Implicit Self-Adjusting Computation for Purely Functional Programs

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

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. Building on this work, recent advances in self-adjusting computation have developed techniques that enable programs to respond automatically and efficiently to dynamic changes in their inputs. Self-adjusting programs have been shown to be efficient for a reasonably broad range of problems but the approach still requires an explicit programming style, where the programmer must use specific monadic types and primitives to identify, create and operate on data that can change over time.

We describe techniques for automatically translating purely functional programs into self-adjusting programs. In this implicit approach, the programmer need only annotate 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.

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

General Terms Algorithms, Languages, Performance

1. Introduction

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 (e.g., 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 [Ley-Wild et al. 2008; Acar et al. 2009]. Self-adjusting computation can deliver asymptotically efficient updates in a reasonably broad range of problem domains [Acar et al. 2007, 2010a], and have even helped solve challenging open problems [Acar et al. 2010b]. Existing self-adjusting computation techniques, however, require the programmer to program *explicitly* by using a certain set of primitives [Carlsson 2002; Ley-Wild et al. 2008; Acar et al. 2009]. 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. As a result, rewriting a conventional program into a self-adjusting program requires extensive changes to the code. For example, a purely functional program will need to be rewritten in imperative style using write-once, monadic references.

In this paper, we present techniques for *implicit* self-adjusting computation that allow conventional programs to be translated automatically into efficient self-adjusting programs. Our approach consists of a type system for inferring self-adjusting computation types from purely functional programs and a type-guided translation algorithm that rewrites purely functional programs into self-adjusting programs.

The type system hinges on a key observation connecting self-adjusting computation to information flow [Pottier and Simonet 2003; Sabelfeld and Myers 2003]: both involve tracking data dependencies (of changeable data and sensitive data, respectively) as well as dependencies between expressions and data. Specifi-

ICFP'11, September 19–21, 2011, Tokyo, Japan.

cally, 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. Our type system admits a constraint-based formulation where the constraints are a strict subset of those needed by traditional information-flow type systems. Consequently, as with 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.

For this work, determination of types via type inference is not an end unto itself but a means for translating purely functional programs into self-adjusting programs. To achieve this, we first present a set of compositional, non-deterministic transformation rules. Guided by the types, the 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. Figure 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. 2006]. 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 6.1).
- **Observational equivalence.** The translated self-adjusting program, when evaluated, produces the same value as the source program (Theorem 6.5).
- **Asymptotic complexity.** The time to evaluate the translated program is asymptotically the same as the time to evaluate the source program (Theorem 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. 2006]). The third property shows that the translated program takes asymptotically as long to evaluate (from scratch) as the corresponding source program. In addition, it places a worst-case bound on the time taken to self-adjust via change propagation, which can and often does take significantly less time when data changes are small. To prove this complexity result, we use a cost semantics [Sands 1990; Sansom and Peyton Jones 1995] that enables precise reasoning about the complexity of the evaluation time. We do not, however, prove tighter bounds on the complexity of self-adjustments; this would be beyond the scope of this paper.

We intend to complete an implementation of our approach as an extension of Standard ML and the MLton compiler [MLton]. However, we expect the proposed approach could be implemented in other languages such as Haskell, where self-adjusting libraries also exist [Carlsson 2002]. In general, since our approach simply generates target code, it is agnostic to implementation details of the explicit self-adjusting-computation mechanisms employed in the target language and thus can be applied broadly.

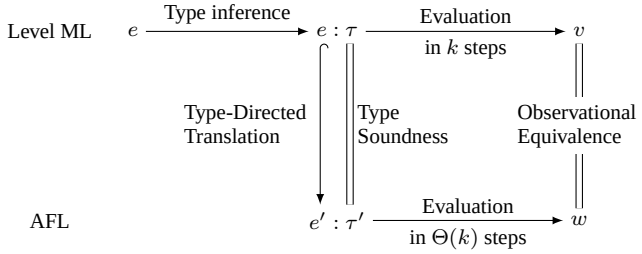

Figure 1. Visualizing the translation between the source language Level ML and the target language AFL, and related properties.

Paper guide. We find it better to give an overview of the proposed approach by focusing on the translation problem and working back to the type system in a "top-down" manner (Section 2). The details of the translation algorithm and our theorems, however, rely on the type system. We therefore take a more "bottom-up" approach in the rest of the paper: we first present the static semantics (the syntax and the type system) (Sections 3 and 4), and then describe the target language AFL (Section 5) and the translation (Section 6). Finally, we discuss related work (Section 7) and conclude. Due to space restrictions, we include all the proofs in the appendix [Chen et al. 2011].

2. Overview

We present an informal overview of our approach via examples. First we briefly describe explicit self-adjusting computation, as laid out in previous work, and which we use as a target language. Then we outline our proposed approach.

2.1 Explicit Self-Adjusting Computation

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. 2006]. 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 graph, enabling efficient *change propagation* when the data changes in small amounts.

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 $\xrightarrow[\mathbb{C}]{}$:

```
squareplus_SC: int * int mod →C 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 delineates the code that depends on the changeable value `y`, and the changeable arrow type ensures a critical consistency property: $\xrightarrow[\mathbb{C}]{}$-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, reusing the computation of the square x2.

Suppose we wish to make x changeable while leaving y stable. We need to 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. (To avoid creating this modifiable would require further structural changes to the code.)

```
squareplus_CS: int mod * int ->C 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.

Conservatively treating all data as changeable would require writing just one version, but treating all data as modifiable can introduce unacceptably high overhead. At the other extreme, making everything stable requires no rewriting, but forgoes the benefits of change propagation. Instead, we take an approach where data is modifiable only where necessary.

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 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).

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 [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].

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 type system of the target language, which is explicitly self-adjusting, is monomorphic in the levels or changeability, while the implicitly self-adjusting source language is polymorphic over levels. Consequently, translation also needs to monomorphize the source code. 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.

```
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 (l,a) = (map inc l, map inc a)
```

Figure 2. Function mapPair in ML

```
datatype α list^δ = nil | cons of α * (α list^δ)

mapPair : ((int^S list^C) * (int^C list^S))
       ->S ((int^S list^C) * (int^C list^S))

... (* inc, map, mapPair same as in Figure 1. *)
```

Figure 3. Function mapPair in Level ML, with level types

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 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 changeability:

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

Then a list of integers allowing insertion and deletion has type $\mathbf{int}^{\mathbb{S}}$ list$^{\mathbb{C}}$, and one with unchanging length has type $\mathbf{int}^{\mathbb{C}}$ list$^{\mathbb{S}}$. Now we can write the type annotation on mapPair shown in Figure 3. 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 3 produces the self-adjusting target code in Figure 4. Note that inc and map have *level-polymorphic* types. In map inc l we increment sta-

```
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 *)
fun𝕊 inc_S (x) = x+1

inc_C : int →ℂ int     (* 'inc' specialized for changeable data *)
fun𝕔 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
fun𝕊 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 list_S)
fun𝕊 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 : ∀δh,δt. (α →δh β) →𝕊 α list^δt →𝕊 β list^δt
val map = select {δh=𝕊, δt=ℂ ⇒ map_SC
                | δh=ℂ, δt=𝕊 ⇒ map_CS}

mapPair :  ((int list_C) mod * (int mod) list_S)
        →𝕊 ((int list_C) mod * (int mod) list_S)
fun𝕊 mapPair (l, a) = (map[δh=𝕊,δt=ℂ] inc[δ=𝕊] l,
                       map[δh=ℂ,δt=𝕊] inc[δ=ℂ] a)
```

Figure 4. Translated `mapPair` with **mod** types and explicit level polymorphism.

ble integers, and in `map inc a` 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 (δ=$\mathbb{S}$ and δ=$\mathbb{C}$): `inc_S` and `inc_C` (in Figure 4). Sine 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:

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

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.)

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.

Levels	δ, ε	$::= \mathbb{S} \mid \mathbb{C} \mid \alpha$
Types	τ	$::= \mathbf{int}^\delta \mid (\tau_1 \times \tau_2)^\delta \mid (\tau_1 + \tau_2)^\delta \mid (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^\delta$
Constraints	C, D	$::= \mathbf{true} \mid \mathbf{false} \mid \exists\vec{\alpha}.C \mid C \wedge D \mid$ $\alpha = \beta \mid \alpha \leq \beta \mid \delta \lhd \tau$
Type schemes	σ	$::= \tau \mid \forall\vec{\alpha}[D].\,\tau$

Figure 5. Levels, constraints, types, and type schemes

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 $\mathbb{C}$.

3. 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 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. Previous work therefore presents languages that enable the programmer to separate stable and changeable data, and type systems that enforce the 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]. Our types 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:

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

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 $(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^\delta$ carry two level annotations ε and δ.

[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.

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

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

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

Figure 6. Subtyping

$$\frac{\delta \le \delta'}{\delta \lhd \mathbf{int}^{\delta'}}\ (\lhd\text{-Int}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 \times \tau_2)^{\delta'}}\ (\lhd\text{-Prod})$$

$$\frac{\delta \le \delta'}{\delta \lhd (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\delta'}}\ (\lhd\text{-Arrow}) \qquad \frac{\delta \le \delta'}{\delta \lhd (\tau_1 + \tau_2)^{\delta'}}\ (\lhd\text{-Sum})$$

Figure 7. Lower bound of a type

$$\overline{\mathbf{int}^{\mathbb{S}}\ \mathsf{O.S.}} \quad \overline{(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}\ \mathsf{O.S.}} \quad \overline{(\tau_1 \times \tau_2)^{\mathbb{S}}\ \mathsf{O.S.}} \quad \overline{(\tau_1 + \tau_2)^{\mathbb{S}}\ \mathsf{O.S.}}$$

$$\overline{\mathbf{int}^{\mathbb{C}}\ \mathsf{O.C.}} \quad \overline{(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{C}}\ \mathsf{O.C.}} \quad \overline{(\tau_1 \times \tau_2)^{\mathbb{C}}\ \mathsf{O.C.}} \quad \overline{(\tau_1 + \tau_2)^{\mathbb{C}}\ \mathsf{O.C.}}$$

$$\mathbf{int}^{\delta_1} \stackrel{\circ}{=} \mathbf{int}^{\delta_2} \qquad (\tau_1 + \tau_2)^{\delta_1} \stackrel{\circ}{=} (\tau_1 + \tau_2)^{\delta_2}$$
$$(\tau_1 \times \tau_2)^{\delta_1} \stackrel{\circ}{=} (\tau_1 \times \tau_2)^{\delta_2} \qquad (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\delta_1} \stackrel{\circ}{=} (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\delta_2}$$

Figure 8. Outer-stable and outer-changeable types, and equality up to outer levels

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.

Subtyping. Figure 6 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.)

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. In other words, δ is a lower bound of the outer level(s) of τ. Figure 7 defines this relation. We distinguish between outer-stable and outer-changeable types (Figure 8). We write τ $\mathsf{O.S.}$ if the outer level of τ is $\mathbb{S}$. Similarly, we write τ $\mathsf{O.C.}$ 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 $\le$ 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 6 and 7 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 of α in ϕ, and $[\phi]\tau$ for the usual substitution operation on types. For example, if $\phi(\alpha) = \mathbb{S}$ then $[\phi]\left(\left(\mathbf{int}^{\alpha} + \mathbf{int}^{\mathbb{C}}\right)^{\alpha}\right) = \left(\mathbf{int}^{\mathbb{S}} + \mathbf{int}^{\mathbb{C}}\right)^{\mathbb{S}}$.

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 $\forall\emptyset[\mathbf{true}].\,\mathbf{int}^{\mathbb{C}}$.

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

Figure 9. Abstract syntax of the source language Level ML

4. Source Language

4.1 Static Semantics

Syntax. Figure 9 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.

Constraint-based type system. We could define types as

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

Such a type system would be completely standard. Instead, we use a richer type system that allows us to directly translate Level ML programs into self-adjusting programs in AFL. This constraint-based type system has the level-decorated types, constraints, and type schemes in Figure 5 and described in Section 3. After discussing the rules themselves, we will look at type inference (Section 4.2).

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

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

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

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

$$\frac{C; \Gamma \vdash_\varepsilon v : \tau_1}{C; \Gamma \vdash_\varepsilon \mathbf{inl}\ v : (\tau_1 + \tau_2)^{\mathbb{S}}} \text{(SSum)}$$

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

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

$$\frac{C; \Gamma \vdash_{\mathbb{S}} x : (\tau_1 \times \tau_2)^{\delta} \qquad C \Vdash \delta \leq \varepsilon}{C; \Gamma \vdash_\varepsilon \mathbf{fst}\ x : \tau_1} \text{(SFst)}$$

$$\frac{\begin{array}{ll} C; \Gamma \vdash_{\varepsilon'} e_1 : \tau' & C; \Gamma, x : \tau'' \vdash_\varepsilon e_2 : \tau \\ C \Vdash \tau' <: \tau'' & C \Vdash \tau' \doteq \tau'' \end{array}}{C; \Gamma \vdash_\varepsilon \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \text{(SLetE)}$$

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

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

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

Figure 10. Typing rules for Level ML

to changeable values in the stable mode. This is a key principle of the type system.

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 (SSum) are otherwise standard (we omit the symmetric judgment **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, $\varepsilon \lhd \tau_2$. As a result, 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. Primitive operators $\oplus$ take two stable integers and return a stable integer result.

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 facilitates adding side effects at a later date. In the first case (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'.[2] 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 $(\mathbf{int}^{\mathbb{S}} \xrightarrow[\mathbb{S}]{} \mathbf{int}^{\mathbb{S}})^{\mathbb{S}} <: (\mathbf{int}^{\mathbb{S}} \xrightarrow[\mathbb{S}]{} \mathbf{int}^{\mathbb{C}})^{\mathbb{C}}$, we can insert *coercions* into the source program before typing it with these rules. (This process could easily be automated.)

In the second case (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.

Function application, $\oplus$, **fst**, and **case** are the forms that eliminate values of changeable type. An application is typed in the mode ε' of the function being applied because changeable functions can operate on changeable values; the typing mode must match ($\varepsilon' = \varepsilon$). Furthermore, the result of the function must be higher than the function's level: if a function is itself changeable, $(\tau_1 \xrightarrow[\varepsilon']{} \tau_2)^{\mathbb{C}}$, then it could be replaced by another function and thus the result of this application must be changeable. (Due to **let**-subsumption, checking this in (SFun) alone is not enough.)

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 Constraints and Type Inference

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].

[2] 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.

$$
\begin{array}{lrcl}
\textit{Levels} & \delta, \varepsilon & ::= & \mathbb{S} \mid \mathbb{C} \\
\textit{Types} & \underline{\tau} & ::= & \textbf{int} \mid \underline{\tau}\ \textbf{mod} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \underline{\tau}_1 + \underline{\tau}_2 \mid \underline{\tau}_1 \xrightarrow{\varepsilon} \underline{\tau}_2 \\
\textit{Type schemes} & \underline{\sigma} & ::= & \Pi\vec{\alpha}[D].\,\tau \\
\textit{Typing environments} & \Gamma & ::= & \cdot \mid \Gamma, x : \underline{\sigma} \mid \Gamma, x : \underline{\tau} \\
\textit{Variables} & \underline{x} & ::= & x \mid x[\vec{\alpha} = \vec{\delta}] \\
\textit{Values} & v & ::= & n \mid \underline{x} \mid \ell \mid (v_1, v_2) \mid \textbf{inl}\ v \mid \textbf{inr}\ v \mid \\
 & & & \textbf{fun}^{\mathbb{S}}\ f(x) = e^{\mathbb{S}} \mid \textbf{fun}^{\mathbb{C}}\ f(x) = e^{\mathbb{C}} \mid \\
 & & & \textbf{select}\ \{(\vec{\alpha_i} = \vec{\delta_i}) \Rightarrow e_i\}_i \\
\textit{Expressions} & e & ::= & e^{\mathbb{S}} \mid e^{\mathbb{C}} \\
\textit{Stable expressions} & e^{\mathbb{S}} & ::= & v \mid \oplus(\underline{x_1}, \underline{x_2}) \mid \textbf{fst}\ \underline{x} \mid \textbf{snd}\ \underline{x} \mid \\
 & & & \textbf{apply}^{\mathbb{S}}(\underline{x_1}, \underline{x_2}) \mid \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e^{\mathbb{S}} \mid \\
 & & & \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e^{\mathbb{S}}\ ,\ x_2 \Rightarrow e^{\mathbb{S}}\} \mid \\
 & & & \textbf{mod}\ e^{\mathbb{C}} \\
\textit{Changeable expressions} & e^{\mathbb{C}} & ::= & \textbf{apply}^{\mathbb{C}}(\underline{x_1}, \underline{x_2}) \mid \textbf{let}\ x = e^{\mathbb{S}}\ \textbf{in}\ e^{\mathbb{C}} \mid \\
 & & & \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e^{\mathbb{C}}\ ,\ x_2 \Rightarrow e^{\mathbb{C}}\} \mid \\
 & & & \textbf{read}\ \underline{x}\ \textbf{as}\ y\ \textbf{in}\ e^{\mathbb{C}} \mid \textbf{write}(\underline{x})
\end{array}
$$

Figure 11. Types and expressions in the target language AFL

4.3 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 13 are standard; we write $[v/x]e$ for capture-avoiding substitution of v for the variable x in e.

5. Target Language

The target language AFL (Figure 11) 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 values 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 **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; **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 computations 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 12 follow the structure of the expressions. Rule (TSelect) checks that each monomorphized expression e_i in a **select** has type $\|[\vec{\delta}/\vec{\alpha}]\tau\|$, where $[\vec{\delta}/\vec{\alpha}]\tau$ is a source-level

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

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

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

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

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

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

$$\frac{\Lambda; \Gamma, x : \underline{\tau}_1, f : (\underline{\tau}_1 \xrightarrow{\varepsilon} \underline{\tau}_2) \vdash_\varepsilon e : \underline{\tau}_2}{\Lambda; \Gamma \vdash_{\mathbb{S}} \textbf{fun}^\varepsilon\ f(x) = e : (\underline{\tau}_1 \xrightarrow{\varepsilon} \underline{\tau}_2)}\ \text{(TFun)}$$

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

$$\frac{\begin{array}{l}\Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x_1} : \textbf{int} \\ \Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x_2} : \textbf{int} \qquad \vdash \oplus : \textbf{int} \times \textbf{int} \to \textbf{int}\end{array}}{\Lambda; \Gamma \vdash_{\mathbb{S}} \oplus(\underline{x_1}, \underline{x_2}) : \textbf{int}}\ \text{(TPrim)}$$

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

$$\frac{\Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x_1} : (\underline{\tau}_1 \xrightarrow{\varepsilon} \underline{\tau}_2) \qquad \Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x_2} : \underline{\tau}_1}{\Lambda; \Gamma \vdash_\varepsilon \textbf{apply}^\varepsilon(\underline{x_1}, \underline{x_2}) : \underline{\tau}_2}\ \text{(TApp)}$$

$$\frac{\Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x} : \underline{\tau}_1 + \underline{\tau}_2 \qquad \begin{array}{l}\Lambda; \Gamma, x_1 : \underline{\tau}_1 \vdash_\varepsilon e_1 : \underline{\tau} \\ \Lambda; \Gamma, x_2 : \underline{\tau}_2 \vdash_\varepsilon e_2 : \underline{\tau}\end{array}}{\Lambda; \Gamma \vdash_\varepsilon \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} : \underline{\tau}}\ \text{(TCase)}$$

$$\frac{\Lambda; \Gamma \vdash_{\mathbb{C}} e : \underline{\tau}}{\Lambda; \Gamma \vdash_{\mathbb{S}} \textbf{mod}\ e : \underline{\tau}\ \textbf{mod}}\ \text{(TMod)} \qquad \frac{\Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x} : \underline{\tau}}{\Lambda; \Gamma \vdash_{\mathbb{C}} \textbf{write}(\underline{x}) : \underline{\tau}}\ \text{(TWrite)}$$

$$\frac{\Lambda; \Gamma \vdash_{\mathbb{S}} \underline{x_1} : \underline{\tau}_1\ \textbf{mod} \qquad \Lambda; \Gamma, x : \underline{\tau}_1 \vdash_{\mathbb{C}} e_2 : \underline{\tau}_2}{\Lambda; \Gamma \vdash_{\mathbb{C}} \textbf{read}\ \underline{x_1}\ \textbf{as}\ x\ \textbf{in}\ e_2 : \underline{\tau}_2}\ \text{(TRead)}$$

Figure 12. Typing rules of the target language AFL

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 Dynamic Semantics

For the source language, our big-step evaluation rules (Figure 13) are standard. In the target language AFL, our rules (Figure 14) 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. Both sets of rules permit expressions that are not in A-normal form, enabling

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

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

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

$$\frac{e \Downarrow (v_1, v_2)}{\textbf{fst}\ e \Downarrow v_1}\ \text{(SEvFst)} \qquad \frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v_2}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v_2}\ \text{(SEvLet)}$$

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

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

Figure 13. Dynamic semantics of source Level ML programs

standard capture-avoiding substitution. To simplify the translation, we call instantiations $x[\vec{\alpha} = \vec{\delta}]$ values, even though $x[\vec{\alpha} = \vec{\delta}]$ does not evaluate to itself. So we distinguish *machine values* w—which do evaluate to themselves—from values v. The only difference is that machine values do not include $x[\vec{\alpha} = \vec{\delta}]$.

$$\begin{array}{lll} \textit{Machine values} & w & ::= n \mid x \mid \ell \mid (w, w) \mid \textbf{inl}\ w \mid \textbf{inr}\ w \mid \\ & & \quad \textbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon} \mid \textbf{select}\ \{(\vec{\alpha_i} = \vec{\delta_i}) \Rightarrow e_i\}_i \end{array}$$

6. Translation

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), the translation is also modal: the translation in the stable mode $\overset{}{\underset{\mathbb{S}}{\hookrightarrow}}$ produces a stable AFL expression $e^{\mathbb{S}}$, and the translation in the changeable mode $\overset{}{\underset{\mathbb{C}}{\hookrightarrow}}$ 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 15 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, $e : \tau \overset{}{\underset{\mathbb{S}}{\hookrightarrow}} e^{\mathbb{S}}$. 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 ($e : \tau \overset{}{\underset{\mathbb{C}}{\hookrightarrow}} e^{\mathbb{C}}$) 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 $\|\textbf{int}^{\mathbb{S}}\|$ of a stable integer is just **int**. In contrast, a changeable integer cannot be inspected or directly created in stable mode, but must be placed into a modifiable: $\|\textbf{int}^{\mathbb{C}}\| = \textbf{int mod}$. 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": $\|(\tau_1 \overset{}{\underset{\mathbb{C}}{\to}} \tau_2)^{\mathbb{S}}\| = \|\tau_1\| \overset{}{\underset{\mathbb{C}}{\to}} \|\tau_2\|^{\to\mathbb{C}}$. 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 while evaluating $e^{\mathbb{C}}$, there is no outer **mod**. Thus the translation $\|\tau\|^{\to\mathbb{C}}$ ignores the outer level (using the function $\left|-\right|^{\mathbb{S}}$, which replaces an outer level $\mathbb{C}$ with $\mathbb{S}$), and never

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

$$\frac{}{\rho \vdash w \Downarrow (\rho \vdash w)}\ \text{(TEvMachineValue)}$$

$$\frac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash w_1) \qquad \rho_1 \vdash e_2 \Downarrow (\rho_2 \vdash w_2)}{\rho \vdash (e_1, e_2) \Downarrow (\rho_2 \vdash (w_1, w_2))}\ \text{(TEvPair)}$$

$$\frac{\rho \vdash e \Downarrow (\rho' \vdash w)}{\rho \vdash \textbf{inl}\ e \Downarrow (\rho' \vdash \textbf{inl}\ w)}\ \text{(TEvSum)}$$

$$\frac{\begin{array}{l} \rho \vdash e_1 \Downarrow (\rho_1 \vdash w_1) \\ \rho_1 \vdash e_2 \Downarrow (\rho_2 \vdash w_2) \qquad \oplus(w_1, w_2) = w' \end{array}}{\rho \vdash \oplus(e_1, e_2) \Downarrow (\rho_2 \vdash w')}\ \text{(TEvPrimop)}$$

$$\frac{\rho \vdash e \Downarrow (\rho' \vdash (w_1, w_2))}{\rho \vdash \textbf{fst}\ e \Downarrow (\rho' \vdash w_1)}\ \text{(TEvFst)}$$

$$\frac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash w_1) \qquad \rho_1 \vdash [w_1/x]e_2 \Downarrow (\rho_2 \vdash w_2)}{\rho \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow (\rho_2 \vdash w_2)}\ \text{(TEvLet)}$$

$$\frac{\begin{array}{c} \rho \vdash e \Downarrow (\rho_1 \vdash \textbf{inl}\ w_1) \\ \rho_1 \vdash [w_1/x_1]e_1 \Downarrow (\rho_2 \vdash w) \end{array}}{\rho \vdash \textbf{case}\ e\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\} \Downarrow (\rho_2 \vdash w)}\ \text{(TEvCaseLeft)}$$

$$\frac{\begin{array}{l} \rho \vdash e_1^{\varepsilon} \Downarrow (\rho_1 \vdash \textbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon}) \\ \rho_1 \vdash e_2^{\varepsilon} \Downarrow (\rho_2 \vdash w_2) \\ \rho_2 \vdash [(\textbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon})/f][w_2/x]e^{\varepsilon} \Downarrow (\rho_3 \vdash w) \end{array}}{\rho \vdash \textbf{apply}^{\varepsilon}(e_1^{\varepsilon}, e_2^{\varepsilon}) \Downarrow (\rho_3 \vdash w)}\ \text{(TEvApply)}$$

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

$$\frac{\rho \vdash e_1 \Downarrow (\rho_1 \vdash \ell) \qquad \rho_1 \vdash [\rho_1(\ell)/x']e^{\mathbb{C}} \Downarrow (\rho_2 \vdash w)}{\rho \vdash \textbf{read}\ e_1\ \textbf{as}\ x'\ \textbf{in}\ e^{\mathbb{C}} \Downarrow (\rho_2 \vdash w)}\ \text{(TEvRead)}$$

$$\frac{\rho_1 \vdash e_1 \Downarrow (\rho' \vdash w)}{\rho \vdash (\textbf{select}\ \{\ldots, \vec{\delta} \Rightarrow e_1, \ldots\})[\vec{\alpha} = \vec{\delta}] \Downarrow (\rho' \vdash w)}\ \text{(TEvSelectE)}$$

$$\frac{\rho \vdash e^{\mathbb{C}} \Downarrow (\rho' \vdash w)}{\rho \vdash \textbf{mod}\ e^{\mathbb{C}} \Downarrow ((\rho', \ell \mapsto w) \vdash \ell)}\ \text{(TEvMod)}$$

Figure 14. Dynamic semantics for first runs of AFL programs

$$
\begin{array}{rcl}
\left|\mathbf{int}^{\mathbb{C}}\right|^{\mathbb{S}} & = & \mathbf{int}^{\mathbb{S}} \\
\left|(\tau_1 \times \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} & = & (\tau_1 \times \tau_2)^{\mathbb{S}} \\
\left|(\tau_1 + \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} & = & (\tau_1 + \tau_2)^{\mathbb{S}} \\
\left|(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} & = & (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}} \\
\\
\|\mathbf{int}^{\mathbb{S}}\| & = & \mathbf{int} \\
\|\mathbf{int}^{\mathbb{C}}\| & = & \mathbf{int\ mod} \\
\|(\tau_1 \xrightarrow[\mathbb{S}]{} \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| \xrightarrow[\mathbb{S}]{} \|\tau_2\| \\
\|(\tau_1 \xrightarrow[\mathbb{S}]{} \tau_2)^{\mathbb{C}}\| & = & \left(\|\tau_1\| \xrightarrow[\mathbb{S}]{} \|\tau_2\|\right) \mathbf{\ mod} \\
\|(\tau_1 \xrightarrow[\mathbb{C}]{} \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| \xrightarrow[\mathbb{C}]{} \|\tau_2\|^{\to\mathbb{C}} \\
\|(\tau_1 \xrightarrow[\mathbb{C}]{} \tau_2)^{\mathbb{C}}\| & = & \left(\|\tau_1\| \xrightarrow[\mathbb{C}]{} \|\tau_2\|^{\to\mathbb{C}}\right) \mathbf{\ mod} \\
\|(\tau_1 \times \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| \times \|\tau_2\| \\
\|(\tau_1 \times \tau_2)^{\mathbb{C}}\| & = & (\|\tau_1\| \times \|\tau_2\|) \mathbf{\ mod} \\
\|(\tau_1 + \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| + \|\tau_2\| \\
\|(\tau_1 + \tau_2)^{\mathbb{C}}\| & = & (\|\tau_1\| + \|\tau_2\|) \mathbf{\ mod} \\
\\
\|\tau\|^{\to\mathbb{C}} & = & \begin{cases} \|\left|\tau\right|^{\mathbb{S}}\| & \text{if } \tau\ \mathsf{O.C.} \\ \|\tau\| & \text{if } \tau\ \mathsf{O.S.} \end{cases}
\end{array}
$$

$$
\begin{array}{rlrl}
\|\cdot\| & = \cdot & \|\tau\|_\phi & = \|[\phi]\tau\| \\
\|\Gamma, x : \forall\emptyset[\mathsf{true}].\,\tau\| & = \|\Gamma\|, x : \|\tau\| & \|\tau\|_\phi^{\to\mathbb{C}} & = \|[\phi]\tau\|^{\to\mathbb{C}} \\
\|\Gamma, x : \forall\vec{\alpha}[D].\,\tau\| & = \|\Gamma\|, x : \Pi\vec{\alpha}[D].\,\tau & \|\Gamma\|_\phi & = \|[\phi]\Gamma\|
\end{array}
$$

Figure 15. Stabilization of types $\left|\tau\right|^{\mathbb{S}}$; translations $\|\tau\|$ and $\|\tau\|^{\to\mathbb{C}}$ of types; translation of typing environments $\|\Gamma\|$

returns a type of the form $(\cdots$ **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. The translation $\|\Gamma\|$ therefore translates only monomorphic types τ; type schemes are left alone (except for replacing the symbol $\forall$ with Π) until instantiation. Once instantiated, the type scheme is an ordinary closed source type, and can be translated by rule (TVar).

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 6.1 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, 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 17 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 $x : \forall\alpha[\mathsf{true}].\,(\mathbf{int}^{\alpha} \xrightarrow[\alpha]{} \mathbf{int}^{\alpha})^{\mathbb{S}}$, we need the type from $C; \Gamma \vdash_\varepsilon x : (\mathbf{int}^{\mathbb{C}} \xrightarrow[\mathbb{C}]{} \mathbf{int}^{\mathbb{C}})^{\mathbb{S}}$ so we can instantiate α in the translated term $x[\alpha = \mathbb{C}]$.

$\boxed{\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau \vdash e')}$ 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 16. Renaming the variable to be read (elimination forms)

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), (Sum), (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 17 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: $\vdash \tau\ \mathsf{O.C.}$

(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 16.

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

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

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

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

$$\frac{\Gamma \vdash v : \tau_1 \underset{\mathbb{S}}{\hookrightarrow} v'}{\Gamma \vdash \mathbf{inl}\ v : (\tau_1 + \tau_2)^{\mathbb{S}} \underset{\mathbb{S}}{\hookrightarrow} \mathbf{inl}\ v'} \text{ (Sum)}$$

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

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

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

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

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

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

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

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

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

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

Figure 17. Monomorphizing translation

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 17 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 **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 18, which examines the source expression e and, using type information, applies the rules necessary to produce an expression of mode ε.

6.4 Properties

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

Theorem 6.1 (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 $\cdot; \|\Gamma\|_{\phi} \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_{\phi}^{\vec{\mathbb{C}}}$.

The proof (in the appendix [Chen et al. 2011]) 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.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 18 (in fact, we extracted the algorithm from the proof).

We also prove that running a translated program gives the same result as running the source program. Theorem 6.5 states that in an initially empty store $\cdot$, if evaluating the translated program e' yields v' with new store ρ', then e evaluates to v where v corresponds to $[\rho']v'$ (the result of substituting values in the store ρ' for locations appearing in v').

To define this correspondence, we use a device somewhat similar to logical relations: a relation $\not\hookrightarrow$ on source and target expressions, allowing us to show that if $e : \tau \not\hookrightarrow e' : \underline{\tau}'$ then $v : \tau \not\hookrightarrow [\rho]v' : \underline{\tau}'$. Both e and e' must be closed. Our definition is weaker than the equivalence relations used in logical relations proofs: $\mathbf{apply}(\mathrm{id}, 4) \not\hookrightarrow 4$, for example. It does not attempt to equate all programs that have the same meaning, but only particu-

```
function trans (e, ε) = case (e, ε) of
  | (n, 𝕊) ⇒ Int
  | (x, 𝕊) ⇒ Var
  | ((v1, v2), 𝕊) ⇒ Pair(trans(v1, 𝕊), trans(v2, 𝕊))
  | (fun f(x) = e' : (τ1 →ε' τ2)^𝕊, 𝕊) ⇒ Fun(trans(e', ε'))
  | (inl v, 𝕊) ⇒ Sum(trans(v, 𝕊))
  | (fst (x : (τ1 × τ2)^δ), ε) ⇒ case (δ, ε) of
      | (𝕊,𝕊) ⇒ Fst(trans(x, 𝕊))
      | (𝕊,ℂ) ⇒ if τ1 O.S. then Write(trans(e, 𝕊))
                 else ReadWrite(trans(e, 𝕊)))
      | (ℂ,ℂ) ⇒ Read(LFst, trans(fst (x' : (τ1 × τ2)^𝕊)), ℂ),
                     trans(x, 𝕊))
  | (⊕(x1 : int^𝕊, x2 : int^𝕊), 𝕊) ⇒
      Prim(trans(x1, 𝕊), trans(x2, 𝕊))
  | (⊕(x1 : int^𝕊, x2 : int^𝕊), ℂ) ⇒ Write(trans(e, 𝕊)))
  | (⊕(x1 : int^ℂ, x2 : int^ℂ), ℂ) ⇒
      Read(LPrimop1,
           Read(LPrimop2, Write(trans(⊕(x1', x2'), 𝕊))))
  | (let x : τ'' = e1 : τ' in e2, ε) ⇒
      LetE(if τ'' O.S. then trans(e1, 𝕊)
           else (if τ' = τ'' then Mod(trans(e1, ℂ))
                 else Lift(trans(e1, ℂ))),
           trans(e2, ε))
  | (let x : ∀α⃗[D]. τ'' = v1 : τ' in e2, ε) ⇒
      let variants = all δ⃗i such that α⃗ = δ⃗i ⊩ D in
      let f = λs. if τ'' O.S. then trans(v1, 𝕊)
                  else (if τ' = τ'' then Mod(trans(v1, ℂ))
                        else Lift(trans(v1, ℂ))) in
        LetV(map f variants, trans(e2, ε))
  | (apply(x1 : (τ1 →ε' τ2)^δ, x2), ε) ⇒ case (ε', δ, ε) of
      | (𝕊,𝕊,𝕊) ⇒ App(trans(x1, 𝕊), trans(x2, 𝕊))
      | (ℂ,𝕊,ℂ) ⇒ App(trans(x1, 𝕊), trans(x2, 𝕊))
      | (𝕊,𝕊,ℂ) ⇒ if τ2 O.S. then Write(trans(e, 𝕊))
                  else ReadWrite(trans(e, 𝕊))
      | (ε',ℂ,ℂ) ⇒ Read(LApply,
                        trans(apply(x' : (τ1 →ε' τ2)^𝕊, x2), ℂ),
                        trans(x1, 𝕊))
      | (ℂ,𝕊,𝕊) ⇒ Mod(trans(e, ℂ))
      | (ε',ℂ,𝕊) ⇒ Mod(trans(e, ℂ))
  | (case x : τ of {x1⇒e1 , x2⇒e2}, ε) ⇒
      if τ O.S. then
        Case(trans(x, 𝕊), trans(e1, ε), trans(e2, ε))
      else Read(LCase,
                trans(case x' : |τ|^𝕊 of {x1⇒e1 , x2⇒e2}, ℂ),
                trans(x, 𝕊))
  | (x : τ, ℂ) ⇒ if τ O.S. then Write(trans(e, 𝕊))
                 else ReadWrite(trans(e, 𝕊))
  | (fun f(x) = e', ℂ) | (inl v, ℂ)
  | (n, ℂ) | ((v1, v2), ℂ) ⇒ Write(trans(e, 𝕊))
```

Figure 18. Translation algorithm

lar Level ML terms to AFL terms that are similarly structured, but have overhead (**mod**, write, etc.). Thus, integers are related to integers, pairs are related if their components are related, and so forth. The definition essentially ignores **mod** and **write** and ignores the mode in **apply**$^{\varepsilon}$. Since translated programs can have "extra" **read** and **let** expressions, these are "substituted out" in the relation, so that $3 \looparrowright$ **let** $x = 3$ **in** x. Functions are related if, given related arguments, they produce related results. Note that we will not induct over this relation; neither the term, nor the type, gets smaller.

We also relate substitutions: a source substitution s and a target substitution $\underline{s}$,

$$s = v_1/x_1, \ldots, v_n/x_n$$
$$\underline{s} = w_1/x_1, \ldots, w_n/x_n$$

are related at their environments $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and $\Gamma' = x_1 : \underline{\tau}'_1, \ldots, x_n : \underline{\tau}'_n$, written

$$(v_1/x_1, \ldots, v_n/x_n) : (x_1 : \tau_1, \ldots, x_n : \tau_n)$$
$$\looparrowright (w_1/x_1, \ldots, w_n/x_n) : (x_1 : \underline{\tau}'_1, \ldots, x_n : \underline{\tau}'_n)$$

if, for all k from 1 to n, we have $v_k : \tau_k \looparrowright w_k : \underline{\tau}_k$.

The key lemma (Lemma 6.3) is that if $e \overset{\varepsilon}{\hookrightarrow} e'$, the target program e' is related to e. Combined with Theorem 6.4, which shows that related programs evaluate to related values, this means that the translated program e' evaluates to the same value that e does. (Actually, e' is a value *related to* that value; only at **int**$^{\delta}$/**int**, and products thereof, are they identical.)

We begin by defining a *store substitution* operation:

Definition 6.2. The *store substitution* $[\rho]e$ is defined as an ordinary substitution, except for $e = \ell$, in which case $[\rho]\ell = [\rho]$(**mod** $\rho(\ell)$).

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

Proofs and several other lemmas can be found in the appendix [Chen et al. 2011].

Lemma 6.3 (Relation of Translation)**.** *If* $\Gamma \vdash e : \tau \overset{\varepsilon}{\hookrightarrow} e'$ *and* $\cdot \vdash s : \Gamma$ *and* $\cdot \vdash \underline{s} : \|\Gamma\|$ *and* $s : \Gamma \looparrowright \underline{s} : \|\Gamma\|$ *then*

$$[s]e : \tau \looparrowright [\underline{s}]e' : \underline{\tau}'$$

where $\underline{\tau}' = \|\tau\|$ *if* $\varepsilon = \mathbb{S}$, *and* $\underline{\tau}' = \|\tau\|^{\to\mathbb{C}}$ *if* $\varepsilon = \mathbb{C}$.

Theorem 6.4 (Generalized Translation Soundness)**.**
If $e : \sigma \looparrowright [\rho]e' : \underline{\sigma}'$ *and* $\mathcal{D} :: \rho \vdash e' \Downarrow (\rho' \vdash w)$
then $\mathcal{D}' :: e \Downarrow v$ *where* $v : \sigma \looparrowright [\rho']w : \underline{\sigma}'$.

Translation soundness now follows from Lemma 6.3 and Theorem 6.4:

Theorem 6.5 (Translation Soundness)**.** *If* $\cdot \vdash e : \tau \overset{\varepsilon}{\hookrightarrow} e'$ *and* $\cdot \vdash e' \Downarrow (\rho' \vdash w)$, *then* $e \Downarrow v$ *where* $v : \tau \looparrowright [\rho']w : \underline{\tau}'$.

Finally, we extend Theorem 6.5 to further show that the size $W(\mathcal{D})$ of the derivation of the target-language evaluation is within a constant factor of the size $W(\mathcal{D}')$ of the derivation of $e \Downarrow v$. We need a few definitions and intermediate results, which can be found in the appendix. The proof hinges on classifying the keywords added by the translation, such as **write**, as "dirty": a dirty keyword will lead 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.

Definition 6.6. 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.

Theorem 6.7. *If* `trans` $(e, \epsilon) = e'$ *then* e' *is deeply* 6*-bounded.*

$e : \sigma \looparrowright e' : \underline{\sigma}'$ Source expression e at type [schema] σ is related to target expression e' at type [schema] $\underline{\sigma}'$

$$
\begin{array}{rcll}
n : \mathbf{int}^{\delta} & \looparrowright & n : \mathbf{int} & \\
(e_1, e_2) : (\tau_1 \times \tau_2)^{\delta} & \looparrowright & (e_1', e_2') : \underline{\tau}_1' \times \underline{\tau}_2' & \text{if } e_1 : \tau_1 \looparrowright e_1' : \underline{\tau}_1' \text{ and } e_2 : \tau_2 \looparrowright e_2' : \underline{\tau}_2' \\
(\mathbf{inl}\ e) : (\tau_1 + \tau_2)^{\delta} & \looparrowright & (\mathbf{inl}\ e') : \underline{\tau}_1' + \underline{\tau}_2' & \text{if } e : \tau_1 \looparrowright e' : \underline{\tau}_1' \\
(\mathbf{fst}\ e) : \tau & \looparrowright & (\mathbf{fst}\ e') : \underline{\tau}' & \text{if } e : (\tau \times \tau_2)^{\delta} \looparrowright e' : \underline{\tau}' \times \underline{\tau}_2' \\
\oplus(e_1, e_2) : \tau & \looparrowright & \oplus(e_1', e_2') : \underline{\tau}' & \text{if } e_1 : \tau_1 \looparrowright e_1' : \underline{\tau}_1' \text{ and } e_2 : \tau_2 \looparrowright e_2' : \underline{\tau}_2' \\
e : \tau & \looparrowright & (\mathbf{mod}\ e^{\mathbb{C}}) : \underline{\tau}'\ \mathbf{mod} & \text{if } e : \tau \looparrowright e^{\mathbb{C}} : \underline{\tau}' \\
e : \tau & \looparrowright & (\mathbf{write}(e^{\mathbb{S}})) : \underline{\tau}' & \text{if } e : \tau \looparrowright e^{\mathbb{S}} : \underline{\tau}' \\
[e_1/x]e : \tau_2 & \looparrowright & (\mathbf{read}\ e^{\mathbb{S}}\ \mathbf{as}\ x\ \mathbf{in}\ e^{\mathbb{C}}) : \underline{\tau}_2' & \text{if } e_1 : \tau_1 \looparrowright e^{\mathbb{S}} : \underline{\tau}_1'\ \mathbf{mod} \text{ and for all } v : \tau_1 \looparrowright w : \underline{\tau}_1', \\
 & & & \quad \text{we have } [v/x]e : \tau_2 \looparrowright [w/x]e^{\mathbb{C}} : \underline{\tau}_2' \\
\mathbf{apply}(e_1, e_2) : \tau_2 & \looparrowright & \mathbf{apply}^{\varepsilon}(e_1', e_2') : \underline{\tau}_2' & \text{if } e_1 : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\delta} \looparrowright e_1' : \underline{\tau}_1' \xrightarrow[\varepsilon]{} \underline{\tau}_2' \text{ and } e_2 : \tau_1 \looparrowright e_2' : \underline{\tau}_1' \\
e : [\vec{\delta}/\vec{\alpha}]\tau & \looparrowright & e'[\vec{\alpha} = \vec{\delta}] : \|[\vec{\delta}/\vec{\alpha}]\tau\| & \text{if } e : \forall\vec{\alpha}[D].\,\tau \looparrowright e' : \Pi\vec{\alpha}[D].\,\tau \\
v : \forall\vec{\alpha}[D].\,\tau & \looparrowright & \mathbf{select}\ \{\vec{\delta}_i \Rightarrow e_i\}_i : \Pi\vec{\alpha}[D].\,\tau & \text{if for all } i \text{ we have } v : [\vec{\delta}/\vec{\alpha}]\tau \looparrowright e_i : \|[\vec{\delta}/\vec{\alpha}]\tau\| \\
(\mathbf{fun}\ f(x) = e) : (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\delta} & \looparrowright & (\mathbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon}) : \underline{\tau}_1' \xrightarrow[\varepsilon]{} \underline{\tau}_2' & \text{if for all } v : \tau_1 \looparrowright w : \underline{\tau}_1', \\
 & & & \quad ([(\mathbf{fun}\ f(x) = e)/f][v/x]e) : \tau_2 \\
 & & & \looparrowright ([(\mathbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon})/f][w/x]e^{\varepsilon}) : \underline{\tau}_2' \\
(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \tau_2 & \looparrowright & (\mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2') : \underline{\tau}_2' & \text{if } e_1 : \tau_1 \looparrowright e_1' : \underline{\tau}_1' \text{ and} \\
 & & & \text{for all } v : \tau_1 \looparrowright w : \underline{\tau}_1',\ [v/x]e_2 : \tau_2 \looparrowright [w/x]e_2' : \underline{\tau}_2' \\
[e_1/x]e_2 : \tau_2 & \looparrowright & (\mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2') : \underline{\tau}_2' & \text{if } e_1 : \tau_1 \looparrowright e_1' : \underline{\tau}_1' \text{ and} \\
 & & & \text{for all } v : \tau_1 \looparrowright w : \underline{\tau}_1',\ [v/x]e_2 : \tau_2 \looparrowright [w/x]e_2' : \underline{\tau}_2' \\
(\mathbf{case}\ e\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}) : \tau & \looparrowright & (\mathbf{case}\ e'\ \mathbf{of}\ \{x_1 \Rightarrow e_1'\ ,\ x_2 \Rightarrow e_2'\}) : \underline{\tau}' & \text{if } e : (\tau_1 + \tau_2)^{\delta} \looparrowright e' : \underline{\tau}_1' + \underline{\tau}_2' \text{ and} \\
 & & & \text{for all } v : \tau_k \looparrowright w : \underline{\tau}_k', \\
 & & & \quad [v/x_k]e_k : \tau \looparrowright [w/x_k]e_k' : \underline{\tau}' \text{ for } k \in \{1, 2\}
\end{array}
$$

Figure 19. Correspondence of source and target expressions, used in Theorem 6.5 and other results

Theorem 6.8 (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 $\frac{k}{k+1} W(\mathcal{D})$.*

The cost theorem follows from Theorem C.5 (in the appendix)—a generalization of Theorem 6.4—and Theorem 6.8:

Theorem 6.9. *If $\mathcal{D}$ derives $\cdot \vdash \mathtt{trans}\,(e, \varepsilon) \Downarrow (\rho' \vdash w)$ then $\mathcal{D}'$ derives $e \Downarrow v$ where $v : \tau \looparrowright [\rho']w : \underline{\tau}'$ and $W(\mathcal{D}) \leq 7W(\mathcal{D}')$.*

Acar et al. [2006] proved that given a well-typed AFL program, change propagation updates the output consistently with an initial run. Using Theorems 6.1 and 6.5, this implies that change propagation is consistent with an initial run of the source program.

7. Related Work

Incremental computation. Self-adjusting computation provides an approach to incremental computation, which has been studied extensively [Ramalingam and Reps 1993; Demers et al. 1981; Pugh and Teitelbaum 1989; Abadi et al. 1996]. Key techniques behind self-adjusting computation include dynamic dependence graphs, which allows a fully general change propagation mechanism [Acar et al. 2006], and a form of memoization that allows inexact computations to be reused via memoized computations that are (recursively) self-adjusting [Acar et al. 2009]. Programming-language features allow writing self-adjusting programs but these require syntactically separating stable and changeable data, as well as code that operates on such data [Acar et al. 2006, 2009; Ley-Wild et al. 2008; Hammer et al. 2009]. DITTO [Shankar and Bodik 2007] shows the benefits of eliminating user annotations. By customizing dependency tracking for invariant checking programs, DITTO provides a fully automatic incremental invariant checker. The approach, however, is domain-specific and only works for certain programs (e.g., functions cannot return arbitrary values): it is unsound in general.

Information flow and constraint-based type inference. A number of information flow type systems have been developed to check security properties, including the SLam calculus [Heintze and Riecke 1998], JFlow [Myers 1999] and a monadic system [Crary et al. 2005]. 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].

8. Conclusion

This paper presents techniques for translating purely functional programs to programs that can automatically self-adjust in response to dynamic changes to their data. Our contributions include a constraint-based type system for inferring self-adjusting-computation types from purely functional programs, a type-directed translation algorithm that rewrites purely functional programs into self-adjusting programs, and proofs of critical properties of the translation: type soundness and observational equivalence, as well as the intrinsic property of time complexity. Perhaps unsurprisingly,

the theorems and their proofs were critical to the determination of the type systems and the translation algorithm: many of our initial attempts at the problem resulted in target programs that were not type sound, that did not ensure observational equivalence, or were asymptotically slower than the source.

These results take an important step towards the development of languages and compilers that can generate code that can respond automatically to dynamically changing data correctly and asymptotically optimally, without substantial programming effort. Remaining open problems include generalization to imperative programs with references, techniques and proofs to determine or improve the asymptotic complexity of dynamic responses, and a complete and careful implementation and its evaluation.

Acknowledgments

We thank the anonymous ICFP reviewers, as well as Arthur Charguéraud, for their useful comments on the submitted version of this paper.

References

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

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

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

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

U. A. Acar, G. E. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoğlu. Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation*, 2010a.

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

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

Y. Chen, J. Dunfield, M. A. Hammer, and U. A. Acar. Online appendix to *Implicit Self-Adjusting Computation for Purely Functional Programs*, 2011. `http://www.mpi-sws.org/~joshua/Chen11appendix.pdf`.

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

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

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212. ACM, 1982.

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

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

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

J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Trans. Prog. Lang. Sys.*, 28: 1035–1087, 2006.

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.

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

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

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

R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

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

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

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

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

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

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

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

D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, Sept. 1990.

P. M. Sansom and S. L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Principles of Programming Languages*, pages 355–366, 1995.

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

V. Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *APLAS*, pages 283–302, 2003.

D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.