e_2^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By i.h. and def. of $\|{-}\|_\phi$

(2)☞ $[\phi]\Gamma \vdash e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e_2^{\mathbb{C}}$ By (LetE)

(2)☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e_2^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TLet)

- **Case** $$\boxed{\dfrac{\begin{array}{lll} & C; \Gamma, x : \forall\vec{\alpha}[D].\,\tau'' \vdash_\varepsilon e_2 : \tau & C \Vdash \tau' <: \tau'' \\ \vec{\alpha} \cap FV(C, \Gamma) = \emptyset & C \wedge D; \Gamma \vdash_{\mathbb{S}} v_1 : \tau' & C \Vdash \tau' \mathrel{\mathring{=}} \tau'' \end{array}}{C \wedge \exists\vec{\alpha}.D; \Gamma \vdash_\varepsilon \underbrace{\textbf{let}\ x = v_1\ \textbf{in}\ e_2}_{e} : \tau}\ \text{(SLetV)}}$$

For all $\vec{\delta_i}$ such that $\vec{\alpha} = \vec{\delta_i} \Vdash D$:

- (a) Suppose $[\phi][\vec{\delta_i}/\vec{\alpha}]\tau''$ O.S., that is, this ith monomorphic instance is outer-stable, and will not need a **mod** in the target.

$C \wedge D; \Gamma \vdash_{\mathbb{S}} v_1 : \tau'$ Subderivation

$\vec{\alpha} \cap FV(C, \Gamma) = \emptyset$ Premise

$C \wedge D; [\vec{\delta_i}/\vec{\alpha}]\Gamma \vdash_{\mathbb{S}} v_1 : [\vec{\delta_i}/\vec{\alpha}]\tau'$ By Lemma A.1.2

$[\phi]([\vec{\delta_i}/\vec{\alpha}]\Gamma) \vdash v_1 : [\phi]([\vec{\delta_i}/\vec{\alpha}]\tau') \underset{\mathbb{S}}{\hookrightarrow} \underline{v_i}$ By i.h., using the lemma's guarantee about derivation height

$\vec{\alpha}$ not free in Γ Above disjointness

$[\phi]\Gamma \vdash v_1 : [\phi]([\vec{\delta_i}/\vec{\alpha}]\tau') \underset{\mathbb{S}}{\hookrightarrow} \underline{v_i}$ By above line

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{v_i} : \|[\vec{\delta_i}/\vec{\alpha}]\tau'\|_\phi$ By i.h. and def. of substitution

Let $\underline{e_i}$ be $\underline{v_i}$.

$$\begin{array}{rll}
C \Vdash & \tau' <: \tau'' & \text{Premise} \\
C \Vdash & \tau' \circeq \tau'' & \text{Premise} \\
 & [\phi]\tau' \circeq [\phi]\tau'' & \text{By Lemma A.1.2} \\
 & [\phi]\tau'' \text{ O.S.} & \text{Subcase (a) assumption} \\
 & [\phi]\tau' = [\phi]\tau'' & \text{By Lemma A.1.3 (1)}
\end{array}$$

$$\begin{array}{rll}
[\phi]\Gamma \vdash & v_1 : [\phi]([\overrightarrow{\delta_i}/\vec{\alpha}]\tau') \xhookrightarrow[\mathbb{S}]{} \underline{v_i} & \text{Above} \\
 & \vec{\alpha} \cup \mathsf{dom}(\phi) = \emptyset & \text{By } \vec{\alpha} \cap FV(C,\Gamma) = \emptyset \\
 & & \quad\text{and appropriateness of } \phi \text{ w.r.t. } C \\
[\phi]\Gamma \vdash & v_1 : [\overrightarrow{\delta_i}/\vec{\alpha}]([\phi]\tau') \xhookrightarrow[\mathbb{S}]{} \underline{v_i} & \text{Property of substitution} \\
[\phi]\Gamma \vdash & v_1 : [\overrightarrow{\delta_i}/\vec{\alpha}]([\phi]\tau'') \xhookrightarrow[\mathbb{S}]{} \underline{e_i} & \text{By } [\phi]\tau' = [\phi]\tau'' \text{ and } \underline{e_i} = \underline{v_i}
\end{array}$$

$$\begin{array}{ll}
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]\tau'\|_\phi & \text{Above and } \underline{e_i} = \underline{v_i} \\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'\| & \text{Definition of } \|-\|_\phi
\end{array}$$

$$\begin{array}{ll}
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]([\phi]\tau')\| & \text{By } \vec{\alpha} \cup \mathsf{dom}(\phi) = \emptyset \\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]([\phi]\tau'')\| & \text{By } [\phi]\tau' = [\phi]\tau'' \\
\qquad\qquad \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\phi]([\overrightarrow{\delta_i}/\vec{\alpha}]\tau'')\| & \text{By } \vec{\alpha} \cup \mathsf{dom}(\phi) = \emptyset \\
\qquad\qquad \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]\tau''\|_\phi & \text{Definition of } \|-\|_\phi
\end{array}$$

$$\begin{array}{ll}
\cdot; \|\Gamma\|_\phi, \{\underline{y_k} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]\tau''\|_\phi\}_k \vdash_{\mathbb{S}} \underline{y_i} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]\tau''\|_\phi & \text{By (TPVar)}
\end{array}$$

End of subcase (a)

- (b) Suppose $[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau''$ O.C., that is, this ith monomorphic instance is outer-changeable, and therefore needs a **mod** in the target.

$$\begin{array}{rll}
C \land D; \Gamma \vdash_{\mathbb{S}} & v_1 : \tau' & \text{Subderivation} \\
 & \vec{\alpha} \cap FV(C,\Gamma) = \emptyset & \text{Premise} \\
C \land D; [\overrightarrow{\delta_i}/\vec{\alpha}]\Gamma \vdash_{\mathbb{S}} & v_1 : [\overrightarrow{\delta_i}/\vec{\alpha}]\tau' & \text{By Lemma A.1.2} \\
[\phi]([\overrightarrow{\delta_i}/\vec{\alpha}]\Gamma) \vdash & v_1 : [\phi]([\overrightarrow{\delta_i}/\vec{\alpha}]\tau') \xhookrightarrow[\mathbb{C}]{} e_i^{\mathbb{C}} & \text{By i.h., using the lemma's} \\
 & & \quad\text{guarantee about derivation height}
\end{array}$$

$$\begin{array}{rl}
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e_i^{\mathbb{C}} : \|[\overrightarrow{\delta_i}/\vec{\alpha}]\tau'\|_\phi^{\to\mathbb{C}} & \text{By i.h. and def. of } \|-\|_\phi^{\to\mathbb{C}} \\
{}[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' <: [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'' & \text{By Lemma A.1.2} \\
{}[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' = \left|[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau''\right|^{\mathbb{S}} \text{ or } \ldots = [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'' & \text{By Lemma A.1.3 (2)}
\end{array}$$

If $[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' = \left|[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau''\right|^{\mathbb{S}}$ then:

$$\begin{array}{ll}
[\phi]\Gamma \vdash v_1 : [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' \xhookrightarrow[\mathbb{C}]{} e_i^{\mathbb{C}} & \text{By } \left|[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau''\right|^{\mathbb{S}} = [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' \\
{}[\phi]\Gamma \vdash v_1 : [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'' \xhookrightarrow[\mathbb{S}]{} \textbf{mod } e_i^{\mathbb{C}} & \text{By (Lift)}
\end{array}$$

Otherwise, $[\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' = [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau''$.

$$\begin{array}{ll}
[\phi]\Gamma \vdash v_1 : [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'' \xhookrightarrow[\mathbb{C}]{} e_i^{\mathbb{C}} & \text{By } [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'' = [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau' \\
{}[\phi]\Gamma \vdash v_1 : [\phi][\overrightarrow{\delta_i}/\vec{\alpha}]\tau'' \xhookrightarrow[\mathbb{S}]{} \textbf{mod } e_i^{\mathbb{C}} & \text{By (Mod)}
\end{array}$$

Now, through either (Lift) or (Mod), we have obtained the same judgment.

Let $\underline{e_i}$ be **mod** $e_i^{\mathbb{C}}$.

$$\begin{array}{ll}
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta_i}/\vec{\alpha}]\tau'\|_\phi^{\to\mathbb{C}}\ \mathbf{mod} & \text{By (TMod)} \\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|\left|[\vec{\delta_i}/\vec{\alpha}]\tau''\right|^{\mathbb{S}}\|_\phi^{\to\mathbb{C}}\ \mathbf{mod} & \text{By } \left|[\phi][\vec{\delta_i}/\vec{\alpha}]\tau''\right|^{\mathbb{S}} = [\phi][\vec{\delta_i}/\vec{\alpha}]\tau' \\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : (\|[\vec{\delta_i}/\vec{\alpha}]\tau''\|_\phi^{\to\mathbb{C}})\ \mathbf{mod} & \text{By definition of } \|{-}\|_\phi^{\to\mathbb{C}} \text{ (ϕ-shuffling)} \\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta_i}/\vec{\alpha}]\tau''\|_\phi & \text{By Lemma A.1.1}
\end{array}$$

End of subcase (b)

This ends the “for all $\vec{\delta_i}$” above. We now have translation judgments for each instance, and target typings for each $\underline{e_i}$ and associated variable $\underline{y_i}$.

$$\begin{array}{rll}
C; \Gamma, x : \forall\vec{\alpha}[D].\,\tau'' \vdash_\varepsilon & e_2 : \tau & \text{Subderivation} \\
[\phi]\Gamma, x : \forall\vec{\alpha}[D].\,\tau'' \vdash & e_2 : [\phi]\tau \xhookrightarrow[\mathbb{S}]{} e_2^{\mathbb{S}} & \text{By i.h. and def. of substitution} \\
[\phi]\Gamma, x : \forall\vec{\alpha}[D].\,\tau'' \vdash & e_2 : [\phi]\tau \xhookrightarrow[\mathbb{C}]{} e_2^{\mathbb{C}} & '' \\
\cdot; \|\Gamma\|_\phi, x : \Pi\vec{\alpha}[D].\,[\phi]\tau'' \vdash_{\mathbb{S}} & e_2^{\mathbb{S}} : \|\tau\|_\phi & \text{By i.h. and def. of } \|{-}\|_\phi \\
\cdot; \|\Gamma\|_\phi, x : \Pi\vec{\alpha}[D].\,[\phi]\tau'' \vdash_C & e_2^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}} & ''
\end{array}$$

Let $e_0^{\mathbb{S}}$ be **let** $x =$ **select** $\{\vec{\delta_i} \Rightarrow \underline{e_i}\}_i$ **in** $e_2^{\mathbb{S}}$, and let $e_0^{\mathbb{C}}$ be **let** $x =$ **select** $\{\vec{\delta_i} \Rightarrow \underline{e_i}\}_i$ **in** $e_2^{\mathbb{C}}$.

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \xhookrightarrow[\mathbb{S}]{} e_0^{\mathbb{S}}$ By (LetV)

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \xhookrightarrow[\mathbb{C}]{} e_0^{\mathbb{C}}$ By (LetV)

$$\begin{array}{rll}
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \underline{e_n} : \|[\vec{\delta_i}/\vec{\alpha}]\tau''\|_\phi & \text{By extending } \Gamma \\
\\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{select}\ \{\vec{\delta_i} \Rightarrow \underline{e_i}\}_i : \Pi\vec{\alpha}[D].\,[\phi]\tau'' & \text{By (TSelect)} \\
\cdot; \|\Gamma\|_\phi, x : \Pi\vec{\alpha}[D].\,\|\tau''\|_\phi \vdash_{\mathbb{S}} & e_2^{\mathbb{S}} : \|\tau\|_\phi & \text{Above}
\end{array}$$

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e_0^{\mathbb{S}} : \|\tau\|_\phi$ By (TLet)

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ Analogous to above

- **Case** $$\boxed{\dfrac{C; \Gamma \vdash_{\mathbb{S}} x_1 : \overbrace{(\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\delta}}^{\tau_f} \qquad C; \Gamma \vdash_{\mathbb{S}} x_2 : \tau_1 \qquad \begin{array}{l} C \Vdash \varepsilon' = \varepsilon \\ C \Vdash \delta \lhd \tau \end{array}}{C; \Gamma \vdash_\varepsilon \underbrace{\mathbf{apply}(x_1, x_2)}_{e} : \tau}\ \text{(SApp)}}$$

  We distinguish four subcases “$\mathbb{S}$-$\mathbb{S}$”, “$\mathbb{C}$-$\mathbb{S}$”, “$\mathbb{C}$-$\mathbb{C}$”, “$\mathbb{S}$-$\mathbb{C}$” according to $[\phi]\varepsilon'$ and $[\phi]\delta$ respectively.

  - **Subcase** “$\mathbb{S}$-$\mathbb{S}$” for $[\phi]\varepsilon' = \mathbb{S}$, $[\phi]\delta = \mathbb{S}$.

    Part (1):

$$\begin{array}{rll}
C; \Gamma \vdash_{\mathbb{S}} & x_2 : \tau_1 & \text{Subderivation} \\
[\phi]\Gamma \vdash & x_2 : [\phi]\tau_1 \xhookrightarrow[\mathbb{S}]{} \underline{x_2} & \text{By i.h.} \\
\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \underline{x_2} : \|\tau_1\|_\phi & '' \\
\\
C; \Gamma \vdash_{\mathbb{S}} & x_1 : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\delta} & \text{Subderivation}
\end{array}$$

$$\begin{array}{rll}
 & [\phi]\Gamma \vdash \; x_1 : [\phi]\tau_f \underset{\mathbb{S}}{\hookrightarrow} \underline{x_1} & \text{By i.h.} \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|(\tau_1 \underset{\varepsilon'}{\rightarrow} \tau)^{\delta}\|_\phi & '' \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \underset{[\phi]\varepsilon'}{\rightarrow} [\phi]\tau)^{[\phi]\delta}\| & \text{By defs. of } \|-\|_\phi \text{ and substitution} \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \underset{\mathbb{S}}{\rightarrow} [\phi]\tau)^{\mathbb{S}}\| & \text{Subcase } \mathbb{S}\text{-}\mathbb{S} \text{ assumption} \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|\tau_1\|_\phi \underset{\mathbb{S}}{\rightarrow} \|\tau\|_\phi & \text{By def. of } \|-\| \\
 & \text{Let } e^{\mathbb{S}} = \mathbf{apply}^{\mathbb{S}}(\underline{x_1}, \underline{x_2}). & \\
☞ & [\phi]\Gamma \vdash \; e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{apply}^{\mathbb{S}}(\underline{x_1}, \underline{x_2}) & \text{By (App)} \\
☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{apply}^{\mathbb{S}}(\underline{x_1}, \underline{x_2}) : \|\tau\|_\phi & \text{By (TApp)}
\end{array}$$

Part (2):

(a) Suppose $[\phi]\tau$ O.S.

$$\begin{array}{rll}
☞ & [\phi]\Gamma \vdash \; e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r) & \text{By (Write)} \\
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi & \text{By (TPVar)} \\
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau\|_\phi & \text{By (TWrite)} \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r) : \|\tau\|_\phi & \text{By (TLet)} \\
 & [\phi]\tau \text{ O.S.} & \text{Subcase (a) assumption} \\
☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r) : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By Lemma A.1.1}
\end{array}$$

(b) Suppose $[\phi]\tau$ O.C.

$$\begin{array}{rll}
☞ & [\phi]\Gamma \vdash \; e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in} & \\
 & \qquad \mathbf{write}(r') & \text{By (ReadWrite)} \\
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi & \text{By (TPVar)} \\
 & [\phi]\tau \text{ O.C.} & \text{Subcase (b) assumption}
\end{array}$$

$$\begin{array}{rll}
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi^{\to\mathbb{C}}\ \mathbf{mod} & \text{By Lemma A.1.1} \\
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi, r' : \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{S}} r' : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By (TPVar)} \\
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi, r' : \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{C}} \mathbf{write}(r') : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By (TWrite)} \\
 & \cdot; \|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r') : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By (TRead)} \\
☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in} & \\
 & \qquad \mathbf{write}(r') : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By (TLet)}
\end{array}$$

- **Subcase** “$\mathbb{C}$-$\mathbb{S}$” where $[\phi]\varepsilon' = \mathbb{C}$ and $[\phi]\delta = \mathbb{S}$.

  Part (2):

$$\begin{array}{ll}
[\phi]\Gamma \vdash x_1 : [\phi]\tau_f \underset{\mathbb{S}}{\hookrightarrow} \underline{x_1} & \text{From subcase } \mathbb{S}\text{-}\mathbb{S} \text{ above} \\
[\phi]\Gamma \vdash x_2 : [\phi]\tau_1 \underset{\mathbb{S}}{\hookrightarrow} \underline{x_2} & \text{From subcase } \mathbb{S}\text{-}\mathbb{S} \text{ above}
\end{array}$$

  Let $e^{\mathbb{C}} = \mathbf{apply}^{\mathbb{C}}(\underline{x_1}, \underline{x_2})$.

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{apply}^{\mathbb{C}}(\underline{x_1}, \underline{x_2})$ By (App)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|(\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\delta}\|_\phi^{\to\mathbb{C}}$ By i.h.

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \xrightarrow[{[\phi]\varepsilon'}]{} [\phi]\tau)^{[\phi]\delta}\|^{\to\mathbb{C}}$ By def. of $\|-\|_\phi^{\to\mathbb{C}}$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}}\|^{\to\mathbb{C}}$ By subcase $\mathbb{C}$-$\mathbb{S}$ assumption

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|[\phi]\tau_1\| \xrightarrow[\mathbb{C}]{} \|[\phi]\tau\|^{\to\mathbb{C}}$ By def. of $\|-\|^{\to\mathbb{C}}$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}}$ By def. of $\|-\|_\phi$ and $\|-\|_\phi^{\to\mathbb{C}}$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_2} : \|\tau_1\|_\phi$ From subcase $\mathbb{S}$-$\mathbb{S}$ above

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TApp)

Part (1):

$[\phi]\tau$ O.C. By $[\phi]\varepsilon' = \mathbb{C}$ and barring $(\tau_1' \xrightarrow[\mathbb{C}]{} \tau_2')^{\delta}$ where τ_2' O.S.

$[\phi]\Gamma \vdash\ e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$ Above

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}}$ By (Mod)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ Above

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}\ \mathbf{mod}$ By (TMod)

$[\phi]\tau$ O.C. Above

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi$ By Lemma A.1.1

- **Subcase** "$\mathbb{C}$-$\mathbb{C}$" where $[\phi]\varepsilon' = \mathbb{C}$ and $[\phi]\delta = \mathbb{C}$:

Part (2):

$(\Gamma, x' : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}})(x') = \forall\vec{\alpha}[\mathsf{true}].\,(\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}}$ By def. of Γ

$C \Vdash \exists\vec{\alpha}.\mathsf{true}$ By def. of $\Vdash$

$C; \Gamma, x' : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}} \vdash_{\mathbb{S}} x' : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}}$ By (SVar)

$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \vdash\ x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} x'$ By (Var)

$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \vdash\ x_2 : [\phi]\tau_1 \underset{\mathbb{S}}{\hookrightarrow} \underline{x_2}$ By extending Γ

$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \vdash \mathbf{apply}(x', x_2) : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$ By (App)

$[\phi]\Gamma, x' : \left|([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}}\right|^{\mathbb{S}} \vdash e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$ By defs. of subst., $\left|-\right|^{\mathbb{S}}$

$[\phi]\Gamma \vdash e \rightsquigarrow (x_1 \gg x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}} \vdash \mathbf{apply}(x', x_2))$ By (LApply)

$([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}}$ O.C. By def. of O.C.

$C; \Gamma \vdash_{\mathbb{S}} x_1 : \tau_f$ Subderivation

$[\phi]\Gamma \vdash\ x_1 : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}} \underset{\mathbb{S}}{\hookrightarrow} \underline{x_1}$ By i.h.

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : (\|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}})\ \mathbf{mod}$ ″

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{read}\ \underline{x_1}\ \mathbf{as}\ x'\ \mathbf{in}\ \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$ By (Read)

Let $e^{\mathbb{C}}$ be $\mathbf{read}\ \underline{x_1}\ \mathbf{as}\ x'\ \mathbf{in}\ \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$.

$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{S}} x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}}$ By (TPVar)

$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{S}} \underline{x_2} : \|\tau_1\|_\phi$ By extending Γ

$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{C}} \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2}) : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TApp)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : (\|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}})\ \textbf{mod}$ Above

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \textbf{read}\ \underline{x_1}\ \textbf{as}\ x'\ \textbf{in}\ \textbf{apply}^{\mathbb{C}}(x', \underline{x_2}) : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TRead) (**)

Part (1):

$C \Vdash \delta \lhd \tau$ Premise

$[\phi]\tau$ O.C. By $[\phi]\delta = \mathbb{C}$

$[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{C}]{} e^{\mathbb{C}}$ Above

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{S}]{} \textbf{mod}\ e^{\mathbb{C}}$ By (Mod)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ Above (**)

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \textbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi$ By reasoning in subcase $\mathbb{C}$-$\mathbb{S}$ for Part (1); note that $[\phi]\tau$ O.C.

- **Subcase** "$\mathbb{S}$-$\mathbb{C}$" where $[\phi]\varepsilon' = \mathbb{S}$ and $[\phi]\delta = \mathbb{C}$:

Part (2):

$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{S}]{} [\phi]\tau)^{\mathbb{S}} \vdash \textbf{apply}(x', x_2) : [\phi]\tau \xhookrightarrow[\mathbb{C}]{} e_0^{\mathbb{C}}$ Above with x' for x_1

$[\phi]\Gamma, x' : \left|[\phi]\tau_f\right|^{\mathbb{S}} \vdash [x'/x_1]e : [\phi]\tau \xhookrightarrow[\mathbb{C}]{} e_0^{\mathbb{C}}$ By defs. of $\left|-\right|^{\mathbb{S}}$, subst.

$[\phi]\Gamma \vdash x_1 : ([\phi]\tau_1 \xrightarrow[\mathbb{S}]{} [\phi]\tau)^{\mathbb{C}} \xhookrightarrow[\mathbb{S}]{} \underline{x_1}$ By i.h.

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \xhookrightarrow[\mathbb{C}]{} \textbf{read}\ \underline{x_1}\ \textbf{as}\ x'\ \textbf{in}\ e_0^{\mathbb{C}}$ By (Read)

Let $e^{\mathbb{C}} = \textbf{read}\ \underline{x_1}\ \textbf{as}\ x'\ \textbf{in}\ e_0^{\mathbb{C}}$.

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ Above with x' for x_1

$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{S}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{C}} e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By extending Γ

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : (\|\tau_1\|_\phi \xrightarrow[\mathbb{S}]{} \|\tau\|_\phi^{\to\mathbb{C}})\ \textbf{mod}$ By i.h.

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \textbf{read}\ \underline{x_1}\ \textbf{as}\ x'\ \textbf{in}\ e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TRead)

Part (1): Similar to Part (1) of subcase $\mathbb{C}$-$\mathbb{C}$.

- **Case** $$\boxed{\dfrac{\begin{array}{l} C; \Gamma \vdash_{\mathbb{S}} x_1 : \textbf{int}^{\delta_1} \\ C; \Gamma \vdash_{\mathbb{S}} x_2 : \textbf{int}^{\delta_2} \\ C \Vdash \delta_1 = \delta_2 \qquad \vdash \oplus : \textbf{int} \times \textbf{int} \to \textbf{int} \end{array}}{C; \Gamma \vdash_\varepsilon \oplus(x_1, x_2) : \textbf{int}^{\delta_1}}\ \text{(SPrim)}}$$

If $[\phi]\delta_1 = [\phi]\delta_2 = \mathbb{S}$ then:

Part (1):

$C; \Gamma \vdash_{\mathbb{S}} x_1 : \textbf{int}^{\delta_1}$ Subderivation

$[\phi]\Gamma \vdash x_1 : \textbf{int}^{\delta_1} \xhookrightarrow[\mathbb{S}]{} \underline{x_1}$ By i.h.

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|\textbf{int}^{\delta_1}\|_\phi$ ″

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \textbf{int}$ By $[\phi]\delta_1 = \mathbb{S}$ and def. of $\|-\|$

$[\phi]\Gamma \vdash x_2 : \textbf{int}^{\delta_2} \xhookrightarrow[\mathbb{S}]{} \underline{x_2}$ Similar to above

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_2} : \textbf{int}$ Similar to above

$[\phi]\Gamma \vdash\ e : \mathbf{int}^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} \oplus(\underline{x_1}, \underline{x_2})$ By (Prim)

☞ $[\phi]\Gamma \vdash\ e : [\phi](\mathbf{int}^{\delta_1}) \underset{\mathbb{S}}{\hookrightarrow} \oplus(\underline{x_1}, \underline{x_2})$ By $[\phi]\delta_1 = \mathbb{S}$

Let $e^{\mathbb{S}} = \oplus(\underline{x_1}, \underline{x_2})$.

$\vdash\ \ \oplus : \mathbf{int} \to \mathbf{int}$ Premise

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \oplus(\underline{x_1}, \underline{x_2}) : \mathbf{int}$ By (TPrim)

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \oplus(\underline{x_1}, \underline{x_2}) : \|\mathbf{int}^{\delta_1}\|_\phi$ By $[\phi]\delta_1 = \mathbb{S}$ and def. of $\|{-}\|$

Part (2): Similar to (SPair), where the outer level is stable ($\tau = \mathbf{int}^{\delta_1} = \mathbf{int}^{\mathbb{S}}$).

If $[\phi]\delta_1 = [\phi]\delta_2 = \mathbb{C}$ then:

Part (2):

$[\phi]\Gamma, y_1{:}\mathbf{int}^{\mathbb{S}}, y_2{:}\mathbf{int}^{\mathbb{S}} \vdash \oplus(y_1, y_2) : \mathbf{int}^{\mathbb{C}} \underset{\mathbb{S}}{\hookrightarrow} \oplus(y_1, y_2)$ By (Var), (Var), (Prim)

$[\phi]\Gamma, \ldots \vdash \oplus(y_1, y_2) : \mathbf{int}^{\mathbb{C}} \underset{\mathbb{C}}{\hookrightarrow} \mathbf{let}\ r = \oplus(y_1, y_2)\ \mathbf{in}\ \mathbf{write}(r)$ By (Write)

$[\phi]\Gamma, y_1{:}\mathbf{int}^{\mathbb{S}} \vdash \oplus(y_1, y_2) : \mathbf{int}^{\mathbb{C}} \underset{\mathbb{C}}{\hookrightarrow} \begin{array}{l}\mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in} \\ \quad \mathbf{let}\ r = \oplus(y_1, y_2)\ \mathbf{in}\ \mathbf{write}(r)\end{array}$ By (LPrimop2) then (Read)

☞ $[\phi]\Gamma \vdash \oplus(y_1, y_2) : \mathbf{int}^{\mathbb{C}} \underset{\mathbb{C}}{\hookrightarrow} \begin{array}{l}\mathbf{read}\ \underline{x_1}\ \mathbf{as}\ y_1\ \mathbf{in}\ \mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in} \\ \quad \mathbf{let}\ r = \oplus(y_1, y_2)\ \mathbf{in}\ \mathbf{write}(r)\end{array}$ By (LPrimop1) then (Read)

$\cdot; \|\Gamma\|_\phi, y_1{:}\mathbf{int}, y_2{:}\mathbf{int}, r{:}\mathbf{int} \vdash_{\mathbb{C}} \mathbf{write}(r) : \mathbf{int}$ By (TVar) then (TWrite)

$\cdot; \|\Gamma\|_\phi, y_1{:}\mathbf{int}, y_2{:}\mathbf{int} \vdash_{\mathbb{S}} \oplus(y_1, y_2) : \mathbf{int}$ By (TVar) and (TVar), then (TPrim)

$\cdot; \|\Gamma\|_\phi, y_1{:}\mathbf{int}, y_2{:}\mathbf{int} \vdash_{\mathbb{C}} (\mathbf{let}\ r = \oplus(y_1, y_2)\ \mathbf{in}\ \mathbf{write}(r)) : \mathbf{int}$ By (TLet)

$\cdot; \|\Gamma\|_\phi, y_1{:}\mathbf{int} \vdash_{\mathbb{C}} (\mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in}\ \mathbf{let}\ r = \ldots\ \mathbf{in}\ \mathbf{write}(r)) : \mathbf{int}$ By (TRead)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} \begin{array}{c}\mathbf{read}\ \underline{x_1}\ \mathbf{as}\ y_1\ \mathbf{in}\ \mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in} \\ \mathbf{let}\ r = \oplus(y_1, y_2)\ \mathbf{in}\ \mathbf{write}(r)\end{array} : \mathbf{int}$ By (TRead)

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}}\ '' : \|\mathbf{int}^{\delta_1}\|_\phi^{\to\mathbb{C}}$ By def. of $\|{-}\|^{\to\mathbb{C}}$ and $[\phi]\delta_1 = \mathbb{C}$

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

- **Case**

$$\frac{C;\Gamma \vdash_{\mathbb{S}} x : (\tau_1 + \tau_2)^{\delta} \qquad \begin{array}{l} C;\Gamma, x_1 : \tau_1 \vdash_{\varepsilon} e_1 : \tau \\ C;\Gamma, x_2 : \tau_2 \vdash_{\varepsilon} e_2 : \tau \end{array} \qquad \begin{array}{l} C \Vdash \delta \le \varepsilon \\ C \Vdash \delta \lhd \tau \end{array}}{C;\Gamma \vdash_{\varepsilon} \underbrace{\mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}}_{e} : \tau}\ (\text{SCase})$$

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

$C;\Gamma \vdash_{\mathbb{S}} x : (\tau_1 + \tau_2)^{\delta}$ Subderivation

$[\phi]\Gamma \vdash\ x : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} \underline{x}$ By i.h.

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x} : \|\tau_1\|_\phi + \|\tau_2\|_\phi$ ″

$$\begin{array}{rll}
C; \Gamma, x_1 : \tau_1 \vdash_\varepsilon & e_1 : \tau & \text{Subderivation} \\
[\phi]\Gamma, x_1 : [\phi]\tau_1 \vdash & e_1 : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e_1^{\mathbb{S}} & \text{By i.h.} \\
\cdot; \|\Gamma\|_\phi, x_1 : \|\tau_1\|_\phi \vdash_{\mathbb{S}} & e_1^{\mathbb{S}} : \|\tau\|_\phi & '' \\
[\phi]\Gamma, x_1 : [\phi]\tau_1 \vdash & e_1 : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e_1^{\mathbb{C}} & '' \\
\cdot; \|\Gamma\|_\phi, x_1 : \|\tau_1\|_\phi \vdash_{\mathbb{C}} & e_1^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}} & ''
\end{array}$$

$$\begin{array}{rll}
C; \Gamma, x_2 : \tau_2 \vdash_\varepsilon & e_2 : \tau & \text{Subderivation} \\
[\phi]\Gamma, x_2 : [\phi]\tau_2 \vdash & e_2 : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} e_2^{\mathbb{S}} & \text{By i.h.} \\
\cdot; \|\Gamma\|_\phi, x_2 : \|\tau_2\|_\phi \vdash_{\mathbb{S}} & e_2^{\mathbb{S}} : \|\tau\|_\phi & '' \\
[\phi]\Gamma, x_2 : [\phi]\tau_2 \vdash & e_2 : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e_2^{\mathbb{C}} & '' \\
\cdot; \|\Gamma\|_\phi, x_2 : \|\tau_2\|_\phi \vdash_{\mathbb{C}} & e_2^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}} & ''
\end{array}$$

$$\begin{array}{lrll}
(1)☞ & [\phi]\Gamma \vdash & e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{case}\ \underline{x}\ \mathbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{S}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{S}}\} & \text{By (Case)} \\
(1)☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{case}\ \underline{x}\ \mathbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{S}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{S}}\} : \|\tau\|_\phi & \text{By (TCase)} \\
(2)☞ & [\phi]\Gamma \vdash & e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{case}\ \underline{x}\ \mathbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{C}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{C}}\} & \text{By (Case)} \\
(2)☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} & \mathbf{case}\ \underline{x}\ \mathbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{C}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{C}}\} : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By (TCase)}
\end{array}$$

(b) Suppose $[\phi]\delta = \mathbb{C}$.

$$\begin{array}{rll}
[\phi]\Gamma, x' : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{S}} \vdash & [x'/x]e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} e_0^{\mathbb{C}} & \text{Above but with } x' \text{ for the first } x \\
\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi + \|\tau_2\|_\phi \vdash_{\mathbb{C}} & e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}} & ''
\end{array}$$

$$\begin{array}{ll}
[\phi]\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} & \\
\qquad \rightsquigarrow (x \gg x' : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{C}} & \\
\qquad\qquad \mathbf{case}\ x'\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}) & \text{By (LCase)}
\end{array}$$

$$\begin{array}{lrll}
 & & ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{C}}\ \text{O.C.} & \text{By def. of O.C.} \\
 & [\phi]\Gamma \vdash & x : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{C}} \underset{\mathbb{S}}{\hookrightarrow} \underline{x} & \text{By i.h.} \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \underline{x} : \|(\tau_1 + \tau_2)^{\mathbb{C}}\|_\phi & '' \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \underline{x} : (\|\tau_1\|_\phi + \|\tau_2\|_\phi)\ \mathbf{mod} & \text{By def. of } \|-\|_\phi \\
(2)☞ & [\phi]\Gamma \vdash & e : [\phi]\tau \underset{\mathbb{C}}{\hookrightarrow} \mathbf{read}\ \underline{x}\ \mathbf{as}\ x'\ \mathbf{in}\ e_0^{\mathbb{C}} & \text{By (Read)} \\
 & \text{Let } e^{\mathbb{C}} = & \mathbf{read}\ \underline{x}\ \mathbf{as}\ x'\ \mathbf{in}\ e_0^{\mathbb{C}}. & \\
(2)☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} & e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}} & \text{By (TRead)}
\end{array}$$

$$\begin{array}{lrll}
 & C \Vdash & \delta \lhd \tau & \text{Premise} \\
 & & [\phi]\tau\ \text{O.C.} & \text{By } [\phi]\delta = \mathbb{C} \text{ and def. of O.C.} \\
(1)☞ & [\phi]\Gamma \vdash & e : [\phi]\tau \underset{\mathbb{S}}{\hookrightarrow} \mathbf{mod}\ e^{\mathbb{C}} & \text{By (Mod)} \\
 & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}\ \mathbf{mod} & \text{By (TMod)} \\
 & & \|\tau\|_\phi^{\to\mathbb{C}}\ \mathbf{mod} = \|\tau\|_\phi & \text{By Lemma A.1.1} \\
(1)☞ & \cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} & \mathbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi & \text{By above equation}
\end{array}$$

□

A.2 Proof of translation soundness

In this proof, we use the store substitution operation $[\rho]e$, which replaces locations ℓ with **mod**s of locations' contents (Definition 6.5.2).

Lemma A.2.1 (Stores Are Monotonic)**.** *If* $\rho_1 \vdash e \Downarrow (\rho_2 \vdash v)$ *then there exists* ρ' *such that* $\rho_2 = \rho_1, \rho'$.

Proof
By induction on the given derivation. All cases are straightforward. □

Lemma A.2.2 (Commuting)**.** *If* e_1 *and* e *are target expressions, then* $\llbracket [e_1/x]e \rrbracket = [\llbracket e_1 \rrbracket \,/\, x]\; \llbracket e \rrbracket$.

Proof
By induction on e, using the definitions of back-translation and substitution. □

Lemma A.2.3. *For all closed target values* w*, the back-translation* $\llbracket w \rrbracket$ *is a (source) value.*

Proof
By induction on w. □

Lemma A.2.4. *The following equivalences hold:*

(i) $(\textbf{let}\ x = e_0\ \textbf{in}\ x) \sim e_0'$, if $e_0 \sim e_0'$
(ii) $(\textbf{let}\ x' = x_1\ \textbf{in}\ \textbf{apply}(x', x_2)) \;\sim\; \textbf{apply}(x_1, x_2)$
(iii) $(\textbf{let}\ x' = x_1\ \textbf{in}\ \oplus(x', x_2)) \;\sim\; \oplus(x_1, x_2)$
(iv) $(\textbf{let}\ x' = x_2\ \textbf{in}\ \oplus(x_1, x')) \;\sim\; \oplus(x_1, x_2)$
(v) $(\textbf{let}\ x' = x\ \textbf{in}\ \textbf{fst}\ x') \;\sim\; \textbf{fst}\ x$
(vi) $(\textbf{let}\ x' = x\ \textbf{in}\ \textbf{case}\ x'\ \textbf{of}\,\{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}) \;\sim\; \textbf{case}\ x\ \textbf{of}\,\{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}$

Proof
Straightforward, using inversion on the given source evaluation derivation, and applying the appropriate evaluation rules. □

Lemma A.2.5. *If* $\Gamma \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$ *then* $\llbracket e' \rrbracket \sim e$.

Proof
By induction on the given derivation.

For (Apply), and other rules for constructs at which substitution happens during evaluation, we use Lemma A.2.2.

The other interesting cases are those in which e cannot be *exactly* $\llbracket e' \rrbracket$: (Read), (Write), and (ReadWrite).

For (Write), $\llbracket e' \rrbracket = \llbracket \textbf{let}\ r = e^{\mathbb{S}}\ \textbf{in}\ \textbf{write}(r) \rrbracket = \textbf{let}\ r = \llbracket e^{\mathbb{S}} \rrbracket\ \textbf{in}\ \llbracket \textbf{write}(r) \rrbracket = \textbf{let}\ r = \llbracket e^{\mathbb{S}} \rrbracket\ \textbf{in}\ r$, but we only have $\llbracket e^{\mathbb{S}} \rrbracket \sim e'$ (by i.h.), so we use Lemma A.2.4 (i).

Rule (ReadWrite) creates two **let**s, so we use the lemma twice.

For (Read), we use Lemma A.2.4. If the rule from Figure 6.3 used to derive the first premise was (LApply), we use part (ii) of the lemma; if (LPrimop1), part (iii); if (LPrimop2), part (iv); if (LFst), part (v); if (LCase), part (vi). □

Roughly, we want to show that, if a target expression e' evaluates to some target value w, that the back-translation $[\![e']\!]$, a source expression, evaluates to the back-translation of w (after replacing locations ℓ in w with their corresponding values). This does not hold in general: the back-translation of **select** uses only the *first* arm, under the assumption that all the arms are "essentially" the same. That is, the back-translation assumes the arms differ only in their use of modifiables, and in how they instantiate polymorphic variables. Fortunately, this condition does hold for all target expressions produced by our translation. We call this condition ***select**-uniformity*.

Definition A.2.6 (Select Uniformity)**.** A target expression e' is ***select**-uniform* if and only if, for all subexpressions of e' of the form **select** $\{(\vec{\alpha}{=}\vec{\delta_1}) \Rightarrow e_1, \ldots, (\vec{\alpha}{=}\vec{\delta_n}) \Rightarrow e_n\}$, all the arms have equivalent back-translations:

$$[\![e_1]\!] \sim [\![e_2]\!] \sim \ \ldots\ \sim [\![e_n]\!]$$

Lemma A.2.7. *If* $\Gamma \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$ *then* e' *is **select**-uniform.*

Proof

By induction on the given derivation. All cases are completely straightforward, except the case for (LetV), where we use Lemma A.2.5. □

Theorem 6.5.3 (Evaluation Soundness)**.**
If $\rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FLV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is **select**-uniform*
then $[\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$.

Proof

By induction on the given derivation. Wherever we apply the induction hypothesis, it is easy to show the condition of **select**-uniformity; in what follows, we omit this reasoning.

- **Case** $$\boxed{\dfrac{}{\rho \vdash w \Downarrow (\rho \vdash w)}\ \text{(TEvValue)}}$$

  Stores only contain values, so $[\rho]w$ is a value. By Lemma A.2.3, $[\![[\rho]w]\!]$ is some source value v. By (SEvValue), $[\![[\rho]w]\!] \Downarrow [\![[\rho]w]\!]$, which was to be shown.

- **Case** $$\boxed{\dfrac{\begin{array}{l}\rho \vdash e_1 \Downarrow (\rho_1 \vdash \mathbf{fun}^{\varepsilon}\ f(x) = e_0) \\ \rho_1 \vdash e_2 \Downarrow (\rho_2 \vdash w_2) \\ \rho_2 \vdash [(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0 \Downarrow (\rho' \vdash w)\end{array}}{\rho \vdash \mathbf{apply}^{\varepsilon}(e_1, e_2) \Downarrow (\rho' \vdash w)}\ \text{(TEvApply)}}$$

$$\begin{array}{ll}
\rho \vdash e_1 \Downarrow (\rho_1 \vdash \mathbf{fun}^{\varepsilon}\ f(x) = e_0) & \text{Subd.} \\
\quad \llbracket[\rho]e_1\rrbracket \Downarrow \llbracket[\rho_1](\mathbf{fun}^{\varepsilon}\ f(x) = e_0)\rrbracket & \text{By i.h.} \\
\quad \llbracket[\rho]e_1\rrbracket \Downarrow \llbracket\mathbf{fun}^{\varepsilon}\ f(x) = [\rho_1]e_0\rrbracket & \text{By definition of } [-] \\
\quad \llbracket[\rho]e_1\rrbracket \Downarrow \mathbf{fun}\ f(x) = \llbracket[\rho_1]e_0\rrbracket & \text{By definition of } \llbracket-\rrbracket \\
\quad \llbracket[\rho]e_1\rrbracket \Downarrow \mathbf{fun}\ f(x) = \llbracket[\rho']e_0\rrbracket & \text{Monotonicity}
\end{array}$$

$$\begin{array}{ll}
\rho_1 \vdash e_2 \Downarrow (\rho_2 \vdash w_2) & \text{Subd.} \\
\quad \llbracket[\rho_1]e_2\rrbracket \Downarrow \llbracket[\rho_2]w_2\rrbracket & \text{By i.h.} \\
\quad \llbracket[\rho]e_2\rrbracket \Downarrow \llbracket[\rho']w_2\rrbracket & \mathsf{FLV}(e_2) \subseteq \mathsf{dom}(\rho) \text{ and monotonicity}
\end{array}$$

$$\begin{array}{ll}
\rho_2 \vdash [(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0 \Downarrow (\rho' \vdash w) & \text{Subd.} \\
\quad \llbracket[\rho_2][(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0\rrbracket \Downarrow \llbracket[\rho']w\rrbracket & \text{By i.h.} \\
\quad \llbracket[\rho'][(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0\rrbracket \Downarrow \llbracket[\rho']w\rrbracket & \text{Monotonicity}
\end{array}$$

$$\begin{array}{ll}
\quad [(\mathbf{fun}\ f(x) = \llbracket[\rho']e_0\rrbracket)/f]\ [\llbracket[\rho']w_2\rrbracket/x]\ \llbracket[\rho']e_0\rrbracket \Downarrow \llbracket[\rho']w\rrbracket & \text{Properties of substitution, } [-],\ \llbracket-\rrbracket
\end{array}$$

☞ $\mathbf{apply}(\llbracket[\rho]e_1\rrbracket, \llbracket[\rho]e_2\rrbracket) \Downarrow \llbracket[\rho']w\rrbracket$ By (SEvApply)

- **Cases** (TEvPair), (TEvSumLeft), (TEvPrimop), (TEvFst), (TEvCaseLeft): By similar reasoning as in the (TEvApply) case, but simpler.
- **Case** TEvLet: By similar reasoning to the (TEvApply) case, but slightly simpler.
- **Case** $$\boxed{\dfrac{\rho \vdash e^{\mathbb{C}} \Downarrow (\rho_0' \vdash w)}{\rho \vdash \mathbf{mod}\ e^{\mathbb{C}} \Downarrow (\underbrace{(\rho_0', \ell \mapsto w)}_{\rho'} \vdash \ell)}\ \text{(TEvMod)}}$$

$$\begin{array}{ll}
\rho \vdash e^{\mathbb{C}} \Downarrow (\rho_0' \vdash w) & \text{Subd.} \\
\quad \llbracket[\rho]e^{\mathbb{C}}\rrbracket \Downarrow \llbracket[\rho_0']w\rrbracket & \text{By i.h.} \\
\quad \llbracket[\rho]e^{\mathbb{C}}\rrbracket \Downarrow \llbracket[\rho_0', \ell \mapsto w]\ell\rrbracket & \text{By def. of } [-] \\
\quad \llbracket[\rho]e^{\mathbb{C}}\rrbracket \Downarrow \llbracket[\rho']\ell\rrbracket & \rho' = \rho_0', \ell \mapsto w \\
\quad \llbracket\mathbf{mod}\ ([\rho]e^{\mathbb{C}})\rrbracket \Downarrow \llbracket[\rho']\ell\rrbracket & \text{By def. of } \llbracket-\rrbracket
\end{array}$$

☞ $\llbracket[\rho](\mathbf{mod}\ e^{\mathbb{C}})\rrbracket \Downarrow \llbracket[\rho']\ell\rrbracket$ By def. of $[-]$

- **Case** $$\boxed{\dfrac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash \ell) \qquad \rho_1 \vdash [\rho_1(\ell)/x']e^{\mathbb{C}} \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{read}\ e_1\ \mathbf{as}\ x'\ \mathbf{in}\ e^{\mathbb{C}} \Downarrow (\rho' \vdash w)}\ \text{(TEvRead)}}$$

$$\begin{array}{ll}
\rho \vdash e_1 \Downarrow (\rho_1 \vdash \ell) & \text{Subd.} \\
\quad \llbracket[\rho]e_1\rrbracket \Downarrow \llbracket[\rho_1]\ell\rrbracket & \text{By i.h.}
\end{array}$$

$$\begin{array}{ll}
\rho_1 \vdash [\rho_1(\ell)/x']e^{\mathbb{C}} \Downarrow (\rho' \vdash w) & \text{Subd.} \\
\quad \llbracket[\rho_1][\rho_1(\ell)\,/\,x']\,e^{\mathbb{C}}\rrbracket\ \Downarrow\ \llbracket[\rho']w\rrbracket & \text{By i.h.} \\
\quad \llbracket[[\rho_1]\ell\,/\,x']\,[\rho_1]e^{\mathbb{C}}\rrbracket\ \Downarrow\ \llbracket[\rho']w\rrbracket & \text{By def. of subst.} \\
\quad \llbracket[[\rho_1]\ell\,/\,x']\,[\rho]e^{\mathbb{C}}\rrbracket\ \Downarrow\ \llbracket[\rho']w\rrbracket & \text{By } \mathsf{FLV}(e^{\mathbb{C}}) \subseteq \mathsf{dom}(\rho) \text{ and Lemma A.2.1} \\
\quad [\llbracket[\rho_1]\ell\rrbracket/x']\,\llbracket[\rho]e^{\mathbb{C}}\rrbracket\ \Downarrow\ \llbracket[\rho']w\rrbracket & \text{By Lemma A.2.2}
\end{array}$$

$\mathbf{let}\ x' = [\![[\rho]e_1]\!]\ \mathbf{in}\ [\![[\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By (SEvLet)

$[\![\mathbf{read}\ [\rho]e_1\ \mathbf{as}\ x'\ \mathbf{in}\ [\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $[\![-]\!]$

☞ $[\![[\rho](\mathbf{read}\ e_1\ \mathbf{as}\ x'\ \mathbf{in}\ e^{\mathbb{C}})]\!] \Downarrow [\![[\rho']w]\!]$ By def. of subst.

- **Case** $\boxed{\dfrac{\rho \vdash e_0 \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{write}(e_0) \Downarrow (\rho' \vdash w)}\ \text{(TEvWrite)}}$

$\rho \vdash e_0 \Downarrow (\rho' \vdash w)$ Subd.

$[\![[\rho]e_0]\!] \Downarrow [\![[\rho']w]\!]$ By i.h.

$[\![\mathbf{write}([\rho]e_0)]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $[\![-]\!]$

☞ $[\![[\rho]\mathbf{write}(e_0)]\!] \Downarrow [\![[\rho']w]\!]$ By def. of subst.

- **Case** $\boxed{\dfrac{\rho \vdash e_i \Downarrow (\rho' \vdash w)}{\rho \vdash (\mathbf{select}\,\{\ldots, (\vec{\alpha}{=}\vec{\delta}) \Rightarrow e_i, \ldots\})[\vec{\alpha} = \vec{\delta}] \Downarrow (\rho' \vdash w)}\ \text{(TEvSelect)}}$

$\rho \vdash e_i \Downarrow (\rho' \vdash w)$ Subd.

$[\![[\rho]e_i]\!] \Downarrow [\![[\rho']w]\!]$ By i.h.

$[\![[\rho]e_1]\!] \sim [\![[\rho]e_i]\!]$ By **select**-uniformity

$[\![[\rho]e_1]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $\sim$

☞ $[\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $[\![-]\!]$ □

Theorem 6.5.4 (Translation Soundness)**.**
If $\cdot \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$ *and* $\cdot \vdash e' \Downarrow (\rho' \vdash w)$ *then* $e \Downarrow [\![[\rho']w]\!]$.

Proof
By Lemma A.2.7, e' is **select**-uniform. The empty store $\cdot$ is trivially **select**-uniform. By Theorem 6.5.3, $[\![[\cdot]e']\!] \Downarrow [\![[\rho']w]\!]$. Since the empty store acts as an identity substitution,

$$[\![e']\!] \Downarrow [\![[\rho']w]\!]$$

By Lemma A.2.5, $[\![e']\!] \sim e$; by Definition 6.5.1, $e \Downarrow [\![[\rho']w]\!]$. □

A.3 Proof of costed soundness

Theorem 6.6.6. *If* `trans` *$(e, \epsilon) = e'$ then e' is deeply* 1*-bounded.*

Proof
By lexicographic induction on e and ϵ, with $\mathbb{S}$ considered smaller than $\mathbb{C}$.

- If $e \in \{n, x, (v_1, v_2), \mathbf{fun} \ldots, \mathbf{inl}\ v\}$ then:
    - If $\epsilon = \mathbb{S}$ then `trans` uses one of its first 5 cases, and the result follows by induction. ($HC(e') = 0$ except for (Var) where $HC(e') = 1$ is possible.)

- If $\epsilon = \mathbb{C}$ then, for $n/(v_1, v_2)/$**fun**/**inl** v, `trans` uses its last case and applies (Write). A **let** has head cost 0 (the **let** appears in the back-translation), so $HC(e') = 0$; however, the head cost of the **write** subterm is 1, so the term is deeply 1-bounded.

  For x, `trans` uses either (Write) or (ReadWrite); in both cases, e' is deeply 1-bounded.

- If e has the form **let** $x = e_1$ **in** e_2, then $HC(e') = 0$; by induction, e_1' and e_2' are deeply 1-bounded, so e' is deeply 1-bounded.
- if e has the form $\oplus(x_1, x_2)$, then: For the stable case, e' is a $\oplus$ so $HC(e') = 0$. For the changeable case, `trans` applies (Var), (Var), (Prim), (Write), (Read) with (LPrimop2), and (Read) with (LPrimop1), producing

$$e' = \textbf{read}\ \underline{x_1}\ \textbf{as}\ y_1\ \textbf{in read}\ \underline{x_2}\ \textbf{as}\ y_2\ \textbf{in let}\ r = \oplus(y_1, y_2)\ \textbf{in write}(r)$$

  so (assuming $HC(\underline{x_1}), HC(\underline{x_2}) \leq 1$) we have $HC(e') = 0$ and all inner head costs bounded by 1.
- If e is an **apply**, then:
  - Case ($\mathbb{S}$, $\mathbb{S}$, $\mathbb{S}$): Here e' is an $\textbf{apply}^{\mathbb{S}}$, so $HC(e') = 0$.
  - Case ($\mathbb{C}$, $\mathbb{S}$, $\mathbb{C}$): Here e' is an $\textbf{apply}^{\mathbb{C}}$, so $HC(e') = 0$.
  - Case ($\mathbb{S}$, $\mathbb{S}$, $\mathbb{C}$): Either (Write) or (ReadWrite), after switching to $\mathbb{S}$ mode, meaning one of the ($-$, $-$, $\mathbb{S}$) cases—which each generate a subterm whose *HC* is 0. For (Write), the **write** subterm of e' has head cost 1, and likewise for (ReadWrite).

    The rules (LApply) and (LCase) guarantee that the **read** has the correct form for $HC(e')$ to be defined.
  - Case (ϵ', $\mathbb{C}$, $\mathbb{C}$): Applies (Read) after devolving to (ϵ', $\mathbb{S}$, $\mathbb{C}$) which returns a term with $HC(e') \leq 1$ (zero if $\epsilon' = \mathbb{C}$, and 1 if $\epsilon' = \mathbb{S}$). Applying (Read) yields a term whose *HC* is 0, and which is deeply 1-bounded.

    Note that $HC(e')$ is defined for the same reason as in the ($\mathbb{S}$, $\mathbb{S}$, $\mathbb{C}$) subcase.
  - Case ($\mathbb{C}$, $\mathbb{S}$, $\mathbb{S}$): Devolves to the ($\mathbb{C}$, $\mathbb{S}$, $\mathbb{C}$) case, yielding a subterm with *HC* of 0; the algorithm then uses (Mod), yielding $HC(e') = 1 + 0 = 1$.
  - Case (ϵ', $\mathbb{C}$, $\mathbb{S}$): Devolves to the (ϵ', $\mathbb{C}$, $\mathbb{C}$) case, where $HC = 0$, then applies (Mod), yielding $HC(e') \leq 1$.

  (Note: We do not use the induction hypothesis as we "devolve"; we are merely reasoning by cases.)
- If $e =$ **fst** x where $x : (\tau_1 \times \tau_2)^{\delta}$, then:
  - Case ($\mathbb{S}$, $\mathbb{S}$): We use (Fst), yielding $HC(e') = 0$.
  - Case ($\mathbb{S}$, $\mathbb{C}$): If τ_1 O.S. then $HC(e') = 0$ (Write). If τ_1 O.C. then we use (ReadWrite), which has *HC* of 0.
  - Case ($\mathbb{C}$, $\mathbb{C}$): We use (Read) with (LFst) and go to the ($\mathbb{S}$, $\mathbb{C}$) case with a new variable x'. The *HC* for the ($\mathbb{S}$, $\mathbb{C}$) case is 0. Using (Read) in this case also has head cost 0.

- If e is a **case** on a variable $x : \tau$, then:
  - If τ is outer stable, the proof is straightforward.
  - If τ is outer changeable, the algorithm applies rule (Read), recursing with $x : \left|\tau\right|^{\mathbb{S}}$, which will apply rule (Case). A **case** has *HC* of 0, so (Read) produces e' where $\mathit{HC}(e') = 0$ and e' is deeply 1-bounded. □

In the following proofs, we assume that in any target evaluation $\rho \vdash e' \Downarrow (\rho' \vdash w)$, the target expression e' is closed (that is, it has no free program variables x, though of course it may contain store locations $\ell \in \mathsf{dom}(\rho)$).

Theorem 6.6.7 (Cost Result)**.** *Given* $\mathcal{D} :: \rho \vdash e' \Downarrow (\rho' \vdash w)$
where for every subderivation $\mathcal{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathcal{D}$ *(including* $\mathcal{D}$*),* $\mathit{HC}(\mathcal{D}^*) \leq k$*,*
then the number of dirty rule applications in $\mathcal{D}$ *is at most* $\frac{k}{k+1}W(\mathcal{D})$*.*

Proof
By the definition of $\mathit{HC}(\mathcal{D})$, if $\mathcal{D}$ is deeply k-bounded, there is no contiguous region of $\mathcal{D}$ consisting only of dirty rule applications that is larger than k; since the only rule with no premises is TEvValue, and TEvValue is clean, at least one of every $k+1$ rule applications is clean. $W(\mathcal{D})$ simply counts the total number of rule applications, so $\mathcal{D}$ contains at least $\frac{W(\mathcal{D})}{k+1}$ clean rule applications, so no more than $\frac{k}{k+1}W(\mathcal{D})$ of $\mathcal{D}$'s rule applications are dirty. □

Theorem 6.6.8 (Costed Stable Evaluation)**.**
If $\mathcal{D} :: \rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select****-uniform*
and $[\rho]e$ *is deeply* k*-bounded*
then $\mathcal{D}' :: [\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$
and $[\rho']w$ *is deeply* k*-bounded*
and for every subderivation $\mathcal{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathcal{D}$ *(including* $\mathcal{D}$*),*
$\qquad \mathit{HC}(\mathcal{D}^*) \leq \mathit{HC}(e^*) \leq k$*,*
and the number of clean rule applications in $\mathcal{D}$ *equals* $W(\mathcal{D}')$*.*

Proof
The differences from Theorem 6.5.3 require additional reasoning:

- The 7 cases for the "clean" rules (TEvValue), (TEvPair), (TEvSumLeft), (TEvPrimop), (TEvFst), (TEvCaseLeft), and (TEvApply) are straightforward: the induction hypothesis shows that the *HC* condition holds for proper subderivations of $\mathcal{D}$, and $\mathit{HC}(\mathcal{D}) = 0$ by definition of $\mathit{HC}(-)$, which is certainly not greater than $\mathit{HC}(e^*)$. Finally, each one of these cases generates a single application of an SEv* rule, which together with the i.h. satisfies the last condition (that the number of clean rule applications in $\mathcal{D}$ equals $W(\mathcal{D}')$.
  For (TEvCaseLeft) and (TEvApply), observe that we are substituting closed values; for all closed values w^* we have $\mathit{HC}(w^*) = 0$, and by i.h. the w^* we substitute are deeply k-bounded, so the result of substitution is deeply k-bounded. (The target expression $x[\vec{\alpha} = \vec{\delta}]$ is not closed, so we need not consider it here.)

Note that this reasoning holds for (TEvValue) even when w is a **select**: (TEvValue) is a clean rule so $HC(\mathcal{D}) = 0$.

- (TEvWrite): We have $\mathcal{D} :: \rho \vdash \mathbf{write}(e_0') \Downarrow (\rho' \vdash w)$ with subderivation $\mathcal{D}_0 :: \rho \vdash e_0' \Downarrow \ldots$. By i.h., $HC(\mathcal{D}_0) \leq HC(\mathbf{write}(e_0'))$. Therefore $HC(\mathcal{D}_0) + 1 \leq HC(e_0')$. By the definitions of HC we have $HC(\mathcal{D}) = HC(\mathcal{D}_0) + 1$ and $HC(\mathbf{write}(e_0')) = 1 + HC(e_0')$, so our inequality becomes $HC(\mathcal{D}) \leq HC(\mathbf{write}(e_0'))$, which was to be shown. Lastly, the $\cdots = W(\mathcal{D}')$ condition from the i.h. is exactly what we need, because the $\mathcal{D}'$ is the same and (TEvWrite) is not clean.
- (TEvMod): Similar to the (TEvWrite) case.
- (TEvLet): According to our definition, $HC(\mathcal{D}) = 0$, and we proceed as with the first 7 cases. Every (TEvLet) in $\mathcal{D}$ corresponds to a (SEvLet) in $\mathcal{D}'$, even if the **let** was created by translation: the theorem concerns the reverse translation $[\![[\rho]e]\!]$, *not* the original source program (which probably has fewer **let**s). (We already assume implicitly that let-expansion preserves asymptotic complexity, because we assume that source programs are in A-normal form, and our goal is to prove an asymptotic equivalence in Theorem 6.6.9.)
- (TEvRead): For $HC(\mathbf{read}\ldots)$ to be defined, the variable bound is used exactly once and contributes to the HC of the term accordingly, justifying the equation

$$HC([\rho_1(\ell)/x']e^{\mathbb{C}}) = HC(e^{\mathbb{C}}) + HC(\rho_1(\ell))$$

- (TEvSelect):

$$\begin{array}{rrll}
 & HC(\mathcal{D}_1) & \leq HC(e_1) & \text{i.h.} \\
\text{(L)} & 1 + HC(\mathcal{D}_1) & \leq 1 + HC(e_1) & +1 \text{ each side} \\
\text{(R)} & 1 + HC(e_1) & \leq HC((\mathbf{select}\,\{\ldots\})[\ldots]) & \text{By def. of } HC(e_1)\text{; property of max.} \\
 & 1 + HC(\mathcal{D}_1) & \leq HC(e') & \text{By (L), (R), transitivity, } e' = (\mathbf{select}\,\{\ldots\})[\ldots] \\
☞ & HC(\mathcal{D}) & \leq HC(e') & \text{By def. of } HC(\mathcal{D})
\end{array}$$

The $HC(e^*) \leq k$ part of the conclusion is easily shown: in each case, it must be shown for each premise and for the conclusion; the induction hypothesis shows it for the premises, and since we know that $[\rho]e'$ is deeply k-bounded, $HC(e') \leq k$ (applying $[\rho]$ cannot decrease head cost).

Showing that the value w is deeply k-bounded is quite easy. For (TEvValue) it follows from the assumption that $e' = w$ is bounded. For any rule whose conclusion has the same w as one of its premises—(TEvLet), (TEvCaseLeft), (TEvApply), (TEvWrite), (TEvRead), (TEvSelect)—it is immediate by the i.h. In (TEvPair), w_1 and w_2 are bounded by i.h., so (w_1, w_2) is too. The value returned by (TEvSumLeft) and (TEvFst) is a subterm of a value in a premise, which is by i.h. deeply k-bounded, so the subterm is too. (TEvMod) returns ℓ where $\ell \mapsto w$, and w is deeply k-bounded. □

Theorem 6.6.9. *If* `trans` $(e, \varepsilon) = e'$ *and* $\mathcal{D}' :: \cdot \vdash e' \Downarrow (\rho' \vdash w)$, *then* $\mathcal{D} :: [\![e']\!] \Downarrow v$ *where* $W(\mathcal{D}') = \Theta(W(\mathcal{D}))$.

Proof

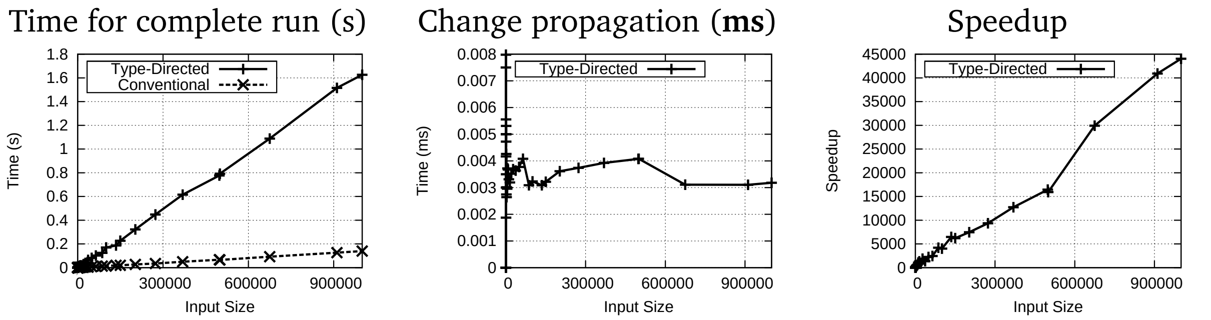
By Theorem 6.6.6, e' is deeply 1-bounded.

The algorithm `trans` merely applies the translation rules, so $\cdot \vdash e : \tau \underset{\varepsilon}{\hookrightarrow} e'$. By Theorem 6.6.8, $\mathcal{D} :: [\![e']\!] \Downarrow v$, and the given derivation $\mathcal{D}'$ and all its subderivations have *HC* bounded by k.

By Theorem 6.6.7, the number of dirty rule applications in $\mathcal{D}'$ is at most $\frac{k}{k+1}W(\mathcal{D}')$. Each rule application is either clean or dirty, so $W(\mathcal{D}') \leq (k+1) \cdot W(\mathcal{D})$, where $k = 1$. By inspecting the evaluation rules, it is clear that $W(\mathcal{D}') \geq W(\mathcal{D})$. Therefore, $W(\mathcal{D}') = \Theta(W(\mathcal{D}))$. □

Appendix B

Benchmark Plots

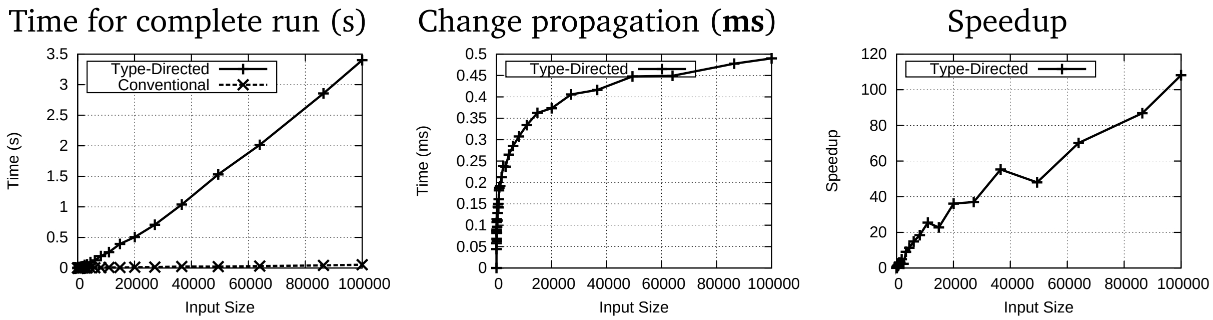


Time for complete run (s)

Change propagation (**ms**)

Speedup

Figure B.1: Time for complete run; time and speedup for change propagation for `split`



Time for complete run (s)

Change propagation (**ms**)

Speedup

Figure B.2: Time for complete run; time and speedup for change propagation for `qsort`

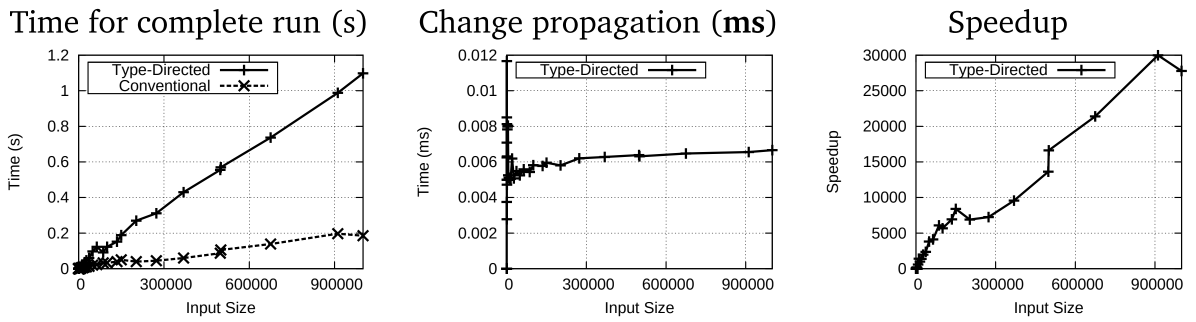


Figure B.3: Time for complete run; time and speedup for change propagation for `vec-mult`

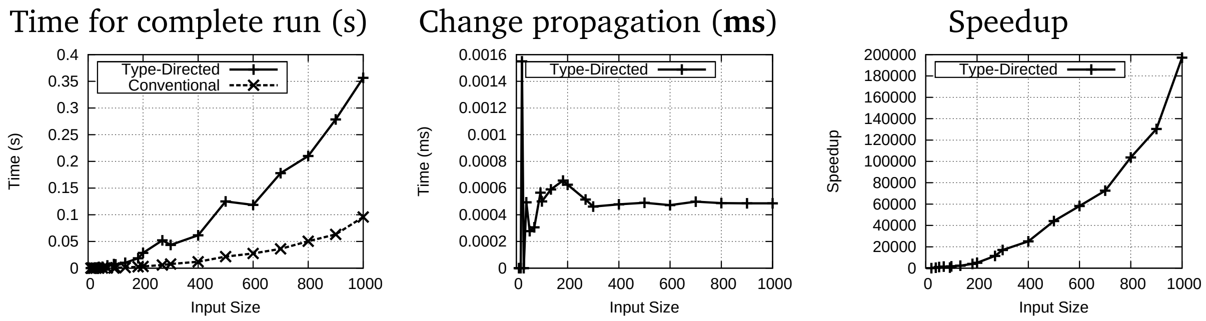


Figure B.4: Time for complete run; time and speedup for change propagation for `mat-add`

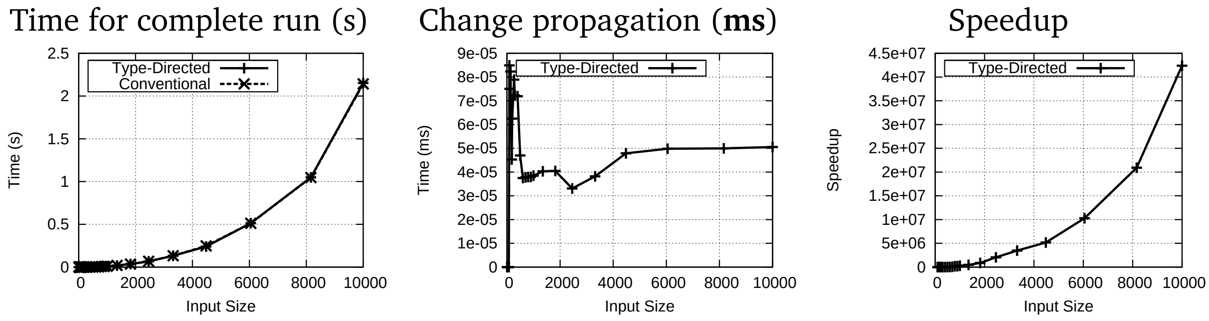


Figure B.5: Time for complete run; time and speedup for change propagation for `transpose`

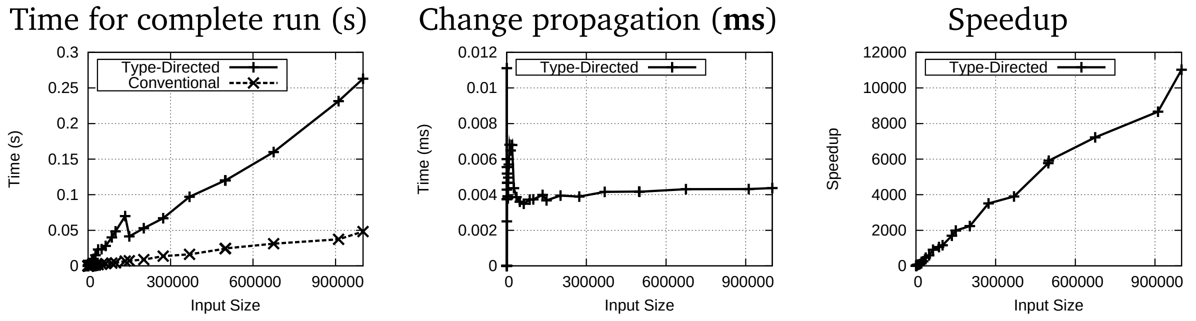


Figure B.6: Time for complete run; time and speedup for change propagation for `vec-reduce`

Bibliography

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

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

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

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

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

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

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212. ACM, 1982.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 11–20. 1982.

Thomas Reps. *Generating Language-Based Environments*. Ph.D. thesis, Department of Computer Science, Cornell University, 1982a.

Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 169–176. 1982b.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

Roger Hoover. *Incremental Graph Evaluation*. Ph.D. thesis, Department of Computer Science, Cornell University, 1987.

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

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

Thomas Reps and Tim Teitelbaum. *The synthesizer generator: a system for constructing language-based editors*, volume 2. Springer-Verlag, 1989.

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

David Sands. *Calculi for Time Analysis of Functional Programs*. Ph.D. thesis, University of London, Imperial College, 1990.

R. S. Sundaresh and Paul Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13. 1991.

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

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

Sofoklis G. Efremidis, John H. Reppy, and Khalid A. Mughal. Attribute grammars in ML. Technical report, 1993.

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

Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Principles of Programming Languages*, pages 355–366. 1995.

Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91. 1996.

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

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

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

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

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160. ACM, New York, NY, USA, 1999.

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

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

Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000.

Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Programming Language Design and Implementation,* pages 311–320. 2000.

Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP*, pages 174–187. 2001.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *SIGPLAN Not.*, 36(10):146–156, 2001.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259. 2002.

Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolfson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

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

Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172. 2002.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*. 2003.

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

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

Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *APLAS*, pages 283–302. 2003.

Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540. 2004.

Camil Demetrescu, Stefano Emiliozzi, and Giuseppe F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *ACM-SIAM Symposium on Discrete*

Algorithms (SODA), pages 369–378. 2004.

L. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.

Umut A. Acar. *Self-Adjusting Computation*. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, 2005.

Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*. 2005.

Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006a.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 2006b.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006c.

Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic algorithms via self-adjusting computation. In *Proceedings of the 14th Annual European Symposium on Algorithms*, pages 636–647. 2006d.

Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*. 2006.

Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Prog. Lang. Sys.*, 28:1035–1087, 2006.

Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan. Kinetic 3D convex hulls via self-adjusting computation (an illustration). In *Proceedings of the 23rd ACM Symposium on Computational Geometry (SCG)*. 2007a.

Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *European Symposium on Programming*. 2007b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*. 2007c.

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

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst.*

Rev., 41(3):59–72, 2007.

Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, 2007.

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

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

Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*. 2008b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive inference on general graphical models. In *Uncertainty in Artificial Intelligence (UAI)*. 2008c.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

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

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*. 2008.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–53, 2009a.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Maintaining MAP configurations with applications to protein sidechain packing. In *IEEE/SP 15th Workshop on Statistical Signal Processing (SSP)*. 2009b.

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

Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*. 2009.

Hai Liu, Eric Cheng, and Paul 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.

Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Türkoğlu.

Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation*. 2010a.

Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*. 2010b.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proc. 7th Symposium on Networked systems design and implementation (NSDI'10)*. 2010.

Jacob Donham. Froc: a library for functional reactive programming in ocaml. 2010.

Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in data centers. In *OSDI'10*. 2010.

Ruy Ley-Wild. *Programmable Self-Adjusting Computation*. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 2010.

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz 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.

Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*. 2010.

Kanat Tangwongsan, Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Efficient similarity estimation for systems exploiting data redundancy. In *INFOCOM*, pages 1487–1495. 2010.

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

Pramod Bhatotia, Alexander Wieder, Istemi Ekin Akkus, Rodrigo Rodrigues, and Umut A. Acar. Large-scale incremental data processing with change propagation. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2011a.

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing*. 2011b.

Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas 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.

Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, pages 129–141. 2011.

Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2011.

Matthew Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2011.

U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011a.

U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011b.

Neelakantan R. Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *LICS '11: Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 257–266. 2011.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174. ACM, New York, NY, USA, 2011.

Özgür Sümer, Umut A. Acar, Alexander Ihler, and Ramgopal Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 8:180–186, 2011.

Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *International Conference on Functional Programming (ICFP)*. 2011.

Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2012.

Matthew A. Hammer. *Self-Adjusting Machines*. Ph.D. thesis, Department of Computer Science, University of Chicago, 2012.

Alan 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.

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 45–58. 2012.

Ruy Ley-Wild, Umut A. Acar, and Guy Blelloch. Non-monotonic self-adjusting computation. In *Proceedings of the 21st European Conference on Programming Languages and Systems,* ESOP'12, pages 476–496. 2012.

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB Endow.*, 5(8):716–727, 2012.

E. Jason Riedy, Henning Meyerhenke, David A. Bader, David Ediger, and Timothy G. Mattson. Analysis of streaming social networks and graphs on multicore architectures. In *ICASSP*, pages 5337–5340. 2012.

Özgür Sümer. *Adaptive Exact Inference in Graphical Models*. Ph.D. thesis, Department of Computer Science, University of Chicago, 2012.

Duru Türkoğlu. *Stable Algorithms and Kinetic Mesh Refinement*. Ph.D. thesis, Computer Science Department, University of Chicago, 2012.

Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. *The Journal of Functional Programming*, 23(3):249–292, 2013a.

Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Dynamic well-spaced point sets. *Journal of Computational Geometry: Theory and Applications*, 2013b.

Evan Czaplicki and Stephen 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.

Alan Jeffrey. Functional reactive programming with liveness guarantees. In *Proceedings of the 18th ACM International Conference on Functional Programming*. 2013. Forthcoming.

Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR 2013*. 2013.

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proc. of SOSP*, pages 439–455. 2013.

Pramod Bhatotia, Umut A. Acar, Flavio P. Junqueira, and Rodrigo Rodrigues. Slider: Incremental sliding window analytics. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 61–72. 2014.

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 145–155. 2014.

Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *International Conference on Functional Programming (ICFP '14)*, pages 227–240. 2014a.

Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. *Journal of Functional Programming*, 24:56–112, 2014b.

Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey 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.

Martín Abadi, Frank McSherry, and Gordon D. Plotkin. Foundations of differential dataflow. In *Proceedings of Foundations of Software Science and Computation Structures*, FoSSaCS '15, pages 71–83. 2015.

Pramod Bhatotia. *Incremental Parallel and Distributed Systems*. Ph.D. thesis, Max Planck Institute for Software Systems, 2015.

Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. ithreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 645–659. 2015.

Ezgi Çiçek, Deepak Garg, and Umut Acar. Refinement types for incremental computational complexity. In *European Symposium on Programming*, volume 9032, pages 406–431. Springer, 2015.

Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. *CoRR*, abs/1503.07792, 2015.

JaneStreet. Jane street incremental. 2015. `https://github.com/janestreet/incremental`.

HaMLet. HaMLet web site. `http://www.mpi-sws.org/~rossberg/hamlet/`.

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

YAN CHEN

Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043

Research Interests

- Self-adjusting computation, Program analysis, Dynamic and parallel computation, Big data systems
- Formal methods, Model checking, Symbolic simulation

Education

- Ph.D. student, Max Planck Institute for Software Systems, Germany *2008 – Present*
  Working on programming languages, and self-adjusting computation
  Adviser: Umut Acar
- M.S., Computer Science, Portland State University, Portland, OR *August 2008*
  Thesis: Equivalence Checking for High-Level Synthesis Flow
- B.E., Computer Science and Technology, Fuzhou University, China *June 2006*

Professional Experience

- Software engineer, Google Inc. *Oct 2015 – Present*
- Intern, Microsoft Research Silicon Valley. *May 2012 – Aug 2012*

Publications

1. **Yan Chen**, Umut A. Acar, and Kanat Tangwongsan. Functional Programming for Dynamic and Large Data with Self-Adjusting Computation. *Proc. of ACM-SIGPLAN International Conference on Functional Programming (**ICFP**)*, 227–240. Sep 2014. *Acceptance Ratio:* 29% (28/97)
2. **Yan Chen**, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit Self-Adjusting Computation for Purely Functional Program. *Journal of Functional Programming (**JFP**)*, 24(1), 56–112. Jan 2014.
3. Umut A. Acar and **Yan Chen**. Streaming Big Data with Self-Adjusting Computation. *Data Driven Functional Programming Workshop (**DDFP**)*, 15–18. Jan 2013.
4. **Yan Chen**, Joshua Dunfield, and Umut A. Acar. Type-Directed Automatic Incrementalization. *ACM SIGPLAN Conference on Programming Language Design and Implementation (**PLDI**)*, 299–310. Jun 2012. *Acceptance Ratio:* 19% (48/255)
5. Matthew A. Hammer, Georg Neis, **Yan Chen**, and Umut A. Acar. Self-Adjusting Stack Machines. *Proc. of ACM-SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (**OOPSLA**)*, 753–772. Oct 2011. *Acceptance Ratio:* 37% (61/166)
6. **Yan Chen**, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit Self-Adjusting Computation for Purely Functional Program. *Proc. of ACM-SIGPLAN International Conference on Functional Programming (**ICFP**)*, 129–141. Sep 2011. *Acceptance Ratio:* 36% (33/92)

7. Matthew A. Hammer, Umut A. Acar, and **Yan Chen**. CEAL: A C-Based Language for Self-Adjusting Computation. *Proc. of ACM-SIGPLAN Conference on Programming Language Design and Implementation (**PLDI**)*, 25–37, Jun 2009. *Acceptance Ratio:* 20% (41/196)
8. Sandip Ray, Kecheng Hao, **Yan Chen**, Fei Xie, and Jin Yang. Formal Verification for High-Assurance Behavioral Synthesis. *Proc. of 7th International Symposium on Automated Technology for Verification and Analysis (**ATVA**)*, 337–351. Oct 2009. *Acceptance Ratio:* 30% (26/84)
9. **Yan Chen**, Fei Xie, and Jin Yang. Optimizing Automatic Abstraction Refinement for GSTE. *Proc. of 45th Design Automation Conference (**DAC**)*, 143–148, Jun 2008. *Acceptance Ratio:* 23% (147/639)
10. **Yan Chen**, Yujing He, Fei Xie, and Jin Yang. Automatic Abstraction Refinement for Generalized Symbolic Trajectory Evaluation. *Proc. of 7th International Conference on Formal Methods in Computer-Aided Design (**FMCAD**)* , 111–118, Nov 2007. *Acceptance Ratio:* 28% (23/80)
11. **Yan Chen**. An Efficient Search Algorithm for Partially Ordered Sets. *Proc. of The IASTED International Conference on Advances in Computer Science and Technology (**ACST**)*, 91–94, Jan 2006.

Technical Reports

1. Matthew A. Hammer, Umut A. Acar, and **Yan Chen**. CEAL: A C-Based Language for Self-Adjusting Computation. Technical Report TTIC-TR-2009-2. Toyota Technological Institute at Chicago. May 2009.
2. Sandip Ray, **Yan Chen**, Fei Xie, and Jin Yang. Combining Theorem Proving and Model Checking for Certification of Behavioral Synthesis Flows. Technical Report TR-08-48, Department of Computer Science, University of Texas at Austin. Dec 2008.

Awards

- Google GRAD CS Scholarship, 01/2010.
- **Silver Medal** (12th Place), 30th ACM International Collegiate Programming Contest (ICPC). Beijing Site, 11/2005.
- **Third Prize**, National Undergraduate Electronic Design Contest Specialized in Embedded System. 09/2004.
- **Second Prize** (for three continuous years), National Olympiad in Informatics Provincial Contest. 1999 – 2001.

Professional Services

- Program Committee for ACM SIGPLAN Workshop on ML. 2016.
- Reviewer for The Mathematical Reviews, American Mathematical Society.
- Reviewer for The Computer Journal, Oxford University Press.
- Reviewer for the Science of Computer Programming, Elsevier.
- Reviewer for ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 2015,2012.
- Reviewer for ACM SIGPLAN International Conference on Functional Programming. 2013,2010.
- Reviewer for ACM SIGPLAN Workshop on ML. 2009.
- Reviewer for ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE). 2009, 2008.
- Assistant Coach. ACM/ICPC World Final Team of Fuzhou University. 2006.

Presentations

- "Implicit Self-Adjusting Computation: From Types to Incremental Systems". *IBM Thomas J. Watson Research Center*. Lab Seminar. Yorktown Heights, NY. Nov 6, 2014.
- "Functional Programming for Dynamic and Large Data with Self-Adjusting Computation". *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Gothenburg, Sweden. Sep 2, 2014.
- "Implicit Self-Adjusting Computation: From Types to Incremental Systems". *Microsoft Research Cambridge*. Lab Seminar. Cambridge, UK. Mar 20, 2014.
- "Streaming Big Data with Self-Adjusting Computation". *Data Driven Functional Programming Workshop*. Rome, Italy. Jan 22, 2013.
- "Refactoring DryadLINQ using the Compiler Forest". *Microsoft Research, Silicon Valley*. Lab Seminar. Mountain View, CA. Aug 16, 2012.
- "Type-Directed Automatic Incrementalization". *Software Engineering Institute, Peking University*. Department Seminar. Beijing, China. Jun 15, 2012.
- "Type-Directed Automatic Incrementalization". *Baidu*. Lab Seminar. Beijing, China. Jun 14, 2012.
- "Type-Directed Automatic Incrementalization". *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China. Jun 12, 2012.
- "Implicit Self-Adjusting Computation for Purely Functional Programs". *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Tokyo, Japan. Sep 19, 2011.
- "Implementing Implicit Self-Adjusting Computation". *ACM SIGPLAN Workshop on ML*. Tokyo, Japan. Sep 18, 2011.
- "Optimizing Automatic Abstraction Refinement for GSTE". *Design Automation Conference (DAC)*. Anaheim, California. Jun 10, 2008.
- "Optimizing Automatic Abstraction Refinement for GSTE". *Progress report to Strategic CAD Lab at Intel*, Portland, Oregon. Apr 7, 2008.
- "Proving Correctness of Computer Systems". *Toyota Technological Institute at Chicago*, Chicago, Illinois. Apr 4, 2008.
- "Automatic Abstraction Refinement for GSTE". *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Austin, Taxes. Nov 13, 2007.
- "Automatic Abstraction Refinement for GSTE". *Progress report to Strategic CAD Lab at Intel*. Portland, Oregon. Jun 15, 2007.

Programming Skills

★ Highly-Proficient Languages: C/C++, Standard ML

★ Proficient Languages: Python, C#, Prolog, Pascal, SQL, ASP/PHP, LaTeX