Implicit self-adjusting computation for purely functional programs

YAN CHEN, JOSHUA DUNFIELD, MATTHEW A. HAMMER, UMUT A. ACAR
Max Planck Institute for Software Systems, Gottlieb-Daimler-Str. 49, 67663 Kaiserslautern, Germany
(e-mail: {chenyan,joshua,hammer,umut}@mpi-sws.org)

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

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

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.
Implicit self-adjusting computation for purely functional programs

Level ML $e : \tau$ Evaluation in $k$ steps $v$

Type-directed Translation

Type Soundness

Observational Equivalence

AFL $e' : \tau'$ Evaluation in $\Theta(k)$ steps $w$

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. 2006). We prove three critical properties of the approach:

- **Type soundness.** On source code of a given type, the translation algorithm produces well-typed self-adjusting code of a corresponding target type (Theorem 6.1).
- **Observational equivalence.** The translated self-adjusting program, when evaluated, produces the same value as the source program (Theorem 6.5).
- **Asymptotic complexity.** The time to evaluate the translated program is asymptotically the same as the time to evaluate the source program (Theorem 6.9).

Type soundness and observational equivalence together imply a critical consistency property: that self-adjusting programs respond correctly to changing data (via the consistency of the target self-adjusting language (Acar et al. 2006)). The third property shows that the translated program takes asymptotically as long to evaluate (from scratch) as the corresponding source program. In addition, it places a worst-case bound on the time taken to self-adjust via change propagation, which can and often does take significantly less time when data changes are small. To prove this complexity result, we use a cost semantics (Sands 1990; Sansom & Peyton Jones 1995) that enables precise reasoning about the complexity of the evaluation time. We do not, however, prove tighter bounds on the complexity of self-adjustments; this would be beyond the scope of this paper.

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

This article is an extended version, with full proofs, of a paper that appeared in the Proceedings of the 2011 International Conference on Functional Programming.

2 Overview

We present an informal overview of our approach via examples. 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. 2006). The programmer operates on modifiabls with mod, read, and write constructs to create, read from, and write into modifiabls. The run-time system of a self-adjusting language uses these constructs to represent the execution as a graph, enabling efficient change propagation when the data changes in small amounts.

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

```plaintext
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 unchanged or stable, we assign y the type int mod (of modifiabls 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 $\rightarrow C$:

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

The read operation delineates the code that depends on the changeable value $y$, and the changeable arrow type ensures a critical consistency property: $\rightarrow C$-functions can only be called within the context of a changeable expression. If we change the value of $y$, change propagation can update the result, re-executing only the read and its body, reusing the computation of the square $x2$. 
Suppose we wish to make \( x \) changeable while leaving \( y \) stable. We need to read \( x \) and place \( x^2 \) into a modifiable (because we can only read within the context of a changeable expression), and immediately read back \( x^2 \) and finish by writing the sum.

\[
\text{squareplus\_CS: int mod * int } \rightarrow \text{C int}
\]

\[
\text{fun squareplus\_CS (x, y) =}
\text{let } x' = \text{mod (read } x \text{ as } x' \text{ in write}(x' \times x')) \text{ in}
\text{read } x2' \text{ as } x2' \text{ in}
\text{let } r = x2' + y \text{ in}
\text{write} (r)
\]

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 \text{squareplus\_SC} and \text{squareplus\_CS}—for instance, if we want to pass \text{squareplus} to various higher-order functions—we must write, and maintain, both versions. If we conservatively treat all data as modifiable, we would only need to write one version of each function, but this would introduce unacceptably high overhead. It is also possible to take the other extreme and treat all data as stable, but this would yield a non-self-adjusting program. Our approach treats data as modifiable only where necessary.

### 2.2 Implicit self-adjusting computation

To make self-adjusting computation implicit, we use type information to insert \text{reads}, \text{writes}, and \text{mods} automatically. The user annotates the input type of the program; we infer types for all expressions, and use this information to guide a translation algorithm. The translation algorithm returns well-typed self-adjusting target programs. The translation requires no expression-level annotations. For the example function \text{squareplus} above, we can automatically derive \text{squareplus\_SC} and \text{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 \text{levels S} (stable) and \text{C} (changeable). The type \text{int}^{\delta} is an integer whose level is \( \delta \); for example, to get \text{squaresum\_CS} we can annotate \text{squaresum}'s argument with the type \text{int}^{\text{C} \times \text{int}^{\text{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 \( C \)), and low-security public data (level \( L \)) behaves like stable data (level \( S \)). 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 type system of the target language, which is explicitly self-adjusting, is monomorphic in the levels or changeability, while the implicitly self-adjusting source language is polymorphic over levels. Consequently, translation also needs to monomorphize the source code. Our translation generates code that is well-typed, has the same input-output behavior as the source program, and is, at worst, a constant factor slower than the source program. Since the source and target languages differ, proving these properties is nontrivial; in fact, the proofs critically guided our formulation of the type system and translation algorithm.

**A more detailed example: mapPair.** To illustrate how our translation works, consider a function `mapPair` that takes two integer lists and increments the elements in both lists. This function can be written by applying the standard higher-order `map` over lists. Figure 2 shows the purely functional code in an ML-like language for an implementation of `mapPair`, with a datatype `α list`, an increment function `inc`, and a polymorphic `map` function. Type signatures give the types of functions.

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

• $\alpha$ list$^C$ for lists of $\alpha$ with changeable tails;
• $\alpha$ list$^S$ for lists of $\alpha$ with stable tails.

Then a list of integers allowing insertion and deletion has type int$^S$ list$^C$, and one with unchanging length has type int$^C$ list$^S$. 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 1 we increment stable integers, and in map inc a we increment changeable integers, so the type inferred for inc must be generic: $\forall \delta. \text{int}^\delta \rightarrow \text{int}^\delta$. Our translation produces two implementations of inc, one per instantiation ($\delta=\text{S}$ and $\delta=\text{C}$): 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:

```plaintext
inc : $\forall \delta. \text{int}^\delta \rightarrow \text{int}^\delta$
val inc = select { $\delta=\text{S}$ \Rightarrow inc$_S$ | $\delta=\text{C}$ \Rightarrow inc$_C$ }
```

In mapPair itself, we pass a level instantiation to the selector: inc[$\delta=\text{S}$]. (This instantiation is known statically, so it could be replaced with inc$_S$ at compile time.)

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

**The interplay of type inference and translation.** Given user annotations on the input, type inference finds a satisfying type assignment, which then guides our translation algorithm to produce self-adjusting code. In many cases, multiple type assignments could satisfy the annotations; for example, subsumption allows any stable type to be promoted to a changeable type. Translation yields target code that satisfies the crucial type soundness, operational equivalence, and complexity properties under any satisfying assignment. But some type assignments are preferable, especially when one considers constant factors. Choosing $\text{C}$ 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 $\text{S}$ over $\text{C}$.

3 A type system for implicit SAC

Self-adjusting computation separates the computation and data into two parts: stable and changeable. Changeable data refers to data that can change over time; all non-changeable
Y. Chen et al.

\[
\begin{align*}
\text{datatype } & \alpha\ \text{list}_S = \text{nil} \mid \text{cons of } \alpha \ast \alpha\ \text{list}_S \\
\text{datatype } & \alpha\ \text{list}_C = \text{nil} \mid \text{cons of } \alpha \ast (\alpha\ \text{list}_C) \mod \\
\text{inc}_S : & \text{int} \rightarrow \text{int} \quad (* \text{inc' specialized for stable data} *) \\
\text{fun} & \text{inc}_S (x) = x + 1 \\
\text{inc}_C : & \text{int} \rightarrow \text{int} \quad (* \text{inc' specialized for changeable data} *) \\
\text{fun} & \text{inc}_C (x) = \text{read } x \text{ as } x' \text{ in write } (x' + 1) \\
\text{inc : } & \forall \delta. \text{int}^\delta \rightarrow \text{int}^\delta \\
\text{val} & \text{inc} = \text{select } \{ \delta^S \Rightarrow \text{inc}_S \\
& | \delta^C \Rightarrow \text{inc}_C \} \\
\text{map}_\text{SC} : & (\alpha \rightarrow \delta) \rightarrow (\alpha\ \text{list}_C) \mod \rightarrow (\beta\ \text{list}_C) \mod \\
\text{fun} & \text{map}_\text{SC} f l = \quad (* \text{map' for stable heads, changeable tails } *) \\
& \text{mod } (\text{read } l \text{ as } x \text{ in} \\
& \text{case } x \text{ of} \\
& \text{nil } \Rightarrow \text{write } \text{nil} \\
& | \text{cons}(h, t) \Rightarrow \text{write } (\text{cons}(f h, \text{map}_\text{SC} f t))) \\
\text{map}_\text{CS} : & (\alpha \rightarrow \delta) \rightarrow (\alpha\ \text{list}_S) \rightarrow (\beta\ \text{list}_S) \\
\text{fun} & \text{map}_\text{CS} f l = \quad (* \text{map' for changeable heads, stable tails } *) \\
& \text{case } l \text{ of} \\
& \text{nil } \Rightarrow \text{nil} \\
& | \text{cons}(h, t) \Rightarrow \text{let val } h' = \text{mod } (f h) \\
& \text{in cons}(h', \text{map}_\text{CS} f t) \\
\text{map : } & \forall \delta_h, \delta. (\alpha \rightarrow \beta) \rightarrow (\alpha\ \text{list}_S) \rightarrow (\beta\ \text{list}_C) \mod \\
\text{val} & \text{map} = \text{select } \{ \delta^S \Rightarrow \text{map}_\text{SC} \\
& | \delta^C \Rightarrow \text{map}_\text{CS} \} \\
\text{mapPair : } & ((\text{int}\ \text{list}_C) \mod \ast (\text{int}\ \text{mod})\ \text{list}_S) \\
& \rightarrow ((\text{int}\ \text{list}_C) \mod \ast (\text{int}\ \text{mod})\ \text{list}_S) \\
\text{fun} & \text{mapPair } (l, a) = \quad (* \text{mapPair' with mod types and explicit level polymorphism} *) \\
& \text{map}[\delta^S = S, \delta^C = C] \text{ inc}[\delta^S = l], \\
& \text{map}[\delta^C = C, \delta^S = S] \text{ inc}[\delta^C = a] \\
\end{align*}
\]

**Fig. 4.** Translated \text{mapPair} with \text{mod} types and explicit level polymorphism.

\[
\begin{align*}
\text{Levels } & \quad \delta, \epsilon ::= S \mid C \mid \alpha \\
\text{Types } & \quad \tau ::= \text{int}^\delta \mid (\tau_1 \times \tau_2)^\delta \mid (\tau_1 + \tau_2)^\delta \mid (\tau_1 \rightarrow \tau_2)^\delta \\
\text{Constraints } & \quad C, D ::= \text{true} \mid \text{false} \mid \exists \alpha. \ C \land D \mid \\
& \quad \alpha = \beta \mid \alpha \leq \beta \mid \delta \triangleleft \tau \\
\text{Type schemes } & \quad \sigma ::= \tau \mid \forall \alpha[D], \tau \\
\end{align*}
\]

**Fig. 5.** Levels, constraints, types, and type schemes.
Implicit self-adjusting computation for purely functional programs

Data is stable. Similarly, changeable expressions refer to expressions that operate (via elimination forms) on changeable data; all non-changeable expressions are stable. Evaluation of changeable expressions (that is, changeable computations) can change as the data that they operate on changes: changes in data cause changes in control flow. These distinctions are critical to effective self-adjustment: previous work shows that it suffices to track and remember changeable data and evaluations of changeable expressions because stable data and evaluations of stable expressions remain invariant over time. Previous work therefore presents languages that enable the programmer to separate stable and changeable data, and type systems that enforce correct usage of these constructs.

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

The starting point for our formulation is Pottier & Simonet (2003). Our types include two (security) levels, stable and changeable. We generally follow their approach and notation. The two key differences are that (1) since Level ML is purely functional, we need no “program counter” level “pc”; (2) we need a mode $\varepsilon$ on function types.

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

$$ S \leq S \quad C \leq C \quad S \leq C $$

To support polymorphism and enable type inference, we allow level variables $\alpha$, $\beta$ to appear in types.

**Types.** Types consist of integers tagged with their level, products and sums with an associated level, and arrow (function) types. Function types $(\tau_1 \rightarrow^{\varepsilon} \tau_2)^{\delta}$ carry two level annotations $\varepsilon$ and $\delta$. The *mode* $\varepsilon$ 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 $\delta$ is the level of the function itself, as a value. We say that a type is *ground* if it contains no level variables.

**Subtyping.** Figure 6 shows the subtyping relation $\tau \ll \tau'$, which is standard except for the levels. It requires that the outer level of the subtype is smaller than the outer level of

---

1 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.
and use a constraint solver to find solutions for the variable \( s \). Our constraints:

- To perform type inference, we extend levels with level variables \( \alpha \) and \( \beta \).
- The supertype and that the modes match in the case of functions: a stable-mode function is never a subtype or supertype of a changeable-mode function. (It would be sound to make stable-mode functions subtypes of changeable-mode functions, but changeable mode functions are more expensive; silent coercion would make performance less predictable.)

**Levels and types.** We rely on several relations between levels and types to ascertain various invariants. A type \( \tau \) is *higher than* \( \delta \), written \( \delta \prec \tau \), if the outer level of the type is at least \( \delta \). In other words, \( \delta \) is a lower bound of the outer level(s) of \( \tau \). Figure 7 defines this relation. We distinguish between outer-stable and outer-changeable types (Figure 8).

We write \( \tau \) O.S. if the outer level of \( \tau \) is \( \subseteq \). Similarly, we write \( \tau \) O.C. if the outer level of \( \tau \) is \( \subsetneq \). Finally, two types \( \tau_1 \) and \( \tau_2 \) are *equal up to their outer levels*, written \( \tau_1 \doteq \tau_2 \), if \( \tau_1 = \tau_2 \) or they differ only in their outer levels.

**Constraints.** To perform type inference, we extend levels with level variables \( \alpha \) and \( \beta \), and use a constraint solver to find solutions for the variables. Our constraints \( C, D \) include
Implicit self-adjusting computation for purely functional programs

Values \[ v ::= n \mid x \mid (v_1, v_2) \mid \text{inl} v \mid \text{inr} v \mid \text{fun} f(x) = e \]

Expressions \[ e ::= v \mid \oplus(x_1, x_2) \mid \text{fst} x \mid \text{snd} x \mid \text{case} x \text{ of } \{ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \} \mid \text{apply}(x_1, x_2) \mid \text{let } x = e_1 \text{ in } e_2 \]

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

level-variable comparisons \( \leq \) and level-type comparisons \( \delta \prec \tau \), 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 \( \phi \), substitutes concrete levels \( S \) and \( C \) for level variables. An assignment \( \phi \) 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 \( \phi \). We say that \( C \) entails \( D \), written \( C \models D \), if and only if every assignment \( \phi \) that satisfies \( C \) also satisfies \( D \). We write \( \phi(\alpha) \) for the solution of \( \alpha \) in \( \phi \), and \( [\phi] \tau \) for the usual substitution operation on types. For example, if \( \phi(\alpha) = S \) then \( [\phi] ((\text{int}^{\alpha} + \text{int}^{C})^{\alpha}) = ([\phi] \text{int}^{\alpha} + [\phi] \text{int}^{C})^{S} \).

Type schemes. A type scheme \( \sigma \) is a type with universally quantified level variables: \( \sigma = \forall \vec{\alpha}[D]. \tau \). We say that the variables \( \vec{\alpha} \) are bound by \( \sigma \). 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. \( \text{int}^{C} \) as \( \forall \emptyset[\text{true}].\text{int}^{C} \).

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. We could define types as
\[
\tau ::= \text{int} \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \rightarrow \tau_2
\]

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

Typing takes place in the context of a constraint formula \( C \) and a typing environment \( \Gamma \) that maps variables to type schemes: \( \Gamma ::= \cdot \mid \Gamma, x : \sigma \). The typing judgment \( C; \Gamma \vdash e : \tau \)
Under constraint $C$ and source typing environment $\Gamma$, source expression $e$ has type $\tau$.

$$C;\Gamma \vdash e : \tau$$

(SInt) $\Gamma(x) = \forall \alpha[D], \tau \quad C \vdash \exists \beta. [\beta/\alpha]D$ (SVar)

$$C;\Gamma \vdash e : \tau$$

(SSum) $C;\Gamma \vdash \text{inr} v : (\tau_1 + \tau_2)^\delta$ (SPrim)

$$C;\Gamma \vdash \text{fist} x : \tau_1$$

(SFst) $C;\Gamma \vdash e' : \tau'$

(SPrim) $C;\Gamma \vdash \text{apply}(x_1, x_2) : \tau_2$

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

Fig. 10. Typing rules for Level ML.

has a constraint $C$ and typing environment $\Gamma$, and infers type $\tau$ for expression $e$ in mode $\epsilon$. 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 $C;\Gamma \vdash e : \tau$ as $\Gamma \vdash e : \tau$, read $C \vdash \delta \triangleleft \tau_2$ as $\delta \triangleleft \tau_2$, 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.

No computation happens in values, so they can be typed in either mode. The typing rules for variables (SVar), integers (SInt), pairs (SPair), and sums (SSum) are otherwise standard (we omit the symmetric judgment $\text{inr} v$). 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 $(\text{SFun})$, we type the body in the mode $\varepsilon$ specified by the function type $(\tau_1 \rightarrow \varepsilon \tau_2)^\delta$, and require the result type $\tau_2$ to be higher than the mode, $\varepsilon \triangleleft \tau_2$. As a result, a changeable-mode function must have a changeable return type. This captures the idea that a changeable-mode function is a computation that depends on changeable data, and thus its result must accommodate changes to that data. Primitive operators $\oplus$ take two stable integers and return a stable integer result.

As is common in Damas-Milner-style systems, when typing $\text{let}$ we can generalize variables in types (in our system, level variables) to yield a polymorphic value only when the bound expression is a value. This value restriction is not essential because Level ML is pure, but facilitates adding side effects at a later date. In the first case (SLetE), the expression bound may be a non-value, so we do not generalize and simply type the body in the same mode as the whole $\text{let}$, assuming that the bound expression has the specified type in any mode $\varepsilon^\delta$. 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 $\text{int}^\delta$ to an assumption $x : \text{int}^C$. This simplifies the translation, with no loss of expressiveness: to handle “deep” subsumption, such as $(\text{int}^S \rightarrow \text{int}^S)^S \ll < (\text{int}^S \rightarrow \text{int}^C)^C$, we can insert coercions into the source program before typing it with these rules. (This process could easily be automated.)

In the second case (SLetV), when the expression bound is a value, we type the $\text{let}$ expression in mode $\varepsilon$ by typing the body in the same mode $\varepsilon$, assuming that the value bound is typed in the stable mode (the mode is ignored in the rules typing values). As in (SLetE), we allow subsumption on the bound value only when the types are equal up to their outer level. Because we are binding a value, we generalize its type by quantifying over the type’s free level variables.

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

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

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

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

---

2 In the target language, bound expressions must be stable-mode, but the translation puts changeable bound expressions inside a $\text{mod}$, yielding a stable-mode bound expression.
Fig. 11. Types and expressions in the target language AFL.

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 $C = \ldots \land (\epsilon \triangleleft \tau_2)$. 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]).

4.3 Dynamic semantics

The call-by-value semantics of source programs is defined by a big-step judgment $e \Downarrow v$, read “$e$ evaluates to value $v$”. Our rules in Figure 13 are standard; we write $[v / x] e$ for capture-avoiding substitution of $v$ for the variable $x$ in $e$.

5 Target language

The target language AFL (Figure 11) is a self-adjusting language with modificables. In addition to integers, products, and sums, the target type system makes a modal distinction between ordinary types (e.g. $\text{int}$) and modificable types (e.g. $\text{int mod}$). It also distinguishes
stable-mode and changeable-mode functions. Level polymorphism is supported through an explicit `select` construct and an explicit polymorphic instantiation. In Section 6, we describe how polymorphic source expressions become `select`s in AFL.

The values of the language are integers, variables, polymorphic variable instantiation \( x[\vec{\alpha} = \vec{\delta}] \), locations \( \ell \) (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 \( \{ (\alpha = S) \Rightarrow e_1 \mid (\alpha = C) \Rightarrow e_2 \} \) then \( x[\alpha = S] \) yields \( e_1 \). The symbol \( x \) stands for a bare variable \( x \) or an instantiation \( x[\vec{\alpha} = \vec{\delta}] \).

We distinguish stable expressions \( e^S \) from changeable expressions \( e^C \). Stable expressions create purely functional values; \( \text{apply}^S \) 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 \( \text{apply}^C \) 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 \( x \) 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. Both sets of rules permit expressions that are not in A-normal form, enabling standard capture-avoiding substitution. To simplify the translation, we call instantiations \( x[\vec{\alpha} = \vec{\delta}] \) values, even though \( x[\vec{\alpha} = \vec{\delta}] \) does not evaluate to itself. So we distinguish machine values \( w \)—which do evaluate to themselves—from values \( v \). The only difference is that machine values do not include \( x[\vec{\alpha} = \vec{\delta}] \).

\[
\text{Machine } w ::= n \mid x \mid \ell \mid (w, w) \mid \text{inl } w \mid \text{inr } w \mid \text{fun}^f f(x) = e^f \mid \text{select} \{ (\vec{\alpha}_i = \vec{\delta}_i) \Rightarrow e_{i_i} \}_{i_i}
\]

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,
Under store typing $\Lambda$ and target typing environment $\Gamma$, target value $v$ has type scheme $\sigma$

for all $\tilde{d}$ such that $\tilde{\alpha} = \tilde{d} \parallel D$

$\Lambda; \Gamma \vdash e : \llbracket \tilde{d}/\tilde{\alpha} \rrbracket \tau$ \hspace{1cm} (TSelect)

$\Lambda; \Gamma \vdash \text{select} \{ \tilde{d}_i \Rightarrow e_i \}_{i \in \Pi} \parallel \llbracket D \rrbracket \tau$

Under store typing $\Lambda$ and target typing environment $\Gamma$, target expression $e^f$ has target type $\tau$

$\Lambda(\ell) = \tau$ \hspace{1cm} (TLoc)

$\Lambda; \Gamma \vdash \ell : \tau$ \hspace{1cm} (TInt)

$\Lambda; \Gamma \vdash n : \text{int}$

$\Gamma(x) = \tau$

$\Lambda; \Gamma \vdash x : \tau$ \hspace{1cm} (TPVar)

$\Gamma(x) = \Pi\tilde{\alpha}[D].\tau$

$\Lambda; \Gamma \vdash x[\tilde{\alpha} = \tilde{d}] : \llbracket \tilde{d}/\tilde{\alpha} \rrbracket \tau$ \hspace{1cm} (TVar)

$\Lambda; \Gamma \vdash v_1 : \tilde{\tau}_1$

$\Lambda; \Gamma \vdash v_2 : \tilde{\tau}_2$

$\Lambda; \Gamma \vdash \text{pair}(v_1, v_2) : \tilde{\tau}_1 \times \tilde{\tau}_2$ \hspace{1cm} (TPair)

$\Lambda; \Gamma \vdash \text{fun}^f f(x) = e : (\tau_1 \rightarrow \tau_2)$ \hspace{1cm} (TFun)

$\Lambda; \Gamma \vdash \text{fst} x : \tau_1$ \hspace{1cm} (Tfst)

$\Lambda; \Gamma \vdash \text{fst} x : \tau_1$

$\Lambda; \Gamma \vdash x_1 : \text{int}$

$\Lambda; \Gamma \vdash x_2 : \text{int}$

$\Lambda; \Gamma \vdash x : \text{int} \times \text{int} \rightarrow \text{int}$ \hspace{1cm} (TPrim)

$\Lambda; \Gamma \vdash f(x_1, x_2) : \text{int}$

$\Lambda; \Gamma \vdash e^\sigma_1 : \sigma$

$\Lambda; \Gamma \vdash e^\tau_2 : \tau$

$\Lambda; \Gamma \vdash \text{let} x = e^\sigma_1 \text{ in } e^\tau_2 : \tau'$ \hspace{1cm} (TLet)

$\Lambda; \Gamma \vdash e_2 = e_1$ \hspace{1cm} (TMatch)

$\Lambda; \Gamma \vdash \text{apply}^f(x_1, x_2) : \tilde{\tau}_2$

$\Lambda; \Gamma \vdash x_1 : \tilde{\tau}_1$

$\Lambda; \Gamma \vdash x_2 : \tilde{\tau}_2$

$\Lambda; \Gamma \vdash \text{case } x \text{ of } \{ e_1 \Rightarrow \text{true}, e_2 \Rightarrow \text{false} \} : \tilde{\tau}_3$ \hspace{1cm} (TCase)

$\Lambda; \Gamma \vdash e : \tilde{\tau}_2$

$\Lambda; \Gamma \vdash \text{mod } e : \tilde{\tau}_2 \mod$ \hspace{1cm} (TMod)

$\Lambda; \Gamma \vdash \text{write}(x) : \tilde{\tau}_2$ \hspace{1cm} (TWrite)

$\Lambda; \Gamma \vdash \text{read } x_1 : \tilde{\tau}_1$

$\Lambda; \Gamma \vdash x_1 : \tilde{\tau}_1$

$\Lambda; \Gamma \vdash \text{read } x_1 \text{ as } x \text{ in } e_2 : \tilde{\tau}_2$ \hspace{1cm} (TRead)

Fig. 12. Typing rules of the target language AFL.
Implicit self-adjusting computation for purely functional programs

\[ e \downarrow v \] Source expression \( e \) evaluates to \( v \)

\[
\begin{align*}
\frac{v \downarrow v}{\text{(SEvValue)}} & \quad \frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{(e_1, e_2) \downarrow (v_1, v_2)} \quad \text{(SEPair)}
\end{align*}
\]

\[
\begin{align*}
\frac{e \downarrow v}{\text{(SEvSum)}} & \quad \frac{e_1 \downarrow v_1}{\text{(SEvLet)}} & \quad \frac{e \downarrow \text{inl } v_1}{\text{(SEvCaseLeft)}}
\end{align*}
\]

\[
\begin{align*}
\frac{[\text{let } x = e_1 \text{ in } e_2 \downarrow v_2]}{(\text{SEvPair})} & \quad \frac{(\text{fun } f(x) = e)/[v_2/x]e \downarrow v}{(\text{SEvApply})}
\end{align*}
\]

\[ 17 \]

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

with a corresponding type system (Section 5), the translation is also modal: the translation in the stable mode \( \overset{\Gamma}{\tau} \rightarrow_{\beta} \) produces a stable AFL expression \( e^S \), and the translation in the changeable mode \( \overset{\Gamma}{\tau} \rightarrow_{\varepsilon} \) produces a changeable expression \( e^C \).

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 \( \tau \). 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 \( e' : \tau^' \) where \( \tau^' \) is a target type that corresponds to \( \tau \). 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 \( e^S \) should have when we translate \( e \) in the stable mode, \( e : \tau \overset{\Gamma}{\rightarrow_{\beta}} e^S \). We also use it to translate the types in the environment \( \Gamma \). The second function, \( \| \tau \|^{-C} \), makes sense in two related situations: translating the type \( \tau \) of an expression \( e \) in the changeable mode \( (e : \tau \overset{\Gamma}{\rightarrow_{\varepsilon}} e^C) \) 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 \( \| \text{int}^S \| \) of a stable integer is just \( \text{int} \). In contrast, a changeable integer cannot be inspected or directly created in stable mode, but must be placed into a modifiable: \( \| \text{int}^C \| = \text{int mod} \). The remaining parts of the definition follow this pattern: the target type is wrapped with \( \text{mod} \) if and only if the outer level of the source type is \( \mathbb{C} \). When we translate a changeable-mode function type (with \( \mathbb{C} \) below the arrow), its codomain is translated “output-changeable”: \( \| (\tau_1 \overset{\Gamma}{\rightarrow_{\varepsilon}} \tau_2)^S \| = \| \tau_1 \| \overset{\Gamma}{\rightarrow_{\varepsilon}} \| \tau_2 \|^{-C} \). The reason is that a
In the store \( \rho \), target expression \( e \) evaluates to \( w \) with updated store \( \rho' \):

\[
\begin{align*}
\rho \vdash e \downarrow (\rho' \vdash w) & \quad & (\text{TEvMachineValue}) \\
\rho' \vdash e \downarrow (\rho' \vdash w) & \quad & (\text{TEvSum}) \\
\rho' \vdash \text{let } x = e \text{ in } e_2 \downarrow (\rho' \vdash w) & \quad & (\text{TEvLet}) \\
\rho' \vdash \text{case } e \text{ of } \{ x_1 \Rightarrow e_1, x_2 \Rightarrow e_2 \} \downarrow (\rho' \vdash w) & \quad & (\text{TEvCaseLeft}) \\
\rho' \vdash \text{apply}^C(e_1, e_2) \downarrow (\rho' \vdash w) & \quad & (\text{TEvApply}) \\
\rho' \vdash \text{write}(e) \downarrow (\rho' \vdash w) & \quad & (\text{TEvWrite}) \\
\rho' \vdash \text{read } x' \text{ in } e_2 \downarrow (\rho' \vdash w) & \quad & (\text{TEvRead}) \\
\rho' \vdash \text{select } \{ \ldots \delta \Rightarrow e_1, \ldots \} \downarrow (\rho' \vdash w) & \quad & (\text{TEvSelectE}) \\
\rho' \vdash \text{mod } e_2 \downarrow ((\rho'`, \ell \mapsto w') \vdash \ell) & \quad & (\text{TEvMod})
\end{align*}
\]

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

...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 \texttt{mod} on the codomain would not match the dynamic semantics of AFL.

The second function \( \| \tau \|^-C \) defines the type of a changeable expression \( e \) that writes to a modifiable containing \( \tau \), yielding a changeable target expression \( e^C \). The source type has an outer \( C \), so when the value is written, it will be placed into a modifiable and have \texttt{mod} type. But while evaluating \( e^C \), there is no outer \texttt{mod}. Thus the translation \( \| \tau \|^-C \) ignores the outer level (using the function \( | |^-S \), which replaces an outer level \( C \) with \( S \)), and never returns a type of the form \( (\cdots \texttt{mod}) \). However, since the value being returned may contain subexpressions that will be placed into modifiabes, we use \( | | \) for the inner types. For instance, \( \| (\tau_1 + \tau_2)^\delta \|^-C = \| \tau_1 + \| \tau_2 \| \). 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...
We define the translation of expressions as a set of type-directed rules. Given (1) a derivation of \( C : \Gamma \vdash e : \tau \) in the constraint-based typing system and (2) a satisfying assignment \( \phi \) for \( C \), it is always possible to produce a correctly typed stable target expression \( \tau^S \) and a correctly typed changeable target expression \( \tau^C \) (see Theorem 6.1 below). The environment \( \Gamma \) in the translation rules is a source typing environment, but must have no free level variables. Given an environment \( \Gamma \) from the constraint typing, we apply the satisfying assignment \( \phi \) to eliminate its free level variables before using it for the translation: \( [\phi] \Gamma \).

With the environment closed, we need not refer to \( C \).

Many of the rules in Figure 17 are purely syntax-directed and are similar to the constraint-based rules. One exception is the (Var) rule, which needs the source type to know how to
Shifting to stable mode. Let-bindings merely satisfy the requirements of A-normal form.)

Shifting to changeable mode. Given a translation of \( e \) in the stable mode to some \( \tau \), the rules (Write) and (ReadWrite) at the bottom of Figure 17 translate \( e \) in the changeable mode, producing an \( \tau' \). If the expression’s type \( \tau \) is outer stable (say, \( \text{int}^\beta \)), the (Write) rule simply binds it to a variable and then writes that variable. If \( \tau \) is outer changeable (say, \( \text{int}^C \)) it will be in a modifiable at runtime, so we read it into \( \tau' \) and then write it. (The let-bindings merely satisfy the requirements of A-normal form.)

Shifting to stable mode. To generate a stable expression \( e^\beta \) based on a changeable expression \( e^C \), we have the (Lift) and (Mod) rules. These rules require the source type \( \tau \) to be outer changeable: in (Lift), the premise \( |\tau|^0 = \tau' \) requires that \( |\tau|^\beta \) is defined, and it is defined only for outer changeable \( \tau \); in (Mod), the requirement is explicit: \( \vdash \tau \ O.C. \)

(Mod) is the simpler of the two: if \( e \) translates to \( e^C \) at type \( \tau \), then \( e \) translates to the stable expression mod \( e^C \) at type \( \tau \). In (Lift), the expression is translated not at the given type \( \tau \) but at its stabilized \( |\tau|^S \), capturing the “shallow subsumption” in the constraint typing rules (SLetE) and (SLetV): a bound expression of type \( \tau^S_0 \) can be translated at type \( \tau^S \) to \( e^S \), and then “promoted” to type \( \tau^S_0 \) by placing it inside a mod.
Implicit self-adjusting computation for purely functional programs

Under closed source typing environment \( \Gamma \), source expression \( e \) is translated at type \( \tau \) in mode \( \varepsilon \) to target expression \( e^\varepsilon \)

\[
\Gamma \vdash e : \tau \xrightarrow{\varepsilon} e^\varepsilon
\]

Reading from changeable data. To use an expression of changeable type in a context where a stable value is needed—such as passing some \( x : \text{int}^C \) to a function expecting \( \text{int}^S \)—the (Read) rule generates a target expression that reads the value out of \( x : \text{int}^C \) into a variable \( x' : \text{int}^S \). The variable-renaming judgment \( \Gamma \vdash e \mapsto (x \gg x' : \tau \vdash e') \) takes the expression \( e \), finds a variable \( x \) about to be used, and yields an expression \( e' \) with that oc-
currence replaced by \( \sigma \). For example, \( \Gamma \vdash \text{case } x \text{ of } \ldots \rightarrow (x \gg \sigma : \tau \vdash \text{case } x' \text{ of } \ldots) \). This judgment is derivable only for \text{apply}, \text{case}, \text{fst}, \text{and } \odot, \text{because these are the elimination forms for outer-changeable data. For } \odot(x_1, x_2), \text{we need to read both variables, so we have one rule for each. The rules are given in Figure 16.}

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

Since (LetV) generates one function for each satisfying \( \delta_i \), 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 \( \text{are} \) used would have been handwritten in an explicit setting, so while the code size is exponential in the worst case, the saved effort is as well.

### 6.3 Algorithm

The system of translation rules in Figure 17 is not deterministic. In fact, if the wrong choices are made it can produce painfully inefficient code. Suppose we have \( 2 : \text{int}^C \), 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 \( e^C \) with a \text{let}, a \text{read} and a \text{write}—then (Mod), which wraps that \( e^C \) in a \text{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 \( \varepsilon \).

### 6.4 Translation type soundness

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

**Theorem 6.1 (Translation Type Soundness).**

If \( C; \Gamma \vdash e : \tau \) and \( \phi \) is a satisfying assignment for \( C \) then

1. there exists \( e^S \) such that \( \Gamma \vdash e : [\phi] \tau \rightleftarrows e^S \) and \( : : \Gamma \parallel_\phi \vdash e^S : [] \tau \parallel_\phi \) and, if \( e \) is a value, then \( e^S \) is a value;

2. there exists \( e^C \) such that \( \Gamma \vdash e : [\phi] \tau \rightleftarrows e^C \) and \( : : \Gamma \parallel_\phi \vdash e^C : [] \tau \parallel_\phi \).

The proof (Appendix A) is by induction on the height of the given derivation of \( C; \Gamma \vdash e : \tau \). If the concluding rule was (SLetE), we use a substitution property (Lemma A.2) for each \( \delta_i \) to get a monomorphic constraint typing derivation; that derivation is not larger than
Implicit self-adjusting computation for purely functional programs

Fig. 18. Translation algorithm.

```haskell
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 → τ₂)^δ, S) ⇒ Fun(trans(e', ε'))
| (inl v, S) ⇒ Sum(trans(v, S))
| (fst x: (τ₁ × τ₂)^δ, ε) ⇒ case (δ, ε) of
  | (S, S) ⇒ Fst(trans(x, S))
  | (S, C) ⇒ if τ₁ O.S. then Write(trans(e, S))
    |     else ReadWrite(trans(e, S))
  | (C, C) ⇒ Read(LFst, trans(fst x: (τ₁ × τ₂)^δ), trans(x, S))
| (get(x₁: int^δ, x₂: int^δ), S) ⇒ Prim(trans(x₁, S), trans(x₂, S))
| (get(x₁: int^δ, x₂: int^δ), C) ⇒ Write(trans(e, S)))
| (get(x₁: int^δ, x₂: int^δ), C) ⇒ Read(LPrim1, Read(LPrimop2, Write(trans(∥x₁,x₂∥, S))))
| (let x: τ'' = e₁: τ' in e₂, ε) ⇒
  LetE(if τ'' O.S. then trans(e₁, S)
    |     else if τ' = τ'' then Mod(trans(e₁, C))
    |     else Lift(trans(e₁, C))), trans(e₂, ε))
| (let x: ∀α[D]. τ'' = v₁: τ' in e₂, ε) ⇒
  let variants = all δ such that Δ = δ ⊢ D in
  let f = λx. if τ'' O.S. then trans(v₁, S)
    |     else if τ' = τ'' then Mod(trans(v₁, C))
    |     else Lift(trans(v₁, C))) in
  LetV(map f variants, trans(e₂, ε))
| (apply(x₁: (τ₁ → τ₂)^δ, x₂, ε) ⇒ case (τ', 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))
  | (τ', C, C) ⇒ Read(LApply, trans(apply(x': (τ₁ → τ₂)^δ, x₂), C), trans(x₁, S))
  | (C, S, S) ⇒ Mod(trans(e, C))
  | (τ', 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]^δ 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))
```

the input derivation, so we can apply the induction hypothesis to get a translated \( e' \). The proof constructs the same translation derivations as the algorithm in Figure 18 (in fact, we extracted the algorithm from the proof).

### 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 in an initially empty store \( \rho \), if evaluating the translated program \( e' \) yields \( v' \) with new store \( \rho' \), then \( e \) evaluates to \( v \) where \( v \) corresponds to \( [\rho']v' \) (the result of substituting values in the store \( \rho' \) for locations appearing in \( v' \)).

To define this correspondence, we use a device somewhat similar to logical relations: a relation \( \equiv \) on source and target expressions, allowing us to show that if \( e : \tau \equiv e' : \tau' \) then \( v : \tau \equiv [\rho]v' : \tau' \). Both \( e \) and \( e' \) must be closed. Our definition is weaker than the equivalence relations used in logical relations proofs: \( \text{apply}(\text{id}, 4) \equiv 4 \), for example. It does not attempt to equate all programs that have the same meaning, but only particular Level ML terms to AFL terms that are similarly structured, but have overhead (\( \text{mod} \), \( \text{write} \), etc.). Thus, integers are related to integers, pairs are related if their components are related, and so forth. The definition essentially ignores \( \text{mod} \) and \( \text{write} \) and ignores the mode in \( \text{apply}^e \). Since translated programs can have “extra” \( \text{read} \) and \( \text{let} \) expressions, these are “substituted out” in the relation, so that \( 3 \equiv \text{let} \ x = 3 \ in \ x \). Functions are related if, given related arguments, they produce related results.

Note that we will not induct over this relation; neither the term, nor the type, gets smaller.

We also relate substitutions: a source substitution \( s \) and a target substitution \( s' \),

\[
\begin{align*}
s &= v_1/x_1, \ldots, v_n/x_n \\
\bar{s} &= w_1/x_1, \ldots, w_n/x_n
\end{align*}
\]

are related at their environments \( \Gamma = \tau_1, \ldots, \tau_n : \tau_{\text{s}} \) and \( \Gamma' = \tau_1', \ldots, \tau_n' : \tau_{\text{s}'} \), written

\[
(v_1/x_1, \ldots, v_n/x_n) : (x_1 : \tau_1, \ldots, x_n : \tau_n) \equiv (w_1/x_1, \ldots, w_n/x_n) : (x_1 : \tau_1', \ldots, x_n : \tau_n')
\]

if, for all \( k \) from 1 to \( n \), we have \( v_k : \tau_k \equiv w_k : \tau_k \).

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

We begin by defining a store substitution operation:

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

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

**Theorem 6.3** (Generalized Translation Soundness).

If \( e : \sigma \equiv [\rho]e' : \sigma' \) and \( D :: \rho \vdash e' \Downarrow (\rho' \Downarrow v) \) then \( D' :: v : \Downarrow v \) where \( v : \sigma \equiv [\rho']w : \sigma' \).

**Proof**

By induction on the derivation of \( \rho \vdash e' \Downarrow (\rho' \Downarrow v) \). See Appendix B.
Let $\tau$ apply $\tau \cdot (\text{fun } e) : \tau' \cdot \tau$ and $\tau$ apply $\tau' \cdot (\text{case } e \text{ of } (x_1 \Rightarrow e_1, x_2 \Rightarrow e_2)) : \tau$.

**Lemma 6.4 (Relation of Translation).**

If $\Gamma \vdash e : \tau \Downarrow \epsilon' \cdot \cdot \vdash s : \Gamma$ and $\cdot \vdash s : \Gamma$ and $\cdot \vdash \emptyset : \|\Gamma\|$ and $s : \Gamma \Rightarrow \emptyset : \|\Gamma\|$ then

$$[s]e : \tau \Rightarrow [s]e' : \tau'$$

where $\tau' = \|\tau\|$ if $\epsilon = \emptyset$, and $\tau' = \|\tau\|^C$ if $\epsilon = C$.

**Proof**

See Appendix B.

**Theorem 6.5 (Translation Soundness).**

If $\cdot \vdash e : \tau \Downarrow \epsilon' \cdot \cdot \vdash e' \Downarrow (\rho' \Downarrow w)$, then $e \Downarrow v$

where $v : \tau \Rightarrow \rho' \Rightarrow \rho^Cw : \tau'$.

**Proof**

Let $\Gamma = \cdot$ and let $s$ and $s'$ be empty substitutions. By Lemma 6.4, $[s]e : \tau \Rightarrow [s]e' : \tau'$. Because $s$ and $s'$ are empty, we have $e : \tau \Rightarrow e' : \tau'$. 

$$\gamma : \sigma \Rightarrow \gamma' : \Gamma$$

Source expression $e$ at type [schema] $\sigma$ is related to target expression $e'$ at type [schema] $\sigma'$.
Let $\rho = \cdot$ (the empty store). By Theorem 6.3, $e \Downarrow v$ and $v : \tau \vdash [\rho']w : \tau'$.

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

### 6.6 Cost of translated code

Our last main result extends Theorem 6.5, showing that the size $W(D)$ of the derivation of the target-language evaluation is within a constant factor of the size $W(D')$ of the derivation of the source-language evaluation $e \Downarrow v$. 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 $\beta$-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.

**Definition 6.6.** A term $e$ is shallowly $k$-bounded if $HC(e) \leq k$. A term $e$ is deeply $k$-bounded if every subterm of $e$ (including $e$ itself) is shallowly $k$-bounded. Similarly, a derivation $D$ is shallowly $k$-bounded if $HC(D) \leq k$, and deeply $k$-bounded if all its subderivations are shallowly $k$-bounded.

**Theorem 6.7.** If $\text{trans}(e, \varepsilon) = e'$ then $e'$ is deeply 6-bounded.

**Theorem 6.8 (Cost Result).** Given $D :: \rho \vdash e' \Downarrow (\rho' \vdash w)$ where for every subderivation $D^* :: \rho^+_3 \vdash e^* \Downarrow (\rho^+_4 \vdash w^*)$ of $D$ (including $D$), $HC(D^*) \leq k$, then the number of dirty rule applications in $D$ is at most $\frac{k}{k+1} W(D)$.

Theorem 6.9 follows from Theorem 6.14—a generalization of Theorem 6.3—and Theorem 6.8.

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

To extend the result above with a guarantee that the evaluation derivation $D$ is not too large—within a constant factor of the source evaluation derivation $D'$—we need several definitions:

**Definition 6.10.** The weight $W(D)$ of a derivation $D$ is the number of rule applications (that is, the number of horizontal lines) in $D$.

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.
Theorem 6.14 (Costed Generalized Translation Soundness)

If all its subderivations are shallowly $k$-bounded.

Definition 6.11. Rules (TEvMachineValue), (TEvPair), (TEvSum), (TEvPrimop), (TEvCase), (TEvFst), and (TEvApply) are clean. The rule (TEvLet) is clean iff it is applied to a let generated by (LetE) or (LetV), and dirty otherwise. The rules (TEvWrite), (TEvMod), (TEvRead) and (TEvSelectE) are dirty.

Definition 6.12. The head cost $HC(D)$ of a derivation $D$ is the number of dirty rule applications reachable from the root of $D$ without passing through any clean rule applications.

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

\[
HC(x) = 0 \\
HC(x_{\bar{a} = \bar{\delta}}) = 1 \\
HC((\text{select } \{ \bar{a}_i = \bar{\delta}_i \Rightarrow e_i \})_{\bar{a}} = \bar{\delta}) = 1 + \max_i(HC(e_i)) \\
HC(\text{select } \{ \bar{a}_i = \bar{\delta}_i \Rightarrow e_i \}) = 0 \\
HC(n) = 0 \\
HC(\text{let } e_0, e_1) = 0 \\
HC(\text{let } e_0, e_1) = 0 \\
HC(\text{fun } f(x) = e') = 0 \\
HC(\text{mod } e^c) = 0 \\
HC(\text{write } e) = 0 \\
HC(\text{case } e \text{ of } \{ x_1 \Rightarrow e_1, x_2 \Rightarrow e_2 \}) = 0 \\
HC(\text{let } x = e^c_1 \text{ in } e_2) = \\
\begin{cases} 
0 & \text{if the let came from (LetE)/(LetV)} \\
1 + HC(e^c_1) + HC(e_2) & \text{if the let came from (Write)/(ReadWrite)} \\
\text{undefined} & \text{otherwise} \\
\end{cases}
\]

\[
HC(\text{mod } e^c) = 1 + HC(e^c) \\
HC(\text{write } e) = 1 + HC(e) \\
HC(\text{read } e_1 \text{ as } y \text{ in } e^c_2) = \\
\begin{cases} 
1 + HC(e_1) + HC(e^c_2) & \text{if (for } y \text{ not free in } e_3, e_4; \\
\text{e}^c_2 \text{ has the form } \text{apply}(y, e_3) & \\
or \text{case } y \text{ of } \{ x_1 \Rightarrow e_3, x_2 \Rightarrow e_4 \} \\
or \text{let } r = \oplus(e_3, y) \text{ in write}(r) & \\
or \text{read } e^c_2 \text{ as } y_2 \text{ in } \text{let } r = \oplus(y, y_2) \text{ in write}(r) & \\
\text{undefined} & \text{otherwise} \\
\end{cases}
\]

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

Definition 6.13. 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 $D$ is shallowly $k$-bounded if $HC(D) \leq k$, and deeply $k$-bounded if all its subderivations are shallowly $k$-bounded.

Theorem 6.14 (Costed Generalized Translation Soundness).

If $e : \sigma \vdash [p]e' : \sigma'$ and $D : \rho \vdash e' \Downarrow (p' \Downarrow w)$

and $[p]e'$ is deeply $k$-bounded

then $D' : e \Downarrow v$ where $v : \sigma \vdash [p']w : \sigma'$.
\[ [\rho']_w \text{ is deeply } k\text{-bounded} \]

\[ \text{and for every subderivation } \mathcal{D}^* :: \rho^*_1 \vdash e^* \Downarrow (\rho^*_2 \vdash w^*) \text{ of } \mathcal{D} \text{ (including } \mathcal{D}) \text{, } HC(\mathcal{D}^*) \leq HC(e^*) \leq k, \]

\[ \text{and the number of clean rule applications in } \mathcal{D} \text{ equals } W(\mathcal{D}). \]

7 Related work

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

Information flow and constraint-based type inference. A number of information flow type systems have been developed to check security properties, including the SLam calculus (Heintze & 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).

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 compil-
ers that can generate code that can respond automatically to dynamically changing data cor-
rectly and asymptotically optimally, without substantial programming effort. Remaining
open problems include generalization to imperative programs with references, techniques
and proofs to determine or improve the asymptotic complexity of dynamic responses, and
a complete and careful implementation and its evaluation.

Acknowledgments

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

A Proof of translation type soundness

As described in the main paper, we define a relation ⊢ on source and target expressions.
This relation essentially ignores mod and write, and substitutes out in read and let expres-
sions. We also relate substitutions, and define a store substitution that replaces locations ℓ
with mods of locations’ contents.

First, we need a few simple lemmas.

Lemma A.1 (Translation of Outer Levels).

\[\phi \vdash \tau \mathcal{O} \text{ C} \iff \|\tau\|_\phi = \|\tau\|_\phi \mathcal{O} \text{ C};\]

\[\phi \vdash \tau \mathcal{O} \text{ S} \iff \|\tau\|_\phi = \|\tau\|_\phi \mathcal{O} \text{ C}.\]

Proof

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

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

1. If \(\mathcal{D}\) derives \(C;\Gamma \vdash e : \tau\), then there exists \(\mathcal{D}'\) deriving \(C;[\bar{\delta}/\bar{\alpha}]\Gamma \vdash e : [\bar{\delta}/\bar{\alpha}]\tau\),
where \(\mathcal{D}'\) has the same height as \(\mathcal{D}\).

2. If \(C \vdash \delta' <\tau\), then \(C \vdash [\bar{\delta}/\bar{\alpha}]\delta' < [\bar{\delta}/\bar{\alpha}]\tau\).

3. If \(C \vdash \tau < \tau''\), then \(C \vdash [\bar{\delta}/\bar{\alpha}]\tau < [\bar{\delta}/\bar{\alpha}]\tau''\).

4. If \(C \vdash \tau' = \tau''\), then \(C \vdash [\bar{\delta}/\bar{\alpha}]\tau' = [\bar{\delta}/\bar{\alpha}]\tau''\).

Proof

By induction on the given derivation.

Lemma A.3. Given \(\tau' <\tau''\) and \(\tau' = \tau''\):

1. If \(\tau''\) O.S. then \(\tau' = \tau''\).
2. If \(\tau''\) O.C. then either \(\tau' = \tau''\) or \(\tau' = [\tau'']^\mathcal{S}\).

\[\phi \vdash \tau \mathcal{O} \text{ C} \iff \|\tau\|_\phi = \|\tau\|_\phi \mathcal{O} \text{ C};\]
Proof

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

- **Case** (subInt): $\tau' = \text{int}^S \delta'$ and $\tau'' = \text{int}^S \delta''$, where $\delta' \leq \delta''$.
  1. If $\tau''$ O.S. then $\delta'' = S$. So $\tau' = \tau''$.
  2. If $\tau''$ O.C. then $\delta'' = C$. If $\delta' = S$ then $|\tau'|^S = \text{int}^S \delta' = \tau'$; if $\delta' = C$ then $\tau'' = \text{int}^C \delta'' = \tau'$.

- **Case** (subProd):
  1. By definition of $\equiv$, $\tau' = \tau''$.
  2. $\tau''$ O.C. is impossible.

- **Case** (subSum):
  1. If $\tau''$ O.S. then $\tau'' = (\tau_1'' + \tau_2'')^S$. By inversion on (subSum), $\tau' = (\tau_1' + \tau_2')^S$. By definition of $\equiv$, $\tau_1' = \tau_1''$ and $\tau_2' = \tau_2''$. Therefore $\tau' = \tau''$.
  2. If $\tau''$ O.C. then $\tau'' = (\tau_1' + \tau_2')^C$. By inversion on (subSum), $\tau' = (\tau_1' + \tau_2')^S$. By definition of $\equiv$, $\tau_1' = \tau_1''$ and $\tau_2' = \tau_2''$. If $\delta' = S$ then $|\tau'|^S = (\tau_1'' + \tau_2'')^S$, which is equal to $\tau'$. If $\delta' = C$ then $\tau'' = (\tau_1' + \tau_2')^C = (\tau_1' + \tau_2')^S = \tau'$.

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

\[\square\]

**Theorem 6.1** (Translation Type Soundness).

If $C; \Gamma \vdash e : \tau$ and $\phi$ is a satisfying assignment for $C$ then

1. there exists $e^S$ such that $[\phi] \Gamma \vdash e : [\phi] \tau \rightarrow^S e^S$ and $\vdash \| e \|_\phi \vdash e^S : \| \tau \|_\phi$,
   and if $e$ is a value, then $e^S$ is a value;
2. there exists $e^C$ such that $[\phi] \Gamma \vdash e : [\phi] \tau \rightarrow^C e^C$ and $\vdash \| e \|_\phi \vdash e^C : \| \tau \|_\phi$.

Proof

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

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

- **Case**

  \[\begin{array}{c}
  C; \Gamma \vdash e : \text{int}^S \\
  \hline
  n : \tau
  \end{array}\]  

(SInt)

Part (1): Let $e^S$ be $n$.

- $[\phi] \Gamma \vdash e : [\phi] \text{int}^S \rightarrow^S n$  
  \[\Rightarrow\]
  
  \[\Rightarrow\]

- $[\phi] \Gamma \vdash e : [\phi] (\text{int}^S) \rightarrow^S e^S$ and $e^S$ is a value

  By $n = e$ and def. of substitution

  $\vdash \| e \|_\phi \vdash n : \text{int}$  
  By (TInt)

  $\vdash \| e \|_\phi \vdash e^S : \| \text{int}^S \|_\phi$  
  By $\text{int} = \| \text{int}^S \| = \| [\phi] \text{int}^S \| = \| \text{int}^S \|_\phi$ and $n = e^S$

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


Implicit self-adjusting computation for purely functional programs

\[ [\phi] \Gamma \vdash n \colon \text{int}^S \cdot \approx n \]
\[ [\phi] \Gamma \vdash n \colon \text{int}^S \cdot \Rightarrow \text{let } r = n \text{ in write}(r) \] By (Write)

\[ [\phi] \Gamma \vdash e \colon [\phi](\text{int}^S) \cdot \approx e^C \]
\[ n = e; \text{def. of subst.; } e^C = \ldots \]

\[ \vdash \|\Gamma\|_{\phi} \vdash n : \text{int} \]
By (TInt)

\[ \vdash \|\Gamma\|_{\phi}, r : \text{int} \vdash r : \text{int} \]
By (TVar)

\[ \vdash \|\Gamma\|_{\phi}, r : \text{int} \vdash \text{write}(r) : \text{int} \]
By (TWrite)

\[ \vdash \|\Gamma\|_{\phi} \vdash \text{let } r = n \text{ in write}(r) : \text{int} \]
By (TLet)

\[ \vdash \|\Gamma\|_{\phi} \vdash e^C : \|\text{int}^S\|_{\phi} \]
By def. of \( e^C \)

\[ \vdash \|\Gamma\|_{\phi} \vdash e^C : \|\tau\|_{\phi}^C \]
By \( \tau = \text{int}^S \)

- Case

\[
\frac{\Gamma(x) = \forall \alpha[D]. \tau_0 \quad C \vdash \exists \beta. [\beta/\alpha]D}{\Gamma \vdash \forall x : [\beta/\alpha] \tau_0} \quad (\text{SVar})
\]

Part (1): Let \( e^S \) be \( x[\alpha = \delta] \).

\[
\Gamma(x) = \forall \alpha[D]. \tau_0 \quad \text{Premise}
\]

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

\[
[\phi] \Gamma \vdash x : [\beta/\alpha]((\phi) \tau_0) \cdot \approx x[\alpha = \delta] \quad \text{By (Var)}
\]

\[
[\phi] \Gamma \vdash x : [\phi][\delta/\alpha](\tau_0) \cdot \approx x[\alpha = \delta] \quad \delta \text{ closed and } \alpha \cap \text{dom}(\phi) = \emptyset
\]

\[
[\phi] \Gamma \vdash x : [\phi][\beta/\alpha](\tau_0) \cdot \approx x[\alpha = \delta] \quad \text{Intermediate subst.}
\]

\[
[\phi] \Gamma \vdash x : [\phi][\beta/\alpha](\tau_0) \cdot \approx x[\alpha = \delta] \quad \phi(\tilde{\beta}) = \tilde{\delta}
\]

\[ [\phi] \Gamma \vdash e : [\phi][\tau \cdot \approx e^S] \text{ and } e^S \text{ is a value} \quad \text{By } e = x; \tau = [\beta/\alpha] \tau_0; e^S = x[\alpha = \delta]
\]

\[ [\|\Gamma\|_{\phi}(x)] = [\phi][\exists \beta[[\phi]D]. [\phi] \tau_0] \quad \text{By def. of } \| \|_{\phi} \text{ and def. of subst.}
\]

\[ \vdash [\|\Gamma\|_{\phi}][S, x[\alpha = \delta] = \delta] = [\delta][\beta/\alpha][\|\phi\|_{\alpha}][S] \quad [\|\phi\|_{\alpha}][S] = [\|\phi\|_{\alpha}][S] \quad \text{By (TVar)}
\]

\[ \vdash [\|\Gamma\|_{\phi}][S, e^S : \|\phi\|_{\alpha}][S] \text{ dom}(\phi) = \emptyset \]

\[ \vdash [\|\Gamma\|_{\phi}][S, e^S : \|\beta/\alpha\|_{\alpha}] \quad \text{Intermediate subst., } \phi(\tilde{\beta}) = \tilde{\delta}, \text{ def. of } \| \|_{\phi}
\]

\[ \vdash [\|\Gamma\|_{\phi}][S, e^S : \|\tau\|_{\phi}] = \tau = [\beta/\alpha] \tau_0 \]

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

\[ [\phi] \Gamma \vdash x : [\phi][\tau \cdot \approx \delta] \quad \text{Above}
\]

\[ [\phi] \Gamma \vdash x : [\phi][\tau \cdot \approx \delta] \quad \text{let } r = x[\alpha = \delta] \text{ in write}(r) \quad \text{By (Write)}
\]

\[ [\phi] \Gamma \vdash e : [\phi][\tau \cdot \approx e^C] \quad \text{By } e = x \text{ and def. of } e^C
\]

\[ \vdash [\|\Gamma\|_{\phi}][S, x[\alpha = \delta] = \delta] = [\|\tau\|_{\phi}] \quad \text{Above}
\]

\[ \vdash [\|\Gamma\|_{\phi}, r : \|\tau\|_{\phi}, r : \|\tau\|_{\phi}] \vdash r : \|\tau\|_{\phi} \quad \text{By (TVar)}
\]

\[ \vdash [\|\Gamma\|_{\phi}, r : \|\tau\|_{\phi}, r : \|\tau\|_{\phi}] \vdash \text{write}(r) : \|\tau\|_{\phi} \quad \text{By (TWrite)}
\]

\[ \vdash [\|\Gamma\|_{\phi}, r : \|\tau\|_{\phi}, r : \|\tau\|_{\phi}] \vdash \text{let } r = x[\alpha = \delta] \text{ in write}(r) : \|\tau\|_{\phi} \quad \text{By (TLet)}
\]

\[ \vdash [\phi][\tau \text{ O.S.}] \quad \text{Subcase (a) assumption}
\]

\[ \|\tau\|_{\phi} = [\|\tau\|_{\phi}]^C \quad \text{By Lemma[A.1]}
\]

\[ \vdash [\|\Gamma\|_{\phi}, r : \|\tau\|_{\phi}, r : \|\tau\|_{\phi}] \vdash \text{let } r = x[\alpha = \delta] \text{ in write}(r) : \|\tau\|_{\phi} \quad \text{By above equality}
\]
Part (2), subcase (b) where $\phi \vdash \tau$. O.C.: Let $e^C = \text{let } r = e^S \text{ in read } r \text{ as } r' \text{ in write} (r')$.

$\vdash [\phi] \Gamma \vdash x : [\phi] \tau \vdash e^S$ Above

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

$\vdash [\phi] \Gamma \vdash x : [\phi] \tau \vdash e^C$ By (ReadWrite)

$\vdash \Gamma[S] e^S : \tau$ O.C. Subcase (b) assumption

$\vdash \Gamma[S], r : \tau \vdash \phi$ mod, $r' : \tau \vdash r : \tau$ C By (TPVar)

$\vdash \Gamma[S], r : \tau \vdash C \text{ write}(r') : \tau$ C By (TWrite)

$\vdash \Gamma[S], r : \tau \vdash \phi$ mod $\vdash S : r : \tau$ C By (TVar)

$\vdash \Gamma[S], r : \tau \vdash \phi$ mod $\vdash C \text{ read } r \text{ as } r'$ C By (TRead)

$\vdash \Gamma[S], v : \tau \vdash \phi$ C By Lemma A.1

$\vdash \Gamma[S] e^S : \tau$ C By above eqn.

$\vdash \Gamma[S] e^S : \tau$ C By (TLet)

\begin{prooftree}
\frac{C; \Gamma \vdash e \, v_1 : \tau_1 \quad C; \Gamma \vdash e \, v_2 : \tau_2}{C; \Gamma \vdash e \, \langle v_1, v_2 \rangle : (\tau_1 \times \tau_2)^S} \quad \text{(SPair)}
\end{prooftree}

---

--- Part (1), stable mode translation:

$\vdash C; \Gamma \vdash \epsilon \, v_1 : \tau_1$ Subderivation

$\vdash \Gamma[S] \vdash \epsilon \, v_1 : \tau_1$ By i.h.

$\vdash \Gamma[S] \vdash \epsilon \, v_2 : \tau_2$ Subderivation

$\vdash \Gamma[S] \vdash \epsilon \, v_2 : \tau_2$ By i.h.

Let $e^S = \langle v_1, v_2 \rangle$.

$\vdash [\phi] \Gamma \vdash (v_1, v_2) : ([\phi] \tau_1 \times [\phi] \tau_2)^S \rightarrow S (v_1, v_2)$ By (Pair)

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

$\vdash \Gamma[S] \vdash e^S : \tau$ C By def. of subst. and $e^S = (v_1, v_2)$

$\vdash \Gamma[S] \vdash \epsilon \, \langle v_1, v_2 \rangle : \tau$ By (TPair)

$\vdash \Gamma[S] \vdash e^S : \tau$ C By def. of $\| - \|$
Implicit self-adjusting computation for purely functional programs

\[ \phi \vdash e : [\phi] \tau \rightarrow e^S \]

Above

\[ (\tau_1 \times \tau_2)^S \]

O.S.

By definition of O.S.

\[ \phi \vdash e : [\phi] \tau \rightarrow \text{let } r = e^S \text{ in write}(r) \]

By (Write)

\[ :: \Gamma \vdash \phi \quad e^S : \| \tau \| \phi \]

Above

\[ :: \Gamma, r : \| \tau \| \phi \vdash r : \| \tau \| \phi \quad \text{By (TPVar)} \]

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

\[ \| \tau \| = \| \tau \|_\phi^- \quad \text{By Lemma } A.1 \]

\[ :: \Gamma, r : \| \tau \| \phi \vdash \text{write}(r) : \| \tau \|_\phi^- \quad \text{By above equality} \]

\[ :: \Gamma \vdash e^C : \| \tau \|_\phi^- \quad \text{By (TLet)} \]

\[ \frac{\text{C; } \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash e : \tau_2}{\text{C; } \Gamma \vdash \text{fun } f(x) = e' : (\tau_1 \rightarrow \tau_2)^S} \quad \text{(SFun)} \]

(a) Suppose \( [\phi] e = S \).

\( C; \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash e' : \tau_2 \)

Subderivation

\[ [\phi](\Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S) \vdash e' : [\phi] \tau_2 \rightarrow e' \]

By i.h. and \( [\phi] e = S \)

\[ :: \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash \phi \quad e' : \| \tau_2 \| \phi \quad \text{By def. of } \| \cdot \| \phi \]

\[ :: \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash \text{fun } f(x) = e' : \| \tau_2 \| \phi \quad \text{By (TFun)} \]

\[ :: \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash \text{fun } f(x) = e' : \| (\tau_1 \rightarrow \tau_2)^S \| \phi \quad \text{By def. of } \| \cdot \| \phi \]

(1)\# \quad [\phi] \vdash e : [\phi] \tau \rightarrow \text{let } r = e^S \text{ in write}(r) \quad \text{By (Write)}

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

\[ :: \Gamma, r : \| \tau \| \phi \vdash r : \| \tau \| \phi \quad \text{By (TPVar)} \]

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

(2)\# \quad [\phi] \vdash e : [\phi] \tau \rightarrow \text{let } r = e^S \text{ in write}(r) \quad \text{By (TLet) and Lemma } A.1 \]

(b) Suppose \([\phi] e = C\).

\[ [\phi](\Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S) \vdash e' : [\phi] \tau_2 \rightarrow e' \]

By i.h. and \([\phi] e = C\)

\[ :: \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash \phi \quad e' : \| \tau_2 \| \phi^- \quad \text{By def. of } \| \cdot \| \phi^- \]

\[ [\phi] \vdash e : [\phi] \tau \rightarrow \text{fun } f(x) = e' \quad \text{(SFun)} \]

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

(1)\# \quad [\phi] \vdash e : [\phi] \tau \rightarrow e^S \quad \text{and } e^S \text{ is a value}
\[ \begin{align*}
\vdash \|\phi\|_\phi, x : \|\tau_1\|_\phi, f : \|\tau_1\|_\phi \rightarrow \|\tau_2\|_\phi &\quad \text{C} \vdash \phi \rightarrow \|\tau_2\|_\phi^C \quad \text{By def. of } \|\cdot\|_\phi \\
(1) &\vdash \|\phi\|_\phi \vdash_S \text{fun}^C f(x) = \delta : \|\tau_1\|_\phi &\quad \text{By (TFun)} \\
(2) &\vdash [\phi] \vdash e : [\phi] \tau_1 \rightarrow [\phi] \text{in write}(r) &\quad \text{Analogous to (a)} \\
(2) &\vdash \|\phi\|_\phi \vdash_S \text{let } r = \text{in write}(r) : \|\tau_2\|_\phi^C &\quad \text{“} \\
\end{align*} \]

\[ \text{Case } \]

\[ \begin{array}{c}
\text{Subderivation} \\
C; \Gamma \vdash_x v : \tau_1 \\
[\phi] \vdash v : [\phi] \tau_1 \rightarrow v \\
\vdash \|\phi\|_\phi \vdash_S v : \|\tau_1\|_\phi \\
\vdash \|\phi\|_\phi \vdash_S [\phi] \vdash [\phi] \text{inl} v &\quad \text{By (Sum)} \\
\end{array} \]

\[ \text{Part (1):} \]

\[ \begin{align*}
C; \Gamma \vdash_x v : \tau_1 &\quad \text{Subderivation} \\
[\phi] \vdash v : [\phi] \tau_1 \rightarrow v &\quad \text{By i.h.} \\
\vdash \|\phi\|_\phi \vdash_S v : \|\tau_1\|_\phi &\quad \text{“} \\
\vdash \|\phi\|_\phi \vdash_S \text{inl} v : [\tau_1 + \tau_2] &\quad \text{By (TSum)} \\
\vdash \|\phi\|_\phi \vdash_S [\phi] \vdash \text{inl} v &\quad \text{By (TSum)} \\
\end{align*} \]

\[ \text{Part (2):} \]

\[ \text{Similar to (SPair), using } (\tau_1 + \tau_2)^S \text{ O.S.} \]

\[ \text{Case } \]

\[ \begin{array}{c}
\vdash \|\phi\|_\phi, x : (\tau_1 \times \tau_2)^\delta \\
\vdash C \vdash \delta \leq \varepsilon \\
C; \Gamma \vdash_x \text{fst} x : \tau_1 \\
\end{array} \]

\[ \text{Part (1):} \]

\[ \begin{align*}
C; \Gamma \vdash_x x : (\tau_1 \times \tau_2)^\delta &\quad \text{Subderivation} \\
[\phi] \vdash x : ([\phi] \tau_1 \times [\phi] \tau_2)^S \rightarrow x &\quad \text{By i.h.} \\
\vdash \|\phi\|_\phi \vdash_S x : \|\tau_1\|_\phi \times \|\tau_2\|_\phi &\quad \text{“} \\
\vdash \|\phi\|_\phi \vdash_S \text{fst} x &\quad \text{By (Fst)} \\
\vdash \|\phi\|_\phi \vdash_S \text{fst} x : \|\tau_1\|_\phi &\quad \text{By (Tfst)} \\
\end{align*} \]

\[ \text{Part (2):} \]

\[ \text{Similar to (SVar):} \]

\[ \begin{align*}
\text{– If } \tau_1 \text{ O.S., let } e^C \text{ be let } r = \text{fst } x \text{ in write}(r) \text{ and apply rule (Write).} \\
\text{– If } \tau_1 \text{ O.C., let } e^C \text{ be let } r = \text{fst } x \text{ in read } r \text{ as } r' \text{ in write}(r') \text{ and apply rule (ReadWrite).} \\
\end{align*} \]

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

\[ \text{Part (1):} \]

\[ \begin{align*}
C; \Gamma \vdash_x x : (\tau_1 \times \tau_2)^\delta &\quad \text{Subderivation} \\
[\phi] \vdash x : ([\phi] \tau_1 \times [\phi] \tau_2)^S \rightarrow x &\quad \text{By i.h.} \\
\vdash \|\phi\|_\phi \vdash_S x : \|\tau_1\|_\phi \times \|\tau_2\|_\phi &\quad \text{“} \\
\vdash \|\phi\|_\phi \vdash_S \text{fst} x &\quad \text{By (Fst)} \\
\vdash \|\phi\|_\phi \vdash_S \text{fst} x : \|\tau_1\|_\phi &\quad \text{By (Tfst)} \\
\end{align*} \]

\[ \text{Part (2):} \]

\[ \text{– If } \tau_1 \text{ O.S., let } e^C \text{ be read } x \text{ as } x' \text{ in let } r = \text{fst } x' \text{ in write}(r) \text{ and apply rule (Read) with (LFst).} \]
Implicit self-adjusting computation for purely functional programs

\[ \ldots, r : \| \tau_1 \| \vdash_s r : \| \tau_1 \| \quad \text{By (TPVar)} \]
\[ \ldots, r : \| \tau_1 \| \vdash_C \text{write}(r) : \| \tau_1 \| \quad \text{By (TWrite)} \]
\[ \ldots, r : \| \tau_1 \| \vdash_C \text{write}(r) : \| \tau_1 \|^{-C} \quad \tau_1 \text{ O.S.} \]
\[ \| \Gamma \|, \cdot : \| \tau_1 \| \times \| \tau_2 \| \vdash_s \text{fst} \cdot \vdash \tau_1 \| \quad \text{By (TPVar) then (TFst)} \]
\[ \| \Gamma \|, \cdot : \| \tau_1 \| \times \| \tau_2 \| \vdash_C \text{let } r = \text{fst} \cdot \text{in write}(r) : \| \tau_1 \|^{-C} \quad \text{By (TLet)} \]
\[ \| \Gamma \| \vdash_s \chi : \| (\tau_1 \times \tau_2)^C \| \quad \text{By i.h.} \]
\[ \| \Gamma \| \vdash_s \chi : \| (\tau_1 \times \tau_2) \| \text{ mod } \quad \text{By def. of } \| - \| \]
\[ \| \Gamma \| \vdash_C \text{read } \chi \text{ as } \cdot \text{in let } r = \text{fst} \cdot \text{in write}(r) : \| \tau_1 \|^{-C} \quad \text{By (TRead)} \]

- If \( \tau_1 \text{ O.C.} \), then \( \| \tau_1 \| = \cdot_1 \text{ mod } \) for some \( \cdot_1 \).

Let \( e^C \) be \( \text{read } \chi \text{ as } \cdot \text{in let } r = \text{fst} \cdot \text{in write}(r) \) and apply rule (Read) with (LFst).

\[ \ldots, r^1 : \| \cdot_1 \| \vdash_C \text{write}(r^1) : \| \cdot_1 \| \quad \text{By (TPVar), (TWrite)} \]
\[ \ldots, r^1 : \| \cdot_1 \| \vdash_C \text{write}(r^1) : \| \cdot_1 \|^{-C} \quad \tau_1 \text{ O.C.} \]
\[ \ldots, r : \| \cdot_1 \| \text{ mod } C \vdash r : \| \cdot_1 \| \text{ mod } \quad \text{By (TPVar)} \]
\[ \ldots, r : \| \cdot_1 \| \text{ mod } \vdash_C r : \| \cdot_1 \| \text{ mod } \quad \text{By (TPVar)} \]

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

\[ \begin{array}{c}
\text{Case} \\
C; \Gamma \vdash e_1 : \tau' \quad C; \Gamma, x : \tau'' \vdash e_2 : \tau_C \vdash \tau' \triangleq \tau'' \\
\hline
C; \Gamma \vdash x = e_1 : \text{in } e_2 : \tau \quad \text{(SLetE)}
\end{array} \]

(a) Subcase for \( [\phi] \tau'' \text{ O.C.} \)

\[ C; \Gamma \vdash e_1 : \tau' \quad \text{Subderivation} \]
\[ [\phi] \Gamma \vdash e_1 : [\phi] \tau' \overset{\text{i.h.}}{\vdash} e^C \]
\[ :: [\phi] \Gamma \vdash e^C : \| \tau' \|^{-C} \quad \tau'' \]
\[ C \vdash \tau' \triangleq \tau'' \quad \text{Premise} \]
\[ [\phi] \tau' \triangleq [\phi] \tau'' \quad \text{By Lemma[\( A.2 \)}} \]
\[ C \vdash \tau'' \overset{\text{O.C.}}{\vdash} \quad \text{Premise} \]
\[ [\phi] \tau'' \overset{\text{O.C.}}{\vdash} \quad \text{Subcase (a) assumption} \]
\[ [\phi] \tau'' = [\phi] \tau'' \text{ or } [\phi] \tau' = \| [\phi] \tau'' \|^C \quad \text{By Lemma[\( A.3 \)](2)} \]

If the former, then:
\[ [\phi] \Gamma \vdash e_1 : [\phi] \tau'' \overset{\triangleq}{\overset{\text{Mod}}{\vdash s}} e^C \quad \text{By (Mod)} \]
\[ [\phi] \Gamma \vdash e_1 : [\phi] \tau'' \overset{\triangleq}{\overset{\text{Mod}}{\vdash s}} e^C \quad \text{By (Mod)} \]

If the latter, then:
\[ [\phi] \Gamma \vdash e_1 : [\phi] \tau'' \overset{\triangleq}{\overset{\text{Mod}}{\vdash s}} e^C \quad \text{By (Lift)} \]

Now we have the same judgment no matter which equation Lemma[\( A.3 \)](2) gave us.
Y. Chen et al.

::||Γ||φ ⊢ C eC : ||τ'||C

::||Γ||φ ⊢S mod eC : ||τ'||C mod

By (TMod)

::||Γ||φ ⊢S mod eC : ||τ'||C mod

or ::||Γ||φ ⊢S mod eC : ||τ'||C mod

By τ' = τ'' or |τ''|φ = τ'

::||Γ||φ ⊢S mod eC : ||τ''||C mod

By def. of |−|C or copying

(1) By Lemma[A.1]

::||Γ||φ ⊢S mod eC : ||τ''||C mod

By abobe equation

(b) Subcase for [φ]|τ'' O.S.

C; Γ ⊢ e1 : τ'

Subderivation

[φ]|Γ ⊢ e1 : [φ]|τ' → eS

By i.h.

::||Γ||φ ⊢S eS : ||τ'||φ

By Lemma[A.3] (1)

::||Γ||φ ⊢S eS : ||τ''||φ

By above equation

For both subcases, we have:

C; Γ, x : τ'' ⊢ e2 : τ

Subderivation

[φ]|Γ, x : [φ]|τ'' → e2 : [φ]|τ'' → eS

By i.h. and def. of subst.

::||Γ||φ, x : ||τ''||φ ⊢S eS : ||τ||φ

By i.h. and def. of |−|C

(1) By (LetE)

(1) By (TLet)

C; Γ, x : τ'' ⊢ e2 : τ

Subderivation

[φ]|Γ, x : [φ]|τ'' → e2 : [φ]|τ'' → eC

By i.h. and def. of subst.

::||Γ||φ, x : ||τ''||φ ⊢C eS : ||τ||C

By i.h. and def. of |−|C

(2) By (LetE)

(2) By (TLet)

\[
\bar{\alpha} \cap FV(C, \Gamma) = \emptyset \quad C \vdash [D] C \wedge v_1 : \tau' \quad C \vdash \tau' < \tau'' \quad C \vdash \tau' = \tau'' \quad C \vdash \tau'' (SLetV)
\]

For all \( \bar{\delta}_i \) such that \( \bar{\alpha} = \bar{\delta}_i \vdash D \):

(a) Suppose \( [\phi]|\bar{\delta}_i \bar{\alpha}|\tau'' \) O.S., that is, this \( i \)th monomorphic instance is out-stable, and will not need a \text{mod} in the target.
Implicit self-adjusting computation for purely functional programs

\[ C \land D; \Gamma \vdash_S v_1 : \tau' \] Subderivation
\[ \bar{\alpha} \cap \text{FV}(C, \Gamma) = \emptyset \] Premise
\[ C \land D; [\bar{\delta}/\bar{\alpha}]\Gamma \vdash_S v_1 : [\bar{\delta}/\bar{\alpha}]\tau' \] By Lemma \([A.2]\
\[ \phi([\bar{\delta}/\bar{\alpha}]\Gamma) \vdash v_1 : [\phi][\bar{\delta}/\bar{\alpha}]\tau' \vdash v_i \] By i.h., using the lemma’s guarantee about derivation height
\[ \bar{\alpha} \text{ not free in } \Gamma \] Above disjointness
\[ [\phi] \Gamma \vdash v_1 : [\phi][\bar{\delta}/\bar{\alpha}]\tau' \vdash v_i \] By above line
\[ \vdash [\Gamma]_\phi \vdash_S v_i : [\mid [\bar{\delta}/\bar{\alpha}]\tau']_\phi \] By i.h. and def. of substitution

Let \( e_i \) be \( v_i \).
\[ C \vdash \tau' \triangleleft \tau'' \] Premise
\[ C \vdash \tau' \triangleleft \tau'' \] Premise
\[ [\phi] \tau' \triangleleft [\phi] \tau'' \] By Lemma \([A.2]\
\[ [\phi] \tau'' \text{ O.S.} \] Subcase (a) assumption
\[ [\phi] \tau' = [\phi] \tau'' \] By Lemma \([A.3](1)\
\[ [\phi] \Gamma \vdash v_1 : [\phi][[\bar{\delta}/\bar{\alpha}]\tau'] \vdash v_i \] Above
\[ \bar{\alpha} \cup \text{dom}(\phi) = \emptyset \] By \( \bar{\alpha} \cap \text{FV}(C, \Gamma) = \emptyset \) and appropriateness of \( \phi \) w.r.t. \( C \)
\[ [\phi] \Gamma \vdash v_1 : [\bar{\delta}/\bar{\alpha}][[\phi] \tau'] \vdash v_i \] Property of substitution
\[ [\phi] \Gamma \vdash v_1 : [\bar{\delta}/\bar{alpha}][[\phi] \tau''] \vdash e_i \] By \([\phi] \tau' = [\phi] \tau'' \) and \( e_i = v_i \)
\[ \vdash [\Gamma]_\phi \vdash_S e_i : [\mid [\bar{\delta}/\bar{\alpha}]\tau']_\phi \] Above and \( e_i = v_i \)
\[ \vdash [\Gamma]_\phi \vdash_S e_i : [\mid [\bar{\delta}/\bar{\alpha}][[\phi] \tau'']]_\phi \] Definition of \( \mid - \mid_\phi \)
\[ \vdash [\Gamma]_\phi \vdash_S e_i : [\mid [\bar{\delta}/\bar{\alpha}][[\phi] \tau'']]_\phi \] By \( \bar{\alpha} \cap \text{dom}(\phi) = \emptyset \)
\[ \vdash [\Gamma]_\phi \vdash_S e_i : [\mid [\bar{\delta}/\bar{\alpha}][[\phi] \tau'']]_\phi \] By def. of \( \mid - \mid_\phi \)
\[ \vdash [\Gamma]_\phi \vdash_S v_i : [\mid [\bar{\delta}/\bar{\alpha}]\tau'']_\phi \] By (TPVar)

End of subcase (a)

(b) Suppose \([\phi][\bar{\delta}/\bar{\alpha}] \tau'' \text{ O.S.} \), that is, this \( i \)th monomorphic instance is outerchangeable, and therefore needs a \textbf{mod} in the target.
\[ C \land D; \Gamma \vdash_S v_1 : \tau' \] Subderivation
\[ \bar{\alpha} \cap \text{FV}(C, \Gamma) = \emptyset \] Premise
\[ C \land D; [\bar{\delta}/\bar{\alpha}]\Gamma \vdash_S v_1 : [\bar{\delta}/\bar{alpha}]\tau' \] By Lemma \([A.2]\
\[ \phi([\bar{\delta}/\bar{alpha}]\Gamma) \vdash v_1 : [\phi][\bar{\delta}/\bar{alpha}]\tau' \vdash e_i \] By i.h., using the lemma’s guarantee about derivation height
\[ \vdash [\Gamma]_\phi \vdash C e_i : [\mid [\bar{\delta}/\bar{alpha}]\tau']_\phi \] By i.h. and def. of \( \mid - \mid_\phi^C \)
\[ [\phi][\bar{\delta}/\bar{alpha}] \tau' \triangleleft [\phi][\bar{\delta}/\bar{alpha}] \tau'' \] By Lemma \([A.2]\
\[ [\phi][\bar{\delta}/\bar{alpha}] \tau' \in \{ [\phi][\bar{\delta}/\bar{alpha}] \tau'', \phi][\bar{\delta}/\bar{alpha}] \tau'' \} \] By Lemma \([A.3](2)\)

If the former, then:
This ends the “for all $\lambda$” above. We now have translation judgments for each instance, and target typings for each $e_i$ and associated variable $y_i$.

```
C; Γ, x : ∀\overline{\alpha}(D), \tau'' \vdash e_2 : \tau
```

Subderivation

```
[Γ, x : ∀\overline{\alpha}(D), \tau'' \vdash e_2 : \phi \tau \rightarrow^c e_2]
```

By i.h. and def. of substitution

- **Case**

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

Part (1):
Implicit self-adjusting computation for purely functional programs

C: \Gamma \vdash x_2 : \tau_1 \quad \text{Subderivation}

[\phi] \Gamma \vdash x_2 : [\phi] \tau_1 \rightarrow^{\delta} x_2 \quad \text{By i.h.}

\vdash \Gamma \|_\phi \vdash x_2 : [\phi] \tau_1 \|_\phi

C: \Gamma \vdash x_1 : (\tau \rightarrow \tau)^\delta \quad \text{Subderivation}

[\phi] \Gamma \vdash x_1 : [\phi] \tau \rightarrow^{\delta} x_1 

\vdash \Gamma \|_\phi \vdash x_1 : [\phi] \tau \rightarrow^{\delta} \|_\phi

Part (2):

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

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

\vdash [\phi] \|_\phi, r : [\phi] \|_\phi \vdash r : [\phi] \|_\phi

\vdash [\phi] \|_\phi, r : [\phi] \|_\phi \vdash \text{write}(r) : [\phi] \|_\phi

\vdash \Gamma \|_\phi \vdash \text{let } r = e^\delta \text{ in write}(r) : [\phi] \|_\phi \quad \text{By (TWrite)}

Subcase (a) assumption

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

\[\phi] \Gamma \vdash e : [\phi] \tau \rightarrow^{\delta} \text{let } r = e^\delta \text{ in read } r \text{ as } r' \in \text{write}(r')

\vdash [\phi] \|_\phi, r : [\phi] \|_\phi \vdash r : [\phi] \|_\phi

\vdash [\phi] \|_\phi \vdash \text{read } r \text{ as } r' \in \text{write}(r') : [\phi] \|_\phi \quad \text{By (TRead)}

Subcase (b) assumption

Subcase "C-S" where \[\phi] e' = C and \[\phi] \delta = S.

Part (2):

\[\phi] \Gamma \vdash x_1 : [\phi] \tau \rightarrow^{\delta} x_1 \quad \text{From subcase S-S above}

\[\phi] \Gamma \vdash x_2 : [\phi] \tau_1 \rightarrow^{\delta} x_2 \quad \text{From subcase S-S above}

Let \[\phi] e^C = \text{apply}^C(x_1, x_2).
Y. Chen et al.

\[ \lbrack \emptyset \rbrack \Gamma \vdash e : \lbrack \emptyset \rbrack \tau \rightsquigarrow \text{apply}^C(x_1, x_2) \]

Part (1):
\[ \lbrack \emptyset \rbrack \tau \text{ O.C.} \]

\[ \lbrack \emptyset \rbrack \Gamma \vdash e : \lbrack \emptyset \rbrack \tau \rightsquigarrow e^C \]

Above

\[ \lbrack \emptyset \rbrack \Gamma \vdash e : \lbrack \emptyset \rbrack \tau \rightsquigarrow \text{mod} e^C \]

By (Mod)

\[ \lbrack \emptyset \rbrack \Gamma \vdash e : \lbrack \emptyset \rbrack \tau \rightsquigarrow e^C : \lbrack \tau \rbrack_{\emptyset}^C \]

Above

\[ \lbrack \emptyset \rbrack \Gamma \vdash e : \lbrack \emptyset \rbrack \tau \rightsquigarrow \text{mod} e^C : \lbrack \tau \rbrack_{\emptyset}^C \]

By (TMod)

\[ \lbrack \emptyset \rbrack \Gamma \vdash e : \lbrack \emptyset \rbrack \tau \rightsquigarrow \text{mod} e^C : \lbrack \tau \rbrack_{\emptyset}^C \]

By (TApp)

Part (2):
\[ (\Gamma, x': (\tau_1 \rightsquigarrow \tau)^S)(x') = \forall \vec{a} \left[ \text{true} \right].(\tau_1 \rightsquigarrow \tau)^S \]

By def. of $\Gamma$

\[ C \vdash \exists \vec{a}. \text{true} \]

By def. of $\vdash$

\[ C, \Gamma, x': (\tau_1 \rightsquigarrow \tau)^S \vdash S \ x' : (\tau_1 \rightsquigarrow \tau)^S \]

By (SVar)

\[ \lbrack \emptyset \rbrack \Gamma, x': (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash x' : (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash x') \]

By (Var)

\[ \lbrack \emptyset \rbrack \Gamma, x': (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash x_2 : (\lbrack (\emptyset) \tau_2 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash x_2) \]

By extending $\Gamma$

\[ \lbrack \emptyset \rbrack \Gamma, x': (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash \text{apply}(x', x_2) : (\emptyset) \tau \rightsquigarrow \text{apply}^C(x', x_2) \]

By (App)

\[ \lbrack \emptyset \rbrack \Gamma, x': (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash e : (\emptyset) \tau \rightsquigarrow \text{apply}^C(x', x_2) \]

By deffs. of subst., $\lbrack \emptyset \rbrack \delta^S$

\[ \lbrack \emptyset \rbrack \Gamma \vdash e \rightsquigarrow (x_1 \Rightarrow x) : (\emptyset) \tau \rightsquigarrow (\emptyset) \tau \rightsquigarrow \text{apply}^C(x', x_2) \]

By (LApply)

\[ (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash \text{O.C.} \]

By def. of $\text{O.C.}$

\[ C, \Gamma \vdash S \ x_1 : \tau_f \]

Subderivation

\[ \lbrack \emptyset \rbrack \Gamma \vdash x_1 : (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash x_1) \]

By i.h.

\[ \vdash (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash \text{mod} \ l_{\emptyset}^C) \]

\[ \text{read} x_1 \text{ as } x' \text{ in } \text{apply}^C(x', x_2) \]

By (Read)

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

\[ \vdash (\lbrack (\emptyset) \tau_1 \rightsquigarrow \tau \rbrack_{\emptyset}^S \vdash \text{mod} \ l_{\emptyset}^C) \]

\[ \text{read} x_1 \text{ as } x' \text{ in } \text{apply}^C(x', x_2) \]

By (TRead) (**)
Implicit self-adjusting computation for purely functional programs

Part (1):

\[ C \vdash \delta < \tau \]
Premise

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

\[ [\phi] \Gamma \vdash e : [\phi] \tau \cong e^C \]
Above

\[ \therefore [\phi] \Gamma \vdash e : [\phi] \tau \cong \text{mod} e^C \]
By (Mod)

\[ \vdash [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
Above (**)

\[ \therefore [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
By reasoning in subcase \( C-S \) for Part (1); note that \( [\phi] \tau \) O.C.

---

Subcase “S-C” where \( [\phi] e' = S \) and \( [\phi] \delta = C \):

Part (2):

\[ [\phi] \Gamma, x' : ([\phi] \tau_1 \cong [\phi] \tau)^S \vdash \text{apply}(x', x_2) : [\phi] \tau \cong e^C_0 \]
Above with \( x' \) for \( x_1 \)

\[ [\phi] \Gamma, x' : ([\phi] \tau_1 \cong [\phi] \tau)^S \vdash [x'/x_1] e : [\phi] \tau \cong e^C_0 \]
By defs. of \( \text{mod} \), subst.

\[ [\phi] \Gamma \vdash x_1 : ([\phi] \tau_1 \cong [\phi] \tau)^C \cong x_1 \]
By i.h.

\[ [\phi] \Gamma \vdash e : [\phi] \tau \cong \text{read} x_1 \text{ as } x' \text{ in } e^C_0 \]
By (Read)

Let \( e^C = \text{read} x_1 \text{ as } x' \text{ in } e^C_0 \).

\[ \vdash [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
Above with \( x' \) for \( x_1 \)

\[ \vdash [\Gamma]_\phi, x' : [\tau]_\phi \cong [\tau]_\phi \cong e^C_0 : [\tau]_\phi \]
By extending \( \Gamma \)

\[ \vdash [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
By i.h.

\[ \therefore [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
By (TRead)

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

---

Case

\[ C, \Gamma \vdash x_1 : \text{int}^\delta_1 \]
\[ C, \Gamma \vdash x_2 : \text{int}^\delta_2 \]

\[ C \vdash \delta_1 = \delta_2 \quad \vdash + : \text{int} \times \text{int} \rightarrow \text{int} \]

(SPrim)

\[ C, \Gamma \vdash e : \text{int}^\delta \]
Subderivation

\[ [\phi] \Gamma \vdash x_1 : \text{int}^\delta_1 \cong x_1 \]
By i.h.

\[ \vdash [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
“”

\[ [\Gamma]_\phi \vdash x_1 : \text{int} \]
By \( [\phi] \delta_1 = S \text{ and def. of } \| - \| \)

\[ [\phi] \Gamma \vdash x_2 : \text{int}^\delta_2 \cong x_2 \]
Similar to above

\[ \vdash [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
Similar to above

\[ [\phi] \Gamma \vdash e : \text{int}^\delta \cong + (x_1, x_2) \]
By (Prim)

\[ [\phi] \Gamma \vdash e : [\phi] (\text{int}^\delta_1) \cong + (x_1, x_2) \]
By \( [\phi] \delta_1 = S \)

Let \( e^S = + (x_1, x_2) \).

\[ \vdash + : \text{int} \rightarrow \text{int} \]
Premise

\[ \vdash [\Gamma]_\phi \vdash + (x_1, x_2) : \text{int} \]
By (TPrim)

\[ \therefore [\Gamma]_\phi \vdash \text{mod} e^C : [\tau]_\phi \]
By \( [\phi] \delta_1 = S \text{ and def. of } \| - \| \)

Part (2): Similar to (SPair); note that \( \tau \) O.S. holds.
If $|\psi|\delta_1 = |\psi|\delta_2 = C$ then:

Part (2):

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

*Case 1.* Suppose $|\phi|\delta = S,

\[ C; \Gamma \vdash S \vdash x : (\tau_1 + \tau_2) \delta \]

\[ C; \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad C \vdash \delta \leq \varepsilon \]

\[ C; \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \quad C \vdash \delta < \varepsilon \]

(by (Case))

\[ C; \Gamma \vdash \{ x_1 \Rightarrow e_1, x_2 \Rightarrow e_2 \} : \tau \]

(by (TCase))

\[ C; \Gamma \vdash \{ x_1 \Rightarrow e_1, x_2 \Rightarrow e_2 \} : \{ \tau\} \]

(by (TCase))
By induction on the given derivation. All cases are straightforward.

Lemma B.2 (Stores Are Monotonic). If \( \rho_1 \vdash e \Downarrow w \) and \( \rho_2 \vdash w' \) then there exists \( \rho' \) such that \( \rho_2 = \rho_1, \rho' \).

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

Lemma B.3 (Values Are Ineffective). For all \( \rho \) and \( w \) such that \( \rho \vdash w \Downarrow (\rho' \vdash v') \), it is the case that \( \rho = \rho' \) and \( w = v' \).

Proof
By induction on the given derivation.

Lemma B.4. For all \( v \), the judgment \( v \Downarrow v \) is derivable.

Proof
By structural induction on \( v \).
Proof

By induction on the derivation of $\Gamma \vdash e : \tau \rightarrow^\delta e'$.

- **Case (LetE):** $e = (\text{let } x = e_1 \text{ in } e_2)$ and $e' = (\text{let } x = e_0 \text{ in } e'_2)$.

Let $\Gamma_1 = \| \tau_1 \|$ and $\Gamma' = \| \tau \|$.

To show that $[s]e$ and $[s]e'$ are related, we need to show $[s]e_1 : \tau_1 \vdash [s]e_0 : \Gamma_1$ and for all $\nu \vdash w$, it is the case that $[\nu/x][s]e_2 : \tau \vdash [w/x][s]e'_2 : \Gamma'$. The first part is straightforward. For the second part, suppose we have some $\nu : \tau_1 \vdash w : \Gamma_1$.

Let $s' = s, v/x$ and $s'' = s, w/x$.

$s' : (\Gamma, x : \tau_1) \vdash \nu : (\| \Gamma', x : \tau'_1 \|)$ and $\nu : \tau_1 \vdash w : \Gamma_1$

$\Gamma, x : \tau_1 \vdash e_2 : \tau \rightarrow e'_2$  
Subd.

$[s']e_2 : \tau \rightarrow [s']e'_2 : \tau'$  
By i.h.

$[s, v/x]e_2 : \tau \rightarrow [s, w/x]e'_2 : \tau'$  
By def. of $s'$ and $s''$

(Popping out of the quantifier, $\vdash \nu \rightarrow \text{let } x = [s]e_1 \text{ in } [s]e_2 : \tau \rightarrow \text{let } x = [s]e_0 \text{ in } [s]e'_2 : \tau'$, from which $e : \tau \rightarrow e' : \tau'$) follows the definition of substitution (moving the $[s]$ and $[s]$ outward).

- **Case (Int):** $e = n$ and $e' = n$.

It follows from the definition of $\vdash$ that $n : \text{Int}^\delta \vdash n : \text{Int}$, which is the same as $[s]n : \text{Int}^\delta \vdash [s]n : \text{Int}$, which was to be shown.

- **Case (Var):**

$\Gamma \vdash x : (\delta/\alpha)[\tau_0] \rightarrow x[\alpha = \delta]$

Given

$\Gamma \vdash x : (\delta/\alpha)[\tau_0] \rightarrow x[\alpha = \delta]$

From $\vdash s : \Gamma$

$\vdash [s]x : \forall \alpha[D], \tau_0$

From $\vdash s : \| \Gamma \|$}

$\vdash [s]x : \Pi \alpha[D], \tau_0 \rightarrow [s]x : \Pi \alpha[D], \tau_0$

From $\vdash s : \| \Gamma \|$

$\vdash [s]x : \delta/\alpha \rightarrow [s]x[\alpha = \delta] : \| \delta/\alpha[\tau_0] \|$

By def. of $\vdash$

$\vdash [s]x : \delta/\alpha \rightarrow [s]x[\alpha = \delta] : \| \delta/\alpha[\tau_0] \|$

By def. of subst.

- **Case (Pair):**

$\Gamma \vdash (v_1, v_2) : (\tau_1 \times \tau_2)^\delta \rightarrow (\nu_1, \nu_2)$

Given

$\Gamma \vdash v_1 : \tau_1 \rightarrow v'_1$

Subd.

$\vdash [s]v_1 : \tau_1 \rightarrow [s]v'_1 : \| \tau_1 \|$

By i.h.

$\Gamma \vdash v_2 : \tau_2 \rightarrow v'_2$

Subd.

$\vdash [s]v_2 : \tau_2 \rightarrow [s]v'_2 : \| \tau_2 \|$

By i.h.

$\vdash \nu_1, \nu_2 : (\tau_1 \times \tau_2)^\delta \rightarrow \nu'_1, \nu'_2 : (\| \tau_1 \| \times \| \tau_2 \|)^\delta$

By def. of $\vdash$

$\vdash \nu_1, \nu_2 : (\tau_1 \times \tau_2)^\delta \rightarrow \nu'_1, \nu'_2 : (\| \tau_1 \| \times \| \tau_2 \|)^\delta$

By def. of subst. and $\vdash$
Implicit self-adjusting computation for purely functional programs

- **Case (Fun):**

  \[ \Gamma \vdash \text{fun } f(x) = e : (\tau_1 \rightarrow \tau_2)^S \rightarrow \text{fun}^S_{\tau_1} \]

  Let \( s' = s, v/x, (\text{fun } f(x) = e) / f \). Let \( s'' = s, w/x, (\text{fun}^S_{\tau_1} f(x) = e^S) / f \).

  We show the case for \( e = C \). The case for \( e = S \) is similar, using \( \| - \| \) instead of \( \| - \|^{-C} \).

  \[ \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2)^S \vdash e : \tau_2 \rightarrow e^S \]

  By definition of \( \bowtie \) for \text{fun}, we have

  \[ \text{fun } f(x) = e : (\tau_1 \rightarrow \tau_2)^S \vdash (\text{fun}^S_{\tau_1} f(x) = e^S) : \| \tau_1 \| \rightarrow \| \tau_2 \|^{-C} \]

- **Case (Sum):**

  \[ \Gamma \vdash \text{inl } v : (\tau_1 + \tau_2)^S \rightarrow \text{inl } v' \]

  Given

  \[ \Gamma \vdash v : \tau_1 \rightarrow \tau_2 \]

  Subd.

  \[ [s]v : \tau_1 \vdash [s]v' : \| \tau_1 \| \]

  By i.h.

  \[ \text{inl } ([s]v) : (\tau_1 + \tau_2)^S \vdash \text{inl } ([s]v') : \| \tau_1 \| \rightarrow \| \tau_2 \| \]

  By def. of \( \vdash \)

  \[ \text{inl } ([s]v) : (\tau_1 + \tau_2)^S \vdash \text{inl } ([s]v') : \| (\tau_1 + \tau_2)^S \| \]

  By def. of \( \| - \| \)

- **Case (Fst):**

  \[ \Gamma \vdash \text{fst } x : \tau \rightarrow \text{fst } x \]

  Given

  \[ \Gamma \vdash x : (\tau \times \tau_2)^S \rightarrow x \]

  Subd.

  \[ [s]x : (\tau \times \tau_2)^S \vdash [s]x : \| (\tau \times \tau_2)^S \| \]

  By i.h.

  \[ [s]x : (\tau \times \tau_2)^S \vdash [s]x : \| \tau \| \rightarrow \| \tau_2 \| \]

  By def. of \( \| - \| \)

  \[ \text{fst } [s]x : \tau \vdash \text{fst } [s]x : \| \tau \| \]

  By def. of subst.

- **Case (Prim):**

  \[ \Gamma \vdash + (x_1, x_2) : \tau \rightarrow + (x_1, x_2) \]

  Given

  \[ \Gamma \vdash x_1 : \tau_1 \rightarrow x_1 \]

  Subd.

  \[ [s]x_1 : \tau_1 \vdash [s]x_1 : \| \tau_1 \| \]

  By i.h.

  \[ [s]x_2 : \tau_2 \vdash [s]x_2 : \| \tau_2 \| \]

  By similar reasoning for \( x_2 \)

  \[ + ([s]x_1, [s]x_2) : \tau \vdash + ([s]x_1, [s]x_2) : \| \tau \| \]

  By def. of \( \vdash \)

  \[ + ([s]x_1, [s]x_2) : \tau \vdash + ([s]x_1, [s]x_2) : \| \tau \| \]

  By def. of subst.

- **Case (App):**

  \[ e = \text{apply}^S_{\tau_1} (x_1, x_2) \]

  Let \( \| - \| \) be \( \| - \| \) when \( e = S \), and \( \| - \|^{-C} \) when \( e = C \).
\[ \Gamma \vdash \text{apply}(x_1, x_2) : \tau_2 \rightarrow^\rho \text{apply}^\rho(x_1, x_2) \quad \text{Given} \]

\[ \Gamma \vdash x_1 : (\tau_1 \rightarrow \tau_2)^S \rightarrow^\rho x_1 \quad \text{Subd.} \]
\[ [s]x_1 : (\tau_1 \rightarrow \tau_2)^S \rightarrow [s]x_1 : \| (\tau_1 \rightarrow \tau_2)^S \| \quad \text{By i.h.} \]
\[ [s]x_1 : (\tau_1 \rightarrow \tau_2)^S \rightarrow [s]x_1 : \| \tau_1 \| \rightarrow \| \tau_2 \| \quad \text{By def. of } \| \| \]
\[ \Gamma \vdash x_2 : \tau_1 \rightarrow x_2 \quad \text{Subd.} \]
\[ [s]x_2 : \tau_1 \rightarrow [s]x_2 : \| \tau_1 \| \quad \text{By i.h.} \]

\[ \text{apply}([s]x_1, [s]x_2) : \tau_2 \rightarrow \text{apply}^\rho(([s]x_1, [s]x_2) : \| \tau_2 \| \)
\]
\[ \text{By def. of } \rightarrow \]

\[ [s]\text{apply}(x_1, x_2) : \tau_2 \rightarrow [s](\text{apply}^\rho(x_1, x_2)) : \| \tau_2 \| \quad \text{By def. of subst.} \]

- **Case (Case):**

\[ \Gamma \vdash \text{case } x \text{ of } \{ x_1 \Rightarrow e_1, x_2 \Rightarrow e_2 \} : \tau \]
\[ \rightarrow^\rho \text{ case } x \text{ of } \{ x_1 \Rightarrow e'_1, x_2 \Rightarrow e'_2 \} \quad \text{Given} \]
\[ \Gamma \vdash x : (\tau_1 + \tau_2)^S \rightarrow^\rho x \quad \text{Subd.} \]
\[ x : (\tau_1 + \tau_2)^S \rightarrow x : \tau'_1 + \tau'_2 \quad \text{By i.h. and def of } \| \| \]

For \( k \in \{1, 2\} \):

For all \( v : \tau_k \rightarrow w : \tau'_k \):

Let \( s' = s, v/x_k \) and \( s'' = s, w/x_k \).
\[ \Gamma, x_k : \tau_k \vdash e_k : \tau \rightarrow e'_k \quad \text{Subd.} \]
\[ [s']e_k : \tau \rightarrow [s']e'_k : \tau' \quad \text{By i.h.} \]
\[ [s, v/x_k]e_k : \tau \rightarrow [s, w/x_k]e'_k : \tau' \quad \text{By def. of } s' \text{ and } s'' \]
\[ [v/x_k][s]e_k : \tau \rightarrow [w/x_k][s]e'_k : \tau' \quad \text{By def. of subst. } (x_k \text{ fresh}) \]

Therefore \text{case } x \text{ of } \{ x_1 \Rightarrow [s]e_1, x_2 \Rightarrow [s]e_2 \} : \tau \rightarrow \text{case } x \text{ of } \{ x_1 \Rightarrow [s]e'_1, x_2 \Rightarrow [s]e'_2 \} : \tau'. \text{ The result follows by pushing } [s] \text{ and } [s] \text{ to the outside.} \]

- **Case (Mod):**

\[ \Gamma \vdash e : \tau \rightarrow e^C \quad \text{mod} \quad \text{Given} \]
\[ \Gamma \vdash e : \tau \rightarrow e^C \quad \text{Subd.} \]
\[ [s]e : \tau \rightarrow [s]e^C : \| \tau \|^{-C} \quad \text{By i.h.} \]
\[ [s]e : \tau \rightarrow \text{mod} [s]e^C : \| \tau \|^{-C} \quad \text{mod} \quad \text{By def. of } \rightarrow \text{mod} \]
\[ [s]e : \tau \rightarrow [s](\text{mod} e^C) : \| \tau \|^{-C} \quad \text{mod} \quad \text{By def. of subst.} \]

\[ \tau \text{ O.C.} \quad \text{Premise} \]
\[ \| \tau \|^{-C} \quad \text{mod} = \| \tau \| \quad \text{By Lemma}[A.1] \]
\[ \text{By preceding equation} \]

- **Case (Lift):**
Implicit self-adjusting computation for purely functional programs

\( \Gamma \vdash e : \tau \rightarrow e^C \mod \) \hspace{2cm} \text{Given}
\( \Gamma \vdash e : \tau \rightarrow e^C \) \hspace{2cm} \text{Subd.}
\( [s]e : \tau \rightarrow [s]e^C : \|\tau^S\| \rightarrow \|e^C\| \) \hspace{2cm} \text{By i.h.}
\( \|\tau^S\| \rightarrow \|e\| \rightarrow \|\tau\| \) \hspace{2cm} \text{By preceding equation}
\( [s]e : \tau \rightarrow [s]e^C : \|\tau\| \rightarrow \|e\| \) \hspace{2cm} \text{By def. of subst.}
\( \tau \rightarrow \|e\| \) \hspace{2cm} \text{By Lemma A.1}
\( s : e : \tau \rightarrow [s]e^C : \|\tau\| \) \hspace{2cm} \text{By preceding equation}

- **Case (LetV):**

\[ (\text{begin}) \]
\( \Gamma, x : \forall \alpha[D]. \tau_0 \vdash e_2 : \tau \rightarrow e_2' \) \hspace{2cm} \text{Subd.}
\( [s,v_0/x]e_2 : \tau \rightarrow [s,v_0/x]e_2' : \) \hspace{2cm} \text{By i.h.}
\( (\text{end}) \]
\[ (\text{let } x = v \text{ in } e_2) : \tau \rightarrow \begin{cases} [s] & \text{by def. of subst.} \\ \tau' & \end{cases} \]

- **Case (Read):**

We have a subderivation of \( \Gamma \vdash e \rightarrow (x \Rightarrow x' : \tau_0 \vdash e') \). By inversion on the rules for such judgments, either

- (LApply) \( e = \text{apply}(x, x_2) \), or
- (LCase) \( e = \text{case } x \text{ of } \{ x_1 \Rightarrow e_L , x_2 \Rightarrow e_R \} \), or
- (LPrimop1) \( e = \oplus(x, x_2) \) and \( e' = \oplus(x', x_2) \), or
- (LPrimop2) \( e = \oplus(x, x_1) \) and \( e' = \oplus(x_1, x') \).

Assume the first case; the others are similar.

Given: \( \Gamma \vdash e : \tau \rightarrow \text{read } x \text{ as } x' \in e^C \).

Let \( \tau_1 = \|\tau_0\| \) and \( \tau'_1 \) be such that \( \|\tau_0\| = \tau'_1 \) \hspace{2cm} \text{mod (it must be the case that } \tau_0 \text{ has this form because } \tau_0 \text{ O.C.)}

Suppose \( \nu : \tau_1 \rightarrow v : \tau'_1 \).
\[ \Gamma \vdash x : \tau_0 \xrightarrow{s} x \quad \text{Subd.} \]
\[ [s]x : \tau_0 \vdash [s]x : [\tau_0] \quad \text{By i.h.} \]
\[ \Gamma, x' : [\tau_0]_{\Sigma} \vdash e' : \tau \xrightarrow{s} e^C \quad \text{Subd.} \]
\[ [s, v/x'](\text{apply}(x', x_2)) : \tau \vdash [s, w/x'](e^C : \tau') \quad \text{By i.h.} \]
\[ [v/x']([s](\text{apply}(x', x_2)) : \tau \vdash [w/x'](e^C : \tau') \quad \text{By def. of subst.} \]

Therefore, by the definition of \( \xrightarrow{s} \), when the target expression is a \textit{read}, we have
\[ [x/x'](\text{apply}(x', x_2)) : \tau \xrightarrow{s} \text{read } [s]x \text{ as } x' \text{ in } [s]e^C : \tau' \]

By def. of subst., \([s][x/x'](\text{apply}(x', x_2)) : \tau \xrightarrow{s} [s](\text{read } x \text{ as } x' \text{ in } e^C) : \tau' \]. By def. of subst., \([s]\text{apply}(x, x_2) : \tau \xrightarrow{s} [s]\text{apply}(x, x_2) : \tau' \] which was to be shown.

• Case (Write):
\[ \Gamma \vdash e : \tau \xrightarrow{s} \text{let } r = e^S \text{ in } \text{write}(r) \quad \text{Given} \]
\[ \Gamma \vdash e : \tau \xrightarrow{s} e^S \quad \text{Subd.} \]
\[ [s]e : e^S \vdash [s]e^S : \tau' \quad \text{By i.h.} \]

Suppose \( v : \tau \vdash w : \tau' \).
\[ [v/r]r : \tau \vdash w : \tau' \quad \text{By def. of subst.} \]
\[ [v/r][s]r : \tau \vdash w : \tau' \quad \text{By def. of subst. (r fresh)} \]
\[ [v/r][s]r : \tau \vdash \text{write}(w) : \tau' \quad \text{By def. of } \vdash r \]
\[ [v/r][s]r : \tau \vdash \text{write}(w/r) : \tau' \quad \text{By def. of subst.} \]
\[ [v/r][s]r : \tau \vdash |w/r|^\langle[s] \text{write}(r) \rangle : \tau' \quad \text{By def. of subst.} \]

By definition of \( \xrightarrow{s} \), we have \([e/r][s]r : \tau \xrightarrow{s} \text{let } r = e^S \text{ in } [s](\text{write}(r)) : \tau' \]. By def. of subst., \([s]e : \tau \xrightarrow{s} [s]\text{let } r = e^S \text{ in } \text{write}(r) : \tau' \], which was to be shown.

• Case (ReadWrite):
Using techniques from the (Write) and (Read) cases.

Lemma B.5. If \( e : \tau \xrightarrow{s} e' : \tau' \) then \( e \) and \( e' \) are closed. (In particular, \( e' \) has no free location variables \( \ell \).)

Proof
Follows from the definition of \( \xrightarrow{s} \).

Lemma B.6. If \( e : \tau \xrightarrow{s} [\rho_1]e' : \tau' \) and \( \rho_2 \) extends \( \rho_1 \) then \( e : \tau \xrightarrow{s} [\rho_2]e' : \tau' \).

Proof
By Lemma B.5 \([\rho_1]e' \) is closed, so \( FV(e') \subseteq dom(\rho_1) \). If \( \rho_2 \) extends \( \rho_1 \) then it agrees on all locations in \( \rho_1 \). Therefore \([\rho_2]e' = [\rho_1]e' \), so replacing equals with equals gives the result.

Lemma B.7. If \([v_1/x]e_2 \Downarrow v \) and \( e_1 \Downarrow v_1 \) then \([e_1/x]e_2 \Downarrow v \).

Proof
By induction on the given derivation.
Theorem [6.3] (Generalized Translation Soundness).

If $e : \sigma \Downarrow \rho | e' : \sigma'$ and $D : \rho \Downarrow (\rho' \Downarrow w)$
then $D' : e \Downarrow v$ where $v : \sigma \Downarrow \rho' | w : \sigma'$.

Proof

- **Case** (TEvApply): $e' = \text{apply}^e(e'_1, e'_2)$

  \[
  \rho \Downarrow \text{apply}^e(e'_1, e'_2) \Downarrow (\rho' \Downarrow w) \quad \text{Given}
  \]
  \[
  \rho \Downarrow e'_1 \Downarrow (\rho_1 \Downarrow \text{fun}^e f(x) = e^f) \quad \text{By inversion (TEvApply)}
  \]
  \[
  \rho_1 \Downarrow e'_2 \Downarrow (\rho_2 \Downarrow w_2) \quad \text{"}
  \]

  $D_3 : \rho_2 \Downarrow [\text{fun}^e f(x) = e^f] [w_2/x] e^f \Downarrow (\rho' \Downarrow w) \quad \text{"}$

  $e \equiv \text{apply}(e_1, e_2)$ and

  \[
  e_1 : (\tau_1 \rightarrow \tau_2)^6 \Downarrow \rho | e'_1 : \tau_1' \rightarrow \tau_2' \quad \text{By def. of } \Downarrow \Downarrow
  \]
  \[
  e_2 : (\tau_1 \rightarrow \tau_2)^6 \Downarrow \rho_2 | e'_2 : \tau_1' \rightarrow \tau_2' \quad \text{By Lemma [B.6]}
  \]

  $v_1 : (\tau_1 \rightarrow \tau_2)^6 \Downarrow (\rho_1) | (\rho_2)[\text{fun}^e f(x) = e^f) : \tau_1' \rightarrow \tau_2' \quad \text{By i.h. (subd.)}$

  $v_1 : (\tau_1 \rightarrow \tau_2)^6 \Downarrow (\rho_2)[\text{fun}^e f(x) = e^f) : \tau_1' \rightarrow \tau_2' \quad \text{By Lemma [B.6]}

  $v_1 : (\tau_1 \rightarrow \tau_2)^6 \Downarrow (\rho_2)[\text{fun}^e f(x) = e^f) : \tau_1' \rightarrow \tau_2' \quad \text{Above}$

  $v_1 : (\text{fun } f(x) = e_0) \quad \text{By def. of } \Downarrow \Downarrow$

  $[\text{fun } f(x) = e_0]/f)[v_2/x] e_0 : \tau \quad \text{By i.h. (for all...)}$

  $v_1 : [\text{fun}^e f(x) = e^f]/f)[v_2/x] e^f : \tau' \quad \text{By def. of } \Downarrow \Downarrow$ ("for all..."

  $D_3 : \rho_2 \Downarrow [\text{fun}^e f(x) = e^f]/f)[w_2/x] e^f \Downarrow (\rho' \Downarrow w) \quad \text{Above}$

  $v_1 : [\text{fun } f(x) = e_0]/f)[v_2/x] e_0 \Downarrow v \quad \text{By i.h. (subd.)}$

  $v_1 : \tau \Downarrow \rho' | w : \tau' \quad \text{By (SEvValue)}$

  $v_1 : [\text{fun } f(x) = e_0]/f)[v_2/x] e_0 \Downarrow v \quad \text{By (SEvValue)}$

- **Case** (TEvPair):

  \[
  e \equiv (e'_1, e'_2) \Downarrow (\rho' \Downarrow (w_1, w_2)) \quad \text{Given}
  \]

  $e : \tau \Downarrow (e'_1, e'_2) : \tau' \quad \text{Given}$

  $v_2 : \tau \Downarrow (e_1, e_2) : \tau' \quad \text{By def. of } \Downarrow \Downarrow$

  $v_2 : \tau_2 \Downarrow (e'_1, e'_2) : \tau'_2 \quad \text{"}$

  $E_1 : (\tau_1 \times \tau_2)^6 \Downarrow (e'_1, e'_2) : \tau'_1 \times \tau'_2 \quad \text{By def. of } \Downarrow \Downarrow$

  $v_1 : (\tau_1 \times \tau_2)^6 \Downarrow (e_1, e_2) : \tau'_1 \times \tau'_2 \quad \text{By (SEvValue)}
\[
\rho \vdash e'_1 \triangleq (\rho_1 \vdash v'_1) \\
\rho_1 \vdash e'_1 \triangleq (\rho'_1 \vdash v'_2) \\
\rho \vdash e'_2 \triangleq (\rho' \vdash v'_2)
\]

\[\rho \vdash \text{inl}(\text{TEvFst})\]
\[\rho \vdash \text{inl}(\text{TEvPrimop})\]
\[\rho \vdash \text{inl}(\text{TEvSum})\]

- **Case (TEvPrimop):**
  \[
  \rho \vdash e'_1 \triangleq (\rho_1 \vdash v_1) \\
  e_1 \vdash v_1 \text{ and } v_1 : \tau_1 \vdash [\rho_1]v'_1 : T' \text{ By i.h.}
  \]
  \[
  \rho_1 \vdash e'_1 \triangleq (\rho'_1 \vdash v_2) \\
  e_2 \vdash v_2 \text{ and } v_2 : \tau_2 \vdash [\rho'_1]v'_2 : T' \text{ By i.h.}
  \]
  \[\vdash (e_1, e_2) \triangleright (v_1, v_2) \text{ By (SEvPair)}\]

- **Case (TEvSum):**
  \[
  \rho \vdash \text{inl}(\text{TEvFst}): \tau \vdash [\rho]\text{inl}(e'_0) : T' \\
  \text{Given}
  \]
  \[
  \text{inl} e_0 : (\tau_1 + \tau_2) \vdash [\rho]\text{inl}(e'_0) : T' + T'' \\
  \text{By def. of subst.}
  \]
  \[
  e_0 : \tau_1 \vdash [\rho]e'_0 : T' \\
  \text{By def. of \triangleright}
  \]
  \[
  \rho \vdash e'_0 \triangleq (\rho'_1 \vdash w_0) \\
  e_0 \vdash v_0 \text{ By i.h.}
  \]
  \[
  v_0 : \tau_1 \vdash [\rho'_1]w_0 : T'' \\
  \text{By def. of subst.}
  \]
  \[\vdash \text{inl}(\text{TEvSum})\]

- **Case (TEvPrimop):**
  \[
  \rho \vdash e'_1 \triangleq (\rho'_1 \vdash w) \\
  e : \tau \vdash [\rho](\text{inl}(e'_1, e'_2)) : \tau' \\
  \text{Given}
  \]
  \[
  e : \tau \vdash (\text{inl}(e'_1, e'_2)) : \tau' \\
  \text{By def. of subst.}
  \]
  \[
  e = \text{inl}(e_1, e_2) \text{ and } e_k : \tau_k \vdash e'_k : T'_k \\
  \text{(for all } k \in \{1, 2\} \text{) By def. of \triangleright}
  \]
  \[
  \rho \vdash e'_k \triangleq (\rho_k \vdash w_k) \\
  \text{(for all } k \in \{1, 2\} \text{) Subd.}
  \]
  \[
  e_k \vdash v_k \text{ and } v_k : \tau_k \vdash w_k : T'_k \\
  \text{(for all } k \in \{1, 2\} \text{) By i.h.}
  \]
  \[
  v_k = w_k \\
  \text{(for all } k \in \{1, 2\} \text{) By def. of \triangleright}
  \]
  \[
  \vdash \text{inl}(\text{TEvPrimop})\]
  \[\vdash \text{inl}(\text{TEvSum})\]

- **Case (TEvFst):**
Implicit self-adjusting computation for purely functional programs

\[ \rho \vdash \textsf{fst} \ e_0 \Downarrow (\rho' \vdash w) \]  
Given

\[ (\textsf{fst} \ e_0) : \tau \Rightarrow [\rho](\textsf{fst} \ e_0') : \tau' \]  
Given

\[ (\textsf{fst} \ e_0) : \tau \Rightarrow \textsf{fst} ([\rho]e_0') : \tau' \]  
By def. of subst.

\[ e_0 : \tau \Rightarrow [\rho]e_0' : \tau' \]  
By def. of \( \Rightarrow \)

\[ \rho \vdash e_0 \Downarrow (\rho' \vdash (w, w_2)) \]  
Subd.

\[ e_0 \Downarrow v \]  
By i.h.

\[ v_{\text{pair}} : \tau_{\text{pair}} \Rightarrow [\rho']((w, w_2) : \tau' \times \tau'_2) \]  
" and def. of \( \Rightarrow \)

\[ (v, v_2) : (\tau \times \tau_2)^0 \Rightarrow [\rho']((w, w_2) : \tau' \times \tau'_2) \]  
By def. of \( \Rightarrow \)

\[ (v, v_2) : (\tau \times \tau_2)^0 \Rightarrow ((\rho'\, w; [\rho]w') : \tau' \times \tau'_2) \]  
By def. of subst.

\[ e : \tau \Rightarrow [\rho] \text{let } x = e_1' \text{ in } e_2' : \tau' \]  
Given

\[ e : \tau \Rightarrow \text{let } x = [\rho]e_1' \text{ in } [\rho]e_2' : \tau' \]  
Def. of substitution

By the definition of \( \Rightarrow \) with a target let, either \( e = (\text{let } x = e_1 \text{ in } e_2) \) or \( e = [e_1/x]e_2 \).

In the former case:

\[ e = (\text{let } x = e_1 \text{ in } e_2) \quad \text{and} \quad e_1 : \tau_1 \Rightarrow [\rho]e_1' : \tau_1' \]  
By def. of \( \Rightarrow \)

\[ [v_1/x]e_2 : \tau \Rightarrow (([\rho_1]w_1/x)[\rho]e_2' : \tau') \]  
"\n
\[ \rho_1 \vdash [v_1/x]e_2' \Downarrow (\rho' \vdash w) \]  
Subd.

\[ v : \tau \Rightarrow [\rho']w : \tau' \]  
i.h.

\[ e \Downarrow v \]  
By (SEvLet)

In the latter case, also use Lemma [8.7]

\[ \rho \vdash \text{case } e_0' \text{ of } \{ x_1 \Rightarrow e_1' , x_2 \Rightarrow e_2' \} \Downarrow (\rho' \vdash w) \]  
Given

\[ \text{case } e_0 \text{ of } \{ x_1 \Rightarrow e_1' , x_2 \Rightarrow e_2' \} : \tau \]  
\( \Rightarrow [\rho](\text{case } e_0' \text{ of } \{ x_1 \Rightarrow e_1' , x_2 \Rightarrow e_2' \}) : \tau' \)  
Given

\[ \rho \vdash e_0' \Downarrow (\rho_1 \vdash \text{inl } w_1) \]  
Subd.

\[ \rho \vdash e_0 \Downarrow v \quad \text{and} \quad v_1 : \tau_0 \Rightarrow [\rho_1](\text{inl } w_1) : \tau_0' \]  
By i.h.

\[ [v_1/x_1]e_1 : \tau \Rightarrow [w_1/x_1]e_1' : \tau' \]  
By def. of \( \Rightarrow \)

\[ \rho_1 \vdash [w_1/x_1]e_1' \Downarrow (\rho' \vdash w) \]  
Subd.

\[ [v_1/x_1]e_1 \Downarrow v \quad \text{and} \quad v : \tau \Rightarrow w : \tau' \]  
i.h.

\[ \text{case } e_0 \text{ of } \{ x_1 \Rightarrow e_1 , x_2 \Rightarrow e_2 \} \Downarrow v \]  
By (SEvCaseLeft)
• Case (TEvRead):

\[ \rho \vdash \text{read } e'_1 \text{ as } x \text{ in } e'_2 \Downarrow (\rho' \vdash w) \quad \text{Given} \]
\[ e : \tau \mapsto [\rho][\text{read } e'_1 \text{ as } x \text{ in } e'_2] : \tau' \quad \text{Given} \]
\[ e = [e_1/x]e_2 \text{ and } e_1 : \tau_1 \mapsto [\rho]e'_1 : \tau'_1 \quad \text{By def. of } \mapsto \]
for all \( v : \tau_1 \mapsto w : \tau'_1 \):
\[ [v/x]e_2 : \tau \mapsto [w/x][\rho]e'_2 : \tau' \quad " \]

\[ \rho \vdash e'_1 \Downarrow (\rho_1 \vdash \ell) \quad \text{Subd.} \]
\[ e_1 \Downarrow v_1 \text{ and } v_1 : \tau_1 \mapsto [\rho_1]\ell : \tau'_1 \quad \text{By i.h.} \]

By definition of store substitution, \( [\rho_1]\ell \) must have the form \text{mod} \( w_1 \).
\[ [v_1/x]e_2 : \tau \mapsto [w_1/x][\rho]e'_2 : \tau' \quad \text{Above with } v = v_1 \text{ and } w = w_1 \]

\[ \rho_1 \vdash [\rho_1(\ell)/x][\rho]e'_2 \Downarrow (\rho' \vdash w) \quad \text{Subd.} \]
\[ \text{By def. of } \mapsto \]

\[ e_1 \Downarrow v_1 \quad \text{Above} \]
\[ e_1/x)e_2 \Downarrow v \quad \text{By Lemma [5.7]} \]

• Case (TEvWrite):

\[ \varnothing : \rho \vdash \text{write}(e_0) \Downarrow (\rho' \vdash w) \quad \text{Given} \]
\[ e : \tau \mapsto \text{write}(e_0) : \tau' \quad \text{Given} \]
\[ e : \tau \mapsto e_0 : \tau' \quad \text{By def. of } \mapsto \]
\[ \varnothing_1 : \rho \vdash e_0 \Downarrow (\rho' \vdash w) \quad \text{Subd.} \]
\[ \varnothing' : e \Downarrow v \quad \text{By i.h.} \]
\[ v : \tau \mapsto [\rho']w : \tau' \quad " \]

• Case (TEvMod):

\[ \rho \vdash \text{mod } e^C \Downarrow (\rho'_0, \ell \mapsto w) \Downarrow \ell) \quad \text{Given} \]
\[ \rho \vdash e^C \Downarrow (\rho'_0 \vdash w) \quad \text{Subd.} \]
\[ e : \tau \mapsto [\rho][\text{mod } e^C] : (\tau'_0 \text{ mod}) \quad \text{Given} \]
\[ e : \tau \mapsto [\rho]e^C : \tau'_0 \quad \text{By def. of } \mapsto \]
\[ e \Downarrow v \text{ and } v : \tau \mapsto [\rho_0]w : \tau'_0 \quad \text{By i.h.} \]
\[ v : \tau \mapsto [\rho_0'](\text{mod } w) : \tau'_0 \text{ mod} \quad \text{By def. of } \mapsto \]
\[ v : \tau \mapsto [\rho_0'](\ell \mapsto w)\ell : \tau'_0 \text{ mod} \quad \text{By def. of store substitution} \]
\[ v : \tau \mapsto [\rho']\ell : \tau' \quad \text{By } \rho' = \rho'_0, \ell \mapsto w \text{ and } \tau' = \tau'_0 \text{ mod} \]

• Case (TEvSelectE):
Implicit self-adjusting computation for purely functional programs

\[ \rho \vdash (\text{select} \{ \ldots, \delta \Rightarrow e_i, \ldots \})[\overline{\alpha} = \delta] \Downarrow (\rho' \vdash w) \quad \text{Given} \]
\[ e : \tau \vdash (\text{select} \{ \ldots, \delta \Rightarrow e_i, \ldots \})[\overline{\alpha} = \delta] : \tau' \quad \text{Given} \]
\[ e : \tau \vdash (\text{select} \{ [\rho], \ldots, \delta \Rightarrow [\rho]e_i, [\rho], \ldots \})[\overline{\alpha} = \delta] : \tau' \quad \text{By def. of subst.} \]
\[ \tau = [\delta/\overline{\alpha}]\tau_0 \quad \text{and} \quad \tau' = \| [\delta/\overline{\alpha}]\tau_0 \| \quad \text{By def. of } \vdash \]
\[ e : \forall \alpha[D]. \tau_0 \vdash (\text{select} \{ [\rho], \ldots, \delta \Rightarrow [\rho]e_i, [\rho], \ldots \}) : \Pi \overline{\alpha}[D]. \tau_0 \quad " \]
\[ e : [\delta/\overline{\alpha}]\tau_0 \vdash [\rho]e_i : \| [\delta/\overline{\alpha}]\tau_0 \| \quad \text{By def. of } \vdash \]
\[ \rho \vdash e_i \Downarrow (\rho' \vdash w) \quad \text{Subd.} \]
\[ e \Downarrow v \quad \text{By i.h.} \]
\[ v \vdash [\rho']w \quad " \]

C Proof of costed soundness

Theorem [6,7] If \( \text{trans}(e, e) = e' \) then \( e' \) is deeply 6-bounded.

Proof

By lexicographic induction on \( e \) and \( e' \), with \( S \) considered smaller than \( C \).

- If \( e \in \{ n, x, (v_1, v_2), \text{fun} \ldots \inl \text{v} \} \) then:
  - If \( e = S \) then \( \text{trans} \) 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 \( e = C \) then, for \( n/(v_1, v_2)/\text{fun}/\text{inl} \text{v} \), \( \text{trans} \) uses its last case and applies (Write), adding a let and a write, so \( HC(e') = 2 \).
    For \( x \), \( \text{trans} \) uses either (Write) or (ReadWrite), giving \( HC(e') \leq 2 + 1 \) or \( HC(e') \leq 3 + 1 \).

- If \( e \) has the form let \( x = e_1 \) in \( e_2 \) where \( x \) has type \( \tau'' \), the resulting let did not come from (Write) or (ReadWrite) so \( HC(e') = 0 \).

- If \( e \) has the form let \( x = e_1 \) in \( e_2 \) where \( x \) is polymorphic, the resulting let did not come from (Write) or (ReadWrite) so \( HC(e') = 0 \).

- 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, \( \text{trans} \) applies (Var), (Var), (Prim), (Write), (Read) (with (LPrimop2)) and (Read) (with (LPrimop1)), producing
  \[ e' = \text{read } x_1 \text{ as } y_1 \text{ in read } x_2 \text{ as } y_2 \text{ in let } r = \oplus(y_1, y_2) \text{ in write}(r) \]
  so (assuming \( HC(x_1) \) and \( HC(x_2) \leq 1 \)),
  \[ HC(e') \leq 1 + 1 + 1 + 1 + HC(\text{let } r = \oplus(y_1, y_2) \text{ in write}(r)) \]
  \[ = 4 + (1 + 0 + 1 + 0) = 6 \]

- If \( e \) is an apply, then:
  - Case (\( S, S, S \)): Here \( e' \) is an \( \text{apply}^{S} \), so \( HC(e') = 0 \).
  - Case (\( C, S, C \)): Here \( e' \) is an \( \text{apply}^{C} \), so \( HC(e') = 0 \).
  - Case (\( S, S, C \)): Either (Write) or (ReadWrite), after switching to \( S \) mode, meaning one of the \( (--, --, S) \) cases—which each generate a subterm whose \( HC \) is 0.
    For (Write), \( HC(e') = 1 + 0 + 1 = 2 \); for (ReadWrite), \( HC(e') = 1 + 0 + (1 + 0 + 1) = 3 \).
The rules (LApply) and (LCase) guarantee that the read has the correct form for \( HC(e') \) to be defined.

- Case \((\epsilon', \mathbb{C}, \mathbb{C})\): Applies (Read) after devolving to \((\epsilon', \mathbb{S}, \mathbb{C})\) which returns a term with \( HC(e') \leq 3 \) (zero if \( e' = \mathbb{C} \), and 2 or 3 if \( e' = \mathbb{S} \)). Applying (Read) yields a term with \( HC(e') \leq 1 + HC(x) + 3 \); since \( HC(x) \leq 1 \) this gives \( HC(e') \leq 5 \).
  Note that \( HC(e') \) is defined for the same reason as in the \((\mathbb{S}, \mathbb{S}, \mathbb{C})\) subcase.

- Case \((\epsilon', \mathbb{S}, \mathbb{S})\): 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 \((\epsilon', \mathbb{C}, \mathbb{C})\): Devolves to \((\epsilon', \mathbb{C}, \mathbb{C})\), with \( HC \leq 5 \), then applies (Mod), yielding \( HC(e') \leq 1 + 5 = 6 \).

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

- If \( e = \text{fst} \ x \) 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 \( \tau_1 \) O.S. then \( HC(e') \leq 2 \) (Write). If \( \tau_1 \) O.C. then we use (Read-Write) followed by the \((\mathbb{S}, \mathbb{S})\) case which has \( HC \) of 0, yielding \( HC(e') \leq 1 + 0 + 1 = 3 \).
  - 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 \((\mathbb{C}, \mathbb{C})\) case is bounded by 3. Using (Read) in this case adds at most 2, so \( HC(e') \leq 5 \).

- If \( e \) is a case on a variable \( x : \tau \), then:
  - If \( \tau \) is outer stable, the proof is straightforward.
  - If \( \tau \) is outer changeable, the algorithm applies rule (Read), recursing with \( x : [\tau]^\delta \), which will apply rule (Case). A case has \( HC \) of 0, so (Read) produces \( e' \) where \( HC(e') = 1 + 0 + 0 = 1 \). \(\square\)

**Theorem 6.8** (Cost Result). Given \( \mathcal{D} :: \rho \vdash e' \Downarrow (\rho' \vdash w) \) where for every subderivation \( \mathcal{D}^* :: \rho^+_1 \vdash e^*_1 \Downarrow (\rho^+_2 \vdash w^*_2) \) of \( \mathcal{D} \) (including \( \mathcal{D} \)), \( HC(\mathcal{D}^*) \leq k \), then the number of dirty rule applications in \( \mathcal{D} \) is at most \( \frac{k}{k+1} W(\mathcal{D}) \).

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

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

**Proof**
By Theorem 6.7 \( e' \) is deeply 6-bounded.

The algorithm \( \text{trans} \) merely applies the translation rules, so \( \cdot \vdash e : \tau \Downarrow e' \). We can show \( e : \tau \Downarrow e' : \tau \) as in Theorem 6.5. By Theorem 6.14 \( \mathcal{D}' :: e \Downarrow v \) where \( v : \sigma \Downarrow [\rho'] w : \sigma' \) and \( \mathcal{D} \) and all its subderivations have \( HC \) bounded by \( k \).
Implicit self-adjusting computation for purely functional programs

By Theorem 6.3, the number of dirty rule applications in \( D \) is \( \leq \frac{k}{k+1} W(D) \). Each rule application is either clean or dirty, so \( W(D) \leq (k+1)W(D) \), where \( k = 6 \).

\[ \square \]

**Theorem 6.14** (Costed Generalized Translation Soundness).
If \( e : \sigma \Rightarrow [\rho][e'] : \sigma' \) and \( D :: \rho \vdash e' \Downarrow (\rho' \vdash w) \)
and \[ [\rho][e'] \text{ is deeply } k\text{-bounded} \]
then \( D' :: e \Downarrow v \) where \( v : \sigma \Rightarrow [\rho'][w] : \sigma' \)
and \[ [\rho'][w] \text{ is deeply } k\text{-bounded} \]
and for every subderivation \( D'' :: \rho_1 \vdash e^* \Downarrow (\rho_2^* \vdash w^*) \) of \( D \) (including \( D' \)), \( HC(D'') \leq HC(e^*) \leq k \),
and the number of clean rule applications in \( D \) equals \( W(D') \).

**Proof**
The differences from Theorem 6.3 require additional reasoning:

- The 7 cases for the “clean” rules (TEvMachineValue), (TEvPair), (TEvSum), (TEvP-rimop), (TEvFst), (TEvCaseLeft), and (TEvApply) are straightforward: the induction hypothesis shows that the \( HC \) condition holds for all proper subderivations of \( D \), and \( HC(D) = 0 \) by definition of \( HC(D) \), 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 \( D \) equals \( W(D') \)).

In (TEvCaseLeft) and (TEvApply), observe that we are substituting machine values; for all machine 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.

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

- (TEvWrite) We have \( D :: \rho \vdash \text{write}(e'_0) \Downarrow (\rho' \vdash w) \) with subderivation \( D_0 :: \rho \vdash e'_0 \Downarrow \ldots \) By i.h., \( HC(D_0) \leq HC(e'_0) \). Therefore \( HC(D_0) + 1 \leq HC(e'_0) \). By the definitions of \( HC \) we have \( HC(D) = HC(D_0) + 1 \) and \( HC(\text{write}(e'_0)) = 1 + HC(e'_0) \), so our inequality becomes \( HC(D) \leq HC(\text{write}(e'_0)) \), which was to be shown. Lastly, the \( \cdots = W(D') \) condition from the i.h. is exactly what we need, because the \( D' \) is the same and (TEvWrite) is not clean.

- (TEvMod) Similar to the (TEvWrite) case.

- (TEvLet) Here we must distinguish **lets** generated by the translation rules (Write) and (ReadWrite), which add entirely new **lets**, from **lets** that are generated by (LetE) and (LetV). The latter kind come from the translation rules (LetE) and (LetV); even though (LetV) replaces the source binding with one that binds a **select**, it does not add a new **let**, it merely replaces one.

If we have the latter kind of **let** then its \( HC(-) \) is 0 in our definition, we consider the rule “clean”, and things go as smoothly as for the 7 unambiguously clean rules (in TEvLet, with reasoning analogous to TEvCaseLeft to deal with the substitution).

On the other hand, if we have **let** \( x = e_1 \) **in** \( e_2 \) resulting from (Write) or (ReadWrite), we rely on the definition of \( HC(\text{let} \ldots) \) to guarantee the that the bound variable \( x \) is used exactly once, justifying the definition’s adding \( HC(e_1) \) and \( HC(e_2) \), which would be nonsensical if \( x \) didn’t appear exactly once in \( e_2 \).
Y. Chen et al.

- (TEvRead) For $HC(\text{read} \ldots)$ to be defined, the variable bound is used exactly once and contributes to the $HC$ of the term accordingly, justifying the equation

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

- (TEvSelectE)

\[
\begin{align*}
(L) & \quad 1 + HC(\mathcal{D}_1) \leq HC(e_1) & \text{i.h.} \\
(R) & \quad 1 + HC(e_1) \leq HC(\{\text{select} \{\ldots\}\}[\ldots]) & \text{By def. of } HC(e_1); \text{ property of max.}
\end{align*}
\]

By (L), (R), transitivity, $e' = \{\text{select} \{\ldots\}\}[\ldots]$ is deeply $k$-bounded.

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 (TEvMachineValue) 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), (TEvSelectE)—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 (TEvSum) 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 $\ell$ where $\ell \mapsto w$, and $w$ is deeply $k$-bounded.

References


Implicit self-adjusting computation for purely functional programs


