Implicit self-adjusting computation for purely functional programs

YAN CHEN and JOSHUA DUNFIELD

Max Planck Institute for Software Systems, Kaiserslautern and Saarbrücken, Germany
(*e-mail:* {chenyan,joshua}@mpi-sws.org)

MATTHEW A. HAMMER

University of Maryland, College Park, Maryland, USA
(*e-mail:* hammer@mpi-sws.org)

UMUT A. ACAR

Carnegie Mellon University, Pittsburgh, Pennsylvania, USA and INRIA Paris-Rocquencourt, France
(*e-mail:* umut@cs.cmu.edu)

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.

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 (Chiang & 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 & 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 & Teitelbaum 1989; Field & 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 can require 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.

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

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

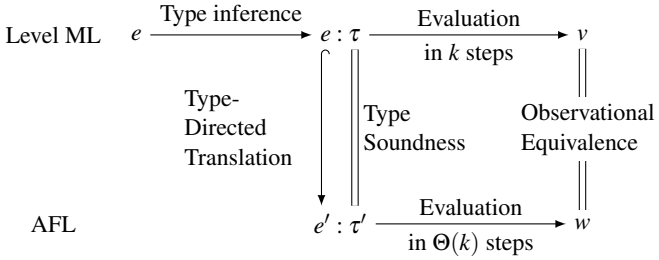

Fig. 1. Visualizing the translation between the source language Level ML and the target language AFL, and related properties.

Taken together, the type system, its inference algorithm, and the translation algorithm enable translating purely functional source programs to self-adjusting target programs using top-level type annotations on the input type of the source program. These top-level type annotations simply mark what part of the input data is subject to change. Type inference assigns types to the rest of the program and the translation algorithm translates the program into self-adjusting target code. 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.* 2006b). 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 6.14).

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.* 2006b)). The third property shows that the translated program takes asymptotically as long to evaluate (from scratch) as the corresponding source program. To prove this complexity result, we use a cost semantics (Sands 1990; Sansom & Peyton Jones 1995) that enables precise reasoning about

the complexity of the evaluation time. The time for incremental updates via change propagation is usually asymptotically more efficient than running from scratch. Proving a tight bound on the complexity of change propagation is beyond the scope of this paper. Interested readers can refer to Ley-Wild *et al.* (2009) for more detail.

We have implemented our approach as an extension of Standard ML and the MLton compiler (MLton). The implementation takes SML code annotated with level types and generates self-adjusting code that can be linked with a previously published, publicly available library for self-adjusting computation (Acar *et al.* 2009). We evaluate the effectiveness of our compiler by considering a range of benchmarks involving lists, vectors, and matrices, as well as a ray tracer. For these benchmarks, our compiler incrementalizes existing code with only trivial amounts of annotation. The resulting programs are often asymptotically more efficient, leading to orders of magnitude speedups in practice. A detailed experimental evaluation is beyond the scope of this paper; interested readers can refer to Chen *et al.* (2012).

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 could be applied more broadly, at least for other strict languages with self-adjusting libraries (Carlsson 2002).

Paper guide. To describe our approach in the overview section (Section 2), we start with the translation problem and work back to the type system, because we feel that motivates well the problem and our proposed solution. When presenting the technical material, however, we start with the type system, because the details of the translation algorithm and our theorems rely on it. 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.

Previous version. This article is an extended version of a paper that appeared in the Proceedings of the 2011 International Conference on Functional Programming (Chen *et al.* 2011). Apart from many smaller improvements and corrections, this version fixes a major problem with one of the proofs. Note that only the proof was flawed, not its conclusion; the results claimed do, in fact, hold.

2 Overview

We present an informal overview of our approach via examples in an extension of SML with features for implicit self-adjusting computation. An implementation of this approach, including support for algebraic datatypes, Hindley-Milner polymorphism and imperative programs, was described in Chen *et al.* (2012). However, to simplify the theoretical presentation, our formalism will only consider a core subset of this language. We start with a brief description of our target language, explicit self-adjusting computation, as laid out in previous work. After this description, we outline our proposed approach.

2.1 Explicit self-adjusting computation

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.* 2006b). The programmer operates on modifiables with **mod**, **read**, and **write** constructs to create, read from, and write into modifiables. The run-time system of a self-adjusting language uses these constructs to represent the execution as a (directed, acyclic) dependency graph, enabling efficient *change propagation* when the data changes in small amounts.

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 delimits the code that can directly inspect the changeable value `y`, and the changeable arrow type ensures an important 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, and thus reusing the computation of the square `x2`.

Note that the result type of `squareplus_SC` is `int`, not `int` **mod**; `squareplus_SC` does not itself create a modifiable, it just writes to the modifiable created by the caller of the function in the context of a (dynamically) enclosing **mod** expression. In fact, programming directly in the explicit self-adjusting computation setting requires extensive knowledge of such non-obvious details of the type system. Our implicit approach also has a nontrivial type system, but manages to expose fewer sharp corners to the user.

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

```
squareplus_CS: int mod * int →C int
fun squareplus_CS (x, y) =
  let x2 = mod (read x as x' in write(x' * x')) in
```

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

```
read x2 as x2' in
  let r = x2' + y in
    write(r)
```

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

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

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

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

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

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

2.2 Implicit self-adjusting computation

To make self-adjusting computation implicit, we use type information to insert **read**s, **write**s, and **mod**s automatically. The user annotates the input type, as well as the corresponding data declarations, of the program; we infer types for all expressions, and use this information to guide a translation algorithm. The translation algorithm returns well-typed self-adjusting target programs. The translation requires no expression-level annotations.

For the example function `squareplus` above, we can automatically derive `squareplus_SC` and `squareplus_CS` from just the type of the function (expressed in a slightly different form, as we discuss next).

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

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

A more detailed example: `mapPair`. To illustrate how our translation works, consider a function `mapPair` that takes two integer lists and increments the elements in both lists. This function can be written by applying the standard higher-order `map` over lists. Figure 2 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

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

inc : int → int
fun inc (x) = x+1

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

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

Fig. 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 2. *)
```

Fig. 3. Function `mapPair` in Level ML, with level types.

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 **int**$^{\mathbb{S}}$ `list`$^{\mathbb{C}}$, and one with unchanging length has type **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 l1` we increment stable integers, and in `map inc l2` we increment changeable integers, so the type inferred for `inc` must be generic: $\forall\delta.\ \mathbf{int}^{\delta} \xrightarrow[\delta]{} \mathbf{int}^{\delta}$. Our translation produces two implementations of `inc`, one per instantiation (δ=$\mathbb{S}$ and δ=$\mathbb{C}$): `inc_S` and `inc_C` (in Figure 4). Since we want to use `inc` with the higher-order function `map`, we need to generate a "selector" function that takes an instantiation and picks out the appropriate implementation:

```
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.) The types of `inc_S` and `inc_C` are produced by a type-level translation that, very roughly, replaces changeable types with **mod** types (Section 6.1).

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

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

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

3 A type system for implicit self-adjusting computation

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

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

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

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

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

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

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

map : ∀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)
fun^S mapPair (l1, l2) = (map[δH=S,δT=C] inc[δ=S] l1,
                          map[δH=C,δT=S] inc[δ=C] l2)
```

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

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 struc-

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

Fig. 5. Levels, constraints, types, and type schemes in Level ML.

ture 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 & Simonet (2003). Types in Level ML (Figure 5) include two *(security) levels*, stable and changeable. We generally follow their approach and notation. The two key differences are that (1) since Level ML is purely functional, we need no "program counter" level "*pc*"; (2) we need a mode ε on function types.

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

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

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

Types. Types consist of integers tagged with their level, products[2] 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. The *mode* ε is the level of the computation encapsulated by the function. This mode determines how a function can manipulate changeable values: a function in stable mode cannot directly manipulate changeable values; it can only pass them around. By contrast, a changeable-mode function can directly manipulate changeable values. The outer level δ is the level of the function itself, as a value. We say that a type is *ground* if it contains no level variables.

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

Subtyping. Figure 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

[2] In Pottier & 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.

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

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

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

Fig. 6. Subtyping.

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

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

Fig. 7. Lower bound of a type.

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

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

$$\mathbf{int}^{\delta_1} \mathrel{\mathring{=}} \mathbf{int}^{\delta_2} \qquad (\tau_1 + \tau_2)^{\delta_1} \mathrel{\mathring{=}} (\tau_1 + \tau_2)^{\delta_2}$$
$$(\tau_1 \times \tau_2)^{\delta_1} \mathrel{\mathring{=}} (\tau_1 \times \tau_2)^{\delta_2} \qquad (\tau_1 \xrightarrow{\varepsilon} \tau_2)^{\delta_1} \mathrel{\mathring{=}} (\tau_1 \xrightarrow{\varepsilon} \tau_2)^{\delta_2}$$

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

make stable-mode functions subtypes of changeable-mode functions, but changeable mode functions are more expensive; silent coercion would make performance less predictable.)

For simplicity, our type system will support only a weaker form of subtyping where only the outer levels can differ. In practice, the more powerful subtyping system could be used; see the discussion of let-expressions in Section 4.1.

Levels and types. We rely on several relations between levels and types to ascertain various invariants. A type τ is *higher than* δ, written $\delta \lhd \tau$, if the outer level of the type is at least δ. Figure 7 defines this relation. We distinguish between outer-stable and outer-changeable types (Figure 8). We write $\tau\ \mathsf{O.S.}$ if the outer level of τ is $\mathbb{S}$. Similarly, we write $\tau\ \mathsf{O.C.}$ 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 $\tau_1 \mathrel{\mathring{=}} \tau_2$, 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

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

Fig. 9. Abstract syntax of the source language Level ML.

level-variable comparisons $\leq$ and level-type comparisons $\delta \lhd \tau$, which type inference composes into conjunctions of satisfiability predicates $\exists \vec{\alpha}.C$.

The subtyping and lower bound relations defined in Figures 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 (instantiation) of α in ϕ, and $[\phi]\tau$ for the usual substitution operation on types. For example, if $\phi(\alpha) = \mathbb{S}$ then $[\phi]\left(\mathbf{int}^{\alpha} + \mathbf{int}^{\mathbb{C}}\right)^{\alpha} = \left(\mathbf{int}^{\mathbb{S}} + \mathbf{int}^{\mathbb{C}}\right)^{\mathbb{S}}$.

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

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. Consider the types defined by the grammar

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

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

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

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

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

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

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

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

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

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

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

Fig. 10. Typing rules for Level ML.

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

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

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

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

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

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

- **(SLetE):** In the first let-rule (SLetE), the expression bound may be a non-value, so we do not generalize and simply type the body in the same mode as the whole **let**, assuming that the bound expression has the specified type in any mode ε'.[3]
- **Subsumption on outer levels:** We allow subsumption only when the subtype and supertype are equal up to their outer levels, e.g. from a bound expression e_1 of subtype $\mathbf{int}^{\mathbb{S}}$ to an assumption $x : \mathbf{int}^{\mathbb{C}}$. This simplifies the translation, with no loss of expressiveness: to handle "deep" subsumption, such as

  $$(\mathbf{int}^{\mathbb{S}} \xrightarrow[\mathbb{S}]{} \mathbf{int}^{\mathbb{S}})^{\mathbb{S}} <: (\mathbf{int}^{\mathbb{S}} \xrightarrow[\mathbb{S}]{} \mathbf{int}^{\mathbb{C}})^{\mathbb{C}}$$

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

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

the types are equal up to their outer level. Because we are binding a value, we generalize its type by quantifying over the type's free level variables.

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

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

- The mode ε' of the function being called must match the current mode ε (the caller's mode): $\varepsilon' = \varepsilon$.
  To see why, first consider the case where we are in stable mode and try to apply a changeable-mode function ($\varepsilon = \mathbb{S}$ and $\varepsilon' = \mathbb{C}$). Changeable data can be directly inspected only in changeable mode; since changeable-mode functions can directly inspect changeable data, the call would allow us to inspect changeable data from stable mode, breaking the property that stable data depends only on stable data.
  Now consider the case where we are in changeable mode, and try to call a stable-mode function ($\varepsilon = \mathbb{C}$ and $\varepsilon' = \mathbb{S}$). This call would not directly violate the same property; we forbid it to simplify translation to a target language that distinguishes stable and changeable modes. Since the rules (SLetV) and (SLetE) can switch from changeable mode to stable mode, we lose no expressive power.
- The outer level of the result of the function, τ_2, must be higher than δ, the function's level: $\delta \lhd \tau_2$.
  The situation we disallow is when $\delta = \mathbb{C}$ and τ_2 is outer-stable, that is, when the called function has a type like $(\tau_1 \xrightarrow{\varepsilon} \mathbf{int}^{\mathbb{S}})^{\mathbb{C}}$. Here, the result type $\mathbf{int}^{\mathbb{S}}$ is stable and therefore must not depend on changeable data. But the type $(\tau_1 \xrightarrow{\varepsilon} \mathbf{int}^{\mathbb{S}})^{\mathbb{C}}$ is changeable: a change in program input could cause it to be entirely replaced by another function, which could of course return a different result!
  (Assuming "deep" subsumption, we lose no expressive power: we can coerce a function of type $(\tau_1 \xrightarrow{\varepsilon} \mathbf{int}^{\mathbb{S}})^{\mathbb{C}}$ to type $(\tau_1 \xrightarrow{\varepsilon} \mathbf{int}^{\mathbb{C}})^{\mathbb{C}}$, which satisfies the constraint.)

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

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

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

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

4.2 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 & 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 & Simonet 2003).

Our formulation of the constraint-based rules follows a standard presentation style (Odersky *et al.* 1999). That style, while relatively concise, obscures how constraints are manipulated in practice: It is tempting to read the typing rules in Figure 10 as taking in a constraint C as input. But in an actual constraint-based typechecker, C cannot be input, because C is not known until the program has been traversed! In practice, C should be thought of as both input and output: at the start of typechecking, C is empty (equivalently, is `true`); as the typechecker traverses the program, C is extended with additional constraints. For example, the premise $C \Vdash \delta \leq \varepsilon$ in (SFst) really corresponds to adding $\delta \leq \varepsilon$ to the "current" C, not to checking $\delta \leq \varepsilon$ under a known constraint.

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

4.3 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. To simplify the presentation, we omit the symmetric rules (SEvSumRight), (SEvSnd) and (SEvCaseRight).

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.

Levels	δ, ε	$::=$	$\mathbb{S} \mid \mathbb{C}$
Types	$\underline{\tau}$	$::=$	$\mathbf{int} \mid \underline{\tau}\ \mathbf{mod} \mid \underline{\tau}_1 \times \underline{\tau}_2 \mid \underline{\tau}_1 + \underline{\tau}_2 \mid \underline{\tau}_1 \xrightarrow{\varepsilon} \underline{\tau}_2$
Type schemes	σ	$::=$	$\forall \vec{\alpha}[D].\, \tau$
Typing environments	Γ	$::=$	$\cdot \mid \Gamma, x : \sigma \mid \Gamma, x : \underline{\tau}$
Variables	$\underline{x}$	$::=$	$x \mid x[\vec{\alpha} = \vec{\delta}]$
Values	w	$::=$	$n \mid \underline{x} \mid \ell \mid (w_1, w_2) \mid \mathbf{inl}\ w \mid \mathbf{inr}\ w \mid$ $\mathbf{fun}^{\mathbb{S}}\ f(x) = e^{\mathbb{S}} \mid \mathbf{fun}^{\mathbb{C}}\ f(x) = e^{\mathbb{C}} \mid \mathbf{select}\ \{(\vec{\alpha}_i = \vec{\delta}_i) \Rightarrow e_i\}_i$
Expressions	e	$::=$	$e^{\mathbb{S}} \mid e^{\mathbb{C}}$
Stable expressions	$e^{\mathbb{S}}$	$::=$	$w \mid \oplus(e^{\mathbb{S}}, e^{\mathbb{S}}) \mid \mathbf{fst}\ e^{\mathbb{S}} \mid \mathbf{snd}\ e^{\mathbb{S}} \mid$ $\mathbf{apply}^{\mathbb{S}}(e^{\mathbb{S}}, e^{\mathbb{S}}) \mid \mathbf{let}\ x = e^{\mathbb{S}}\ \mathbf{in}\ e^{\mathbb{S}} \mid$ $\mathbf{case}\ e^{\mathbb{S}}\ \mathbf{of}\ \{x_1 \Rightarrow e^{\mathbb{S}}\ ,\ x_2 \Rightarrow e^{\mathbb{S}}\} \mid$ $\mathbf{mod}\ e^{\mathbb{C}}$
Changeable expressions	$e^{\mathbb{C}}$	$::=$	$\mathbf{apply}^{\mathbb{C}}(e^{\mathbb{S}}, e^{\mathbb{S}}) \mid \mathbf{let}\ x = e^{\mathbb{S}}\ \mathbf{in}\ e^{\mathbb{C}} \mid$ $\mathbf{case}\ e^{\mathbb{S}}\ \mathbf{of}\ \{x_1 \Rightarrow e^{\mathbb{C}}\ ,\ x_2 \Rightarrow e^{\mathbb{C}}\} \mid$ $\mathbf{read}\ e^{\mathbb{S}}\ \mathbf{as}\ y\ \mathbf{in}\ e^{\mathbb{C}} \mid \mathbf{write}(e^{\mathbb{S}})$

Fig. 11. Types and expressions in the target language AFL.

Level polymorphism is supported through an explicit **select** construct and an explicit polymorphic instantiation. In Section 6, we describe how polymorphic source expressions become **select**s in AFL. The type schemes used in the target are identical to those in the source language; $\sigma = \forall \vec{\alpha}[D].\, \tau$ quantifies over *source* types τ (from Figure 5), not target types $\underline{\tau}$. We cannot quantify over target types here, because no single type scheme over target types can represent exactly the set of types corresponding to the instances of a source type scheme. For example, the source type scheme $\forall \alpha[\mathsf{true}].\, \mathbf{int}^{\alpha}$ corresponds to **int** if α is instantiated with $\mathbb{S}$, and to **int mod** if α is instantiated with $\mathbb{C}$, but the set of types $\{\mathbf{int}, (\mathbf{int}\ \mathbf{mod})\}$ does not correspond to the instances of any type scheme.

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

We distinguish stable expressions $e^{\mathbb{S}}$ from changeable expressions $e^{\mathbb{C}}$. Stable expressions create purely functional values; $\mathbf{apply}^{\mathbb{S}}$ applies a stable-mode function. The **mod** construct evaluates a changeable expression and writes the output value to a modifiable, yielding a location, which is a stable expression. Changeable expressions are computations that end in a **write** of a pure value. Changeable-mode application $\mathbf{apply}^{\mathbb{C}}$ 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 within a **select** has type $\|[\vec{\delta}/\vec{\alpha}]\tau\|$, where $[\vec{\delta}/\vec{\alpha}]\tau$ is a source-level polymorphic type with the levels $\vec{\delta}$ substituted for the variables $\vec{\alpha}$, and $\|-\|$ translates source types to target types (see Section 6.1). Rule (TPVar) is a standard rule for variables of monomorphic type, but rule (TVar) gives the instantiation $x[\vec{\alpha} = \vec{\delta}]$, of a variable x of polymorphic type, the type $\|[\vec{\delta}/\vec{\alpha}]\tau\|$—matching the monomorphic expression from the **select** to which x is bound.

5.1 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. Again, we omit symmetric rules such as (SEvSumRight).

According to the grammar in Figure 11, $x[\vec{\alpha} = \vec{\delta}]$ is a value. It might seem that evaluation (Figure 14) could replace the variable x by a **select** expression, yielding **select** $\{\ldots\}[\vec{\alpha} = \vec{\delta}]$, which does not evaluate to itself. However, $x[\vec{\alpha} = \vec{\delta}]$ is not closed, and we only evaluate closed target expressions.

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

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

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

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

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

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

$$\frac{\Lambda;\Gamma \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : \underline{\tau}_1 \qquad \Lambda;\Gamma \vdash_{\mathbb{S}} e_2^{\mathbb{S}} : \underline{\tau}_2}{\Lambda;\Gamma \vdash_{\mathbb{S}} (e_1^{\mathbb{S}}, e_2^{\mathbb{S}}) : \underline{\tau}_1 \times \underline{\tau}_2}\ \text{(TPair)}$$

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

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

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

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

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

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

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

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

Fig. 12. Typing rules of the target language AFL.

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

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

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

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

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

Fig. 13. Dynamic semantics of source Level ML programs.

the types in the environment Γ. The second function, $\|\tau\|^{\to\mathbb{C}}$, makes sense in two related situations: translating the type τ of an expression e in the changeable mode ($e : \tau \underset{\mathbb{C}}{\hookrightarrow} e^{\mathbb{C}}$) and translating the codomain of changeable functions.

In the stable mode, values of stable type can be used and created directly, so the "stable" translation $\|\textbf{int}^{\mathbb{S}}\|$ 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": $\|(\tau_1 \underset{\mathbb{C}}{\to} \tau_2)^{\mathbb{S}}\| = \|\tau_1\| \underset{\mathbb{C}}{\to} \|\tau_2\|^{\to\mathbb{C}}$. The reason is that a changeable-mode function can only be applied in the changeable mode; the function result is not placed into a modifiable until we return to the stable mode, so putting a **mod** on the codomain would not match the dynamic semantics of AFL.

The second function $\|\tau\|^{\to\mathbb{C}}$ defines the type of a changeable expression e that writes to a modifiable containing τ, yielding a changeable target expression $e^{\mathbb{C}}$. The source type has an outer $\mathbb{C}$, so when the value is written, it will be placed into a modifiable and have **mod** type. But *inside* the evaluation of $e^{\mathbb{C}}$, there is no outer **mod** on the type.[4] 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 returns a type of the form $(\cdots\ \textbf{mod})$. 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.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 14. Dynamic semantics for first runs of AFL programs.

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

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

No single type scheme over target types can represent exactly the set of types $\{\mathbf{int}, \mathbf{int\ mod}\}$. The translation $\|\Gamma\|$, therefore, translates only monomorphic types τ; type schemes are left alone until instantiation. Once instantiated, the type scheme is an ordinary closed source type, and can be translated by rule (Var) in Figure 16.

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

$$\begin{array}{rcl}
\left|\mathbf{int}^{\mathbb{C}}\right|^{\mathbb{S}} & = & \mathbf{int}^{\mathbb{S}} \\
\left|(\tau_1 \times \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} & = & (\tau_1 \times \tau_2)^{\mathbb{S}} \\
\left|(\tau_1 + \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} & = & (\tau_1 + \tau_2)^{\mathbb{S}} \\
\left|(\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{C}}\right|^{\mathbb{S}} & = & (\tau_1 \xrightarrow[\varepsilon]{} \tau_2)^{\mathbb{S}}
\end{array}$$

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

$$\begin{array}{rcl}
\|\mathbf{int}^{\mathbb{S}}\| & = & \mathbf{int} \\
\|\mathbf{int}^{\mathbb{C}}\| & = & \mathbf{int}\ \mathbf{mod} \\
\|(\tau_1 \xrightarrow[\mathbb{S}]{} \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| \xrightarrow[\mathbb{S}]{} \|\tau_2\| \\
\|(\tau_1 \xrightarrow[\mathbb{S}]{} \tau_2)^{\mathbb{C}}\| & = & \left(\|\tau_1\| \xrightarrow[\mathbb{S}]{} \|\tau_2\|\right)\ \mathbf{mod} \\
\|(\tau_1 \xrightarrow[\mathbb{C}]{} \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| \xrightarrow[\mathbb{C}]{} \|\tau_2\|^{\to\mathbb{C}} \\
\|(\tau_1 \xrightarrow[\mathbb{C}]{} \tau_2)^{\mathbb{C}}\| & = & \left(\|\tau_1\| \xrightarrow[\mathbb{C}]{} \|\tau_2\|^{\to\mathbb{C}}\right)\ \mathbf{mod} \\
\|(\tau_1 \times \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| \times \|\tau_2\| \\
\|(\tau_1 \times \tau_2)^{\mathbb{C}}\| & = & (\|\tau_1\| \times \|\tau_2\|)\ \mathbf{mod} \\
\|(\tau_1 + \tau_2)^{\mathbb{S}}\| & = & \|\tau_1\| + \|\tau_2\| \\
\|(\tau_1 + \tau_2)^{\mathbb{C}}\| & = & (\|\tau_1\| + \|\tau_2\|)\ \mathbf{mod}
\end{array}$$

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

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

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

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

Translations under substitution ϕ

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l}\Gamma \vdash e \rightsquigarrow (x \gg x' : \tau' \vdash e') \qquad \tau'\ \mathsf{O.C.} \\ \Gamma, x' : \left|\tau'\right|^{\mathbb{S}} \vdash e' : \tau \hookrightarrow_{\mathbb{C}} e^{\mathbb{C}} \qquad \Gamma \vdash x : \tau' \hookrightarrow_{\mathbb{S}} \underline{x}\end{array}}{\Gamma \vdash e : \tau \hookrightarrow_{\mathbb{C}} \mathbf{read}\, \underline{x}\, \mathbf{as}\, x'\, \mathbf{in}\, e^{\mathbb{C}}} \text{ (Read)} \qquad \frac{\Gamma \vdash e : \tau \hookrightarrow_{\mathbb{S}} e^{\mathbb{S}} \qquad \tau\ \mathsf{O.S.}}{\Gamma \vdash e : \tau \hookrightarrow_{\mathbb{C}} \mathbf{let}\, r = e^{\mathbb{S}}\, \mathbf{in}\, \mathbf{write}(r)} \text{ (Write)}$$

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

Fig. 16. Monomorphizing translation.

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

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

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

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

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

Fig. 17. Renaming the variable to be read (elimination forms).

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

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

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

Shifting to changeable mode. Given a translation of e in the stable mode to some $e^{\mathbb{S}}$, the rules (Write) and (ReadWrite) at the bottom of Figure 16 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, $\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \ldots \rightsquigarrow (x \gg x' : \tau \vdash \mathbf{case}\ x'\ \mathbf{of}\ \ldots)$. 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 17.

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 16 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.

```
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) ⇒ SumLeft(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), 𝕊) ⇒ Mod(trans(e, ℂ))
  | (⊕(x1 : int^ℂ, x2 : int^ℂ), ℂ) ⇒ Read(LPrimop1,
                                         Read(LPrimop2,
                                              Write(trans(⊕(x1', x2'), 𝕊)),
                                              trans(x2,𝕊)),
                                         trans(x1,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 = λδ⃗. if [δ⃗/α⃗]τ'' O.S. then trans(v1, 𝕊)
                  else (if τ' = [δ⃗/α⃗]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))
```

Fig. 18. Translation algorithm.

6.4 Translation type soundness

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

Theorem 6.1 (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* $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{S}} e^{\mathbb{S}}$ *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* $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}} e^{\mathbb{C}}$ *and* $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$.

The proof (Appendix A) 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).

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

6.5 Translation soundness

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

Theorem 6.5 states that if evaluating the translated program e' (in an initially-empty store) yields a (target-language) value w under a new store ρ', then the source program e evaluates to v where v corresponds to $[\rho']w$ (the result of substituting values in the store ρ' for locations appearing in w).[5]

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

Concretely, the back-translation removes all the constructs related to the store: **write**(x) becomes x; **read** expressions are replaced with **let**s; **mod** expressions are replaced with

[5] Our original conference paper attempted to prove this result by defining a relation between source and target terms, and showing that (1) the translation rules produce related terms, and (2) the relation is preserved under evaluation. Unfortunately, as pointed out by one of the journal reviewers, this relation was not well-founded! Fortunately, as we show in this version, we can prove essentially the same result via a clearly well-defined function: the back-translation in Figure 19.

their bodies. The back-translation also removes level superscripts: $\mathbf{apply}^{\varepsilon}$ becomes **apply**, etc. Finally, the back-translation drops instantiations of polymorphic variables, replacing $x[\vec{\alpha}=\vec{\delta}]$ with x, and replaces **select** expressions with the back-translation of a branch. (The translation guarantees that all branches of a **select** will have semantically equivalent back-translations, a property we call **select**-uniformity.) We give the full definition in Figure 19.

$$
\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![e[\vec{\alpha}=\vec{\delta}]]\!] & = & [\![e]\!] \\
[\![\mathbf{apply}^{\varepsilon}(e_1,e_2)]\!] & = & \mathbf{apply}([\![e_1]\!],[\![e_2]\!]) \\
[\![\mathbf{fun}^{\varepsilon}\ f(x)=e]\!] & = & \mathbf{fun}\ f(x)=[\![e]\!] \\
[\![\mathbf{select}\ \{(\vec{\alpha}_i{=}\vec{\delta}_i)\Rightarrow e_i\}_i]\!] & = & [\![e_1]\!] \\
[\![\mathbf{mod}\ e]\!] & = & [\![e]\!] \\
[\![\mathbf{write}(e)]\!] & = & [\![e]\!] \\
[\![\mathbf{read}\ e_1\ \mathbf{as}\ y\ \mathbf{in}\ e_2]\!] & = & \mathbf{let}\ y{=}[\![e_1]\!]\ \mathbf{in}\ [\![e_2]\!] \\
[\![n]\!] & = & n \\
[\![(e_1,e_2)]\!] & = & ([\![e_1]\!],[\![e_2]\!]) \\
[\![\mathbf{fst}\ e]\!] & = & \mathbf{fst}\ [\![e]\!] \\
[\![\mathbf{snd}\ e]\!] & = & \mathbf{snd}\ [\![e]\!] \\
[\![\mathbf{inl}\ e]\!] & = & \mathbf{inl}\ [\![e]\!] \\
[\![\mathbf{inr}\ e]\!] & = & \mathbf{inr}\ [\![e]\!] \\
[\![\mathbf{case}\ e\ \mathbf{of}\ \{x_1\Rightarrow e_1\ ,\ x_2\Rightarrow e_2\}]\!] & = & \mathbf{case}\ [\![e]\!]\ \mathbf{of}\ \{x_1\Rightarrow [\![e_1]\!]\ ,\ x_2\Rightarrow [\![e_2]\!]\} \\
[\![\oplus(e_1,e_2)]\!] & = & \oplus([\![e_1]\!],[\![e_2]\!]) \\
[\![\mathbf{let}\ x{=}e_1\ \mathbf{in}\ e_2]\!] & = & \mathbf{let}\ x{=}[\![e_1]\!]\ \mathbf{in}\ [\![e_2]\!]
\end{array}
$$

Fig. 19. Back-translation, used for correspondence between target and source dynamic semantics

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

Definition 6.2. Source expressions e_1, e_2 are *equivalent*, $e_1 \sim e_2$, iff both evaluate to the same value, or both diverge: if there exists v_1 such that $e_1 \Downarrow v_1$ then $e_2 \Downarrow v_1$, and if there exists v_2 such that $e_2 \Downarrow v_2$ then $e_1 \Downarrow v_2$.

Definition 6.3. Given a store ρ, which maps locations ℓ to target terms $\rho(\ell)$, and a target term e, the *store substitution* operation $[\rho]e$, for all $\ell \in \mathsf{dom}(\rho)$, replaces each occurrence of ℓ in e with $[\rho](\rho(\ell))$.

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

Theorem 6.4 (Evaluation Soundness)**.**
If $\rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FLV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select****-uniform*
then $[\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$.

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

Acar *et al.* (2006b) 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.

6.6 Cost of translated code

Our last main result extends Theorem 6.5, showing that the size $W(\mathscr{D})$ of the derivation of the target-language evaluation $\cdot \vdash e' \Downarrow (\rho \vdash w)$ is asymptotically the same as the size $W(\mathscr{D}')$ of the derivation of the source-language evaluation[6], $[\![e']\!] \Downarrow [\![[\rho]w]\!]$. To prove Theorem 6.14, the extended version of Theorem 6.5, we need a few definitions and intermediate results. The proof hinges on classifying keywords added by the translation, such as **write**, as "dirty": a dirty keyword leads to applications of the dirty rule (TEvWrite) in the evaluation derivation; such applications have no equivalent in the source-language evaluation.

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

The omitted proofs can be found in Appendix C.

To extend the evaluation soundness result above (Theorem 6.4) with a guarantee that the evaluation derivation $\mathscr{D}$ is not too large—within a constant factor of the source evaluation derivation $\mathscr{D}'$—we need several definitions:

Definition 6.6. The *weight* $W(\mathscr{D})$ of a derivation $\mathscr{D}$ is the number of rule applications (that is, the number of horizontal lines) in $\mathscr{D}$.

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

Definition 6.7. Rules (TEvValue), (TEvPair), (TEvSumLeft), (TEvPrimop), (TEvCase), (TEvFst), and (TEvApply) are clean. Rule (TEvLet) is clean, since each **let** in the target expression becomes a **let** in the back-translation. The rules (TEvWrite), (TEvMod), (TEvRead) and (TEvSelect) are dirty.

Definition 6.8. The *head cost* $HC(\mathscr{D})$ of a derivation $\mathscr{D}$ is the number of dirty rule applications reachable from the root of $\mathscr{D}$ without passing through any clean rule applications.

Definition 6.9. The *head cost* $HC(e)$ of a term e is defined in Figure 20.

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

Definition 6.10. 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 $\mathscr{D}$ is shallowly k-bounded if $HC(\mathscr{D}) \leq k$, and deeply k-bounded if all its subderivations are shallowly k-bounded.

$$
\begin{aligned}
HC(x) &= 0 \\
HC(x[\vec{\alpha} = \vec{\delta}]) &= 1 \\
HC((\textbf{select}\ \{\vec{\alpha}_i = \vec{\delta}_i \Rightarrow e_i\}_i)[\vec{\alpha} = \vec{\delta}]) &= 1 + \max_i(HC(e_i)) \\
HC(\textbf{select}\ \{\vec{\alpha}_i = \vec{\delta}_i \Rightarrow e_i\}_i) &= 0 \\
HC(n) &= 0 \\
HC((e_1, e_2)) &= 0 \\
HC(\textbf{inl}\ e) &= 0 \\
HC(\textbf{fun}^{\varepsilon}\ f(x) = e^{\varepsilon}) &= 0 \\
HC(\oplus(e_1, e_2)) &= 0 \\
HC(\textbf{fst}\ e) &= 0 \\
HC(\textbf{apply}^{\varepsilon}(e_1, e_2)) &= 0 \\
HC(\textbf{case}\ e\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}) &= 0 \\
HC(\textbf{let}\ x = e_1^{\mathbb{S}}\ \textbf{in}\ e_2) &= 0 \\
HC(\textbf{mod}\ e^{\mathbb{C}}) &= 1 + HC(e^{\mathbb{C}}) \\
HC(\textbf{write}(e)) &= 1 + HC(e)
\end{aligned}
$$

$$
HC(\textbf{read}\ e_1\ \textbf{as}\ y\ \textbf{in}\ e_2^{\mathbb{C}}) = \begin{cases}
1 + HC(e_1) + HC(e_2^{\mathbb{C}}) & \text{if (for } y \text{ not free in } e_3, e_4\text{):} \\
 & \quad e_2^{\mathbb{C}} \text{ has the form } \textbf{apply}^{\varepsilon}(y, e_3) \\
 & \quad \text{or } \textbf{case}\ y\ \textbf{of}\ \{x_1 \Rightarrow e_3\ ,\ x_2 \Rightarrow e_4\} \\
 & \quad \text{or } \textbf{let}\ r = \oplus(e_3, y)\ \textbf{in write}(r) \\
 & \quad \text{or } \textbf{read}\ e_2'\ \textbf{as}\ y_2\ \textbf{in} \\
 & \qquad \textbf{let}\ r = \oplus(y, y_2)\ \textbf{in write}(r) \\
\text{undefined} & \text{otherwise}
\end{cases}
$$

Fig. 20. Definition of the "head cost" $HC(e)$ of a target expression e.

Theorem 6.11. *If* $\texttt{trans}\ (e, \varepsilon) = e'$ *then* e' *is deeply* 1*-bounded.*

Theorem 6.12 (Cost Result)**.** *Given* $\mathscr{D} :: \rho \vdash e' \Downarrow (\rho' \vdash w)$ *where for every subderivation* $\mathscr{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathscr{D}$ *(including* $\mathscr{D}$*),* $HC(\mathscr{D}^*) \leq k$*, then the number of dirty rule applications in* $\mathscr{D}$ *is at most* $\frac{k}{k+1} W(\mathscr{D})$*.*

The extended soundness result, Theorem 6.14 below, will follow from Theorem 6.13, which generalizes Theorem 6.4, and Theorem 6.12. The parts that differ from the uncosted result (Theorem 6.4) are shaded.

Theorem 6.13 (Costed Evaluation Soundness)**.**
If $\mathscr{D} :: \rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FLV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select****-uniform*
and $[\rho]e$ *is deeply k-bounded*
then $\mathscr{D}' :: [\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$
and $[\rho']w$ *is deeply k-bounded*
and for every subderivation $\mathscr{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathscr{D}$ *(including* $\mathscr{D}$*),*

$HC(\mathscr{D}^*) \leq HC(e^*) \leq k$,
and the number of clean rule applications in $\mathscr{D}$ *equals* $W(\mathscr{D}')$.

Theorem 6.14. *If* $\texttt{trans}\ (e, \varepsilon) = e'$ *and* $\mathscr{D}' :: \cdot \vdash e' \Downarrow (\rho' \vdash w)$, *then* $\mathscr{D} :: [\![e']\!] \Downarrow v$ *where* $W(\mathscr{D}') = \Theta(W(\mathscr{D}))$.

7 Related work

Incremental computation. Self-adjusting computation builds on incremental computation (Ramalingam & Reps 1993). Here we briefly review some techniques. Dependence-graph techniques (Demers *et al.* 1981) enable a change-propagation algorithm to update the computation. Self-adjusting computation performs a generalized form of this dependence tracking for changeable data. Memoization, an idea that dates back to the late 1950s (Bellman 1957), was first used for incremental computation by Pugh & Teitelbaum (1989) and then by many others (e.g., Abadi *et al.* (1996)). Self-adjusting computation uses a form of memoization that allows reuse of computations that themselves can be incrementally updated, also within an imperative computation model.

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.* 2006b, 2009; Ley-Wild *et al.* 2008; Hammer *et al.* 2009). DITTO (Shankar & Bodik 2007) shows the benefits of eliminating user annotations in self-adjusting computation and similar incremental computation techniques. By customizing the computation graph data structures for invariant checking programs, they implemented a fully automatic incremental invariant checker that can speed up invariant checks by an order of magnitude. DITTO, however, is domain-specific, only works for certain programs (e.g., functions cannot return arbitrary values) and is unsound in general.

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

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

algorithm used in these systems is similar to that of self-adjusting computation, but in a less general setting.

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 & Riecke 1998), JFlow (Myers 1999) and a monadic system (Crary *et al.* 2005). Our type system uses many ideas from Pottier & 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 & 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).

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

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

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

Even though functional reactive programming may be viewed as naturally amenable to incremental computation—it should be possible to incorporate time-varying values into a computation by performing an incremental update—most existing research does not employ incremental computation techniques, focusing instead on taming the time and space consumption of the "one-shot", re-computation-based approach. Some recent work took steps in the direction of connecting functional programming and incremental computation (Cooper & Krishnamurthi 2006). In their setting, the compiler transparently lifts computations into the reactive domain, which is similar to marking every value as changeable in our implicit self-adjusting computation. Since this can lead to redundant signals in the runtime system, **?** develop an optimization technique, called "lowering", which rewrites expressions that depend on non-time-varying values into pure computations. This optimization coarsens the dependency graph and eliminates many redundant "stable" signals. Both their system and our system aim to efficiently translate functional programs into a monadic language. The lowering technique starts with programs with all changeable data, and gradually lowers stable subcomputations. In contrast, we start with a pure source program and lift changeable data into modifiables.

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 JFP reviewers for their extensive and careful comments, as well as the anonymous ICFP reviewers, and Arthur Charguéraud, for their useful comments on earlier versions of this paper.

A Proof of translation type soundness

First, we need a few simple lemmas.

Lemma A.1 (Translation of Outer Levels)**.**

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

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

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

Lemma A.2 (Substitution)**.** *Suppose ϕ is a satisfying assignment for C, and $\phi(\vec{\alpha}) = \vec{\delta}$, where $\vec{\alpha} \subseteq FV(C)$.*

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

Proof
By induction on the given derivation. □

Lemma A.3. *Given $\tau' <: \tau''$ and $\tau' \mathrel{\mathring{=}} \tau''$:*

(1) If τ'' O.S. *then $\tau' = \tau''$.*
(2) If τ'' O.C. *then either $\tau' = \tau''$ or $\tau' = \left|\tau''\right|^{\mathbb{S}}$.*

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

- **Case** (subInt): $\tau' = \mathbf{int}^{\delta'}$ and $\tau'' = \mathbf{int}^{\delta''}$, where $\delta' \leq \delta''$.
  (1) If τ'' O.S. then $\delta'' = \mathbb{S}$. So $\tau' = \tau''$.
  (2) If τ'' O.C. then $\delta'' = \mathbb{C}$. If $\delta' = \mathbb{S}$ then $\left|\tau''\right|^{\mathbb{S}} = \mathbf{int}^{\mathbb{S}} = \mathbf{int}^{\delta'} = \tau'$; if $\delta' = \mathbb{C}$ then $\tau'' = \mathbf{int}^{\mathbb{C}} = \mathbf{int}^{\delta'} = \tau'$.
- **Case** (subProd):
  (1) By definition of $\mathring{=}$, $\tau' = \tau''$.
  (2) τ'' O.C. is impossible.
- **Case** (subSum):
  (1) If τ'' O.S. then $\tau'' = (\tau_1'' + \tau_2'')^{\mathbb{S}}$. By inversion on (subSum), $\tau' = (\tau_1' + \tau_2')^{\mathbb{S}}$. By definition of $\mathring{=}$, $\tau_1' = \tau_1''$ and $\tau_2' = \tau_2''$. Therefore $\tau' = \tau''$.
  (2) If τ'' O.C. then $\tau'' = (\tau_1'' + \tau_2'')^{\mathbb{C}}$. By inversion on (subSum), $\tau' = (\tau_1' + \tau_2')^{\delta'}$. By definition of $\mathring{=}$, $\tau_1' = \tau_1''$ and $\tau_2' = \tau_2''$. If $\delta' = \mathbb{S}$ then $\left|\tau''\right|^{\mathbb{S}} = (\tau_1'' + \tau_2'')^{\mathbb{S}}$, which is equal to τ'. If $\delta' = \mathbb{C}$ then $\tau'' = (\tau_1'' + \tau_2'')^{\mathbb{C}} = (\tau_1' + \tau_2')^{\mathbb{C}} = (\tau_1' + \tau_2')^{\delta'} = \tau'$.
- **Case** (subArrow): Similar to the (subSum) case. □

Theorem 6.1 (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 $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{S}} e^{\mathbb{S}}$ 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* $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}} e^{\mathbb{C}}$ *and* $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$.

Proof

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

We present the proof in a line-by-line style, with the justification for each step on the right. Since we need to show that four different judgments are derivable (translation in the $\mathbb{S}$ mode, typing in the $\mathbb{S}$ mode, translation in the $\mathbb{C}$ mode, and typing in the $\mathbb{C}$ mode), and often arrive at some of them early, we indicate them with "☞".

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

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

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

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

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

  $[\phi]\Gamma \vdash n : \mathbf{int}^{\mathbb{S}} \hookrightarrow_{\mathbb{S}} n$ By (Int)

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

  $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} n : \mathbf{int}$ By (TInt)

  $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\mathbf{int}^{\mathbb{S}}\|_\phi$ By $\mathbf{int} = \|\mathbf{int}^{\mathbb{S}}\| = \|[\phi]\mathbf{int}^{\mathbb{S}}\| = \|\mathbf{int}^{\mathbb{S}}\|_\phi$ and $n = e^{\mathbb{S}}$

  ☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_\phi$ By $\tau = \mathbf{int}^{\mathbb{S}}$

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

  $[\phi]\Gamma \vdash n : \mathbf{int}^{\mathbb{S}} \hookrightarrow_{\mathbb{S}} n$ Above

  $[\phi]\Gamma \vdash n : \mathbf{int}^{\mathbb{S}} \hookrightarrow_{\mathbb{C}} \mathbf{let}\ r{=}n\ \mathbf{in}\ \mathbf{write}(r)$ By (Write)

  ☞ $[\phi]\Gamma \vdash e : [\phi](\mathbf{int}^{\mathbb{S}}) \hookrightarrow_{\mathbb{C}} e^{\mathbb{C}}$ $n = e$; def. of subst.; $e^{\mathbb{C}} = \ldots$

  $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} n : \mathbf{int}$ By (TInt)

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

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

  $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r{=}n\ \mathbf{in}\ \mathbf{write}(r) : \mathbf{int}$ By (TLet)

  $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \mathbf{int}$ By def. of $e^{\mathbb{C}}$

  $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\mathbf{int}^{\mathbb{S}}\|_\phi^{\to\mathbb{C}}$ By $\mathbf{int} = \|\mathbf{int}^{\mathbb{S}}\|_\phi^{\to\mathbb{C}}$

  ☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By $\tau = \mathbf{int}^{\mathbb{S}}$

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

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

$\Gamma(x) = \forall\vec{\alpha}[D].\,\tau_0$ Premise

$([\phi]\Gamma)(x) = [\phi](\forall\vec{\alpha}[D].\,\tau_0) = \forall\vec{\alpha}[[\phi]D].\,[\phi]\tau_0$ By def. of substitution

$[\phi]\Gamma \vdash x : [\vec{\delta}/\vec{\alpha}]([\phi]\tau_0) \hookrightarrow_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}]$ By (Var)

$[\phi]\Gamma \vdash x : [\phi][\vec{\delta}/\vec{\alpha}]\tau_0 \hookrightarrow_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}]$ $\vec{\delta}$ closed and $\vec{\alpha} \cap \mathsf{dom}(\phi) = \emptyset$

$[\phi]\Gamma \vdash x : [\phi][\vec{\delta}/\vec{\beta}]([\vec{\beta}/\vec{\alpha}]\tau_0) \hookrightarrow_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}]$ Intermediate subst.

$[\phi]\Gamma \vdash x : [\phi](\underbrace{[\vec{\beta}/\vec{\alpha}]\tau_0}_{\tau}) \hookrightarrow_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}]$ $\phi(\vec{\beta}) = \vec{\delta}$

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{S}} e^{\mathbb{S}}$ and $e^{\mathbb{S}}$ is a value By $e = x$; $\tau = [\vec{\beta}/\vec{\alpha}]\tau_0$; $e^{\mathbb{S}} = x[\vec{\alpha} = \vec{\delta}]$

$(\|\Gamma\|_\phi)(x) = \forall\vec{\alpha}[[\phi]D].\,[\phi]\tau_0$ By def. of $\|-\|_\phi$ and def. of subst.

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} x[\vec{\alpha} = \vec{\delta}] : \|[\vec{\delta}/\vec{\alpha}]([\phi]\tau_0)\|$ By (TVar)

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|[\phi][\vec{\delta}/\vec{\alpha}]\tau_0\|$ $\vec{\alpha} \cap \mathsf{dom}(\phi) = \emptyset$

$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|[\vec{\beta}/\vec{\alpha}]\tau_0\|_\phi$ Intermed. subst., $\phi(\vec{\beta}) = \vec{\delta}$, def. of $\|-\|_\phi$

☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} e^{\mathbb{S}} : \|\tau\|_\phi$ $\tau = [\vec{\beta}/\vec{\alpha}]\tau_0$

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

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

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

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

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

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

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

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

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

$\|\tau\|_\phi = \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By Lemma A.1

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

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

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

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

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

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

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

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

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

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

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

$\|\tau\|_\phi = \|\tau\|_\phi^{\to\mathbb{C}}$ **mod** By Lemma A.1
$\cdot;\|\Gamma\|_\phi \vdash_\mathbb{S} e^\mathbb{S} : \|\tau\|_\phi^{\to\mathbb{C}}$ **mod** By above eqn.

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

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

— Part (1), stable mode translation:

$C;\Gamma \vdash_\varepsilon v_1 : \tau_1$ Subderivation
$[\phi]\Gamma \vdash\ v_1 : [\phi]\tau_1 \hookrightarrow_\mathbb{S} \underline{v_1}$ By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_\mathbb{S} \underline{v_1} : \|\tau_1\|_\phi$ ″

$C;\Gamma \vdash_\varepsilon v_2 : \tau_2$ Subderivation
$[\phi]\Gamma \vdash\ v_2 : [\phi]\tau_2 \hookrightarrow_\mathbb{S} \underline{v_2}$ By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_\mathbb{S} \underline{v_2} : \|\tau_2\|_\phi$ ″

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

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

$\cdot;\|\Gamma\|_\phi \vdash_\mathbb{S} (\underline{v_1}, \underline{v_2}) : \|\tau_1\|_\phi \times \|\tau_2\|_\phi$ By (TPair)
☞ $\cdot;\|\Gamma\|_\phi \vdash_\mathbb{S} e^\mathbb{S} : \|\underbrace{(\tau_1 \times \tau_2)^\mathbb{S}}_{\tau}\|_\phi$ By def. of $\|-\|_\phi$

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

$[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_\mathbb{S} e^\mathbb{S}$ Above
$\underbrace{(\tau_1 \times \tau_2)^\mathbb{S}}_{\tau}$ O.S. By definition of O.S.

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

$\cdot;\|\Gamma\|_\phi \vdash_\mathbb{S} e^\mathbb{S} : \|\tau\|_\phi$ Above
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_\mathbb{S} r : \|\tau\|_\phi$ By (TPVar)
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_\mathbb{C}$ **write**$(r) : \|\tau\|_\phi$ By (TWrite)
$\|\tau\|_\phi = \|\tau\|_\phi^{\to\mathbb{C}}$ By Lemma A.1
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_\mathbb{C}$ **write**$(r) : \|\tau\|_\phi^{\to\mathbb{C}}$ By above equality
☞ $\cdot;\|\Gamma\|_\phi \vdash_\mathbb{C} e^\mathbb{C} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TLet)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(2)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\vec{\mathbb{C}}}$ By (TLet) and Lemma A.1

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

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

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

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

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

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

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

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

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

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

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

Part (1):

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

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

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

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

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

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

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

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

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

— Suppose $[\phi]\delta = \mathbb{S}$.
Part (1):

$C;\Gamma \vdash_\varepsilon x : (\tau_1 \times \tau_2)^\delta$	Subderivation
$[\phi]\Gamma \vdash\ x : ([\phi]\tau_1 \times [\phi]\tau_2)^{\mathbb{S}} \hookrightarrow_{\mathbb{S}} \underline{x}$	By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x} : \|\tau_1\|_\phi \times \|\tau_2\|_\phi$	$''$
☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau_1 \hookrightarrow_{\mathbb{S}} \mathbf{fst}\ \underline{x}$	By (Fst)

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

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

Part (2): Similar to (SVar):

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

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

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

$\ldots, r : \|\tau_1\| \vdash_{\mathbb{S}} r : \|\tau_1\|$	By (TPVar)
$\ldots, r : \|\tau_1\| \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau_1\|$	By (TWrite)
$\ldots, r : \|\tau_1\| \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau_1\|^{\to\mathbb{C}}$	τ_1 O.S.
$\|\Gamma\|, x' : \|\tau_1\| \times \|\tau_2\| \vdash_{\mathbb{S}} \mathbf{fst}\ x' : \|\tau_1\|$	By (TPVar) then (TFst)
$\|\Gamma\|, x' : \|\tau_1\| \times \|\tau_2\| \vdash_{\mathbb{C}} \mathbf{let}\ r$=**fst** x' **in write**$(r) : \|\tau_1\|^{\to\mathbb{C}}$	By (TLet)
$\|\Gamma\| \vdash_{\mathbb{S}} \underline{x} : \|(\tau_1 \times \tau_2)^{\mathbb{C}}\|$	By i.h.
$\|\Gamma\| \vdash_{\mathbb{S}} \underline{x} : (\|\tau_1\| \times \|\tau_2\|)$ **mod**	By def. of $\|-\|$
$\|\Gamma\| \vdash_{\mathbb{C}}$ **read** $\underline{x}$ **as** x' **in let** r=**fst** x' **in write**$(r) : \|\tau_1\|^{\to\mathbb{C}}$	By (TRead)

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

$\ldots, r' : \underline{\tau}_1' \vdash_{\mathbb{C}} \mathbf{write}(r') : \underline{\tau}_1'$	By (TPVar), (TWrite)
$\ldots, r' : \underline{\tau}_1' \vdash_{\mathbb{C}} \mathbf{write}(r') : \|\tau_1\|^{\to\mathbb{C}}$	τ_1 O.C.
$\ldots, r : \underline{\tau}_1'$ **mod** $\vdash_{\mathbb{C}} r : \underline{\tau}_1'$ **mod**	By (TPVar)
$\ldots, r : \underline{\tau}_1'$ **mod** $\vdash_{\mathbb{C}}$ **read** r **as** r' **in write**$(r) : \|\tau_1\|^{\to\mathbb{C}}$	By (TRead)

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

- **Case**

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

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

$C;\Gamma \vdash_{\varepsilon'}$	$e_1 : \tau'$	Subderivation
$[\phi]\Gamma \vdash$	$e_1 : [\phi]\tau' \hookrightarrow_{\mathbb{C}} e^{\mathbb{C}}$	By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}}$	$e^{\mathbb{C}} : \|\tau'\|_\phi^{\to\mathbb{C}}$	$''$

$C \Vdash$	$\tau' <: \tau''$	Premise
	$[\phi]\tau' <: [\phi]\tau''$	By Lemma A.2
$C \Vdash$	$\tau' \mathrel{\mathring{=}} \tau''$	Premise
	$[\phi]\tau' \mathrel{\mathring{=}} [\phi]\tau''$	By Lemma A.2
	$[\phi]\tau''$ O.C.	Subcase (a) assumption
	$[\phi]\tau' = [\phi]\tau''$ or $[\phi]\tau' = \lvert[\phi]\tau''\rvert^{\mathbb{S}}$	By Lemma A.3 (2)

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

	$[\phi]\Gamma \vdash e_1 : [\phi]\tau' \hookrightarrow_{\mathbb{S}} \textbf{mod } e^{\mathbb{C}}$	By (Mod)
	$[\phi]\Gamma \vdash e_1 : [\phi]\tau'' \hookrightarrow_{\mathbb{S}} \textbf{mod } e^{\mathbb{C}}$	By $[\phi]\tau' = [\phi]\tau''$

Otherwise, $[\phi]\tau' = \lvert[\phi]\tau''\rvert^{\mathbb{S}}$.

	$[\phi]\Gamma \vdash e_1 : [\phi]\tau'' \hookrightarrow_{\mathbb{S}} \textbf{mod } e^{\mathbb{C}}$	By (Lift)

Now we have the same judgment no matter which equation Lemma A.3 gave us.

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}}$	$e^{\mathbb{C}} : \|\tau'\|_\phi^{\to\mathbb{C}}$	Above
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$\textbf{mod } e^{\mathbb{C}} : \|\tau'\|_\phi^{\to\mathbb{C}}$ **mod**	By (TMod)
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$\textbf{mod } e^{\mathbb{C}} : \|\lvert\tau''\rvert^{\mathbb{S}}\|_\phi^{\to\mathbb{C}}$ **mod**	
or $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$\textbf{mod } e^{\mathbb{C}} : \|\tau''\|_\phi^{\to\mathbb{C}}$ **mod**	By $\tau' = \tau''$ or $\lvert\tau''\rvert^{\mathbb{S}} = \tau'$
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$\textbf{mod } e^{\mathbb{C}} : \|\tau''\|_\phi^{\to\mathbb{C}}$ **mod**	By def. of $\lvert-\rvert^{\mathbb{S}}$ or copying
	$[\phi]\tau''$ O.C.	Subcase (a) assumption
	$\|\tau''\|_\phi^{\to\mathbb{C}} = \|\tau''\|_\phi$ **mod**	By Lemma A.1
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$\underbrace{\textbf{mod } e^{\mathbb{C}}}_{e^{\mathbb{S}}} : \|\tau''\|_\phi$	By above equation

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

$C;\Gamma \vdash_{\varepsilon'}$	$e_1 : \tau'$	Subderivation
$[\phi]\Gamma \vdash$	$e_1 : [\phi]\tau' \hookrightarrow_{\mathbb{S}} e^{\mathbb{S}}$	By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$e^{\mathbb{S}} : \|\tau'\|_\phi$	$''$
	$[\phi]\tau''$ O.S.	Subcase (b) assumption
	$[\phi]\tau'' = [\phi]\tau'$	By Lemma A.3 (1)
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$	$e^{\mathbb{S}} : \|\tau''\|_\phi$	By above equation

For both subcases, we have:

$C;\Gamma,x:\tau'' \vdash_\varepsilon e_2:\tau$ Subderivation
$[\phi]\Gamma,x:[\phi]\tau'' \vdash e_2:[\phi]\tau \hookrightarrow_{\mathbb{S}} e_2^{\mathbb{S}}$ By i.h. and def. of subst.
$\cdot;\|\Gamma\|_\phi,x:\|\tau''\|_\phi \vdash_{\mathbb{S}} e_2^{\mathbb{S}}:\|\tau\|_\phi$ By i.h. and def. of $\|-\|_\phi$
(1)☞ $[\phi]\Gamma \vdash e:[\phi]\tau \hookrightarrow_{\mathbb{S}} \textbf{let } x=e^{\mathbb{S}} \textbf{ in } e_2^{\mathbb{S}}$ By (LetE)
(1)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \textbf{let } x=e^{\mathbb{S}} \textbf{ in } e_2^{\mathbb{S}}:\|\tau\|_\phi$ By (TLet)

$C;\Gamma,x:\tau'' \vdash_\varepsilon e_2:\tau$ Subderivation
$[\phi]\Gamma,x:[\phi]\tau'' \vdash e_2:[\phi]\tau \hookrightarrow_{\mathbb{C}} e_2^{\mathbb{C}}$ By i.h. and def. of subst.
$\cdot;\|\Gamma\|_\phi,x:\|\tau''\|_\phi \vdash_{\mathbb{C}} e_2^{\mathbb{C}}:\|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By i.h. and def. of $\|-\|_\phi$
(2)☞ $[\phi]\Gamma \vdash e:[\phi]\tau \hookrightarrow_{\mathbb{C}} \textbf{let } x=e^{\mathbb{S}} \textbf{ in } e_2^{\mathbb{C}}$ By (LetE)
(2)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \textbf{let } x=e^{\mathbb{S}} \textbf{ in } e_2^{\mathbb{C}}:\|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By (TLet)

- **Case**

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

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

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

$C\wedge D;\Gamma\vdash_{\mathbb{S}} v_1:\tau'$ Subderivation
$\vec{\alpha}\cap FV(C,\Gamma)=\emptyset$ Premise
$C\wedge D;[\vec{\delta}_i/\vec{\alpha}]\Gamma\vdash_{\mathbb{S}} v_1:[\vec{\delta}_i/\vec{\alpha}]\tau'$ By Lemma A.2
$[\phi]([\vec{\delta}_i/\vec{\alpha}]\Gamma)\vdash v_1:[\phi]([\vec{\delta}_i/\vec{\alpha}]\tau') \hookrightarrow_{\mathbb{S}} \underline{v_i}$ By i.h., using the lemma's guarantee about derivation height
$\vec{\alpha}$ not free in Γ Above disjointness
$[\phi]\Gamma\vdash v_1:[\phi]([\vec{\delta}_i/\vec{\alpha}]\tau') \hookrightarrow_{\mathbb{S}} \underline{v_i}$ By above line
$\cdot;\|\Gamma\|_\phi\vdash_{\mathbb{S}} \underline{v_i}:\|[\vec{\delta}_i/\vec{\alpha}]\tau'\|_\phi$ By i.h. and def. of substitution

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

$C \Vdash \tau' <: \tau''$ Premise
$C \Vdash \tau' \mathrel{\mathring{=}} \tau''$ Premise
$[\phi]\tau' \mathrel{\mathring{=}} [\phi]\tau''$ By Lemma A.2
$[\phi]\tau''$ O.S. Subcase (a) assumption
$[\phi]\tau' = [\phi]\tau''$ By Lemma A.3 (1)

$[\phi]\Gamma \vdash v_1 : [\phi]([\vec{\delta}_i/\vec{\alpha}]\tau') \hookrightarrow_{\mathbb{S}} \underline{v_i}$ Above
$\vec{\alpha} \cup \mathsf{dom}(\phi) = \emptyset$ By $\vec{\alpha} \cap FV(C,\Gamma) = \emptyset$ and appropriateness of ϕ w.r.t. C
$[\phi]\Gamma \vdash v_1 : [\vec{\delta}_i/\vec{\alpha}]([\phi]\tau') \hookrightarrow_{\mathbb{S}} \underline{v_i}$ Property of substitution
$[\phi]\Gamma \vdash v_1 : [\vec{\delta}_i/\vec{\alpha}]([\phi]\tau'') \hookrightarrow_{\mathbb{S}} \underline{e_i}$ By $[\phi]\tau' = [\phi]\tau''$ and $\underline{e_i} = \underline{v_i}$

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta}_i/\vec{\alpha}]\tau'\|_\phi$ Above and $\underline{e_i} = \underline{v_i}$
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\phi][\vec{\delta}_i/\vec{\alpha}]\tau'\|$ Definition of $\|-\|_\phi$

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta}_i/\vec{\alpha}]([\phi]\tau')\|$ By $\vec{\alpha} \cup \mathsf{dom}(\phi) = \emptyset$
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta}_i/\vec{\alpha}]([\phi]\tau'')\|$ By $[\phi]\tau' = [\phi]\tau''$
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\phi]([\vec{\delta}_i/\vec{\alpha}]\tau'')\|$ By $\vec{\alpha} \cup \mathsf{dom}(\phi) = \emptyset$
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta}_i/\vec{\alpha}]\tau''\|_\phi$ Definition of $\|-\|_\phi$

$\cdot;\|\Gamma\|_\phi, \{\underline{y_k} : \|[\vec{\delta}_i/\vec{\alpha}]\tau''\|_\phi\}_k \vdash_{\mathbb{S}} \underline{y_i} : \|[\vec{\delta}_i/\vec{\alpha}]\tau''\|_\phi$ By (TPVar)

End of subcase (a)

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

$C \wedge D; \Gamma \vdash_{\mathbb{S}} v_1 : \tau'$ Subderivation
$\vec{\alpha} \cap FV(C,\Gamma) = \emptyset$ Premise
$C \wedge D; [\vec{\delta}_i/\vec{\alpha}]\Gamma \vdash_{\mathbb{S}} v_1 : [\vec{\delta}_i/\vec{\alpha}]\tau'$ By Lemma A.2
$[\phi]([\vec{\delta}_i/\vec{\alpha}]\Gamma) \vdash v_1 : [\phi]([\vec{\delta}_i/\vec{\alpha}]\tau') \hookrightarrow_{\mathbb{C}} e_i^{\mathbb{C}}$ By i.h., using the lemma's guarantee about derivation height
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e_i^{\mathbb{C}} : \|[\vec{\delta}_i/\vec{\alpha}]\tau'\|_\phi^{\vec{\mathbb{C}}}$ By i.h. and def. of $\|-\|_\phi^{\vec{\mathbb{C}}}$
$[\phi][\vec{\delta}_i/\vec{\alpha}]\tau' <: [\phi][\vec{\delta}_i/\vec{\alpha}]\tau''$ By Lemma A.2
$[\phi][\vec{\delta}_i/\vec{\alpha}]\tau' = \left|[\phi][\vec{\delta}_i/\vec{\alpha}]\tau''\right|^{\mathbb{S}}$ or $\ldots = [\phi][\vec{\delta}_i/\vec{\alpha}]\tau''$ By Lemma A.3 (2)

If $[\phi][\vec{\delta}_i/\vec{\alpha}]\tau' = \left|[\phi][\vec{\delta}_i/\vec{\alpha}]\tau''\right|^{\mathbb{S}}$ then:
$[\phi]\Gamma \vdash v_1 : [\phi][\vec{\delta}_i/\vec{\alpha}]\tau' \hookrightarrow_{\mathbb{C}} e_i^{\mathbb{C}}$ By $\left|[\phi][\vec{\delta}_i/\vec{\alpha}]\tau''\right|^{\mathbb{S}} = [\phi][\vec{\delta}_i/\vec{\alpha}]\tau'$
$[\phi]\Gamma \vdash v_1 : [\phi][\vec{\delta}_i/\vec{\alpha}]\tau'' \hookrightarrow_{\mathbb{S}} \mathbf{mod}\ e_i^{\mathbb{C}}$ By (Lift)

Otherwise, $[\phi][\vec{\delta}_i/\vec{\alpha}]\tau' = [\phi][\vec{\delta}_i/\vec{\alpha}]\tau''$.
$[\phi]\Gamma \vdash v_1 : [\phi][\vec{\delta}_i/\vec{\alpha}]\tau'' \hookrightarrow_{\mathbb{C}} e_i^{\mathbb{C}}$ By $[\phi][\vec{\delta}_i/\vec{\alpha}]\tau'' = [\phi][\vec{\delta}_i/\vec{\alpha}]\tau'$
$[\phi]\Gamma \vdash v_1 : [\phi][\vec{\delta}_i/\vec{\alpha}]\tau'' \hookrightarrow_{\mathbb{S}} \mathbf{mod}\ e_i^{\mathbb{C}}$ By (Mod)

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

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

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta}_i/\vec{\alpha}]\tau'\|_\phi^{\to\mathbb{C}}$ **mod** By (TMod)
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|\left|[\vec{\delta}_i/\vec{\alpha}]\tau''\right|^{\mathbb{S}}\|_\phi^{\to\mathbb{C}}$ **mod** By $\left|[\phi][\vec{\delta}_i/\vec{\alpha}]\tau''\right|^{\mathbb{S}} = [\phi][\vec{\delta}_i/\vec{\alpha}]\tau'$
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : (\|[\vec{\delta}_i/\vec{\alpha}]\tau''\|_\phi^{\to\mathbb{C}})$ **mod** By definition of $\|-\|_\phi^{\to\mathbb{C}}$ (ϕ-shuffling)
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_i} : \|[\vec{\delta}_i/\vec{\alpha}]\tau''\|_\phi$ By Lemma A.1

End of subcase (b)

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

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

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

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

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{e_n} : \|[\vec{\delta}_i/\vec{\alpha}]\tau''\|_\phi$ By extending Γ

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$ **select** $\{\vec{\delta}_i \Rightarrow \underline{e_i}\}_i : \forall\vec{\alpha}[D].\,[\phi]\tau''$ By (TSelect)
$\cdot;\|\Gamma\|_\phi,x:\forall\vec{\alpha}[D].\,\|\tau''\|_\phi \vdash_{\mathbb{S}} e_2^{\mathbb{S}} : \|\tau\|_\phi$ Above

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

- **Case**

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

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

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

Part (1):

$C;\Gamma \vdash_{\mathbb{S}} x_2 : \tau_1$ Subderivation
$[\phi]\Gamma \vdash\ x_2 : [\phi]\tau_1 \hookrightarrow_{\mathbb{S}} \underline{x_2}$ By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_2} : \|\tau_1\|_\phi$ $''$

$C;\Gamma \vdash_{\mathbb{S}} x_1 : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^\delta$ Subderivation

$[\phi]\Gamma \vdash\ x_1 : [\phi]\tau_f \hookrightarrow_{\mathbb{S}} \underline{x_1}$ By i.h.
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|(\tau_1 \xrightarrow[\epsilon']{} \tau)^{\delta}\|_\phi$ ″
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \xrightarrow[[\phi]\epsilon']{} [\phi]\tau)^{[\phi]\delta}\|$ By defs. of $\|-\|_\phi$ and substitution
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \xrightarrow[\mathbb{S}]{} [\phi]\tau)^{\mathbb{S}}\|$ Subcase $\mathbb{S}$-$\mathbb{S}$ assumption
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|\tau_1\|_\phi \xrightarrow[\mathbb{S}]{} \|\tau\|_\phi$ By def. of $\|-\|$

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

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \hookrightarrow_{\mathbb{S}} \mathbf{apply}^{\mathbb{S}}(\underline{x_1}, \underline{x_2})$ By (App)
☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{apply}^{\mathbb{S}}(\underline{x_1}, \underline{x_2}) : \|\tau\|_\phi$ By (TApp)

Part (2):

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

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r)$ By (Write)
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi$ By (TPVar)
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{C}} \mathbf{write}(r) : \|\tau\|_\phi$ By (TWrite)
$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r) : \|\tau\|_\phi$ By (TLet)
$[\phi]\tau$ O.S. Subcase (a) assumption
☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{write}(r) : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By Lemma A.1

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

☞ $[\phi]\Gamma \vdash\ e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r')$ By (ReadWrite)
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi$ By (TPVar)
$[\phi]\tau$ O.C. Subcase (b) assumption
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} r : \|\tau\|_\phi^{\rightarrow\mathbb{C}}\ \mathbf{mod}$ By Lemma A.1
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi, r' : \|\tau\|_\phi^{\rightarrow\mathbb{C}} \vdash_{\mathbb{S}} r' : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By (TPVar)
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi, r' : \|\tau\|_\phi^{\rightarrow\mathbb{C}} \vdash_{\mathbb{C}} \mathbf{write}(r') : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By (TWrite)
$\cdot;\|\Gamma\|_\phi, r : \|\tau\|_\phi \vdash_{\mathbb{S}} \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r') : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By (TRead)
☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{let}\ r = e^{\mathbb{S}}\ \mathbf{in}\ \mathbf{read}\ r\ \mathbf{as}\ r'\ \mathbf{in}\ \mathbf{write}(r') : \|\tau\|_\phi^{\rightarrow\mathbb{C}}$ By (TLet)

— **Subcase** "$\mathbb{C}$-$\mathbb{S}$" where $[\phi]\epsilon' = \mathbb{C}$ and $[\phi]\delta = \mathbb{S}$.

Part (2):

$[\phi]\Gamma \vdash x_1 : [\phi]\tau_f \hookrightarrow_{\mathbb{S}} \underline{x_1}$ From subcase $\mathbb{S}$-$\mathbb{S}$ above
$[\phi]\Gamma \vdash x_2 : [\phi]\tau_1 \hookrightarrow_{\mathbb{S}} \underline{x_2}$ From subcase $\mathbb{S}$-$\mathbb{S}$ above
Let $e^{\mathbb{C}} = \mathbf{apply}^{\mathbb{C}}(\underline{x_1}, \underline{x_2})$.

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \mathbf{apply}^{\mathbb{C}}(\underline{x_1}, \underline{x_2})$ By (App)
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|(\tau_1 \xrightarrow[\varepsilon']{} \tau)^\delta\|_\phi^{\to\mathbb{C}}$ By i.h.
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \xrightarrow[[\phi]\varepsilon']{} [\phi]\tau)^{[\phi]\delta}\|^{\to\mathbb{C}}$ By def. of $\|-\|_\phi^{\to\mathbb{C}}$
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}}\|^{\to\mathbb{C}}$ By subcase $\mathbb{C}$-$\mathbb{S}$ assumption
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|[\phi]\tau_1\| \xrightarrow[\mathbb{C}]{} \|[\phi]\tau\|^{\to\mathbb{C}}$ By def. of $\|-\|^{\to\mathbb{C}}$
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}}$ By def. of $\|-\|_\phi$ and $\|-\|_\phi^{\to\mathbb{C}}$
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_2} : \|\tau_1\|_\phi$ From subcase $\mathbb{S}$-$\mathbb{S}$ above
☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TApp)

Part (1):
$[\phi]\tau$ O.C. By $[\phi]\varepsilon' = \mathbb{C}$ and barring $(\tau_1' \xrightarrow[\mathbb{C}]{} \tau_2')^\delta$ where τ_2' O.S.

$[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}} e^{\mathbb{C}}$ Above
☞ $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{S}} \mathbf{mod}\ e^{\mathbb{C}}$ By (Mod)
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ Above
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}\ \mathbf{mod}$ By (TMod)
$[\phi]\tau$ O.C. Above
☞ $\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \mathbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi$ By Lemma A.1

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

Part (2):

$(\Gamma, x' : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}})(x') = \forall\vec{\alpha}[\mathsf{true}].(\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}}$ By def. of Γ
$C \Vdash \exists\vec{\alpha}.\mathsf{true}$ By def. of $\Vdash$
$C; \Gamma, x' : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}} \vdash_{\mathbb{S}} x' : (\tau_1 \xrightarrow[\varepsilon']{} \tau)^{\mathbb{S}}$ By (SVar)
$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \vdash x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \hookrightarrow_{\mathbb{S}} x'$ By (Var)
$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \vdash x_2 : [\phi]\tau_1 \hookrightarrow_{\mathbb{S}} \underline{x_2}$ By extending Γ
$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{S}} \vdash \mathbf{apply}(x', x_2) : [\phi]\tau \hookrightarrow_{\mathbb{C}} \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$ By (App)
$[\phi]\Gamma, x' : \left|([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}}\right|^{\mathbb{S}} \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$ By defs. of subst., $\left|-\right|^{\mathbb{S}}$

$[\phi]\Gamma \vdash e \rightsquigarrow (x_1 \gg x' : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}} \vdash \mathbf{apply}(x', x_2))$ By (LApply)
$([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}}$ O.C. By def. of O.C.
$C; \Gamma \vdash_{\mathbb{S}} x_1 : \tau_f$ Subderivation
$[\phi]\Gamma \vdash x_1 : ([\phi]\tau_1 \xrightarrow[\mathbb{C}]{} [\phi]\tau)^{\mathbb{C}} \hookrightarrow_{\mathbb{S}} \underline{x_1}$ By i.h.
$\cdot; \|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : (\|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}})\ \mathbf{mod}$ ″

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \mathbf{read}\ \underline{x_1}\ \mathbf{as}\ x'\ \mathbf{in}\ \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$ By (Read)
Let $e^{\mathbb{C}}$ be $\mathbf{read}\ \underline{x_1}\ \mathbf{as}\ x'\ \mathbf{in}\ \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2})$.
$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{S}} x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}}$ By (TPVar)
$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{S}} \underline{x_2} : \|\tau_1\|_\phi$ By extending Γ
$\cdot; \|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{C}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{C}} \mathbf{apply}^{\mathbb{C}}(x', \underline{x_2}) : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TApp)

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

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

Part (1):

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

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

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

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

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

☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}}$ **mod** $e^{\mathbb{C}} : \|\tau\|_\phi$ By reasoning in subcase $\mathbb{C}$-$\mathbb{S}$ for Part (1); note that $[\phi]\tau$ O.C.

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

Part (2):

$[\phi]\Gamma, x' : ([\phi]\tau_1 \xrightarrow[\mathbb{S}]{} [\phi]\tau)^{\mathbb{S}} \vdash$ **apply**$(x', x_2) : [\phi]\tau \hookrightarrow_{\mathbb{C}} e_0^{\mathbb{C}}$ Above with x' for x_1

$[\phi]\Gamma, x' : \left|[\phi]\tau_f\right|^{\mathbb{S}} \vdash [x'/x_1]e : [\phi]\tau \hookrightarrow_{\mathbb{C}} e_0^{\mathbb{C}}$ By defs. of $\left|-\right|^{\mathbb{S}}$, subst.

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

☞ $[\phi]\Gamma \vdash e : [\phi]\tau \hookrightarrow_{\mathbb{C}}$ **read** $\underline{x_1}$ **as** x' **in** $e_0^{\mathbb{C}}$ By (Read)

Let $e^{\mathbb{C}} =$ **read** $\underline{x_1}$ **as** x' **in** $e_0^{\mathbb{C}}$.

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ Above with x' for x_1

$\cdot;\|\Gamma\|_\phi, x' : \|\tau_1\|_\phi \xrightarrow[\mathbb{S}]{} \|\tau\|_\phi^{\to\mathbb{C}} \vdash_{\mathbb{C}} e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By extending Γ

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

☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}}$ **read** $\underline{x_1}$ **as** x' **in** $e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TRead)

Part (1): Similar to Part (1) of subcase $\mathbb{C}$-$\mathbb{C}$.

- **Case**

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

If $[\phi]\delta_1 = [\phi]\delta_2 = \mathbb{S}$ then:

Part (1):

$C;\Gamma \vdash_{\mathbb{S}} x_1 : \mathbf{int}^{\delta_1}$ Subderivation

$[\phi]\Gamma \vdash x_1 : \mathbf{int}^{\delta_1} \hookrightarrow_{\mathbb{S}} \underline{x_1}$ By i.h.

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \|\mathbf{int}^{\delta_1}\|_\phi$ $''$

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_1} : \mathbf{int}$ By $[\phi]\delta_1 = \mathbb{S}$ and def. of $\|-\|$

$[\phi]\Gamma \vdash x_2 : \mathbf{int}^{\delta_2} \hookrightarrow_{\mathbb{S}} \underline{x_2}$ Similar to above

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x_2} : \mathbf{int}$ Similar to above

$[\phi]\Gamma \vdash e : \mathbf{int}^{\mathbb{S}} \hookrightarrow_{\mathbb{S}} \oplus(\underline{x_1}, \underline{x_2})$ By (Prim)

☞ $[\phi]\Gamma \vdash e : [\phi](\mathbf{int}^{\delta_1}) \hookrightarrow_{\mathbb{S}} \oplus(\underline{x_1}, \underline{x_2})$ By $[\phi]\delta_1 = \mathbb{S}$

Let $e^{\mathbb{S}} = \oplus(\underline{x_1}, \underline{x_2})$.

$\vdash \ \oplus : \mathbf{int} \to \mathbf{int}$ Premise

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \oplus(\underline{x_1},\underline{x_2}) : \mathbf{int}$ By (TPrim)

☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \oplus(\underline{x_1},\underline{x_2}) : \|\mathbf{int}^{\delta_1}\|_\phi$ By $[\phi]\delta_1 = \mathbb{S}$ and def. of $\|-\|$

Part (2): Similar to (SPair), where the outer level is stable ($\tau = \mathbf{int}^{\delta_1} = \mathbf{int}^{\mathbb{S}}$).

If $[\phi]\delta_1 = [\phi]\delta_2 = \mathbb{C}$ then:

Part (2):

$[\phi]\Gamma, y_1{:}\mathbf{int}^{\mathbb{S}}, y_2{:}\mathbf{int}^{\mathbb{S}} \vdash \oplus(y_1,y_2) : \mathbf{int}^{\mathbb{C}} \hookrightarrow_{\mathbb{S}} \oplus(y_1,y_2)$ By (Var), (Var), (Prim)

$[\phi]\Gamma, \ldots \vdash \oplus(y_1,y_2) : \mathbf{int}^{\mathbb{C}} \hookrightarrow_{\mathbb{C}} \mathbf{let}\ r = \oplus(y_1,y_2)\ \mathbf{in}\ \mathbf{write}(r)$ By (Write)

$[\phi]\Gamma, y_1{:}\mathbf{int}^{\mathbb{S}} \vdash \oplus(y_1,y_2) : \mathbf{int}^{\mathbb{C}} \hookrightarrow_{\mathbb{C}} \mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in}$ By (LPrimop2)
$\mathbf{let}\ r = \oplus(y_1,y_2)\ \mathbf{in}\ \mathbf{write}(r)$ then (Read)

☞ $[\phi]\Gamma \vdash \oplus(y_1,y_2) : \mathbf{int}^{\mathbb{C}} \hookrightarrow_{\mathbb{C}} \mathbf{read}\ \underline{x_1}\ \mathbf{as}\ y_1\ \mathbf{in}\ \mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in}$ By (LPrimop1)
$\mathbf{let}\ r = \oplus(y_1,y_2)\ \mathbf{in}\ \mathbf{write}(r)$ then (Read)

$\cdot;\|\Gamma\|_\phi, y_1{:}\mathbf{int}, y_2{:}\mathbf{int}, r{:}\mathbf{int} \vdash_{\mathbb{C}} \mathbf{write}(r) : \mathbf{int}$ By (TVar) then (TWrite)

$\cdot;\|\Gamma\|_\phi, y_1{:}\mathbf{int}, y_2{:}\mathbf{int} \vdash_{\mathbb{S}} \oplus(y_1,y_2) : \mathbf{int}$ By (TVar) and (TVar), then (TPrim)

$\cdot;\|\Gamma\|_\phi, y_1{:}\mathbf{int}, y_2{:}\mathbf{int} \vdash_{\mathbb{C}} (\mathbf{let}\ r = \oplus(y_1,y_2)\ \mathbf{in}\ \mathbf{write}(r)) : \mathbf{int}$ By (TLet)

$\cdot;\|\Gamma\|_\phi, y_1{:}\mathbf{int} \vdash_{\mathbb{C}} (\mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in}\ \mathbf{let}\ r = \ldots\ \mathbf{in}\ \mathbf{write}(r)) : \mathbf{int}$ By (TRead)

$\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \mathbf{read}\ \underline{x_1}\ \mathbf{as}\ y_1\ \mathbf{in}\ \mathbf{read}\ \underline{x_2}\ \mathbf{as}\ y_2\ \mathbf{in}\ \mathbf{let}\ r = \oplus(y_1,y_2)\ \mathbf{in}\ \mathbf{write}(r) : \mathbf{int}$ By (TRead)

☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} '' : \|\mathbf{int}^{\delta_1}\|_\phi^{\to\mathbb{C}}$ By def. of $\|-\|^{\to\mathbb{C}}$ and $[\phi]\delta_1 = \mathbb{C}$

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

- **Case**

$$\frac{C;\Gamma \vdash_{\mathbb{S}} x : (\tau_1+\tau_2)^{\delta} \qquad C;\Gamma, x_1:\tau_1 \vdash_{\varepsilon} e_1 : \tau \qquad C \Vdash \delta \le \varepsilon \qquad C;\Gamma, x_2:\tau_2 \vdash_{\varepsilon} e_2 : \tau \qquad C \Vdash \delta \lhd \tau}{C;\Gamma \vdash_{\varepsilon} \underbrace{\mathbf{case}\ x\ \mathbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}}_{e} : \tau}\ (\text{SCase})$$

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

$C;\Gamma \vdash_{\mathbb{S}} x : (\tau_1+\tau_2)^{\delta}$ Subderivation

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

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

$C;\Gamma, x_1 : \tau_1 \vdash_{\varepsilon} e_1 : \tau$ Subderivation

$[\phi]\Gamma, x_1 : [\phi]\tau_1 \vdash \ e_1 : [\phi]\tau \hookrightarrow_{\mathbb{S}} e_1^{\mathbb{S}}$ By i.h.

$\cdot;\|\Gamma\|_\phi, x_1 : \|\tau_1\|_\phi \vdash_{\mathbb{S}} e_1^{\mathbb{S}} : \|\tau\|_\phi$ ″

$[\phi]\Gamma, x_1 : [\phi]\tau_1 \vdash \ e_1 : [\phi]\tau \hookrightarrow_{\mathbb{C}} e_1^{\mathbb{C}}$ ″

$\cdot;\|\Gamma\|_\phi, x_1 : \|\tau_1\|_\phi \vdash_{\mathbb{C}} e_1^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ ″

$C;\Gamma, x_2 : \tau_2 \vdash_{\varepsilon} e_2 : \tau$ Subderivation

$[\phi]\Gamma, x_2 : [\phi]\tau_2 \vdash \ e_2 : [\phi]\tau \hookrightarrow_{\mathbb{S}} e_2^{\mathbb{S}}$ By i.h.

$\cdot;\|\Gamma\|_\phi, x_2 : \|\tau_2\|_\phi \vdash_{\mathbb{S}} e_2^{\mathbb{S}} : \|\tau\|_\phi$ ″

$[\phi]\Gamma, x_2 : [\phi]\tau_2 \vdash \ e_2 : [\phi]\tau \hookrightarrow_{\mathbb{C}} e_2^{\mathbb{C}}$ ″

$\cdot;\|\Gamma\|_\phi, x_2 : \|\tau_2\|_\phi \vdash_{\mathbb{C}} e_2^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ ″

(1)☞ $[\phi]\Gamma \vdash \ e : [\phi]\tau \hookrightarrow_{\mathbb{S}} \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{S}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{S}}\}$ By (Case)
(1)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{S}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{S}}\} : \|\tau\|_\phi$ By (TCase)

(2)☞ $[\phi]\Gamma \vdash \ e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{C}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{C}}\}$ By (Case)
(2)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} \textbf{case}\ \underline{x}\ \textbf{of}\ \{x_1 \Rightarrow e_1^{\mathbb{C}}\ ,\ x_2 \Rightarrow e_2^{\mathbb{C}}\} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TCase)

(b) Suppose $[\phi]\delta = \mathbb{C}$.
$[\phi]\Gamma, x' : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{S}} \vdash \ [x'/x]e : [\phi]\tau \hookrightarrow_{\mathbb{C}} e_0^{\mathbb{C}}$ Above but with x' for the first x
$\cdot;\|\Gamma\|_\phi, x' : \|\tau_1\|_\phi + \|\tau_2\|_\phi \vdash_{\mathbb{C}} e_0^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ ″
$[\phi]\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\}$
$\quad \rightsquigarrow (x \gg x' : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{C}}$
$\quad\quad \textbf{case}\ x'\ \textbf{of}\ \{x_1 \Rightarrow e_1\ ,\ x_2 \Rightarrow e_2\})$ By (LCase)
$\quad\quad\quad ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{C}}\ \mathsf{O.C.}$ By def. of O.C.
$\quad\quad [\phi]\Gamma \vdash \ x : ([\phi]\tau_1 + [\phi]\tau_2)^{\mathbb{C}} \hookrightarrow_{\mathbb{S}} \underline{x}$ By i.h.
$\quad\quad \cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x} : \|(\tau_1 + \tau_2)^{\mathbb{C}}\|_\phi$ ″
$\quad\quad \cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \underline{x} : \left(\|\tau_1\|_\phi + \|\tau_2\|_\phi\right)\ \textbf{mod}$ By def. of $\|-\|_\phi$
(2)☞ $[\phi]\Gamma \vdash \ e : [\phi]\tau \hookrightarrow_{\mathbb{C}} \textbf{read}\ \underline{x}\ \textbf{as}\ x'\ \textbf{in}\ e_0^{\mathbb{C}}$ By (Read)
$\quad$ Let $e^{\mathbb{C}} =\ \textbf{read}\ \underline{x}\ \textbf{as}\ x'\ \textbf{in}\ e_0^{\mathbb{C}}$.
(2)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{C}} e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}$ By (TRead)
$\quad\quad C \Vdash \ \delta \lhd \tau$ Premise
$\quad\quad\quad [\phi]\tau\ \mathsf{O.C.}$ By $[\phi]\delta = \mathbb{C}$ and def. of O.C.
(1)☞ $[\phi]\Gamma \vdash \ e : [\phi]\tau \hookrightarrow_{\mathbb{S}} \textbf{mod}\ e^{\mathbb{C}}$ By (Mod)
$\quad\quad \cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \textbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi^{\to\mathbb{C}}\ \textbf{mod}$ By (TMod)
$\quad\quad\quad \|\tau\|_\phi^{\to\mathbb{C}}\ \textbf{mod} = \|\tau\|_\phi$ By Lemma A.1
(1)☞ $\cdot;\|\Gamma\|_\phi \vdash_{\mathbb{S}} \textbf{mod}\ e^{\mathbb{C}} : \|\tau\|_\phi$ By above equation □

B Proof of translation soundness

In this proof, we use the store substitution operation $[\rho]e$, which replaces locations ℓ with **mod**s of locations' contents (Definition 6.3).

Lemma B.1 (Stores Are Monotonic)**.** *If $\rho_1 \vdash e \Downarrow (\rho_2 \vdash v)$ then there exists ρ' such that $\rho_2 = \rho_1, \rho'$.*

Proof
By induction on the given derivation. All cases are straightforward. □

Lemma B.2 (Commuting)**.** *If e_1 and e are target expressions, then $[\![[e_1/x]e]\!] = [[\![e_1]\!]/x]\,[\![e]\!]$.*

Proof
By induction on e, using the definitions of back-translation and substitution. □

Lemma B.3. *For all closed target values w, the back-translation $[\![w]\!]$ is a (source) value.*

Proof
By induction on w. □

Lemma B.4. *The following equivalences hold:*

(i) $(\textbf{let } x = e_0 \textbf{ in } x) \sim e_0'$, if $e_0 \sim e_0'$
(ii) $(\textbf{let } x' = x_1 \textbf{ in } \textbf{apply}(x', x_2)) \sim \textbf{apply}(x_1, x_2)$
(iii) $(\textbf{let } x' = x_1 \textbf{ in } \oplus(x', x_2)) \sim \oplus(x_1, x_2)$
(iv) $(\textbf{let } x' = x_2 \textbf{ in } \oplus(x_1, x')) \sim \oplus(x_1, x_2)$
(v) $(\textbf{let } x' = x \textbf{ in } \textbf{fst } x') \sim \textbf{fst } x$
(vi) $(\textbf{let } x' = x \textbf{ in case } x' \textbf{ of } \{x_1 \Rightarrow e_1 \ , \ x_2 \Rightarrow e_2\}) \sim \textbf{case } x \textbf{ of } \{x_1 \Rightarrow e_1 \ , \ x_2 \Rightarrow e_2\}$

Proof
Straightforward, using inversion on the given source evaluation derivation, and applying the appropriate evaluation rules. □

Lemma B.5. *If* $\Gamma \vdash e : \tau \hookrightarrow_{\varepsilon} e'$ *then* $[\![e']\!] \sim e$.

Proof
By induction on the given derivation.

For (Apply), and other rules for constructs at which substitution happens during evaluation, we use Lemma B.2.

The other interesting cases are those in which e cannot be *exactly* $[\![e']\!]$: (Read), (Write), and (ReadWrite).

For (Write), $[\![e']\!] = [\![\textbf{let } r = e^{\mathbb{S}} \textbf{ in } \textbf{write}(r)]\!] = \textbf{let } r = [\![e^{\mathbb{S}}]\!] \textbf{ in } [\![\textbf{write}(r)]\!] = \textbf{let } r = [\![e^{\mathbb{S}}]\!] \textbf{ in } r$, but we only have $[\![e^{\mathbb{S}}]\!] \sim e'$ (by i.h.), so we use Lemma B.4 (i).

Rule (ReadWrite) creates two **let**s, so we use the lemma twice.

For (Read), we use Lemma B.4. If the rule from Figure 17 used to derive the first premise was (LApply), we use part (ii) of the lemma; if (LPrimop1), part (iii); if (LPrimop2), part (iv); if (LFst), part (v); if (LCase), part (vi). □

Roughly, we want to show that, if a target expression e' evaluates to some target value w, that the back-translation $[\![e']\!]$, a source expression, evaluates to the back-translation of w (after replacing locations ℓ in w with their corresponding values). This does not hold in general: the back-translation of **select** uses only the *first* arm, under the assumption that all the arms are "essentially" the same. That is, the back-translation assumes the arms differ only in their use of modifiables, and in how they instantiate polymorphic variables. Fortunately, this condition does hold for all target expressions produced by our translation. We call this condition ***select**-uniformity*.

Definition B.6 (Select Uniformity)**.** A target expression e' is ***select**-uniform* if and only if, for all subexpressions of e' of the form $\textbf{select } \{(\vec{\alpha} = \vec{\delta}_1) \Rightarrow e_1, \ldots, (\vec{\alpha} = \vec{\delta}_n) \Rightarrow e_n\}$, all the arms have equivalent back-translations:

$$[\![e_1]\!] \sim [\![e_2]\!] \sim \ldots \sim [\![e_n]\!]$$

Lemma B.7. *If* $\Gamma \vdash e : \tau \hookrightarrow_{\varepsilon} e'$ *then* e' *is* ***select**-uniform.*

Proof
By induction on the given derivation. All cases are completely straightforward, except the case for (LetV), where we use Lemma B.5. □

Theorem 6.4 (Evaluation Soundness)**.**
If $\rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FLV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select****-uniform*
then $[\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$.

Proof
By induction on the given derivation. Wherever we apply the induction hypothesis, it is easy to show the condition of **select**-uniformity; in what follows, we omit this reasoning.

- **Case** $\dfrac{}{\rho \vdash w \Downarrow (\rho \vdash w)}$ (TEvValue)
  Stores only contain values, so $[\rho]w$ is a value. By Lemma B.3, $[\![[\rho]w]\!]$ is some source value v. By (SEvValue), $[\![[\rho]w]\!] \Downarrow [\![[\rho]w]\!]$, which was to be shown.

- **Case** $\dfrac{\begin{array}{l}\rho \vdash e_1 \Downarrow (\rho_1 \vdash \mathbf{fun}^{\varepsilon}\ f(x) = e_0) \\ \rho_1 \vdash e_2 \Downarrow (\rho_2 \vdash w_2) \\ \rho_2 \vdash [(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0 \Downarrow (\rho' \vdash w)\end{array}}{\rho \vdash \mathbf{apply}^{\varepsilon}(e_1, e_2) \Downarrow (\rho' \vdash w)}$ (TEvApply)

  $\rho \vdash e_1 \Downarrow (\rho_1 \vdash \mathbf{fun}^{\varepsilon}\ f(x) = e_0)$ — Subd.
  $[\![[\rho]e_1]\!] \Downarrow [\![[\rho_1](\mathbf{fun}^{\varepsilon}\ f(x) = e_0)]\!]$ — By i.h.
  $[\![[\rho]e_1]\!] \Downarrow [\![\mathbf{fun}^{\varepsilon}\ f(x) = [\rho_1]e_0]\!]$ — By definition of $[-]$
  $[\![[\rho]e_1]\!] \Downarrow \mathbf{fun}\ f(x) = [\![[\rho_1]e_0]\!]$ — By definition of $[\![-]\!]$
  $[\![[\rho]e_1]\!] \Downarrow \mathbf{fun}\ f(x) = [\![[\rho']e_0]\!]$ — Monotonicity

  $\rho_1 \vdash e_2 \Downarrow (\rho_2 \vdash w_2)$ — Subd.
  $[\![[\rho_1]e_2]\!] \Downarrow [\![[\rho_2]w_2]\!]$ — By i.h.
  $[\![[\rho]e_2]\!] \Downarrow [\![[\rho']w_2]\!]$ — $\mathsf{FLV}(e_2) \subseteq \mathsf{dom}(\rho)$ and monotonicity

  $\rho_2 \vdash [(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0 \Downarrow (\rho' \vdash w)$ — Subd.
  $[\![[\rho_2][(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0]\!] \Downarrow [\![[\rho']w]\!]$ — By i.h.
  $[\![[\rho'][(\mathbf{fun}^{\varepsilon}\ f(x) = e_0)/f][w_2/x]e_0]\!] \Downarrow [\![[\rho']w]\!]$ — Monotonicity

  $[(\mathbf{fun}\ f(x) = [\![[\rho']e_0]\!])/f]\ [[\![[\rho']w_2]\!]/x]\ [\![[\rho']e_0]\!] \Downarrow [\![[\rho']w]\!]$ — Properties of substitution, $[-]$, $[\![-]\!]$

  ☞ $\mathbf{apply}([\![[\rho]e_1]\!], [\![[\rho]e_2]\!]) \Downarrow [\![[\rho']w]\!]$ — By (SEvApply)

- **Cases** (TEvPair), (TEvSumLeft), (TEvPrimop), (TEvFst), (TEvCaseLeft): By similar reasoning as in the (TEvApply) case, but simpler.
- **Case** TEvLet: By similar reasoning to the (TEvApply) case, but slightly simpler.
- **Case** $\dfrac{\rho \vdash e^{\mathbb{C}} \Downarrow (\rho_0' \vdash w)}{\rho \vdash \mathbf{mod}\ e^{\mathbb{C}} \Downarrow (\underbrace{(\rho_0', \ell \mapsto w)}_{\rho'} \vdash \ell)}$ (TEvMod)

$\rho \vdash e^{\mathbb{C}} \Downarrow (\rho_0' \vdash w)$ Subd.
$[\![[\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho_0']w]\!]$ By i.h.
$[\![[\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho_0', \ell \mapsto w]\ell]\!]$ By def. of $[-]$
$[\![[\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']\ell]\!]$ $\rho' = \rho_0', \ell \mapsto w$
$[\![\mathbf{mod}\ ([\rho]e^{\mathbb{C}})]\!] \Downarrow [\![[\rho']\ell]\!]$ By def. of $[\![-]\!]$
☞ $[\![[\rho](\mathbf{mod}\ e^{\mathbb{C}})]\!] \Downarrow [\![[\rho']\ell]\!]$ By def. of $[-]$

- **Case**

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

$\rho \vdash e_1 \Downarrow (\rho_1 \vdash \ell)$ Subd.
$[\![[\rho]e_1]\!] \Downarrow [\![[\rho_1]\ell]\!]$ By i.h.
$\rho_1 \vdash [\rho_1(\ell)/x']e^{\mathbb{C}} \Downarrow (\rho' \vdash w)$ Subd.
$[\![[\rho_1][\rho_1(\ell)/x']e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By i.h.
$[\![[[\rho_1]\ell/x'][\rho_1]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By def. of subst.
$[\![[[\rho_1]\ell/x'][\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By $\mathsf{FLV}(e^{\mathbb{C}}) \subseteq \mathsf{dom}(\rho)$ and Lemma B.1
$[[\![[\rho_1]\ell]\!]/x'][\![[\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By Lemma B.2
$\mathbf{let}\ x' = [\![[\rho]e_1]\!]\ \mathbf{in}\ [\![[\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By (SEvLet)
$[\![\mathbf{read}\ [\rho]e_1\ \mathbf{as}\ x'\ \mathbf{in}\ [\rho]e^{\mathbb{C}}]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $[\![-]\!]$
☞ $[\![[\rho](\mathbf{read}\ e_1\ \mathbf{as}\ x'\ \mathbf{in}\ e^{\mathbb{C}})]\!] \Downarrow [\![[\rho']w]\!]$ By def. of subst.

- **Case**

$$\frac{\rho \vdash e_0 \Downarrow (\rho' \vdash w)}{\rho \vdash \mathbf{write}(e_0) \Downarrow (\rho' \vdash w)}\ \text{(TEvWrite)}$$

$\rho \vdash e_0 \Downarrow (\rho' \vdash w)$ Subd.
$[\![[\rho]e_0]\!] \Downarrow [\![[\rho']w]\!]$ By i.h.
$[\![\mathbf{write}([\rho]e_0)]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $[\![-]\!]$
☞ $[\![[\rho]\mathbf{write}(e_0)]\!] \Downarrow [\![[\rho']w]\!]$ By def. of subst.

- **Case**

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

$\rho \vdash e_i \Downarrow (\rho' \vdash w)$ Subd.
$[\![[\rho]e_i]\!] \Downarrow [\![[\rho']w]\!]$ By i.h.
$[\![[\rho]e_1]\!] \sim [\![[\rho]e_i]\!]$ By **select**-uniformity
$[\![[\rho]e_1]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $\sim$
☞ $[\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$ By def. of $[\![-]\!]$ □

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

Proof
By Lemma B.7, e' is **select**-uniform. The empty store $\cdot$ is trivially **select**-uniform. By Theorem 6.4, $[\![[\cdot]e']\!] \Downarrow [\![[\rho']w]\!]$. Since the empty store acts as an identity substitution,

$$[\![e']\!] \Downarrow [\![[\rho']w]\!]$$

By Lemma B.5, $[\![e']\!] \sim e$; by Definition 6.2, $e \Downarrow [\![[\rho']w]\!]$. □

C Proof of costed soundness

Theorem 6.11. *If* `trans` $(e, \varepsilon) = e'$ *then* e' *is deeply* 1*-bounded.*

Proof
By lexicographic induction on e and ε, with $\mathbb{S}$ considered smaller than $\mathbb{C}$.

- If $e \in \{n, x, (v_1, v_2), \textbf{fun} \ldots, \textbf{inl}\ v\}$ then:
  - — If $\varepsilon = \mathbb{S}$ then `trans` uses one of its first 5 cases, and the result follows by induction. ($HC(e') = 0$ except for (Var) where $HC(e') = 1$ is possible.)
  - — If $\varepsilon = \mathbb{C}$ then, for $n/(v_1, v_2)$/**fun/inl** v, `trans` uses its last case and applies (Write). A **let** has head cost 0 (the **let** appears in the back-translation), so $HC(e') = 0$; however, the head cost of the **write** subterm is 1, so the term is deeply 1-bounded. For x, `trans` uses either (Write) or (ReadWrite); in both cases, e' is deeply 1-bounded.
- If e has the form **let** $x = e_1$ **in** e_2, then $HC(e') = 0$; by induction, e'_1 and e'_2 are deeply 1-bounded, so e' is deeply 1-bounded.
- if e has the form $\oplus(x_1, x_2)$, then: For the stable case, e' is a $\oplus$ so $HC(e') = 0$. For the changeable case, `trans` applies (Var), (Var), (Prim), (Write), (Read) with (LPrimop2), and (Read) with (LPrimop1), producing

  $$e' = \textbf{read}\ \underline{x_1}\ \textbf{as}\ y_1\ \textbf{in read}\ \underline{x_2}\ \textbf{as}\ y_2\ \textbf{in let}\ r = \oplus(y_1, y_2)\ \textbf{in write}(r)$$

  so (assuming $HC(\underline{x_1}), HC(\underline{x_2}) \leq 1$) we have $HC(e') = 0$ and all inner head costs bounded by 1.
- If e is an **apply**, then:
  - — Case ($\mathbb{S}$, $\mathbb{S}$, $\mathbb{S}$): Here e' is an $\textbf{apply}^{\mathbb{S}}$, so $HC(e') = 0$.
  - — Case ($\mathbb{C}$, $\mathbb{S}$, $\mathbb{C}$): Here e' is an $\textbf{apply}^{\mathbb{C}}$, so $HC(e') = 0$.
  - — Case ($\mathbb{S}$, $\mathbb{S}$, $\mathbb{C}$): Either (Write) or (ReadWrite), after switching to $\mathbb{S}$ mode, meaning one of the ($-$, $-$, $\mathbb{S}$) cases—which each generate a subterm whose HC is 0. For (Write), the **write** subterm of e' has head cost 1, and likewise for (ReadWrite).
    The rules (LApply) and (LCase) guarantee that the **read** has the correct form for $HC(e')$ to be defined.
  - — Case (ε', $\mathbb{C}$, $\mathbb{C}$): Applies (Read) after devolving to (ε', $\mathbb{S}$, $\mathbb{C}$) which returns a term with $HC(e') \leq 1$ (zero if $\varepsilon' = \mathbb{C}$, and 1 if $\varepsilon' = \mathbb{S}$). Applying (Read) yields a term whose HC is 0, and which is deeply 1-bounded.
    Note that $HC(e')$ is defined for the same reason as in the ($\mathbb{S}$, $\mathbb{S}$, $\mathbb{C}$) subcase.
  - — Case ($\mathbb{C}$, $\mathbb{S}$, $\mathbb{S}$): Devolves to the ($\mathbb{C}$, $\mathbb{S}$, $\mathbb{C}$) case, yielding a subterm with HC of 0; the algorithm then uses (Mod), yielding $HC(e') = 1 + 0 = 1$.
  - — Case (ε', $\mathbb{C}$, $\mathbb{S}$): Devolves to the (ε', $\mathbb{C}$, $\mathbb{C}$) case, where $HC = 0$, then applies (Mod), yielding $HC(e') \leq 1$.

  (Note: We do not use the induction hypothesis as we "devolve"; we are merely reasoning by cases.)

- If $e = \mathbf{fst}\ x$ where $x : (\tau_1 \times \tau_2)^{\delta}$, then:
  - — Case ($\mathbb{S}$, $\mathbb{S}$): We use (Fst), yielding $HC(e') = 0$.
  - — Case ($\mathbb{S}$, $\mathbb{C}$): If τ_1 O.S. then $HC(e') = 0$ (Write). If τ_1 O.C. then we use (Read-Write), which has HC of 0.
  - — Case ($\mathbb{C}$, $\mathbb{C}$): We use (Read) with (LFst) and go to the ($\mathbb{S}$, $\mathbb{C}$) case with a new variable x'. The HC for the ($\mathbb{S}$, $\mathbb{C}$) case is 0. Using (Read) in this case also has head cost 0.
- If e is a **case** on a variable $x : \tau$, then:
  - — If τ is outer stable, the proof is straightforward.
  - — If τ is outer changeable, the algorithm applies rule (Read), recursing with $x : \left|\tau\right|^{\mathbb{S}}$, which will apply rule (Case). A **case** has HC of 0, so (Read) produces e' where $HC(e') = 0$ and e' is deeply 1-bounded. □

In the following proofs, we assume that in any target evaluation $\rho \vdash e' \Downarrow (\rho' \vdash w)$, the target expression e' is closed (that is, it has no free program variables x, though of course it may contain store locations $\ell \in \mathsf{dom}(\rho)$).

Theorem 6.12 (Cost Result)**.** *Given* $\mathscr{D} :: \rho \vdash e' \Downarrow (\rho' \vdash w)$
where for every subderivation $\mathscr{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathscr{D}$ *(including* $\mathscr{D}$*),* $HC(\mathscr{D}^*) \leq k$*, then the number of dirty rule applications in* $\mathscr{D}$ *is at most* $\frac{k}{k+1} W(\mathscr{D})$.

Proof
By the definition of $HC(\mathscr{D})$, if $\mathscr{D}$ is deeply k-bounded, there is no contiguous region of $\mathscr{D}$ consisting only of dirty rule applications that is larger than k; since the only rule with no premises is TEvValue, and TEvValue is clean, at least one of every $k+1$ rule applications is clean. $W(\mathscr{D})$ simply counts the total number of rule applications, so $\mathscr{D}$ contains at least $\frac{W(\mathscr{D})}{k+1}$ clean rule applications, so no more than $\frac{k}{k+1} W(\mathscr{D})$ of $\mathscr{D}$'s rule applications are dirty. □

Theorem 6.13 (Costed Stable Evaluation)**.**
If $\mathscr{D} :: \rho \vdash e \Downarrow (\rho' \vdash w)$ *where* $\mathsf{FV}(e) \subseteq \mathsf{dom}(\rho)$ *and* $[\rho]e$ *is* ***select****-uniform*
and $[\rho]e$ *is deeply* k*-bounded*
then $\mathscr{D}' :: [\![[\rho]e]\!] \Downarrow [\![[\rho']w]\!]$
and $[\rho']w$ *is deeply* k*-bounded*
and for every subderivation $\mathscr{D}^* :: \rho_1^* \vdash e^* \Downarrow (\rho_2^* \vdash w^*)$ *of* $\mathscr{D}$ *(including* $\mathscr{D}$*),*
$HC(\mathscr{D}^*) \leq HC(e^*) \leq k$*,*
and the number of clean rule applications in $\mathscr{D}$ *equals* $W(\mathscr{D}')$.

Proof
The differences from Theorem 6.4 require additional reasoning:

- The 7 cases for the "clean" rules (TEvValue), (TEvPair), (TEvSumLeft), (TEvPrimop), (TEvFst), (TEvCaseLeft), and (TEvApply) are straightforward: the induction hypothesis shows that the HC condition holds for proper subderivations of $\mathscr{D}$, and $HC(\mathscr{D}) = 0$ by definition of $HC(-)$, which is certainly not greater than $HC(e^*)$. Finally, each one of these cases generates a single application of an SEv* rule,

which together with the i.h. satisfies the last condition (that the number of clean rule applications in $\mathscr{D}$ equals $W(\mathscr{D}')$.

For (TEvCaseLeft) and (TEvApply), observe that we are substituting closed values; for all closed values w^* we have $HC(w^*) = 0$, and by i.h. the w^* we substitute are deeply k-bounded, so the result of substitution is deeply k-bounded. (The target expression $x[\vec{\alpha} = \vec{\delta}]$ is not closed, so we need not consider it here.)

Note that this reasoning holds for (TEvValue) even when w is a **select**: (TEvValue) is a clean rule so $HC(\mathscr{D}) = 0$.

- (TEvWrite): We have $\mathscr{D} :: \rho \vdash \mathbf{write}(e_0') \Downarrow (\rho' \vdash w)$ with subderivation $\mathscr{D}_0 :: \rho \vdash e_0' \Downarrow \ldots$. By i.h., $HC(\mathscr{D}_0) \leq HC(\mathbf{write}(e_0'))$. Therefore $HC(\mathscr{D}_0) + 1 \leq HC(e_0')$. By the definitions of HC we have $HC(\mathscr{D}) = HC(\mathscr{D}_0) + 1$ and $HC(\mathbf{write}(e_0')) = 1 + HC(e_0')$, so our inequality becomes $HC(\mathscr{D}) \leq HC(\mathbf{write}(e_0'))$, which was to be shown. Lastly, the $\cdots = W(\mathscr{D}')$ condition from the i.h. is exactly what we need, because the $\mathscr{D}'$ is the same and (TEvWrite) is not clean.
- (TEvMod): Similar to the (TEvWrite) case.
- (TEvLet): According to our definition, $HC(\mathscr{D}) = 0$, and we proceed as with the first 7 cases. Every (TEvLet) in $\mathscr{D}$ corresponds to a (SEvLet) in $\mathscr{D}'$, even if the **let** was created by translation: the theorem concerns the reverse translation $[\![[\rho]e]\!]$, *not* the original source program (which probably has fewer **let**s). (We already assume implicitly that let-expansion preserves asymptotic complexity, because we assume that source programs are in A-normal form, and our goal is to prove an asymptotic equivalence in Theorem 6.14.)
- (TEvRead): For $HC(\mathbf{read}\ldots)$ to be defined, the variable bound is used exactly once and contributes to the HC of the term accordingly, justifying the equation

$$HC([\rho_1(\ell)/x']e^{\mathbb{C}}) = HC(e^{\mathbb{C}}) + HC(\rho_1(\ell))$$

- (TEvSelect):

	$HC(\mathscr{D}_1) \leq HC(e_1)$	i.h.
(L)	$1 + HC(\mathscr{D}_1) \leq 1 + HC(e_1)$	$+1$ each side
(R)	$1 + HC(e_1) \leq HC((\mathbf{select}\ \{\ldots\})[\ldots])$	By def. of $HC(e_1)$; property of max.
	$1 + HC(\mathscr{D}_1) \leq HC(e')$	By (L), (R), transitivity, $e' = (\mathbf{select}\ \{\ldots\})[\ldots]$
☞	$HC(\mathscr{D}) \leq HC(e')$	By def. of $HC(\mathscr{D})$

The $HC(e^*) \leq k$ part of the conclusion is easily shown: in each case, it must be shown for each premise and for the conclusion; the induction hypothesis shows it for the premises, and since we know that $[\rho]e'$ is deeply k-bounded, $HC(e') \leq k$ (applying $[\rho]$ cannot decrease head cost).

Showing that the value w is deeply k-bounded is quite easy. For (TEvValue) it follows from the assumption that $e' = w$ is bounded. For any rule whose conclusion has the same w as one of its premises—(TEvLet), (TEvCaseLeft), (TEvApply), (TEvWrite), (TEvRead), (TEvSelect)—it is immediate by the i.h. In (TEvPair), w_1 and w_2 are bounded by i.h., so (w_1, w_2) is too. The value returned by (TEvSumLeft) and (TEvFst) is a subterm of a value in a premise, which is by i.h. deeply k-bounded, so the subterm is too. (TEvMod) returns ℓ where $\ell \mapsto w$, and w is deeply k-bounded. □

Theorem 6.14. *If* `trans` $(e, \varepsilon) = e'$ *and* $\mathscr{D}' :: \cdot \vdash e' \Downarrow (\rho' \vdash w)$, *then* $\mathscr{D} :: [\![e']\!] \Downarrow v$ *where* $W(\mathscr{D}') = \Theta(W(\mathscr{D}))$.

Proof

By Theorem 6.11, e' is deeply 1-bounded.

The algorithm `trans` merely applies the translation rules, so $\cdot \vdash e : \tau \hookrightarrow_{\varepsilon} e'$. By Theorem 6.13, $\mathscr{D} :: [\![e']\!] \Downarrow v$, and the given derivation $\mathscr{D}'$ and all its subderivations have *HC* bounded by k.

By Theorem 6.12, the number of dirty rule applications in $\mathscr{D}'$ is at most $\frac{k}{k+1} W(\mathscr{D}')$. Each rule application is either clean or dirty, so $W(\mathscr{D}') \leq (k+1) \cdot W(\mathscr{D})$, where $k = 1$. By inspecting the evaluation rules, it is clear that $W(\mathscr{D}') \geq W(\mathscr{D})$. Therefore, $W(\mathscr{D}') = \Theta(W(\mathscr{D}))$. □

References

Martín Abadi, Butler W. Lampson, & Jean-Jacques Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, & Kanat Tangwongsan. A library for self-adjusting computation. *Electronic Notes in Theoretical Computer Science*, 148(2), 2006a.

Umut A. Acar, Guy E. Blelloch, & Robert Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006b.

Umut A. Acar, Alexander Ihler, Ramgopal Mettu, & Özgür Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, & Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–53, 2009.

Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, & Duru Türkoğlu. Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation*, 2010a.

Umut A. Acar, Andrew Cotter, Benoît Hudson, & Duru Türkoğlu. Dynamic well-spaced point sets. In *Symposium on Computational Geometry*, 2010b.

Henk Barendregt, Mario Coppo, & Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

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

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

Yan Chen, Joshua Dunfield, Matthew A. Hammer, & Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *Int'l Conference on Functional Programming (ICFP '11)*, pages 129–141, September 2011.

Yan Chen, Joshua Dunfield, & Umut A. Acar. Type-directed automatic incrementalization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2012.

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

Gregory H. Cooper & Shriram Krishnamurthi. FrTime: Functional Reactive Programming in PLT Scheme. Technical Report CS-03-20, Department of Computer Science, Brown University, April 2004.

Gregory H. Cooper & Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*, 2006.

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

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

Luis Damas & Robin Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212. ACM, 1982.

Alan Demers, Thomas Reps, & Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax-directed editors. In *Principles of Programming Languages*, pages 105–116, 1981.

Camil Demetrescu, Irene Finocchi, & Giuseppe F. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005.

Sofoklis G. Efremidis, John H. Reppy, & Khalid A. Mughal. Attribute grammars in ML. Technical report, 1993.

Conal Elliott & Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

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

Jeffrey S. Foster, Robert Johnson, John Kodumal, & Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Prog. Lang. Sys.*, 28:1035–1087, 2006.

L. Guibas. Modeling motion. In J. Goodman & J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.

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

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

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

Alan Jeffrey. Functional reactive programming with liveness guarantees. In *Proceedings of the 18th ACM International Conference on Functional Programming*, 2013. forthcoming.

Neelakantan R. Krishnaswami & Nick Benton. Ultrametric semantics of reactive programs. In *LICS '11: Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 257–266, 2011. ISBN 978-0-7695-4412-0.

Neelakantan R. Krishnaswami, Nick Benton, & Jan Hoffmann. Higher-order functional reactive programming in bounded space. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 45–58, 2012. ISBN 978-1-4503-1083-3.

Ruy Ley-Wild, Matthew Fluet, & Umut A. Acar. Compiling self-adjusting programs with continuations. In *Int'l Conference on Functional Programming*, 2008.

Ruy Ley-Wild, Umut A. Acar, & Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

Hai Liu & Paul Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, November 2007.

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

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

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

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

Frank Pfenning & Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

François Pottier & Vincent Simonet. Information flow inference for ML. *ACM Trans. Prog. Lang. Sys.*, 25(1):117–158, January 2003.

William Pugh & Tim Teitelbaum. Incremental computation via function caching. In *Principles of Programming Languages*, pages 315–328, 1989.

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

Thomas Reps. *Generating Language-Based Environments*. PhD thesis, Department of Computer Science, Cornell University, August 1982a.

Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 169–176, 1982b.

Thomas Reps & Tim Teitelbaum. *The synthesizer generator: a system for constructing language-based editors*, volume 2. Springer-Verlag, 1989.

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

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

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

Neil Sculthorpe & Henrik Nilsson. Safe functional reactive programming through dependent types. *SIGPLAN Not.*, 44(9):23–34, August 2009. ISSN 0362-1340.

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

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

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, & Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

R. S. Sundaresh & Paul Hudak. Incremental compilation via partial evaluation. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, 1991.

Zhanyong Wan, Walid Taha, & Paul Hudak. Real-time FRP. *SIGPLAN Not.*, 36(10):146–156, 2001.

Zhanyong Wan, Walid Taha, & Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, 2002.