Affine Contracts for Affine Types

Jesse A. Tov Riccardo Pucella
Northeastern University
{tov,riccardo}@ccs.neu.edu

Abstract
Affine and other substructural type systems offer a range of expressiveness and performance benefits but cannot easily interact with components written in non-affine languages in a safe manner. We propose a technique for regulating the interaction between code written in an affine language and code written in a conventional typed language by means of contracts.

We formalize our approach via a typed calculus with both affine-typed and conventionally typed modules. An affine type system is used for affine modules and a conventional type system for conventional modules, and neither type system is aware of the other. We show how to preserve the properties guaranteed by the two type systems despite modules being able to call each other and exchange values arbitrarily, and we establish the correctness of our approach through a syntactic type soundness theorem.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Programming by contract; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—Affine Logic

General Terms Languages, Reliability

Keywords Contracts, affine types, substructural logic, multi-language systems

1. Introduction
Substructural type systems augment conventional type systems with the ability to control the number and order of uses of a data structure or operation (Walker 2005). The most common substructural type systems are linear type systems (Wadler 1990; Plotkin 1993; Benton 1995; Ahmed et al. 2004). Roughly speaking, a linear type system ensures that values with linear type cannot be duplicated or dropped: a value of linear type must be eliminated exactly once. Other substructural type systems refine these constraints. Affine type systems, for instance, enforce that values cannot be duplicated, but allow them to be dropped: a value of affine type may be used once or not at all.

A common use case for substructural type systems is to type check reference cells that support strong updates, that is, where the type of a value stored in a cell can be different at different points during the execution of a program. The following example in an ML-like language illustrates why conventional type systems are inadequate to the task of typing strong updates:

```
fun tricky (r1: int → int) ref, r2: (int → int) ref): int =
let val f = λr1
val _ = r1 := 0
val g = λr2
in f (g 2) end
```

If we invoke `tricky(r, r)`, where `r` is a reference cell containing an `int → int` function, then `g` in the code above will be an integer rather than a function, resulting in a run-time fault. The problem occurs only if the two arguments to `tricky` are aliases for the same reference cell, which a linear or affine type system can prevent.

Beyond strong updates, substructural types have been used for memory management (Jim et al. 2002), for optimization of lazy languages (Turner et al. 1995), and to handle effects in pure languages (Barendsen and Smetsers 1996). Typestate (Strom and Yemini 1986) and session types (Gay and Hole 1999) can also be understood as substructural type systems, although they are not always presented as such.

Given the range of language features that substructural types can express, a programmer may wish to take advantage of these features in writing real-world programs. Writing real systems, however, often requires access to comprehensive libraries, which mainstream programming languages generally provide but prototype implementations often do not. The prospect of rewriting libraries to work in a substructural language strikes these authors as unappealing.

This suggests allowing conventional and substructural languages to interoperate. We envision the following scenarios:

- The programmer wishes to import a legacy library into a substructural language. Unfortunately, if the library is unaware of the substructural conditions then it may duplicate values received from the substructural language.
- The programmer wishes to write a library in a substructural language but provide access to the library from a conventional language. The naive client may duplicate values it receives from a substructural library and resubmit them, causing aliasing that the substructural library could not produce by itself and bypassing the guarantees of the substructural type system.

Our Contributions. The core contribution of this paper is a novel approach to regulating the interaction between an affine language and a conventionally typed language. We present a calculus having several notable features:

- The non-affine language may gain access to affine values, including strongly-updateable references, and may apply affine-language functions.
- The non-affine type system has no awareness of the affine type system.
- And yet, the resulting system enjoys type soundness.
We expect our technique to be efficiently implementable on real hardware.

Our calculus consists of two sublanguages with different type systems: a simply-typed \( \lambda \) calculus with a completely standard type system and a \( \lambda \) calculus with an affine type system. We have chosen an affine rather than linear type system because what it means for an affine invariant to be violated is significantly clearer than what it means for a linear invariant to be violated. (We revisit this point in §6.) Our affine language provides strong updates as a simple stand-in for the variety of features such as typestate that they subsume.

A program is a collection of modules and a main expression, and each module may be written in either of the two sublanguages. Modules in each language have access to modules written in the other language, though they view foreign types through a translation into the native type system. Affine modules are checked by an affine type system, and non-affine modules are checked by the conventional type system. Notably, the conventional type system has no knowledge of affine types and their properties.

To allow modules to invoke each other and exchange values while preserving the safety properties guaranteed by the individual type systems, we need to perform runtime checks in cases where the non-affine type system is too weak to express the affine type system’s invariants for values that flow between the languages. For instance, the affine type system guarantees that an affine value created in an affine module will not be duplicated within the affine sublanguage. If the value flows into a non-affine module, however, static bets are off. In that case, we resort to a dynamic check that prevents the value from flowing back into an affine context—where it can be used—more than once. Since our calculus is higher-order, we use a form of higher-order contract (Findler and Felleisen 2002) to keep track of each module’s obligations toward maintaining the affine invariants.

In Findler and Felleisen’s formulation, a software contract is an agreement between two software components, or parties, about some property of a value. The positive party produces a value, which must satisfy the specified property. The negative party consumes the value and is held responsible to treat it appropriately. Contracts are concerned with catching violations of the property and blaming the guilty party, which may help locate the source of a bug. For first-order values, the contract may be immediately checkable, but for functional values, the property is likely undecidable, so the check must wait until the negative party applies the function, at which point the negative party is responsible for providing a suitable argument and the positive party for producing a suitable result. Thus, for higher-order functions, checks are delayed until first-order values are reached.

Our approach to integrating affine and conventional types borrows heavily from recent literature on multi-language interoperability, in particular work that uses contracts to mediate between typed and untyped code. A variety of approaches to partial typing appear in the literature, including hybrid types (Flanagan 2006) and gradual types (Siek and Taha 2006). We follow the approaches of Typed Scheme and its predecessor (Tobin-Hochstadt and Felleisen 2008, 2006) and of Matthews and Findler (2007), both of which handle multi-language programs that combine an untyped, Scheme-like language with a typed language. Both use contracts to regulate the inter-language boundary. Unlike these other systems, both of our languages are typed. Our affine type system is strictly stronger than our non-affine type system, in the sense that the former is capable of enforcing all of the latter’s invariants. The typed/untyped systems have this property in a trivial sense only, since the untyped (or trivially typed, rather) side enforces no representation invariants.

Road Map. In §2, we give the syntax and semantics of our affine sublanguage, followed by a very brief presentation of our non-affine sublanguage in §3. In §4 we introduce our multi-language calculus and attempt to justify some of its design. The impatient reader may wish to skip to §4.3 on page 7 for examples. We explain our soundness criterion, state a standard type soundness theorem, and discuss highlights of the proof in §5. We explore alternate design ideas, extensions, and future research possibilities in §6 and conclude in §7.

To distinguish the two languages of our calculus, we typeset our affine language \( \mathcal{A} \) in a sans-serif font and our non-affine language \( \mathcal{C} \) in a bold serif font.

## 2. The Affine Sublanguage \( \mathcal{A} \)

We begin with our affine sublanguage, which we call \( \mathcal{A} \) (for “affine,” typeset in sans-serif). Figure 1 presents the syntax of types and terms.

In preparation for adding contracts, \( \mathcal{A} \) includes modules, which act as contractual parties. For simplicity, a module declares only one name bound to one value. A program comprises a mutually recursive collection of modules \( M \) and a main expression \( e \).

Expressions are mostly conventional: values, which include \( \lambda \) expressions, several constants, pairs of values, and \( [z] \) for integer literals; variables; application; if expressions; pair construction; and pair elimination. Less conventionally, expressions also include module names \( f \), which reduce to the value of the named module. We define the free variables of an expression in the usual way, but note that this includes only regular variables (e.g., \( y \)), not module names (e.g., \( g \)), which we assume are distinguished syntactically.

Types are syntactically partitioned into two sorts, the unlimited types \( \sigma^u \), which permit value duplication, and the affine types \( \sigma^a \), which do not. In particular, the type system forbids aliasing of affine-typed variables, which is crucial for supporting strong updates. The unlimited types include integers, unlimited functions \( (\sigma \bowtie \sigma) \), and products of unlimited types. The affine types include references, one-shot functions \( (\sigma \oslash \sigma) \), and products with at least one affine component. While a one-shot function \( \sigma \oslash \sigma \) can only be applied once and cannot be duplicated, an unlimited function \( \sigma \bowtie \sigma \) can be duplicated and used freely. (Our grammar also defines a non-terminal \( \bowtie \), which is useful when discussing both one-shot and unlimited functions.)

---

1 This calculus is designed to be simple, but the multi-language system described in this paper should work equally well with a “low level” affine calculus having explicit !-suspension and duplication, with a “high level” polymorphic calculus, or with \( \lambda^{\text{PRAL}} \)-style (Ahmed et al. 2005) qualifier polymorphism.
It takes two references to swap that may result in aliasing the value. While writing is all right in reading frequently observed (Ahmed et al. 2004), freely
(A-Figure 2 presents the operational semantics of the functions applied to value $\mapsto\varepsilon$ contexts are wholly conventional as are stores, which map locations $\ell$ to value $\mapsto \varepsilon$.

We use a function $\text{swap} : \ell \mapsto \varepsilon$, where the store maps $\ell$ to value $\mapsto \varepsilon$.

$$\delta_{\sigma}(s, c v) = (s, v)$$

2.2.2 Operational Semantics

Expressions in $\mathcal{A}$ are typed by a judgment $\Gamma \vdash_{\mathcal{A}} e : \sigma$, in figure 3, where $\Gamma$ is an environment mapping variables to types. As in other substructural type systems, our type system pays special attention to managing the type environment. For example, in TA-APP, the rule for application, the environment $\Gamma$ is split between the operator and operand rather than being wholly available to both.

As in Girard’s (1987) linear logic, we do without the structural rule for contraction (duplicating), but only for some types. Environment splitting is defined so that variables with affine type are available in only one subexpression or the other:

$$\Gamma_1 \not\in \Gamma_2 \vdash_\mathcal{A} \Gamma_1 \not\in \Gamma_2 \vdash_\mathcal{A} \Gamma_3$$

Unlimited variables, on the other hand, are subject to contraction, so they are available on both sides of the split:

$$\Gamma_1 \not\in \Gamma_2 \vdash_\mathcal{A} \Gamma_1 \not\in \Gamma_2 \vdash_\mathcal{A} \Gamma_3$$

Of course, the empty environment splits into itself:

$$\bullet \not\in \bullet \vdash_\mathcal{A} \bullet$$

We implicitly allow exchange (reordering) of $\Gamma$. Our affine language, unlike a linear language, does implicitly permit weakening (discarding variables), since the base-case rules TA-CON, TA-VAR, and TA-MOD allow for ignoring an arbitrary $\Gamma$.

Note that two rules for typing $\lambda$ expressions, TA-LAM and TA-LAMM. The former applies only in cases where the free variables of the term do not include any affine ($\sigma^\ast$) types, and the latter applies only in cases where the free variables do include affine types; thus, at most one of these rules applies in any case. In the former case, it is safe to use the function an unlimited number of times, so we give it an unlimited (\(\infty\)) type. In the latter case, the function closes over some affine values, so it should be used at most once, and we give it an affine ($\sigma^\ast$) type.

The affine sublanguage also admits a simple form of subtyping wherein an unlimited function may be used where a one-shot function is expected. The reverse, as we have seen, must be avoided. We

$$\text{let } r : (\sigma \to int) \not\in \text{ref } (\lambda x : \sigma \cdot x + [1]) \text{ in tricky } r \tau$$

The first swap writes $[0]$ to the same location that the second swap reads, which means that this program gets stuck trying to apply an integer:

$$\left(\ell \mapsto [1], (\lambda x : int \cdot x + [1]) ([0] [2])\right)$$

For the strong update to be safe, it is necessary but not sufficient to ensure that the two references passed to tricky are unaliased, either to each other or elsewhere. A similar problem occurs if tricky is partially applied and the resulting function is then applied twice:

$$\text{let } f : (\sigma \to int) \not\in \text{ref } \lambda s \cdot \text{tricky } (\lambda x : int \cdot x + [1]) \text{ in } f (\text{new } \lambda x : int \cdot x + [2]) + f (\text{new } \lambda x : int \cdot x + [3])$$

When we partially apply tricky to some new reference $\ell$, it closes over the reference. The first call to $f$ then stores $[0]$ at $\ell$, which the second call to $f$ reads out and tries to apply to $[4]$, thereby getting stuck. Thus, it is not enough to avoid aliasing references. We must also avoid aliasing values, such as closures, that contain references.
define the subtyping relation (<:) in figure 4. The subtyping relation is reflexive and transitive, covariant in products and function results, and contravariant in function arguments, as usual. Given this subtyping relation, the subsumption rule TA-SUBSUME states that any expression may be used at a more restrictive (higher) type.

To type a whole program (figure 5), we check that each module is checked in the context of all the modules, which means that all modules must have an unlimited type σω. Then, the main expression must type in the context of all the modules.

Figure 3. $\sigma$ static semantics (expressions)

\[
\begin{align*}
\text{(TA-CON)} & \quad \Gamma \vdash^M e : \sigma' \\
\text{(TA-LAMROM)} & \quad \Gamma, x : \sigma \vdash^M e : \sigma' \\
\text{(TA-APP)} & \quad \Gamma_1 \sqcap \Gamma_2 \vdash^M e_1 : \sigma_1, e_2 : \sigma_2 \\
\text{(TA-LAMAFF)} & \quad \Gamma, x : \sigma \vdash^M e : \sigma' \\
\text{(TA-IF0)} & \quad \Gamma_1 \vdash^M e_1 : \text{int}, \Gamma_2 \vdash^M e_2 : \sigma \\
\text{(TA-LET)} & \quad \Gamma_1 \sqcap \Gamma_2 \vdash^M \text{let}(x, y) = e_1 \in e_2 : \sigma \\
\end{align*}
\]

\[
\begin{align*}
\text{ty}^M(c) &= \sigma \\
\text{ty}^M([z]) &= \text{int} \\
\text{ty}^M(\text{new}) &= \sigma_1 \otimes \sigma_2 \text{ref} \\
\text{ty}^M(\text{swap}) &= \sigma_1 \text{ref} \otimes \sigma_2 \text{ref} \otimes \sigma_1 \\
\end{align*}
\]

Figure 4. $\sigma$ static semantics (subtyping)

\[
\begin{align*}
\sigma &<: \sigma \\
\text{(O-DERELICT)} & \quad \sigma_1 \otimes \sigma_2 <: \sigma_1 \otimes \sigma_2 \\
\text{(O-REFL)} & \quad \sigma <: \sigma \\
\text{(O-TRANS)} & \quad \sigma_1 <: \sigma_2, \sigma_2 <: \sigma_3 \\
\text{(O-PROD)} & \quad \sigma_1 <: \sigma_1' \otimes \sigma_2 <: \sigma_2' \otimes \sigma_2, \sigma_2 <: \sigma_3 \\
\text{(O-ARROW)} & \quad \sigma_1 <: \sigma_1' \otimes \sigma_2 <: \sigma_2' \otimes \sigma_2 \\
\end{align*}
\]

Figure 5. $\sigma$ static semantics (programs and modules)

\[
\begin{align*}
\text{(T-PROGA)} & \quad \forall m \in M, a^M \text{okay} \\
\text{(T-LOADA)} & \quad a^M \text{m okay} \\
\text{(TM-A)} & \quad a^M \text{module } f : \sigma^m = v \text{ okay} \\
\end{align*}
\]

variables \(x, y\)
module names \(f, g\)
expressions \(e := v | x | f | e e\)
values \(v := \lambda x : \tau. e | c\)
constants \(c := [z] | \ldots\)
types \(\tau := \text{int} | \tau \rightarrow \tau\)
evaluation contexts \(E := [E]_E | E e | v E\)
configurations \(C := (s, e)\)

Figure 6. $\mathcal{C}$ syntax

3. The Non-Affine Sublanguage $\mathcal{C}$

Our calculus includes a second, non-affine sublanguage, which we call $\mathcal{C}$ (for “conventional”). Sublanguage $\mathcal{C}$ is a completely standard simply-typed $\lambda$ calculus augmented with modules as in $\sigma$. Figure 6 presents the syntax of types and terms, and the semantics appear in appendix A. We typeset $\mathcal{C}$ terms in a **bold serif font** to distinguish them visually from $\sigma$.

This language omits features such as references and if expressions that are inessential to the story we are telling. Such features would not be difficult to add.

4. Mixing It Up

Our goal in this work is to construct (type-safe) programs by mixing modules written in an affine language and modules written in a non-affine language, and to have them interoperate as seamlessly as possible. This might represent an affine program calling into a library written in a legacy language, or calling into a library written in an affine language from a program written in a conventional language, or both. In either case, we must ensure that the non-affine
programs \( P ::= M e \)
declarations \( M ::= \bullet \mid M m \)
modules \( m ::= \bullet \mid m \mid \text{interface } f ::= \sigma^u = g \)
\( \mathcal{A} \) expressions \( e ::= \ldots \mid f^\mathcal{A} \)
\( \mathcal{C} \) expressions \( e ::= \ldots \mid f^\mathcal{C} \)
opaque \( \mathcal{A} \) types \( \sigma^o ::= \sigma \mathcal{R} \mid \sigma \mathcal{O} \)
\( \mathcal{C} \) types \( \tau ::= \ldots \mid \sigma^o \mathcal{A} \)

Figure 7. Mixed syntax

\[
\begin{align*}
(\sigma)^\mathcal{C} &= \tau \\
(\int)^\mathcal{C} &= \text{int} \\
(\sigma_1 \approx_0 \sigma_2)^\mathcal{C} &= (\sigma_1)^\mathcal{C} \to (\sigma_2)^\mathcal{C} \\
(\sigma^o)^\mathcal{C} &= \sigma^o \mathcal{A}
\end{align*}
\]

\[
(\tau_1 \to \tau_2)^\mathcal{C} = (\tau_1)^\mathcal{C} \approx_0 (\tau_2)^\mathcal{C} \\
(\sigma^o \mathcal{A})^\mathcal{C} = \sigma^o
\]

Figure 8. Mixed static semantics (type conversions)

portions of the program cannot break the affine portions’ invariants. We accomplish this via run-time checks in the style of higher-order contracts (Findler and Fellesen 2002).

The additional syntax for mixed programs is shown in Figure 7. The main expression in a mixed program is in the \( \mathcal{C} \) language, though this decision is arbitrary, and \( \mathcal{A} \) would work just as well. Modules now include \( \mathcal{A} \) modules, \( \mathcal{C} \) modules, and interface modules, which are used to ascribe an \( \mathcal{A} \) type to a \( \mathcal{C} \) module. We discuss interface modules in more detail in §4.1.

We add a production to each language’s expressions allowing it to refer to modules from the other language. We decorate each such module name with the name of the module in which it statically appears (e.g., \( f^\mathcal{C} \) for a reference to \( \mathcal{C} \) module \( f \) from \( \mathcal{A} \) module \( g \)) and use this name as the negative party in contracts regulating the inter-language interaction. This is not necessary for soundness, but we use it to assign blame. In an implementation this annotation would not appear in source programs, as it is easily computed by a compiler.

We define a new syntactic class of types for the \( \mathcal{A} \) language: Opaque types \( (\sigma^o) \) do not translate into existing types in the \( \mathcal{C} \) language. Rather, values of opaque type appear as opaque values if they flow into \( \mathcal{C} \), and all \( \mathcal{C} \) may do is shuffle them around or return them back into \( \mathcal{A} \). To \( \mathcal{C} \) we add a new type constructor \( \mathcal{A} \) that is used to refer to opaque \( \mathcal{A} \) types in \( \mathcal{C} \).

4.1 Static Semantics

The type system for the mixed calculus is the union of the type systems for \( \mathcal{A} \) and \( \mathcal{C} \), along with additional typing rules that describe how to type module invocations in \( \mathcal{A} \) expressions and \( \mathcal{C} \) module invocations in \( \mathcal{A} \) expressions.

Rule TC-MODA (Figure 9) is used to type \( \mathcal{A} \) modules used in \( \mathcal{C} \) expressions. The rule uses a function \( \langle \cdot \rangle^\mathcal{C} \), defined in Figure 8, to convert the \( \mathcal{A} \) module’s type into the \( \mathcal{C} \) type for the expression. Intuitively, \( \mathcal{A} \) types are richer than \( \mathcal{C} \); in particular, \( \mathcal{A} \) has two constructors for function types whereas \( \mathcal{C} \) has only one. As a consequence, for any \( \mathcal{A} \) type \( \sigma \), the only reasonable view from \( \mathcal{C} \) is \( (\sigma)^\mathcal{C} \).

For example, suppose that some \( \mathcal{A} \) module \( g \) has the type \( \approx_0 \text{int} \perp_0 \text{int} \); that is, \( g \) is an unlimited-use function that, given an integer, returns a one-shot function. By the conversion rule, a \( \mathcal{C} \) expression that refers to \( g \) will see it with \( \mathcal{C} \) type \( \text{int} \to \text{int} \to \text{int} \), which means that \( \mathcal{C} \)’s type system cannot enforce that the functional result of partially applying \( g \) be used at most once. We will need to insert a dynamic check.

For \( \mathcal{C} \) modules used in \( \mathcal{A} \) expressions, we use rule TA-MODC. Given a \( \mathcal{C} \) type \( \tau \), any \( \mathcal{A} \) type \( \sigma \) such that \( (\sigma)^\mathcal{C} = \tau \) may be a reasonable view from \( \mathcal{A} \). The type conversion function used by rule TA-MODC, \( \langle \cdot \rangle^\mathcal{A} \), is a conservative choice:

- If \( f : \text{int} \to \text{int} \) in \( \mathcal{C} \), then \( f^\mathcal{A} : (\text{int} \approx_0 \text{int}) \approx_0 \text{int} \) is the right type in \( \mathcal{A} \).
- There is no reason to limit \( f \) to a one-shot function type because subsumption means we can use it where a one-shot function is allowed if necessary.
- If \( f : (\text{int} \to \text{int}) \to \text{int} \) in \( \mathcal{C} \), then \( f^\mathcal{A} : (\text{int} \approx_0 \text{int}) \approx_0 \text{int} \) will allow the imported function to be passed unlimited functions but not one-shot functions. This is a safe choice, because \( \mathcal{C} \)’s type system cannot tell whether it is safe to pass a one-shot function to \( f \).

In the latter case, where \( f : (\text{int} \to \text{int}) \to \text{int} \) in \( \mathcal{C} \), what if the programmer knows, perhaps by inspecting its code, that function \( f \) applies its argument at most once? For example, it will not cause a problem to pass a one-shot function to the \( \mathcal{C} \) function \( \lambda x. \text{int} \to x \cdot [0] \), but \( \mathcal{A} \)’s subtyping cannot safely allow this in general.

To this end, we introduce interface modules (Figure 7) to allow explicit assertions that a \( \mathcal{C} \) value conforms to a particular \( \mathcal{A} \) type. A module interface \( f ::= \sigma^u = g \) means that \( f \) is defined to have the same value as \( g \), but additionally promises to behave as if it had \( \mathcal{A} \) type \( \sigma^u \). By rule TA-MODI, \( \mathcal{A} \) expressions then see invocations of the \( f \) at its asserted type \( \sigma^u \). We do not take its word for it, however. Statically, rule TM-I checks that the declared \( \mathcal{A} \) type converts to the actual \( \mathcal{C} \) type. This ensures that we get integers where integers are expected and functions where functions are expected. This of course does not suffice—we also need to insert a run-time check that the \( \mathcal{C} \) function actually behaves as the interface claims.

In an indication that \( \mathcal{A} \) is in some sense stronger than \( \mathcal{C} \), we observe that \( \mathcal{C} \) types embedded into \( \mathcal{A} \) return faithfully to \( \mathcal{C} \):
\( \mathcal{E} \) expressions: \( e ::= \cdots \mid \sigma \mathcal{A} \ell (e) \)

\( \mathcal{C} \) expressions: \( e ::= \cdots \mid \mathcal{C} \sigma \ell (e) \mid \blame f \)

\( \mathcal{E} \) values: \( v ::= \cdots \mid \mathcal{C} \sigma \ell (v) \)

\( \mathcal{C} \) values: \( v ::= \cdots \mid \mathcal{C} \sigma \ell (v) \)

\[\begin{align*}
\text{Lemma 4.1} & \quad \text{(Type Conversion). For any } \mathcal{A} \text{ type } \tau, (\tau)^\mathcal{C} \in \mathcal{C} = \tau.
\end{align*}\]

The other direction does not hold, since \( (\cdot)^\mathcal{C} \) is not injective.

## 4.2 Operational Semantics

To express the intermediate states reached by our reduction semantics, we extend the syntax of our language to include several new forms (figure 10). This is required because, while our program syntax segregates the two sublanguages into separate modules, a module invocation should reduce to the body of the module, which leads expressions of both sublanguages to nest arbitrarily at run time. Rather than simply allow \( \mathcal{A} \) expressions to appear directly in \( \mathcal{C} \) expressions, and vice versa, we need a way to cordon off expressions to appear directly in \( \mathcal{E} \) expressions. Some contracts, for example affine, then the negative party promises to use the resulting value properly. In particular, if the contract is affine, then the negative party promises to use the resulting value at most once. This is not checked statically in the \( \mathcal{C} \) language.

The right subscript of a boundary—a module name in the inner language—is the positive party: It promises that if the enclosed subexpression reduces to a value, then the value will obey the contract \( \sigma \). The left subscript is the negative party, which promises to treat the resulting value properly. In particular, if the contract is affine, then the negative party promises to use the resulting value at most once. If an affine type appears in a negative portion of the contract, then this imposes an obligation on the positive party.

Inter-language boundaries first arise when a module in one language refers to a module in the other language. When the name of a \( \mathcal{C} \) module appears in an \( \mathcal{A} \) term, A-MOD\( \mathcal{C} \) wraps the module name with an \( \mathcal{A} \) boundary, with the \( \mathcal{A} \)-conversion of the module’s \( \tau \) type as the contract. The (outer) \( \mathcal{A} \) module is the negative party and the (inner) \( \mathcal{C} \) module is the positive party. When the name of an interface module appears, the contract is as declared by the interface, and the name of the interface is the positive party (A-
be treated specially.

A

assign blame reasonably, only

as a trivial corollary to our soundness theorem, provided that we

Therefore, we add the rule $A$-CTX$^t$ for reducing an $A$ subexpression in a $C$ configuration.

Eventually, the expression under a boundary may reduce to a

culture. This structure means that it is now possible

for boundaries should cancel out and the value appear unwrapped

to become unwrapped a second time.

Other copies of the value may remain sealed on the

C

but only once!

function returned by

ap

other hand, closes over the affine argument

module bodies, has no free variables. The inner function, on the

other hand, closes over the affine argument $f$, which is why the

function returned by ap is given an affine type by TA-LAMAFF.

Function inc takes an int, and then applies ap to an unlimited int $\bowtie$ int function and the given integer. By TA-SUBSUME, the int $\bowtie$ int function may be used at the stronger affine type int $\bowtie$ int, so inc types as well.

Finally, the main expression (given party name main), applies inc to an integer.

This program produces the value [0]. During reduction, the integer argument to inc passes transparently through an AC boundary, the two $\forall$ modules compute the result entirely within $\forall$, and finally the result passes back through a CA boundary in the last reduction step.

An Ill-Typed $\forall$ Module. Consider now a slight modification to module inc:

module inc2 : int $\bowtie$ int =
  $\lambda$y: int.
  let g : int $\bowtie$ int = ap ($\lambda$z: int. z + [1]) in
  g (g y)

Unlike the previous case, the new program does not type, because $\forall$ cannot type inc2. In particular, because the affine variable g is applied twice, we have a type error: There is no type $\sigma$ such that the judgment

$\forall y : int. g : int \bowtie int f \stackrel{M}{\forall f} (g y) : \sigma$

is provable in $\forall$, because there is no way to split the type environment in order to type g in both subexpressions of the application expression.

A Blameworthy $\forall$ Module. On the other hand, if we rewrite inc2 in $\forall$, the type system has no problem with it:

module ap : (int $\bowtie$ int) $\bowtie$ int =
  $\lambda$f: int $\bowtie$ int.
  $\lambda$x: int.
  f x

module inc2 : int $\rightarrow$ int =
  $\lambda$y: int.
  let g : int $\rightarrow$ int = ap$^{inc2}$ ($\lambda$z: int. z + [1]) in
  g (g y)

inc2 [5]

This program types because

$$((int \bowtie int) \bowtie int) \forall = (int \rightarrow int) \rightarrow int \rightarrow int,$$

which is the type that $\forall$ sees for ap. While $\forall$'s type system cannot detect the problematic reuse of g, the second application of g is prevented at run time by the contract on ap, which assigns blame

4.3 Examples

A Type-Correct, Blame-Free Program. The following small program consists of two $\forall$ modules and a main expression (in $\forall$) that invokes one of them:

module ap : (int $\bowtie$ int) $\bowtie$ int =
  $\lambda$f: int $\bowtie$ int.
  $\lambda$x: int.
  f x

module inc : int $\bowtie$ int =
  $\lambda$y: int. ap ($\lambda$z: int. z + [1]) y

inc$^{main}$ [5]

Function ap takes a one-shot function and an integer and applies the function to the integer; this types in $\forall$. By rule TA-LAMROM, ap is given an unlimited function type, since it, like all well-typed module bodies, has no free variables. The inner function, on the other hand, closes over the affine argument $f$, which is why the function returned by ap is given an affine type by TA-LAMAFF.

Function inc takes an int, and then applies ap to an unlimited int $\bowtie$ int function and the given integer. By TA-SUBSUME, the int $\bowtie$ int function may be used at the stronger affine type int $\bowtie$ int, so inc types as well.

Finally, the main expression (given party name main), applies inc to an integer.

This program produces the value [0]. During reduction, the integer argument to inc passes transparently through an AC boundary, the two $\forall$ modules compute the result entirely within $\forall$, and finally the result passes back through a CA boundary in the last reduction step.
Our run of the program revealed a bug in inc2, so we know that we need to repair it:

\[
\text{module inc2 : int \rightarrow int =}
\]

Now the program runs without attempting to apply a one-shot function twice, and thus it computes \([7]\) with no trouble.

---

**Figure 11.** Mixed language reduction sequence

to the negative party, inc2. See figure 11 for an example reduction sequence for this program.

The \(\mathcal{C}\) Module, Corrected. Our run of the program revealed a bug in inc2, so we know that we need to repair it:
A Call from $\mathcal{A}$ into $\mathcal{C}$. We have seen how $\mathcal{C}$ code can call into $\mathcal{A}$ code, but the reverse is also true:

\[
\text{module } \text{ap} : (\text{int} \to \text{int}) \to \text{int} \to \text{int} = \\
\lambda f : \text{int} \to \text{int}. \\
\lambda x : \text{int}. \\
\Gamma x \\
\text{module } \text{inc} : \text{int} \cong \text{int} = \\
\lambda y : \text{int}.\text{ap}^\approx (\lambda z : \text{int}. z + [1]) y \\
\text{inc}^\approx [5]
\]

This program types, and in particular, inc types, because the function it passes to ap has unlimited type int $\cong$ int rather than an affine type.

An Ill-Typed $\mathcal{A}$-to-$\mathcal{C}$ Call. However, if we explicitly give that function an affine type (by subsumption), then $\mathcal{A}$’s type system rejects the module:

\[
\text{module } \text{inc} : \text{int} \cong \text{int} = \\
\lambda y : \text{int}. \\
\text{let } g : \text{int} \cong \text{int} = \lambda z : \text{int}. z + [1] \text{ in } \\
\text{ap}^\approx g y
\]

The problem above is that $\mathcal{A}$ sees ap at type

\[
\left( (\text{int} \to \text{int}) \to \text{int} \to \text{int} \right)^{\mathcal{A}} = (\text{int} \cong \text{int}) \cong \text{int} \cong \text{int},
\]

and $\mathcal{A}$ does not allow passing a one-shot function where an unlimited function is required.

An Interface Module Intervenes. This is where interfaces are useful, since they allow us to assert that ap actually conforms to the desired $\mathcal{A}$ type:

\[
\text{module } \text{ap} : (\text{int} \to \text{int}) \to \text{int} \to \text{int} = \\
\lambda f : \text{int} \to \text{int}. \\
\lambda x : \text{int}. \\
\Gamma x \\
\text{interface } \text{iap} : > (\text{int} \cong \text{int}) \cong \text{int} \cong \text{int} = \text{ap} \\
\text{module } \text{inc} : \text{int} \cong \text{int} = \\
\lambda y : \text{int}. \\
\text{let } g : \text{int} \cong \text{int} = \lambda z : \text{int}. z + [1] \text{ in } \\
\text{iap}^\approx g y \\
\text{inc}^\approx [5]
\]

When type checking inc, we provisionally believe the contract applied to iap, so its use type checks. In this case, ap indeed behaves as promised. If, however, we modify ap to apply f twice, then blame accrues to iap at run time.

5. Type Soundness

The presence of strong updates in our calculi makes it easy to write programs, such as the example with tricky in §2.1, that get stuck, in the sense that they reach a state that is not an answer configuration but cannot take a step. Our soundness criterion is that no such program can be assigned a type.

Some similar linear systems, such as L, use a semantic notion of soundness that involves the interpretation of types as sets of pairs of store fragments and values that have exclusive access to those fragments (Ahmed et al. 2004). In their successor language $\lambda_{\text{LAPP}}$, Ahmed et al. (2005) instrument values in order to detect when relevant and linear values would be implicitly discarded. Since our affine language does not purport to protect against discarding, that kind of machinery is unnecessary here. Because strong updates are sufficiently capable of leading to stuck configurations, we are satisfied with a syntactic soundness theorem that well-typed programs do not reach such configurations. Our syntactic soundness theorem, formalized below, says that if a program $M \vdash e$ type checks, then either $(s, e)$ diverges, or it reduces to an answer of the same type as $M \vdash e$.

In order to prove a Wright-Felleisen-style type soundness theorem (1994), it is necessary to identify precisely what property is preserved in a preservation lemma. Because we reduce configurations rather than programs, we need a type judgment for configurations, and because our expressions now contain locations, we need a store environment ($\Sigma$) mapping locations to types. Augmenting our existing expression type rules with a store environment is not enough, however. The expression rules for $\mathcal{C}$, naively extended, are too weak to express the property that we need. Consider, for example, a hypothetical typing judgment:

\[
\Sigma; h : \text{int} \cong \text{int} \cong \text{int} \\
\vdash M \Gamma \left[ \text{CA}^\left\langle \text{inc}^\approx \right\rangle \left[ \text{h (h [2])} \left[ \text{CA}^\left\langle \text{inc}^\approx \right\rangle \left[ \text{h [3]} \right] \left[ 4 \right] \right] \right] : \text{int} \right].
\]

If we use TC-APP, which duplicates the type environment to both subterms, there might be a derivation of this judgment. We do not want this to be the case. The term being typed above is liable to go wrong, as both applications under CA boundaries will evaluate before any contract has an opportunity to say otherwise. Such a configuration is not reachable from any well-typed program, so preservation requires type judgments for run-time terms that demonstrate that fact.

To this end, our soundness theorem relies on an auxiliary type system for configurations with mixed terms. The auxiliary type system has judgments of the form $\Sigma; \Gamma \vdash e : \tau$, and it is careful about environment splitting even in rules for $\mathcal{C}$ terms:

\[
\text{(RTC-APP)} \\
\Sigma_1; \Gamma_1 \vdash M \left[ \text{e}_1 : \tau' \right] \rightarrow \tau \\
\Sigma_2; \Gamma_2 \vdash M \left[ \text{e}_1 \right] : \tau \\
\Sigma_1 \boxplus \Sigma_2; \Gamma_1 \boxplus \Gamma_2 \vdash \left[ \text{e}_2 : \tau \right] : \tau
\]

We emphasize that this auxiliary type system is merely a proof technique that generalizes our type system enough to state our preservation invariant. In no way does it require that typing $\mathcal{C}$ code in source programs be aware of context splitting and other substructural trivia. It does, however, indicate why programs in our language require language boundaries between modules rather than arbitrarily nested expressions as in Matthews and Findler (2007): Arbitrary nesting means that environments with bindings for $\mathcal{A}$ may need to be managed by the type system for $\mathcal{C}$, which forces $\mathcal{C}$ to become as complex as $\mathcal{A}$. Selected rules from the auxiliary type system may be found in appendix B.

Additional machinery is required to deal with blessed boundaries, which allow locations to be duplicated temporarily and scattered through a $\mathcal{C}$ term.

Our full set of run-time type rules in hand, we prove a lemma that well-typed programs produce well-typed configurations:

Lemma 5.1 (Programs to Configurations).

- If $\Gamma \vdash M \left[ \text{e} : \tau \right] \rightarrow \tau$ then $\Gamma \vdash M \left[ \text{e} : \tau \right] : \tau$.
- If $\Gamma \vdash M \left[ \text{e} : \sigma \right] \rightarrow \tau$ then $\Gamma \vdash M \left[ \text{e} : \sigma \right] : \tau$.
- If $\Gamma \vdash M \left[ \text{e} : \tau \right] \rightarrow M \left[ \text{e} : \tau \right] : \tau$.

Configurations now enjoy standard progress and preservation lemmas under the run-time type judgments, which allows us to state and prove a syntactic type soundness theorem.

Definition 5.2 (Divergence). A configuration $\text{C}$ diverges in $M$ if for all $\mathcal{C}'$ such that $\text{C} \rightarrow_{\delta M} \text{C}'$, there exists a $\mathcal{C}''$ such that $\text{C}' \rightarrow_{\delta M} \text{C}''$.

Theorem 5.3 (Type Soundness). If $\Gamma \vdash M \left[ \text{e} : \tau \right]$, then either $(s, e)$ diverges in $M$, or there is some answer $a$ such that $(s, e) \rightarrow_{\delta M} a$ and $\Gamma \vdash M \left[ \text{a} : \tau \right]$.
6. Discussion and Future Work

The work presented here is part of an ongoing program to investigate practical aspects of substructural type systems. Our answers naturally raise more questions.

Exceptions. In a production language with a contract system, contract violations should not always terminate the program. Real programs may catch an exception and either try to mitigate the condition that caused it, try something easier instead, or report an error and go on with some other task. To ensure soundness, it suffices to prevent the questionable actions from occurring.

On one hand, we believe that ML-style exceptions should not provide too much difficulty in an affine setting. We expect that handle expressions should be multiplicative, in the sense that the type environment needs to be split between an expression and its exception handler, not given in whole to both.

On the other hand, we do not know how exceptions or any sort of blame might work in a linear setting—this is one reason why we chose an affine rather than linear calculus. Terminating the program is dodgy because of the implicit discarding of linear values, but catching an exception once values on the stack have been lost seems even worse. Exceptions in linear languages remain a very open question.

Linearity. Our work emphasizes contract-based interaction with affine type systems rather than linear type systems because it remains unclear to us what linear contracts ought to mean. We may want a conventional language to interoperate with a language that (at least sometimes) prohibits discarding values. However, as Ahmed et al. (2005) point out, detecting illicit discarding may require special instrumentation. Unlike affine guarantees, which correspond to safety properties, relevance guarantees—that a value is used at some point in the future—are a form of liveness property (Alpern and Schneider 1987), a violation of which may result only in long-term resource leaks.

Conventional contracts regulate only sins of commission: when a party agrees to a conventional contract, it essentially agrees not to do something, such as not passing a non-integer to a particular function, or not returning anything other than a string.3 If the party ever does what it agreed not to do, that constitutes a contact violation. Likewise, for affine contracts, a party agrees not to use some value more than once; if it ever does, then a violation is reported. When exactly should we report a violation when a party takes on the positive obligation to do something?

Perhaps the most conservative answer is this: when a party assumes responsibility to use a value, the party is obliged not to allow the program to terminate before using the value. This may seem rather pointless: The program is about to exit, but we detect a contact violation and thus terminate the program instead! While it is too late to prevent a resource leak, it is not too late to file a bug report.

With such an interpretation of linear contracts, we could reasonably consider the contract to be violated if at any point we can determine that the contract necessarily will be violated. Detecting the violation of such a liveness property is undecidable for any nontrivial language, but tracing garbage collection may be used to detect violations of linearity approximately and somewhat late. In an idealized semantics, we might garbage collect the store after each reduction step and signal a violation if the seal location of a not-yet-used linear value has become unreachable. In a real implementation, finalizers on linear values could detect sins of omission. Even if we detect a violation early, however, it is not clear that we can do anything to prevent it.

3For conventional contracts, non-termination is an acceptable outcome.

\[
\begin{align*}
\varphi_{\lambda}[x]^r_{g} &= x \\
\varphi_{\lambda}[x]^r_{g} \cdot \sigma &= \lambda y:(\sigma_{1}) \cdot \mathbf{let~} z = x \cdot \sigma_{1} \cdot \mathbf{in} \varphi_{\lambda}[z]^r_{g} \\
\varphi_{\lambda}[x]^r_{g} \cdot \sigma &= \lambda y:1 \cdot \mathbf{let~} u = \mathbf{new} [0] \mathbf{in} \\
&\quad \lambda y:1 \cdot \mathbf{if}0 \mathbf{(}u\mathbf{)} \mathbf{in} \\
&\quad \mathbf{let} z = x \cdot \sigma_{1} \cdot \mathbf{in} \varphi_{\lambda}[z]^r_{g} \\
&\quad \mathbf{blame} f \\
\varphi_{\lambda}[x]^r_{g} &= \lambda y:1 \cdot \mathbf{let~} u = \mathbf{new} [0] ~ \mathbf{in} \\
&\quad \lambda y:1 \cdot \mathbf{if}0 \mathbf{(}u\mathbf{)} \mathbf{in} \\
&\quad \mathbf{let} z = x \cdot \sigma_{1} \cdot \mathbf{in} \varphi_{\lambda}[z]^r_{g} \\
&\quad \mathbf{blame} f \\
\end{align*}
\]

Figure 12. Sketch of a type-directed translation

Implementation. We believe that our technique admits a reasonable implementation strategy, given an existing language implementation standing in for \( \lambda \) and a similar but affine language for \( \lambda_{\alpha} \). Significantly, the \( \lambda \) implementation need not be modified, and \( \lambda_{\alpha} \) does not require access to the sources of \( \lambda \) modules in a mixed program. We require that \( \lambda \) support references in order to track blessed versus defunct values, and that it have some mechanism for signaling blame.

We type check the whole program in the mixed type system, and then translate the \( \lambda_{\alpha} \) modules to \( \lambda \), renaming and wrapping so that \( \lambda_{\alpha} \) modules access each other directly, but only communicate with \( \lambda \) modules through wrappers implementing the necessary contracts. Finally, we compile the whole program with an existing \( \lambda \) compiler.

Matthews and Findler (2007) show how to implement the contracts in their multi-language system as checks and error signaling written in their object language. For us, the key is replacing the C-BLESS reduction rule with an appropriate translation for boundaries. A sketch of a potential type-directed translation, which assumes several additional but uncontroversial features in \( \lambda \), may be found in figure 12. When an \( \lambda_{\alpha} \) module \( g \) with type \( \sigma \) appears in a \( \lambda \) module \( f \), we translate it as \( \varphi_{\lambda}[g]^r_{g} \). Likewise, when a \( \lambda \) module \( g \) having \( \lambda_{\alpha} \)-type \( \sigma \) appears in an \( \lambda_{\alpha} \) module \( g \), we translate it as \( \varphi_{\lambda}[f]^r_{g} \).

We conjecture that a translated program faithfully simulates the original program.

Affine Contracts without Affine Types. The dynamic properties enforced by affine contracts may be useful even in a language that lacks affine types. Affine contracts can provide a conventional typed language or an untyped language a way to specify formally and then check resource usage invariants. We believe an implementation along the lines of the above sketch could easily be added to some existing contract system such as PLT Scheme’s (Findler and Felleisen 2002; Flatt 2007).

Our calculus can express affine contracts directly between \( \lambda \) modules by a simple translation. Assume we have a \( \lambda \) module,
module f : τ = v, to which we wish to add the contract σ₁ \(\bowtie\) σ₂, where \(\sigma_1 \bowtie \sigma_2\) \(\bowtie\) = \(\tau\). We simply rename \(f\) and wrap it with an interface having the contract, and wrap that in turn with a \(\mathcal{A}\) module having the original name:

\[
\begin{align*}
\text{module } f_u : \tau & = v \\
\text{module } f_l : \sigma_1 \bowtie \sigma_2 & = f_u \\
\text{module } f & : \sigma_1 \rightarrow \sigma_2 = \lambda x : \sigma_1, f_l^x
\end{align*}
\]

The \(\mathcal{A}\) module is necessary here only because our system does not place contract boundaries between same-language modules; it is not difficult to lift this restriction and provide a more direct implementation of affine contracts in a conventional language.

Dynamic Promotion Several linear type systems offer some way for linear values to become temporarily unlimited. In \(\mathbb{L}^3\), rather roughly, reference cells may be frozen, which makes them aliasable but allows only weak updates; provided a suitable witness that a frozen cell is not aliased, it may be thawed to provide strong updates again (Ahmed et al. 2004).\(^4\) Wadler (1990) introduces left, which temporarily treats a single-threaded, mutable value as unlimited but read-only.

Affine contracts point to another way to loosen the duplication restriction. We allow one-shot functions to be provisionally promoted to unlimited functions:

\[
\Gamma \vdash_M \text{promote} e : \sigma_1 \bowtie \sigma_2 \\
\Gamma \vdash_M f : \tau
\]

At run time, promote must wrap its argument in a contract that ensures it is actually applied at most once.

In fact, promote (for a given \(\sigma_1\) and \(\sigma_2\)) is definable in our calculus:

\[
\begin{align*}
\text{module promote'} : (\sigma_1 \bowtie \sigma_2) \rightarrow (\sigma_1 \bowtie \sigma_2) = \lambda x. (\sigma_1 \bowtie \sigma_2), x \\
\text{module promote'} & = \text{promote'}
\end{align*}
\]

7. Conclusion

This paper describes one step in our research program to study substructural type systems for practical programming. Here, we have focused on the problem of interaction between substructural and non-substructural code, each governed by its own type system.

Our solution synthesizes two previously disparate technologies, substructural type systems and higher-order contracts. We have illustrated our solution through a calculus that combines modules written in an affine-typed language and modules written in a conventionally typed language. The two languages call one another and exchange values freely, using contracts to prevent the conventional language from breaking the affine language’s stronger invariants. A soundness theorem validates the correctness of our approach.

Our work suggests that adding the ability to write substructural libraries in a conventional programming language such as ML, Haskell, or Scheme does not require a particularly complicated implementation, and our results yield a realistic contract-based implementation design.

Acknowledgments

We greatly appreciate Sam Tobin-Hochstadt’s help in thinking about contracts and inter-language systems. We wish to thank Daniel Brown, Ryan Culpepper, Jed Davis, Alec Heller, and Aaron Turon for their helpful comments and corrections.

\(^4\)In reality, \(\mathbb{L}^3\) distinguishes reference cells from capabilities to access those cells. Only the capabilities are linear, but our characterization applies if we treat cells and their capabilities as a single entity.

A. Semantics of \(\mathcal{C}\)

\[
\begin{align*}
\text{(C-\(\beta\))} & \quad (s, (\lambda x : \tau, e) v) \rightarrow (s, e[v/x]) \\
\text{(C-MOD)} & \quad (s, f) \rightarrow (s, v) \\
\text{(C-CTX)} & \quad (s, e) \rightarrow (s, e')
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash_M \text{promote} e : \tau & = \text{okay} \\
\Gamma \vdash_M \text{promote} e : \tau & = \text{okay}
\end{align*}
\]

B. The Auxiliary Type System

As explained in §5, we use an auxiliary type system for proving the soundness theorem. We present selected portions of that system here.

A store type maps locations to \(\mathcal{C}\) types, \(\mathcal{A}\) types, and sealed \(\mathcal{A}\) types, which are used to type locations that appear under bounded boundaries.

\[
\text{store types } \Sigma ::= \bullet | \Sigma, \ell : \tau | \Sigma, \ell : \sigma | \Sigma, \ell : [\sigma]^\ell
\]

Environment splitting \((\Gamma \oslash \Gamma = \Gamma)\) is extended to treat \(\mathcal{C}\) types as duplicable, and store type splitting \((\Sigma \oslash \Sigma = \Sigma)\) is defined to duplicate \(\mathcal{C}\) types and sealed \(\mathcal{A}\) types but not ordinary \(\mathcal{A}\) types.

To type the store, we type each element in it. Locations holding \(\mathcal{A}\) values may be given either sealed or ordinary \(\mathcal{A}\) types.

\[
\begin{align*}
\Sigma \vdash_M \text{M} s : \Sigma \\
\text{(TH-EMPTY)} & \quad \Sigma_1 \vdash_M \text{M} s : \Sigma' | \Sigma_2 : \bullet \vdash_M v : \tau \\
\Sigma \vdash_M \bullet : \bullet & \quad \Sigma_1 \oslash \Sigma_2 \vdash_M (s, \ell \mapsto v) : (\Sigma', \ell : \tau)
\end{align*}
\]

\[
\begin{align*}
\text{(TH-ALOC)} & \quad \Sigma_1 \vdash_M \text{M} s : \Sigma' | \Sigma_2 : \bullet \vdash_M v : \sigma \\
\Sigma_1 \oslash \Sigma_2 \vdash_M (s, \ell \mapsto v) : (\Sigma', \ell : \sigma)
\end{align*}
\]

\[
\begin{align*}
\text{(TH-ALLOC)} & \quad \Sigma_1 \vdash_M \text{M} s : \Sigma' | \Sigma_2 : \bullet \vdash_M v : \sigma \\
\Sigma_1 \oslash \Sigma_2 \vdash_M (s, \ell \mapsto v) : (\Sigma', \ell : [\sigma]^\ell)
\end{align*}
\]
To type a configuration, all the modules must be okay; then we type the store, and whatever part of the store type is not used in typing the store itself (as locations may contain other locations), we use in typing the main expression:

\[ □^M C : Τ \]

\[ (T\text{-CONF}) \]

\[ ∀m ∈ M. □^M m \text{ okay} \]

\[ Σ_1 □^M s_1 : Σ_1 ⊗ Σ_2 \quad Σ_2 ; □^M e : Τ \]

\[ □^M (s, e) : Τ \]

\[ Σ_1; Γ □^M e : σ \quad \text{and} \quad Σ_1; Γ □^M e : Τ \]

\[ (RTA-LOC) \]

\[ Σ, ℓ : σ; Γ □^M e \quad (σ)^{ref} \]

\[ Σ_1; Γ □^M \text{ blame } e : Τ \]

\[ (RTC-BLAIME) \]

\[ Σ, ℓ : σ; Γ □^M e \quad (σ)^{ref} \]

\[ Σ_1; Γ □^M e : σ \]

\[ Σ_1; Γ □^M CA^σ(e) : (σ)^{ref} \]

\[ (RTC-BLONED) \]

\[ Σ, ℓ : ty^e(\text{BLSSD}); Γ □^M CA^σ(e) : (σ)^{ref} \]

\[ (RTC-DEFUNCT) \]

\[ [Σ]^{ref} = Σ \]

\[ Σ, ℓ : ty^e(\text{DFNC}); Γ □^M CA^σ(e) : (σ)^{ref} \]

\[ (RTC-LAM) \]

\[ Σ, Γ, x : Τ □^M e : Τ \quad \text{worthy} \]

\[ Σ; Γ □^M λx:Τ. e \quad \text{worthy} \]

\[ (RTA-LAMROMP) \]

\[ Σ, Γ, x : σ □^M e : σ' \quad \text{worthy} \]

\[ Σ; Γ □^M λx:σ. e \quad \text{worthy} \]

\[ (RTA-LAMAP) \]

\[ Σ, Γ, x : σ □^M e : σ' \quad \text{worthy} \]

\[ Σ; Γ □^M λx:σ. e \quad \text{worthy} \]

\[ (WORTHY-A) \]

\[ [σ]^f = [σ] \quad \text{and} \quad [σ']^f = [σ'] \]

\[ (WORTHY-C) \]

\[ [σ', σ]^[f] = [σ, σ']^[f] \quad \text{and} \quad [σ', σ]^[f] = [σ, σ']^[f] \]

References


A PLT Redex model of our calculus, including an algorithmic variant of the type system, may be found at http://www.ccs.neu.edu/~tov/substructural/affine-contracts.ss.