Stateful Contracts for Affine Types[*]

Jesse A. Tov and Riccardo Pucella

Northeastern University, Boston, MA 02115, USA
{tov,riccardo}@ccs.neu.edu

Abstract. Affine type systems manage resources by preventing some values from being used more than once. This offers expressiveness and performance benefits, but difficulty arises in interacting with components written in a conventional language whose type system provides no way to maintain the affine type system's aliasing invariants. We propose and implement a technique that uses behavioral contracts to mediate between code written in an affine language and code in a conventional typed language. We formalize our approach via a typed calculus with both affine-typed and conventionally-typed modules. We show how to preserve the guarantees of both type systems despite both languages being able to call into each other and exchange higher-order values.

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 [20]. Linear type systems [19, 11, 3, 1], for example, ensure that values with linear type cannot be duplicated or dropped, but must be eliminated exactly once. Other substructural type systems refine these constraints. Affine type systems, which we consider here, prevent values from being duplicated but allow them to be dropped: a value of affine type may be used once or not at all.

Affine types are useful to support language features that rely on avoidance of aliasing. One example is session types [6], which are a method to represent and statically check communication protocols. Suppose that the type declared by

$$\textbf{type}_{\mathscr{A}}\ \mathsf{prot} = (\mathsf{int\ send} \to \mathsf{string\ recv} \to \mathsf{unit})\ \mathsf{chan} \tag{1}$$

represents a channel whose protocol allows us to to send an integer, then receive a string, and finally end the session. Further, suppose that *send* and *recv* consume a channel whose type allows sending or receiving, as appropriate, and return a channel whose type is advanced to the next step in the protocol. Then we might write a function that takes two such channels and runs their protocols in parallel:

```
let_𝒜 twice (c1: prot, c2: prot, z: int): string ⊗ string =
  let once (c: prot) (_: unit) =
    let c      = send c z in
    let (s, _) = recv c    in s
  in (once c1) ||| (once c2)                                    (2)
```

[*] Our prototype implementation and the full details of our soundness theorem may be found at http://www.ccs.neu.edu/~tov/pubs/affine-contracts/.

The protocol is followed correctly provided that *c1* and *c2* are *different* channels. Calling *twice*(*c*, *c*, 5), for instance, would violate the protocol. An affine type system can prevent this.

In addition to session types and other forms of typestate [15], substructural types have been used for memory management [8], for optimization of lazy languages [18], and to handle effects in pure languages [2]. Given this range of features, a programmer may wish to take advantage of substructural types in real-world programs. Writing real systems, however, often requires access to comprehensive libraries, which mainstream programming languages usually provide but experimental implementations often do not. The prospect of rewriting a large library to work in a substructural language strikes these authors as unappealing.

It is therefore compelling to allow conventional and substructural languages to interoperate. We envision complementary scenarios:

– A programmer wishes to import legacy code for use by affine-typed client code. Unfortunately, legacy code unaware of the substructural conditions may duplicate values received from the substructural language.
– A programmer wishes to export substructural library code for access from a conventional language. A client may duplicate values received from the library and resubmit them, causing aliasing that the library could not produce on its own and bypassing the substructural type system's guarantees.

Our Contributions. We present a novel approach to regulating the interaction between an affine language and a conventionally-typed language and implement a multi-language system having several notable features:

– The non-affine language may gain access to affine values and may apply affine-language functions.
– The non-affine type system is utterly standard, making no concessions to the affine type system.
– And yet, the composite system preserves the affine language's invariants.

We model the principal features of our implementation in a multi-language calculus that enjoys type soundness. In particular, the conventional language, although it has access to the affine language's functions and values, cannot be used to subvert the affine type system.

Our solution is to wrap each exchanged value in a software contract [4], which uses *one bit* of state to track when an affine value has been used. While this idea is simple, the details can be subtle.

Design Rationale and Background. Our multi-language system combines two sublanguages with different type systems. The $\mathscr{C}$ ("conventional") language is based on the call-by-value, polymorphic λ calculus [7, 12] with algebraic datatypes and SML-style *abstype* [10]. The $\mathscr{A}$ ("affine") language adds affine types and the ability to declare new abstract affine types, allowing us to implement affine abstractions such as session types and static read-write locks.

A program in our language consists of top-level module, value, and type definitions, each of which may be written in either of the two sublanguages. (In the example above (2), the subscripts on **type**$_{\mathscr{A}}$ and **let**$_{\mathscr{A}}$ indicate the $\mathscr{A}$ language.) Each language has access to modules written in the other language, although 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 a conventional type system. Notably, non-functional affine types appear as abstract types to the conventional type system, which requires no special knowledge about affine types other than comparing them for equality.

In our introductory example, a protocol violation occurs only if the two arguments to *twice* are aliases for the same session-typed channel, which the $\mathscr{A}$ language type system prevents. Problems would arise if we could use the $\mathscr{C}$ language to subvert $\mathscr{A}$ language's type system non-aliasing invariants. To preserve the safety properties guaranteed by each individual type system and allow the two sublanguages to invoke one another and exchange values, we need to perform run-time checks in cases where the non-affine type system is too weak to express the affine type system's invariants. Because the affine type system can enforce all of the conventional type system's invariants, we may dispense with checks in the other direction.

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, however, the value flows into a non-affine module, then static bets are off. In that case, we resort to a dynamic check that prevents the value from flowing back into an affine context more than once. Since our language is higher-order, we use a form of higher-order contract [4] to keep track of each module's obligations toward maintaining the affine invariants.

Our approach to integrating affine and conventional types borrows heavily from recent literature on multi-language interoperability [5, 13]. Our approach borrows from that of Typed Scheme [17, 16] and of Matthews and Findler [9], both of which use contracts to mediate between an untyped, Scheme-like language and a typed language.

2 Example: Taming the Berkeley Sockets API

The key feature of our system is the ability to write programs that safely mix code written in an affine-typed language and a conventionally-typed language. As an example, we develop a small networking library and application, using both of our sublanguages where appropriate.

The Berkeley sockets API is the standard C language interface to network communication [14]. Transmission Control Protocol (TCP), which provides reliable byte streams, is the standard transport layer protocol used by most internet applications (*e.g.*, SMTP, HTTP, and SSH). Setting up a TCP session using Berkeley sockets is a multi-step process (Fig. 1). A client must first create a communication end-point, called a *socket*, via the `socket` system call. It may optionally select a port to use with `bind`, and then it establishes a connection

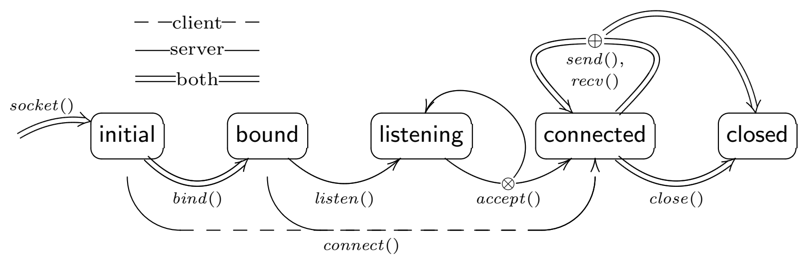

Fig. 1. States and transitions for TCP (simplified)

```
module𝒞 Socket : sig
  type socket
  val socket : unit → socket                (* ∅ ⇒ initial *)
  val bind   : socket → port → unit         (* initial ⇒ bound *)
  val listen : socket → unit                (* bound ⇒ listening *)
  val accept : socket → socket              (* listening ⇒ connected ⊗ listening *)
  val send   : socket → string → bool       (* connected ⇒ connected ⊕ closed *)
··· end
```

Fig. 2. Selected $\mathscr{C}$ language socket operations, annotated with state transitions

with `connect`. Once a connection is established, the client may `send` and `recv` until either the client or the other side closes the connection.

For a server, the process is more involved: it begins with `socket` and `bind` as the client does, and then it calls `listen` to allow connection requests to begin queuing. The server calls `accept` to accept a connection request. When `accept` succeeds, it returns a *new* socket that is connected to a client, and the old, listening socket is available for further `accept` calls. (For simplicity, we omit error transitions, except for failure of `send` and `recv`.)

Our $\mathscr{C}$ sublanguage provides the interface to sockets shown in Fig. 2. The socket operations are annotated with their pre- and post-conditions, but the implementation detects and signals state errors dynamically. For example, calling *listen* on a socket in state initial or calling *connect* on a socket that is already connected will raise an exception. If the other side hangs up, send and recv raise exceptions, but nothing in this interface prevents further communication attempts that are bound to fail.[1]

By reimplementing the sockets API in language $\mathscr{A}$, we can use language $\mathscr{A}$'s type system to move the state transition information from comments into the type system itself. For example, we give *listen* in sublanguage $\mathscr{A}$ the type

$$\forall\alpha.\ \alpha\ \mathsf{socket} \to \alpha\ \mathsf{bound} \to \alpha\ \mathsf{listening}\,, \tag{3}$$

[1] This simplifies the Berkeley sockets API by omitting address families, protocols, half-closed sockets, non-blocking IO, etc., but the stateful essence remains.

$$
\begin{array}{ll}
\textbf{val}\ \textit{socket} & : \mathsf{unit} \to \exists\alpha.\,\alpha\ \mathsf{socket} \otimes \alpha\ \mathsf{initial} \\
\textbf{val}\ \textit{bind} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \mathsf{port} \to \alpha\ \mathsf{initial} \to \alpha\ \mathsf{bound} \\
\textbf{val}\ \textit{listen} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \alpha\ \mathsf{bound} \to \alpha\ \mathsf{listening} \\
\textbf{val}\ \textit{accept} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \alpha\ \mathsf{listening} \to (\alpha\ \mathsf{listening} \otimes \exists\beta.\,\beta\ \mathsf{socket} \otimes \beta\ \mathsf{initial}) \\
\textbf{val}\ \textit{connect} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \mathsf{host} \to \mathsf{port} \to (\alpha\ \mathsf{initial} \oplus \alpha\ \mathsf{bound}) \to \alpha\ \mathsf{connected} \\
\textbf{val}\ \textit{send} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \mathsf{string} \to \alpha\ \mathsf{connected} \to \alpha\ \mathsf{connected} \\
\textbf{val}\ \textit{recv} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \mathsf{int} \to \alpha\ \mathsf{connected} \to \mathsf{string} \otimes \alpha\ \mathsf{connected} \\
\textbf{val}\ \textit{close} & : \forall\alpha.\,\alpha\ \mathsf{socket} \to \alpha\ \mathsf{connected} \to \mathsf{unit}
\end{array}
$$

Fig. 3. The $\mathscr{A}$ language sockets API

which means that given a socket and evidence that the socket is bound, *listen* changes the state to listening and returns evidence to that effect. These evidence tokens are *capabilities*, and the type parameter on each capability ties it to the particular socket whose state it describes. These capabilities have affine type so that when *listen* consumes the bound capability, we cannot call *listen* again on the same socket.

We reimplement the sockets API in language $\mathscr{A}$ in terms of the language $\mathscr{C}$ operations. From the vantage of language $\mathscr{A}$, $\mathscr{C}$ function types are mapped to $\mathscr{A}$ function types, but the $\mathscr{C}$ type Socket.socket is mapped to an *opaque type* {Socket.socket}. Type constructor $\{\cdot\}$ delimits foreign types referenced from the other sublanguage.

We declare a new abstract type for sockets in language $\mathscr{A}$, along with a type to represent each of the states:

$$
\begin{array}{rll}
\textbf{abstype}_{\mathscr{A}} & \alpha\ \mathsf{socket} = \mathsf{Sock}\ \textbf{of}\ \{\mathsf{Socket.socket}\} & \\
\textbf{and} & \alpha\ \mathsf{initial} & \textbf{qualifier A} = \mathsf{Initial} \\
\textbf{and} & \alpha\ \mathsf{bound} & \textbf{qualifier A} = \mathsf{Bound} \\
\textbf{and} & \alpha\ \mathsf{listening} & \textbf{qualifier A} = \mathsf{Listening} \\
\textbf{and} & \alpha\ \mathsf{connected} & \textbf{qualifier A} = \mathsf{Connected} \\
\textbf{with} & \cdots\ (*\ \mathsf{operations\ detailed\ below}\ *)\ \cdots\ \textbf{end} &
\end{array} \tag{4}
$$

Several aspects of this abstype declaration bear further explanation:

- Each type has a phantom parameter α, which is used to associate a socket with the type witnessing its state.
- The syntax **qualifier A** on each the state type declares that outside the abstraction boundary, values of those types will appear as affine. Code inside the abstype declaration sees that they are ordinary, non-affine data types.
- Because each of the capabilities has only one constructor with no values, they need not be represented at run time.

The $\mathscr{A}$ language sockets interface appears in Fig. 3. The $\mathscr{A}$ sockets implementation relies on delegating to $\mathscr{C}$ language functions. From within $\mathscr{A}$, $\mathscr{C}$ types are viewed through a simple translation: function types, quantified types, and a few base types such as int pass through transparently, whereas other types are

```
let_𝒜 clientLoop[α] (sock: α socket) (f: string → string) (cap: α connected) =
  let rec loop (cap: α connected): unit =
      let (str, cap) = recv sock 1024 cap in
      let cap        = send sock (f str) cap in
      loop cap
  in try loop cap with SocketError _ → ()

let interface threadFork :> (unit a⊸ unit) → {thread}_𝒞 = threadFork_𝒞

let rec_𝒜 acceptLoop[α] (sock: α socket) (f: string → string) (cap: α listening): unit =
  let (cap, Pack(β, (clientsock, clientcap))) = accept sock cap in
    threadFork (fun () → clientLoop clientsock f clientcap);
    acceptLoop sock f cap

let_𝒜 echoServe (port: int) (f: string → string) =
  let Pack(α, (sock, cap)) = socket () in
  let cap = bind sock port cap in
  let cap = listen sock cap in
    acceptLoop sock f cap
```

Fig. 4. An echo server in language $\mathscr{A}$

wrapped opaquely as Socket.socket was above. Thus, the type of $\mathit{Socket.socket}_{\mathscr{C}}$ becomes unit $\rightarrow$ {Socket.socket} when viewed from $\mathscr{A}$. Each $\mathscr{A}$ function is a minimal wrapper around its $\mathscr{C}$ counterpart:

```
let_𝒜 socket () =
  let sock = Socket.socket_𝒞 () in
    in Pack(unit, (Sock[unit] sock, Initial[unit])) as ∃β. β socket ⊗ β initial
let_𝒜 listen[α] (Sock sock as s: α socket) (_: α bound) =
  try Socket.listen_𝒞 sock;
      Listening[α]
  with IOError msg → raise (StillBound (freezeBound s cap, msg))            (5)
```

For $\mathit{socket}_{\mathscr{A}}$, we call $\mathit{Socket.socket}_{\mathscr{C}}$ to create the new socket, which we wrap in the Sock constructor and pack into an existential with a new initial capability. (The type abstracted by the existential is immaterial; unit will do.) Function $\mathit{listen}_{\mathscr{A}}$ calls its $\mathscr{C}$ counterpart on the socket and returns a listening capability tied by α to the socket. On failure, the socket is still in state bound, so it raises an exception containing the bound capability. The remaining functions are equally straightforward, but when we're done, provided we got this trusted kernel correct, we have an $\mathscr{A}$ library that enforces the correct ordering of socket operations.

Calling the various $\mathscr{C}$ socket operations from $\mathscr{A}$ is safe because none has a type that enables it to gain access to an $\mathscr{A}$ language value. Other situations are not as simple. Figure 4 shows an implementation of an echo server in language $\mathscr{A}$. (The working code is included with our prototype implementation on our web site.) The server sends back the data it receives from each client after passing it through an unspecified $\mathsf{string} \to \mathsf{string}$ function f. The main function *echoServe* creates a socket, binds it to the requested port, and begins to listen. The type system ensures that *echoServe* performs these operations in the right order, and because the capabilities have affine types, it disallows referring to any one of them more then once. Function *echoServe* calls *acceptLoop*, which blocks in *accept* waiting for clients. For each client, it spawns a thread to handle that client and continues waiting for another client. Spawning the thread is where the multi-language interaction becomes tricky.

As in other substructural type systems, $\mathscr{A}$ requires that a function be given a type whose usage (unlimited or affine) is at least as restrictive as any variable that it closes over. Thus far, we have seen only unlimited function types ($\to$), also written $\overset{\mathsf{u}}{\multimap}$. Language $\mathscr{A}$ also has affine function types, written $\overset{\mathsf{a}}{\multimap}$.

The new client capability *clientcap*, returned by *accept*, has affine type β $\mathsf{connected}$. Because the thunk for the new thread, (**fun** () $\to$ *clientLoop clientsock f clientcap*), closes over *clientcap*, it has affine type as well: $\mathsf{unit} \overset{\mathsf{a}}{\multimap} \mathsf{unit}$. This causes a problem: To create a new thread, we must pass the thunk to the $\mathscr{C}$ function $\mathit{threadFork}_{\mathscr{C}}$, whose type as viewed from $\mathscr{A}$ is $(\mathsf{unit} \to \mathsf{unit}) \to \{\mathsf{thread}\}_{\mathscr{C}}$. Such a type makes *no guarantee* about how many times $\mathit{threadFork}_{\mathscr{C}}$ applies its argument. In order to pass the affine thunk to it, we assert that $\mathit{threadFork}_{\mathscr{C}}$ has the desired behavior:

$$\textbf{let interface } \mathit{threadFork} :> (\mathsf{unit} \overset{\mathsf{a}}{\multimap} \mathsf{unit}) \to \{\mathsf{thread}\}_{\mathscr{C}} = \mathit{threadFork}_{\mathscr{C}} \qquad (6)$$

This constitutes a checked assertion that the $\mathscr{C}$ value actually behaves according to the given $\mathscr{A}$ type. This gets the program past $\mathscr{A}$'s type checker, and if $\mathit{threadFork}_{\mathscr{C}}$ attempts to apply its argument twice at run time, a dynamic check prevents it from doing so and signals an error.

The two sublanguages can interact in other ways:

– We may call $\mathit{echoServe}_{\mathscr{A}}$ from the $\mathscr{C}$ language, passing it a $\mathscr{C}$ function for f. This is safe because function f has type $\mathsf{string} \to \mathsf{string}$, and thus can never gain access to an affine value.
– We may use the $\mathscr{A}$ language sockets library from a $\mathscr{C}$ program:

```
let𝒞 sneaky () =
  let Pack(α, (sock, cap1)) = socket𝒜 () in
  let cap2 = connect𝒜 sock "sneaky.example.org" 25 cap1 in
  let cap3 = connect𝒜 sock "sneaky2.example.org" 25 cap1 in
    ...                                                      (7)
```

This program passes $\mathscr{C}$'s type checker but is caught when it attempts to reuse the $\mathsf{initial}$ capability *cap1* at run time. This misbehavior is detected because *sneaky*'s interaction with $\mathscr{A}$ is mediated by a behavioral contract.

3 Implementing Stateful Contracts

In Findler and Felleisen's formulation [4], a 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 for treating 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 nontrivial properties are 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.

In our language, the parties to contracts are modules, which must be in entirely one language or the other, and top-level functions, which we consider as singleton modules.

Contracts on first-order values check assertions about their arguments, and either return the argument or signal an error. Contracts on functions return functions that defer checking until first-order values are reached. The result of applying a contract should contextually approximate the argument. We represent a contract for a type α as a function taking two parties and a value of type α, and returning a value of the same type α:

$$\textbf{type}\ \alpha\ \mathsf{contract} = \mathsf{party} \times \mathsf{party} \to \alpha \to \alpha \tag{8}$$

A simple contract might assert something about a first-order value:

```
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x then x else blame pos                                  (9)
```

The contract is instantiated with the identities of the contracted parties, and then may be applied to a value. We may also construct contracts for functional values, given contracts for the domain and codomain:

```
let makeFunctionContract[α, β] (dom: α contract, codom: β contract)
                               (neg: party, pos: party) (f: α → β) =
  fun (x: α) → codom (neg, pos) (f (dom (pos, neg) x))              (10)
```

When this contract is applied to a function, it can perform no checks immediately. Instead, it wraps the function so that, when the resulting function is applied, the domain contract is applied to the actual parameter and the codomain contract to the actual result.

We follow this approach closely, but with one small change—contracts for affine functions are stateful:

```
let makeAffineFunContract[α, β] (dom: α contract, codom: β contract)
                                (neg: party, pos: party) (f: α → β) =
  let stillGood = ref true in
    fun (x: α) →
      if ! stillGood
        then stillGood ← false;
             codom (neg, pos) (f (dom (pos, neg) x))
        else blame neg                                               (11)
```

This approach works for functions because we can wrap a function to modify its behavior. But what about for other affine values such as the socket capabilities in Sect. 2? We must consider how non-functional values move between the two sublanguages.

In order to understand the solution, we need to show in greater detail how types are mapped between the two sublanguages. (The rest of the type system appears in the next section.) We define mappings $(\cdot)^{\mathscr{A}}$ and $(\cdot)^{\mathscr{C}}$ from $\mathscr{C}$ types to $\mathscr{A}$ types and $\mathscr{A}$ types to $\mathscr{C}$ types, respectively. Base types such as int and bool, which may be duplicated without restriction in both languages, map to themselves:

$$(\mathcal{B})^{\mathscr{A}} = \mathcal{B} \qquad\qquad (\mathcal{B})^{\mathscr{C}} = \mathcal{B} \tag{12}$$

Function types convert to function types. $\mathscr{C}$ function types go to unlimited functions in $\mathscr{A}$, and both unlimited and affine $\mathscr{A}$ functions collapse to ordinary $(\rightarrow)$ functions in $\mathscr{C}$ (where q ranges over a and u):

$$(\tau_1 \rightarrow \tau_2)^{\mathscr{A}} = (\tau_2)^{\mathscr{A}} \overset{\mathsf{u}}{\multimap} (\tau_2)^{\mathscr{A}} \qquad (\sigma_1 \overset{\mathsf{q}}{\multimap} \sigma_2)^{\mathscr{C}} = (\sigma_1)^{\mathscr{C}} \rightarrow (\sigma_2)^{\mathscr{C}} \tag{13}$$

Quantified types map to quantified types, but they require renaming because we distinguish type variables between the two languages. In particular, $\mathscr{A}$ language type variables carry usage qualifiers, which indicate whether they may be instantiated to any type or only to unlimited types. (All type variables in Sect. 2 were of the u kind.)

$$(\forall \alpha.\, \tau)^{\mathscr{A}} = \forall \beta^{\mathsf{u}}.\, (\tau_1[\{\beta^{\mathsf{u}}\}/\alpha])^{\mathscr{A}} \qquad (\forall \alpha^{\mathsf{q}}.\, \sigma)^{\mathscr{C}} = \forall \beta.\, (\sigma_1[\{\beta\}/\alpha^{\mathsf{q}}])^{\mathscr{C}} \tag{14}$$

Finally, the remaining types are uninterpreted by the mapping, and merely enclosed in $\{\cdot\}$:

$$(\tau^o)^{\mathscr{A}} = \{\tau^o\}, \text{ otherwise} \qquad\qquad (\sigma^o)^{\mathscr{C}} = \{\sigma^o\}, \text{ otherwise} \tag{15}$$

Values in this class of types are inert: they have no available operations other than passing them back to their native sublanguage, which removes the $\{\cdot\}$. (We take $\{\{\tau\}\}$ to be equivalent to τ.)

This mapping implies that all non-functional, affine types in $\mathscr{A}$ map to opaque types in $\mathscr{C}$.[2] Since all that the $\mathscr{C}$ language can do with values of opaque type is pass them back to $\mathscr{A}$, we are free to wrap such values when they flow into $\mathscr{C}$ and unwrap them when they return to $\mathscr{A}$. Specifically, when an affine value v passes into $\mathscr{C}$, we wrap it in a λ abstraction, **fun** $(\_ \colon \mathsf{unit}) \rightarrow v$, and wrap that thunk with an affine function contract. If the wrapped value flows back into

[2] Opaque types may seem limiting, but Matthews and Findler [9] have shown that it is possible, in what they call the "lump embedding," for each sublanguage to marshal its opaque values for the other sublanguage as desired. In practice, this amounts to exporting a fold to the other sublanguage.

$$\begin{aligned}
\mathscr{C}\mathscr{A}\llbracket \mathsf{int} \rrbracket(n,p) &= \mathit{id} \\
\mathscr{C}\mathscr{A}\llbracket \sigma_1 \overset{\mathsf{u}}{\multimap} \sigma_2 \rrbracket(n,p) &= \mathsf{makeFunctionContract}\ (\mathscr{A}\mathscr{C}\llbracket \sigma_1 \rrbracket,\ \mathscr{C}\mathscr{A}\llbracket \sigma_2 \rrbracket)\ (n,\ p) \\
\mathscr{C}\mathscr{A}\llbracket \sigma_1 \overset{\mathsf{a}}{\multimap} \sigma_2 \rrbracket(n,p) &= \mathsf{makeAffineFunContract}\ (\mathscr{A}\mathscr{C}\llbracket \sigma_1 \rrbracket,\ \mathscr{C}\mathscr{A}\llbracket \sigma_2 \rrbracket)\ (n,\ p) \\
\mathscr{C}\mathscr{A}\llbracket \sigma^o \rrbracket(n,p) &= \textbf{fun}\ (\mathsf{v}\colon \sigma^o) \to \mathsf{makeAffineFunContract} \qquad \textit{(if } \sigma^o \textit{ is} \\
&\qquad\qquad (\mathit{id},\ \mathit{id})\ (n,\ p)\ (\textbf{fun}\ () \to \mathsf{v}) \qquad \textit{affine)}
\end{aligned}$$

$$\begin{aligned}
\mathscr{A}\mathscr{C}\llbracket \mathsf{int} \rrbracket(n,p) &= \mathit{id} \\
\mathscr{A}\mathscr{C}\llbracket \sigma_1 \overset{\mathsf{q}}{\multimap} \sigma_2 \rrbracket(n,p) &= \mathsf{makeFunctionContract}\ (\mathscr{C}\mathscr{A}\llbracket \sigma_1 \rrbracket,\ \mathscr{A}\mathscr{C}\llbracket \sigma_2 \rrbracket)\ (n,\ p) \\
\mathscr{A}\mathscr{C}\llbracket \sigma^o \rrbracket(n,p) &= \textbf{fun}\ (\mathsf{v}\colon \mathsf{unit} \to \sigma^o) \to \mathsf{v}\ () \qquad \textit{(if } \sigma^o \textit{ is affine)}
\end{aligned}$$

Fig. 5. Type-directed generation of coercions

$\mathscr{A}$, we unwrap it by applying the thunk, which produces a contract error if we attempt unwrapping it more than once.

After type checking, our implementation translates $\mathscr{A}$ modules to $\mathscr{C}$ and wraps all interlanguage variable references with contracts that enforce the $\mathscr{A}$ language's view of the variable. In Fig. 5, we show several cases from a pair of metafunctions $\mathscr{A}\mathscr{C}\llbracket\cdot\rrbracket$ and $\mathscr{C}\mathscr{A}\llbracket\cdot\rrbracket$, which perform this wrapping. Metafunction $\mathscr{A}\mathscr{C}\llbracket\cdot\rrbracket$ produces the coercion for references to $\mathscr{C}$ values from $\mathscr{A}$, and $\mathscr{C}\mathscr{A}\llbracket\cdot\rrbracket$ is for references to $\mathscr{A}$ values from $\mathscr{C}$. Our formalization does not use this translation, but gives a semantics to the multi-language system directly.

4 Formalization

We model our language with a pair of calculi corresponding to the two sublanguages in the implementation. In this section, we first describe the two calculi independently, and then move on to explain how they interact.

To distinguish the two calculi, we typeset our affine calculus $\lambda^{\mathscr{A}}$ in a sans-serif font and our non-affine calculus $\lambda_{\mathscr{C}}$ in a **bold serif font**.

4.1 The Calculi $\lambda^{\mathscr{A}}$ and $\lambda_{\mathscr{C}}$

We model sublanguage $\mathscr{C}$ with calculus $\lambda_{\mathscr{C}}$, which is merely call-by-value System F [7] equipped with singleton modules, each of which for simplicity declares only one name bound to one value. The syntax of $\lambda_{\mathscr{C}}$ appears in Fig. 6, including module names, which are disjoint from variable names. We include integer literals, which serve as first-order values that should pass transparently into the affine subcalculus. A program comprises a mutually recursive collection of modules M and a main expression $\mathbf{e}$. We give only the semantics relevant to modules, as the rest is standard. The expression typing judgment has the form $\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash^{M}_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$, and it carries a module context M, which rule TC-Mod uses to type module expressions. To type a program, we must type each module with rule TM-C;

$$
\begin{array}{rlcl}
\textit{variables} & \mathbf{x}, \mathbf{y} & \in & \mathit{Var}_{\mathscr{C}} \\
\textit{type variables} & \boldsymbol{\alpha}, \boldsymbol{\beta} & \in & \mathit{TVar}_{\mathscr{C}} \\
\textit{module names} & \mathbf{f}, \mathbf{g} & \in & \mathit{MVar}_{\mathscr{C}} \\
\textit{programs} & \mathbf{P} & ::= & M\ \mathbf{e} \\
\textit{module contexts} & M & ::= & \mathbf{m_1} \ldots \mathbf{m_k} \\
\textit{modules} & \mathbf{m} & ::= & \mathbf{module\,f} : \boldsymbol{\tau} = \mathbf{v} \\
\textit{types} & \boldsymbol{\tau} & ::= & \forall \boldsymbol{\alpha}.\, \boldsymbol{\tau} \mid \boldsymbol{\alpha} \\
 & & \mid & \boldsymbol{\tau} \to \boldsymbol{\tau} \mid \mathbf{int} \\
\textit{expressions} & \mathbf{e} & ::= & \mathbf{x} \mid \mathbf{f} \mid \mathbf{e}[\boldsymbol{\tau}] \mid \mathbf{e}\,\mathbf{e} \\
 & & \mid & \boldsymbol{\Lambda\alpha}.\, \mathbf{v} \mid \lceil z \rceil \mid \cdots
\end{array}
$$

$$
\textsc{TC-Mod}\quad \frac{\mathbf{module\,f} : \boldsymbol{\tau} = \mathbf{v} \in M \qquad \cdot \vdash_{\mathscr{C}} \boldsymbol{\tau}}{\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash^{M}_{\mathscr{C}} \mathbf{f} : \boldsymbol{\tau}}
$$

$$
\textsc{TM-C}\quad \frac{\cdot; \cdot \vdash^{M}_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}}{\vdash^{M} \mathbf{module\,f} : \boldsymbol{\tau} = \mathbf{v}\ \text{okay}}
$$

$$
\textsc{C-Mod}\quad \frac{(\mathbf{module\,f} : \boldsymbol{\tau} = \mathbf{v}) \in M}{\mathbf{f} \underset{M}{\longmapsto} \mathbf{v}}
$$

Fig. 6. Selected syntax and semantics of $\lambda_{\mathscr{C}}$

note that the whole module context is available to each module, allowing for recursion. Finally, C-Mod shows that module names reduce to the value of the module.

We model sublanguage $\mathscr{A}$ with calculus $\lambda^{\mathscr{A}}$, which extends $\lambda_{\mathscr{C}}$ with affine types. While $\lambda^{\mathscr{A}}$ includes all of $\lambda_{\mathscr{C}}$, we choose not to embed $\lambda_{\mathscr{C}}$ in $\lambda^{\mathscr{A}}$ to emphasize the generality of our approach, anticipating conventional language features that we do not know how to type in an affine language. The syntax of $\lambda^{\mathscr{A}}$ may be found in Fig. 7. Expressions are mostly conventional: values, which include λ and Λ abstractions, constants, and pairs; variables; application and type 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 include integers, function types with qualifier q, universals, and the syntactically distinguished opaque types, which include type variables, products, and reference cells. Figure 8 defines a lattice on qualifiers, of which there are only two: u is bottom and a is top. A qualifier is assigned to each type, with the notation $|\sigma| = \mathsf{q}$. Integers are always assigned the unlimited qualifier u, whereas references always have the affine qualifier a. Function types and type variables are annotated with their qualifiers, and products get the stronger qualifier of either of their components. We define the qualifier of a value context Γ as well, to be the maximum qualifier of any type bound in it; in other words, Γ is affine if *any* variable is affine, but if none is then it is unlimited.

The subtyping relation appears in Fig. 8. It is reflexive and transitive, covariant on both pair components and function codomains, and contravariant on function domains, as usual. Subtyping arises from the qualifier lattice in two ways: an unlimited function may be used where an affine function is expected (but not vice versa), and a universal type whose bound variable has qualifier a may be instantiated by a type with qualifier u (but not vice versa).

$$
\begin{array}{rlll}
\textit{variables} & \mathsf{x},\mathsf{y} & \in \mathit{Var}_{\mathscr{A}} \\
\textit{qualifiers} & \mathsf{q} & \in \{\mathsf{a},\mathsf{u}\} \\
\textit{type variables} & \alpha^{\mathsf{q}},\beta^{\mathsf{q}} & \in \mathit{TVar}_{\mathscr{A}} \\
\textit{module names} & \mathsf{f},\mathsf{g} & \in \mathit{MVar}_{\mathscr{A}} \\[4pt]
\textit{modules} & \mathsf{m} & ::= & \mathsf{module}\ \mathsf{f} : \sigma = \mathsf{v} \\
\textit{types} & \sigma & ::= & \mathsf{int} \mid \sigma \overset{\mathsf{q}}{\multimap} \sigma \mid \forall\alpha^{\mathsf{q}}.\,\sigma \mid \sigma^{\circ} \\
\textit{opaque types} & \sigma^{\circ} & ::= & \alpha \mid \sigma \otimes \sigma \mid \sigma\ \mathsf{ref} \\
\textit{expressions} & \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{x} \mid \mathsf{f} \mid \mathsf{e}\ \mathsf{e} \mid \mathsf{e}[\sigma] \mid \mathsf{if0}\ \mathsf{e}\ \mathsf{e}\ \mathsf{e} \\
 & & \mid & \langle \mathsf{e},\mathsf{e}\rangle \mid \mathsf{let}\ \langle \mathsf{x},\mathsf{x}\rangle = \mathsf{e}\ \mathsf{in}\ \mathsf{e} \\
\textit{values} & \mathsf{v} & ::= & \mathsf{c} \mid \lambda \mathsf{x}{:}\sigma.\mathsf{e} \mid \Lambda\alpha^{\mathsf{q}}.\,\mathsf{v} \mid \langle \mathsf{v},\mathsf{v}\rangle \\
\textit{constants} & \mathsf{c} & ::= & \mathsf{new}[\sigma] \mid \mathsf{swap}[\sigma][\sigma] \mid \lceil z \rceil \mid - \mid (z-) \mid \cdots \\[4pt]
\textit{value contexts} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x}{:}\sigma \\
\textit{type contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha^{\mathsf{q}}
\end{array}
$$

Fig. 7. Syntax of $\lambda^{\mathscr{A}}$

$\boxed{\mathsf{q} \sqsubseteq \mathsf{q}}$, $\boxed{|\tau| = \mathsf{q}}$, $\boxed{|\Gamma| = \mathsf{q}}$

$$
\mathsf{u} \sqsubseteq \mathsf{q} \qquad \mathsf{q} \sqsubseteq \mathsf{a} \qquad |\mathsf{int}| = \mathsf{u} \qquad |\sigma_1 \overset{\mathsf{q}}{\multimap} \sigma_2| = \mathsf{q} \qquad |\forall\alpha^{\mathsf{q}}.\,\sigma| = |\sigma| \qquad |\alpha^{\mathsf{q}}| = \mathsf{q}
$$

$$
|\sigma_1 \otimes \sigma_2| = |\sigma_1| \sqcup |\sigma_2| \qquad |\sigma\ \mathsf{ref}\,| = \mathsf{a} \qquad |\Gamma| = \bigsqcup_{\mathsf{x}\in\mathrm{dom}(\Gamma)} |\Gamma(\mathsf{x})|
$$

$\boxed{\sigma <: \sigma}$

$$
\textsc{S-Refl}\ \frac{}{\sigma <: \sigma} \qquad \textsc{S-Trans}\ \frac{\sigma_1 <: \sigma_2 \qquad \sigma_2 <: \sigma_3}{\sigma_1 <: \sigma_3} \qquad \textsc{S-Prod}\ \frac{\sigma_1 <: \sigma_1' \qquad \sigma_2 <: \sigma_2'}{\sigma_1 \otimes \sigma_2 <: \sigma_1' \otimes \sigma_2'}
$$

$$
\textsc{S-Arrow}\ \frac{\sigma_1' <: \sigma_1 \qquad \sigma_2 <: \sigma_2' \qquad \mathsf{q} \sqsubseteq \mathsf{q}'}{\sigma_1 \overset{\mathsf{q}}{\multimap} \sigma_2 <: \sigma_1' \overset{\mathsf{q}'}{\multimap} \sigma_2'} \qquad \textsc{S-Forall}\ \frac{\mathsf{q}_2 \sqsubseteq \mathsf{q}_1 \qquad \sigma_1[\beta^{\mathsf{q}_2}/\alpha^{\mathsf{q}_1}] <: \sigma_2}{\forall\alpha^{\mathsf{q}_1}.\,\sigma_1 <: \forall\beta^{\mathsf{q}_2}.\,\sigma_2}
$$

$\boxed{\Gamma \boxplus \Gamma = \Gamma}$

$$
\frac{}{\cdot \boxplus \cdot = \cdot} \qquad \frac{\Gamma_1 \boxplus \Gamma_2 = \Gamma_3 \qquad |\sigma| = \mathsf{a}}{\Gamma_1 \boxplus \Gamma_2, \mathsf{x}{:}\sigma = \Gamma_3, \mathsf{x}{:}\sigma} \qquad \frac{\Gamma_1 \boxplus \Gamma_2 = \Gamma_3 \qquad |\sigma| = \mathsf{a}}{\Gamma_1, \mathsf{x}{:}\sigma \boxplus \Gamma_2 = \Gamma_3, \mathsf{x}{:}\sigma}
$$

$$
\frac{\Gamma_1 \boxplus \Gamma_2 = \Gamma_3 \qquad |\sigma| = \mathsf{u}}{\Gamma_1, \mathsf{x}{:}\sigma \boxplus \Gamma_2, \mathsf{x}{:}\sigma = \Gamma_3, \mathsf{x}{:}\sigma}
$$

Fig. 8. Statics of $\lambda^{\mathscr{A}}$ (qualifiers, subtyping, contexts)

$$\boxed{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{e} : \sigma}$$

TA-Subsume
$$\frac{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{e} : \sigma \qquad \sigma <: \sigma'}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{e} : \sigma'}$$

TA-Lam
$$\frac{\Delta; \Gamma, \mathsf{x} : \sigma \vdash^{M}_{\mathscr{A}} \mathsf{e} : \sigma' \qquad \Delta \vdash_{\mathscr{A}} \sigma \qquad \left|\Gamma|_{\mathrm{FV}(\lambda \mathsf{x}:\sigma.\,\mathsf{e})}\right| = \mathsf{q}}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \lambda \mathsf{x}{:}\sigma.\, \mathsf{e} : \sigma \overset{\mathsf{q}}{\multimap} \sigma'}$$

TA-TApp
$$\frac{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{e} : \forall \alpha^{\mathsf{q}}.\, \sigma' \qquad \Delta \vdash_{\mathscr{A}} \sigma \qquad |\sigma| \sqsubseteq \mathsf{q}}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{e}[\sigma] : \sigma'[\sigma/\alpha^{\mathsf{q}}]}$$

TA-App
$$\frac{\Delta; \Gamma_1 \vdash^{M}_{\mathscr{A}} \mathsf{e}_1 : \sigma' \overset{\mathsf{q}}{\multimap} \sigma \qquad \Delta; \Gamma_2 \vdash^{M}_{\mathscr{A}} \mathsf{e}_2 : \sigma'}{\Delta; \Gamma_1 \boxplus \Gamma_2 \vdash^{M}_{\mathscr{A}} \mathsf{e}_1\ \mathsf{e}_2 : \sigma}$$

TA-Mod
$$\frac{\mathsf{module}\ \mathsf{f} : \sigma = \mathsf{v} \in M \qquad \cdot \vdash_{\mathscr{A}} \sigma}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{f} : \sigma}$$

TA-New
$$\frac{}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{new}[\sigma] : \sigma \overset{\mathsf{u}}{\multimap} \sigma\ \mathsf{ref}}$$

TA-Swap
$$\frac{}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{swap}[\sigma_1][\sigma_2] : (\sigma_1\ \mathsf{ref} \otimes \sigma_2) \overset{\mathsf{u}}{\multimap} (\sigma_1 \otimes \sigma_2\ \mathsf{ref})}$$

Fig. 9. Statics of $\lambda^{\mathscr{A}}$ (selected expressions)

Figure 8 defines context splitting, which is used by expression typing to distribute affine assumptions to only one use in a term, but unlimited variables to an unlimited number of mentions. When a value context must be split to type two subexpressions, in an application expression, for example (Fig. 9), variables of affine type are made available to either the operator or operand, but not both.

Selected expression typing rules appear in Fig. 9. Rules TA-Lam and TA-App are the usual substructural rules for typing λ expressions and applications: for λ expressions, the qualifier q given to the resulting $\overset{\mathsf{q}}{\multimap}$ type is the qualifier of the context Γ limited to the free variables of the expression; thus, the function is at least as restricted as any values it closes over. The type application rule TA-TApp requires that a type variable be at least as restrictive as any type with which it is instantiated.

Finally, rule TA-Swap takes a pair of a σ_1 reference and a σ_2, and returns a σ_1 and a σ_2 reference. From the operational semantics, a small selection of which appears in Fig. 10, it should be clear that swap swaps the σ_2 argument into the location and returns the value previously in the location. Since TA-Swap does not require these two types to be the same, swap performs a *strong update*—that is, it may change the type of the value residing in a reference cell. This is why the qualifier given to references must be a: if a reference is aliased, then it becomes possible to observe the type change in a way the destroys type safety. This feature of the calculus is a stand-in for the variety of invariants that an affine type system might enforce. In the mixed calculus, $\lambda_{\mathscr{C}}$ may gain access to $\lambda^{\mathscr{A}}$ references. It has no operations available to read or write them, but it must be prevented from passing an aliased reference cell back into $\lambda^{\mathscr{A}}$ where it can cause trouble.

$$\begin{array}{rl}
\textit{locations} & \ell \in \mathit{Loc} \\
\textit{values} & \mathsf{v} ::= \cdots \mid \ell \\
\textit{stores} & s ::= \{\ell \mapsto \mathsf{v}, \ldots, \ell \mapsto \mathsf{v}\} \\
\textit{configurations} & \mathsf{C} ::= (s, \mathsf{e}) \\
\textit{evaluation contexts} & \mathsf{E} ::= [\,]_{\mathscr{A}} \mid \mathsf{E}[\sigma] \mid \mathsf{E}\,\mathsf{e} \mid \mathsf{v}\,\mathsf{E} \mid \langle \mathsf{E}, \mathsf{e}\rangle \mid \langle \mathsf{v}, \mathsf{E}\rangle \mid \cdots
\end{array}$$

$\boxed{C \longmapsto_M C}$

$$\begin{array}{ll}
\text{(A-New)} & (s, \mathsf{new}[\sigma]\,\mathsf{v}) \underset{M}{\longmapsto} (s \uplus \{\ell \mapsto \mathsf{v}\}, \ell) \\
\text{(A-Swap)} & (s \uplus \{\ell \mapsto \mathsf{v}_1\}, \mathsf{swap}[\sigma_1][\sigma_2]\,\langle \ell, \mathsf{v}_2\rangle) \underset{M}{\longmapsto} (s \uplus \{\ell \mapsto \mathsf{v}_2\}, \langle \mathsf{v}_1, \ell\rangle)
\end{array}$$

Fig. 10. Dynamics of $\lambda^{\mathscr{A}}$ (selected rules)

$$\begin{array}{rlrl}
\textit{programs} & P ::= M\,\mathbf{e} & \lambda_{\mathscr{C}}\ \textit{expressions} & \mathbf{e} ::= \cdots \mid \mathsf{f}^{\mathbf{g}} \\
\textit{module contexts} & M ::= m_1 \ldots m_k & \lambda_{\mathscr{C}}\ \textit{types} & \boldsymbol{\tau} ::= \cdots \mid \{\sigma\} \\
\textit{modules} & m ::= \mathsf{m} \mid \mathbf{m} & \lambda^{\mathscr{A}}\ \textit{expressions} & \mathsf{e} ::= \cdots \mid \mathbf{f}^{\mathsf{g}} \\
& \quad\mid \textbf{interface}\ \mathbf{f} :> \sigma = \mathbf{g} & \lambda^{\mathscr{A}}\ \textit{types} & \sigma ::= \cdots \mid \{\boldsymbol{\tau}\}
\end{array}$$

Fig. 11. New syntax for $\lambda^{\mathscr{A}}_{\mathscr{C}}$

4.2 Mixing It Up with $\lambda^{\mathscr{A}}_{\mathscr{C}}$

The primary aim of 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. We can then model an affine program calling into a library written in a legacy language, or a conventional program calling into code written in an affine language. In either case, we must ensure that the non-affine portions of the program do not break the affine portions' invariants. As noted in Sect. 3, we accomplish this via run-time checks in the style of higher-order contracts [4].

The additional syntax for mixed programs is in Fig. 11. The main expression in a mixed program is in subcalculus $\lambda_{\mathscr{C}}$. Modules now include $\lambda^{\mathscr{A}}$ modules, $\lambda_{\mathscr{C}}$ modules, and *interface* modules, which are used to assert a $\lambda^{\mathscr{A}}$ type about a $\lambda_{\mathscr{C}}$ module as we saw in Sect. 2.

We add to each subcalculus's expressions a production referring to modules from the other subcalculus. We decorate each such module name with the name of the module in which it appears (*e.g.*, $\mathbf{f}^{\mathsf{g}}$ for a reference to $\lambda_{\mathscr{C}}$ module $\mathbf{f}$ from $\lambda^{\mathscr{A}}$ module g) and use this name as the negative party in contracts regulating the intercalculus boundary, in order to assign blame.

Static Semantics. The type system for the mixed calculus is the union of the type systems for $\lambda^{\mathscr{A}}$ and $\lambda_{\mathscr{C}}$ (Figs. 6, 8, and 9), along with additional typing rules (Fig. 12) for $\lambda^{\mathscr{A}}$ module invocations in $\lambda_{\mathscr{C}}$ expressions and $\lambda_{\mathscr{C}}$ module invocations in $\lambda^{\mathscr{A}}$ expressions.

$\boxed{\vdash P : \boldsymbol{\tau}}$, $\boxed{\vdash^{M} m \text{ okay}}$

$$\textsc{Prog}\quad \frac{\forall m \in M, \vdash^{M} m \text{ okay} \qquad \cdot;\cdot \vdash^{M}_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}}{\vdash M\ \mathbf{e} : \boldsymbol{\tau}} \qquad\qquad \text{TM-I}\quad \frac{(\mathbf{module}\ \mathbf{g} : (\sigma)^{\mathscr{C}} = \mathbf{v}) \in M \qquad |\sigma| = \mathsf{u}}{\vdash^{M} \mathbf{interface}\ \mathbf{f} :> \sigma = \mathbf{g} \text{ okay}}$$

$\boxed{\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash^{M}_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}}$, $\boxed{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathsf{e} : \sigma}$

$$\text{TA-}\textsc{ModC}\quad \frac{(\mathbf{module}\ \mathbf{f} : \boldsymbol{\tau} = \mathbf{v}) \in M \qquad \cdot \vdash_{\mathscr{C}} \boldsymbol{\tau}}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathbf{f} : (\boldsymbol{\tau})^{\mathscr{A}}} \qquad\qquad \text{TC-}\textsc{ModA}\quad \frac{(\mathsf{module}\ \mathsf{f} : \sigma = \mathsf{v}) \in M \qquad \cdot \vdash_{\mathscr{A}} \sigma}{\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash^{M}_{\mathscr{C}} \mathsf{f} : (\sigma)^{\mathscr{C}}}$$

$$\text{TA-}\textsc{ModI}\quad \frac{(\mathbf{interface}\ \mathbf{f} :> \sigma = \mathbf{g}) \in M \qquad \cdot \vdash_{\mathscr{A}} \sigma}{\Delta; \Gamma \vdash^{M}_{\mathscr{A}} \mathbf{f} : \sigma}$$

Fig. 12. New statics for $\lambda^{\mathscr{A}}_{\mathscr{C}}$

Rule TC-ModA (Fig. 12) types occurrences of $\lambda^{\mathscr{A}}$ module names in $\lambda_{\mathscr{C}}$ expressions. The rule uses the type conversion function $(\cdot)^{\mathscr{C}}$, defined in Sect. 3 (p. 9) to give a $\lambda_{\mathscr{C}}$ type to the $\lambda^{\mathscr{A}}$ module invocation. Because $\lambda^{\mathscr{A}}$ types are richer than $\lambda_{\mathscr{C}}$ types—$\lambda^{\mathscr{A}}$ function types carry extra information in the qualifier—the conversion loses information, which may need to be recovered through dynamic checks. For example, given a $\lambda^{\mathscr{A}}$ module g with type $\mathsf{int} \overset{\mathsf{u}}{\multimap} \mathsf{int} \overset{\mathsf{a}}{\multimap} \mathsf{int}$, the conversion rule assigns it the $\lambda_{\mathscr{C}}$ type $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$. Calculus $\lambda_{\mathscr{C}}$'s type system cannot enforce that the result of applying g be applied at most once, which will need to be checked at run time.

For a $\lambda_{\mathscr{C}}$ module with type $\boldsymbol{\tau}$ invoked from a $\lambda^{\mathscr{A}}$ expression, we use the module at type $(\boldsymbol{\tau})^{\mathscr{A}}$. It would be reasonable for TA-ModC to give it any $\lambda^{\mathscr{A}}$ type in the pre-image of the $\lambda^{\mathscr{A}}$-to-$\lambda_{\mathscr{C}}$ mapping, but $(\cdot)^{\mathscr{A}}$ makes the most permissive, statically safe choice, which is to map all $\lambda_{\mathscr{C}}$ arrows $(\to)$ to the unlimited $\lambda^{\mathscr{A}}$ arrow $(\overset{\mathsf{u}}{\multimap})$. Consider:

- If $\mathbf{f} : \mathbf{int} \to \mathbf{int}$ in $\lambda_{\mathscr{C}}$, then $\mathsf{int} \overset{\mathsf{u}}{\multimap} \mathsf{int}$ is the right type in $\lambda^{\mathscr{A}}$. There is no reason to limit $\mathbf{f}$ to an affine function type, because $\lambda_{\mathscr{C}}$ does not impose that requirement, and subtyping allows us to use it at $\mathsf{int} \overset{\mathsf{a}}{\multimap} \mathsf{int}$, if necessary.
- If $\mathbf{f} : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$ in $\lambda_{\mathscr{C}}$, then $(\mathsf{int} \overset{\mathsf{u}}{\multimap} \mathsf{int}) \overset{\mathsf{u}}{\multimap} \mathsf{int}$ will allow the imported function to be passed unlimited functions but not affine functions. This is a safe choice, because $\lambda_{\mathscr{C}}$'s type system does not tell us whether $\mathbf{f}$ may call its argument more than once.

In the latter case, what if the programmer somehow knows that function $\mathbf{f}$ applies its argument at most once, as in the example of $\mathit{threadFork}_{\mathscr{C}}$ (p. 7)? It should not violate $\lambda^{\mathscr{A}}$'s invariants to pass an affine function to $\mathit{threadFork}_{\mathscr{C}}$, but $\lambda^{\mathscr{A}}$ cannot know this. Therefore, rule TA-ModC gives $\lambda_{\mathscr{C}}$ modules a conservative $\lambda^{\mathscr{A}}$ type that requires no run-time checks. We can use an **interface** module to coerce a $\lambda_{\mathscr{C}}$ module's type $\boldsymbol{\tau}$ to a more permissive $\lambda^{\mathscr{A}}$ type in the pre-image of $\boldsymbol{\tau}$, and this, too, requires a dynamic check.

Operational Semantics. We extend the syntax of our mixed language with several new forms (Fig. 13). Whereas our source syntax segregates the two subcalculi into separate modules, module invocation reduces to the body of the module, which leads expressions of both subcalculi to nest at run time. Rather than allow $\lambda^{\mathscr{A}}$ terms to appear directly in $\lambda_{\mathscr{C}}$, and vice versa, we need a way to cordon off terms from one calculus embedded in the other and to ensure that the interaction is well-behaved. We call these new expression forms *boundaries.*

The new run-time syntax includes both boundary expressions ${}^{\sigma}_{\mathsf{f}}\mathsf{AC}_{\mathbf{g}}(\mathbf{e})$ for embedding $\lambda_{\mathscr{C}}$ expressions in $\lambda^{\mathscr{A}}$ and boundary expressions ${}_{\mathbf{f}}\mathbf{CA}^{\sigma}_{\mathsf{g}}(\mathsf{e})$ for embedding $\lambda^{\mathscr{A}}$ expressions in $\lambda_{\mathscr{C}}$. Each of these forms has a superscript σ, written on the $\lambda^{\mathscr{A}}$ side, which represents a contract between the two modules that gave rise to the nested expression. Some contracts, for example int, are fully enforced by both type systems. Other contracts, such as $\mathsf{int} \overset{\mathsf{a}}{\multimap} \mathsf{int}$, require dynamic checks. The type system guarantees that such a function receives and returns only integers, but this type also imposes an obligation on the negative party to apply the function at most once, which the $\lambda_{\mathscr{C}}$ type system alone does not enforce.

The right subscript of a boundary is a module name in the inner subcalculus, representing the positive party to the contract: It promises that if the enclosed subexpression reduces to a value, then the value will obey contract σ. 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.

Boundaries first arise when a module in one calculus refers to a module in the other calculus. When the name of a $\lambda_{\mathscr{C}}$ module appears in a $\lambda^{\mathscr{A}}$ term, A-ModC wraps the module name with an AC boundary, using the $\lambda^{\mathscr{A}}$-conversion of the module's type $\boldsymbol{\tau}$ as the contract. For interface modules, the contract is as declared by the interface, and the name of the interface is the positive party (A-ModI). From the other direction, a $\lambda^{\mathscr{A}}$ module invoked from a $\lambda_{\mathscr{C}}$ expression is wrapped in a $\mathbf{CA}$ boundary by rule C-ModA.

We add evaluation contexts for reduction under boundaries, which means it is now possible to construct a $\lambda_{\mathscr{C}}$ evaluation context with a $\lambda^{\mathscr{A}}$ hole, and vice versa. If the expression under a boundary reduces to a value, it is time to apply the boundary's contract to the value. There are three possibilities:

- Some values, such as integers, always satisfy the contract, so the boundary is discarded.
- Functional values and opaque affine values must have their checks deferred: functions until application time, and opaque values until they pass back into their original subcalculus. For deferred checks, we leave the value in a "sealed" boundary, ${}_{\mathbf{f}}\mathbf{CA}[\ell]^{\sigma}_{\mathsf{g}}(\mathsf{v})$ or ${}^{\sigma}_{\mathsf{f}}\mathsf{AC}[\,]_{\mathbf{g}}(\mathbf{v})$, which is itself a value form.
- When a previously sealed opaque value reaches a boundary back to its original subcalculus, both that boundary and the sealed boundary are discarded.

$$
\begin{array}{rlrl}
\lambda_{\mathscr{C}}\ \textit{terms} & \mathbf{e} ::= \cdots \mid \underset{\mathbf{f}\,\mathsf{f}}{\mathbf{CA}}^{\sigma}(\mathsf{e}) & \lambda^{\mathscr{A}}\ \textit{terms} & \mathsf{e} ::= \cdots \mid {}^{\sigma}\underset{\mathsf{f}\,\mathbf{f}}{\mathsf{AC}}(\mathbf{e}) \\
\lambda_{\mathscr{C}}\ \textit{values} & \mathbf{v} ::= \cdots \mid \underset{\mathbf{f}\,\mathsf{f}}{\mathbf{CA}}[\ell]^{\sigma}(\mathsf{v}) & \lambda^{\mathscr{A}}\ \textit{values} & \mathsf{v} ::= \cdots \mid {}^{\sigma}\underset{\mathsf{f}\,\mathbf{f}}{\mathsf{AC}}[\,](\mathbf{v}) \\
\lambda_{\mathscr{C}}\ \textit{eval. cxts.} & \mathbf{E} ::= \cdots \mid \underset{\mathbf{f}\,\mathsf{f}}{\mathbf{CA}}^{\sigma}(\mathsf{E}) & \lambda^{\mathscr{A}}\ \textit{eval. cxts.} & \mathsf{E} ::= \cdots \mid {}^{\sigma}\underset{\mathsf{f}\,\mathbf{f}}{\mathsf{AC}}(\mathbf{E})
\end{array}
$$

$$
\begin{array}{rl}
\textit{configurations} & C ::= (s, \mathbf{e}) \mid \mathbf{blame\,f} \\
\textit{answers} & A ::= (s, \mathbf{v}) \mid \mathbf{blame\,f} \\
\textit{stores} & s ::= \{\} \mid s \uplus \{\ell \mapsto \mathbf{v}\} \mid s \uplus \{\ell \mapsto \mathsf{v}\}
\end{array}
$$

$$
\begin{array}{lrcll}
(\text{C-M{\scriptsize OD}A}) & (s, \mathsf{f}^{\mathbf{g}}) & \underset{M}{\longmapsto} & (s,\ \underset{\mathbf{g}\,\mathsf{f}}{\mathbf{CA}}^{\sigma}(\mathsf{f})) & (\mathsf{module\,f} : \sigma = \mathsf{v}) \in M \\
(\text{A-M{\scriptsize OD}C}) & (s, \mathbf{f}^{\mathsf{g}}) & \underset{M}{\longmapsto} & (s, {}^{(\boldsymbol{\tau})^{\mathscr{A}}}\underset{\mathsf{g}\,\mathbf{f}}{\mathsf{AC}}(\mathbf{f})) & (\mathbf{module\,f} : \boldsymbol{\tau} = \mathbf{v}) \in M \\
(\text{A-M{\scriptsize OD}I}) & (s, \mathbf{f}^{\mathsf{g}}) & \underset{M}{\longmapsto} & (s, {}^{\sigma}\underset{\mathsf{g}\,\mathbf{f}}{\mathsf{AC}}(\mathbf{f}')) & (\mathbf{interface\,f} :> \sigma = \mathbf{f}') \in M \\
(\text{C-W{\scriptsize RAP}}) & (s,\ \underset{\mathbf{f}\,\mathsf{g}}{\mathbf{CA}}^{\sigma}(\mathsf{v})) & \underset{M}{\longmapsto} & \textit{coerce}_{\mathscr{C}}(s, \sigma, \mathsf{v}, \mathbf{f}, \mathsf{g}) & \\
(\text{A-W{\scriptsize RAP}}) & (s, {}^{\sigma}\underset{\mathsf{f}\,\mathbf{g}}{\mathsf{AC}}(\mathbf{v})) & \underset{M}{\longmapsto} & \textit{coerce}_{\mathscr{A}}(s, \sigma, \mathbf{v}, \mathsf{f}, \mathbf{g}) & \\
(\text{C-}B\text{-A}) & (s,\ \underset{\mathbf{f}\,\mathsf{g}}{\mathbf{CA}}[\ell]^{\forall\alpha^{\mathsf{q}}.\sigma}(\mathsf{v})[\boldsymbol{\tau}]) & \underset{M}{\longmapsto} & \textit{check}(s, \ell, |\sigma|,\ \underset{\mathbf{f}\,\mathsf{g}}{\mathbf{CA}}^{\sigma[(\boldsymbol{\tau})^{\mathscr{A}}/\alpha^{\mathsf{q}}]}\Big(\mathsf{v}[(\boldsymbol{\tau})^{\mathscr{A}}]\Big), \mathbf{f}) & \\
(\text{C-}\beta\text{-A}) & (s,\ \underset{\mathbf{f}\,\mathsf{g}}{\mathbf{CA}}[\ell]^{\sigma_1 \overset{\mathsf{q}}{\multimap} \sigma_2}(\mathsf{v}_1)\ \mathbf{v_2}) & \underset{M}{\longmapsto} & \textit{check}(s, \ell, \mathsf{q},\ \underset{\mathbf{f}\,\mathsf{g}}{\mathbf{CA}}^{\sigma_2}\Bigg(\mathsf{v}_1\ {}^{\sigma_1}\underset{\mathsf{g}\,\mathbf{f}}{\mathsf{AC}}(\mathbf{v_2})\Bigg), \mathbf{f}) & \\
(\text{A-}B\text{-C}) & (s, {}^{\forall\alpha^{\mathsf{q}}.\sigma}\underset{\mathsf{f}\,\mathbf{g}}{\mathsf{AC}}[\,](\mathbf{v})[\sigma_{\mathsf{a}}]) & \underset{M}{\longmapsto} & (s, {}^{\sigma[\sigma_{\mathsf{a}}/\alpha^{\mathsf{q}}]}\underset{\mathsf{f}\,\mathbf{g}}{\mathsf{AC}}\Big(\mathbf{v}[(\sigma_{\mathsf{a}})^{\mathscr{C}}]\Big)) & \\
(\text{A-}\beta\text{-C}) & (s, {}^{\sigma_1 \overset{\mathsf{q}}{\multimap} \sigma_2}\underset{\mathsf{f}\,\mathbf{g}}{\mathsf{AC}}[\,](\mathbf{v_1})\ \mathsf{v}_2) & \underset{M}{\longmapsto} & (s, {}^{\sigma_2}\underset{\mathsf{f}\,\mathbf{g}}{\mathsf{AC}}\Bigg(\mathbf{v_1}\ \ \underset{\mathbf{g}\,\mathsf{f}}{\mathbf{CA}}^{\sigma_1}(\mathsf{v}_2)\Bigg)) &
\end{array}
$$

$$
\textit{coerce}_{\mathscr{C}}(s, \sigma, \mathsf{v}, \mathbf{f}, \mathsf{g}) = \begin{cases} (s, \boldsymbol{\lceil} z \boldsymbol{\rceil}) & \text{if } \mathsf{v} = \lceil z \rceil \\ (s, \mathbf{v}') & \text{if } \mathsf{v} = {}^{\{\boldsymbol{\tau}^{\mathbf{o}}\}}_{\mathsf{g}'}\mathsf{AC}[\,]_{\mathbf{f}'}(\mathbf{v}') \\ (s \uplus \{\ell \mapsto \mathbf{blssd}\},\ \underset{\mathbf{f}\,\mathsf{g}}{\mathbf{CA}}[\ell]^{\sigma}(\mathsf{v})) & \text{otherwise} \end{cases}
$$

$$
\textit{coerce}_{\mathscr{A}}(s, \sigma, \mathbf{v}, \mathsf{f}, \mathbf{g}) = \begin{cases} (s, \lceil z \rceil) & \text{if } \mathbf{v} = \boldsymbol{\lceil} z \boldsymbol{\rceil} \\ \textit{check}(s, \ell, |\sigma^{\circ}|, \mathsf{v}', \mathbf{g}') & \text{if } \mathbf{v} = {}_{\mathbf{g}'}\mathbf{CA}[\ell]^{\sigma^{\circ}}_{\mathsf{f}'}(\mathsf{v}') \\ (s, {}^{\sigma}\underset{\mathsf{f}\,\mathbf{g}}{\mathsf{AC}}[\,](\mathbf{v})) & \text{otherwise} \end{cases}
$$

$$
\textit{check}(s, \ell, \mathsf{q}, e, \mathbf{f}) = \begin{cases} (s, e) & \text{if } \mathsf{q} = \mathsf{u} \\ (s' \uplus \{\ell \mapsto \mathbf{dfnct}\}, e) & \text{if } s = s' \uplus \{\ell \mapsto \mathbf{blssd}\} \\ (s, \mathbf{blame\,f}) & \text{otherwise} \end{cases}
$$

Fig. 13. Dynamics of $\lambda^{\mathscr{A}}_{\mathscr{C}}$ (run-time syntax and reduction rules)

Rule C-Wrap implements contract application for $\lambda^{\mathscr{A}}$ values embedded in $\lambda_{\mathscr{C}}$ expressions, as indicated by metafunction $\mathit{coerce}_{\mathscr{C}}$. The first case of $\mathit{coerce}_{\mathscr{C}}$ handles immediate checks, and its second case unseals previously sealed $\lambda_{\mathscr{C}}$ values that have returned home. The second case of $\mathit{coerce}_{\mathscr{C}}$ seals and *blesses* a $\lambda^{\mathscr{A}}$ value, by allocating a location ℓ, to which it stores a distinguished value **blssd**; it adds this location to the boundary, which marks the sealed value as not yet used. This corresponds directly to the reference cell allocated by *makeAffineFunContract* in Sect. 3.

Rule A-Wrap implements contracts for $\lambda_{\mathscr{C}}$ values in $\lambda^{\mathscr{A}}$ expressions. Metafunction $\mathit{coerce}_{\mathscr{A}}$'s first case is the same as $\mathit{coerce}_{\mathscr{C}}$'s, and the third case seals a value for deferred checking; it need not allocate a location to track the usage of a $\lambda_{\mathscr{C}}$ value. The third case unseals a previously sealed $\lambda^{\mathscr{A}}$ value on its way back to $\lambda^{\mathscr{A}}$, and this requires checking that an affine value has not been previously unsealed. This step is specified by metafunction *check*, which also has three cases. Unlimited values are unsealed with no check. If an affine value remains blessed, *check* updates the store to mark it "defunct" and returns the unsealed value. If, on the other hand, there is an attempt to unseal a defunct affine value, *check* blames the negative party. This is the key dynamic check that enforces the affine invariant for non-functional values.

Rules C-B-A, C-β-A, A-B-C, and A-β-C all handle sealed abstractions, which are unsealed when they are applied. For sealed $\lambda^{\mathscr{A}}$ abstractions, the seal location ℓ must be checked, to ensure that an affine function or type abstraction is not unsealed and applied more than once. This is the dynamic check that enforces the affine invariant for functions.

Type Soundness. The presence of strong updates means that aliasing a location can result in a program getting "stuck": if an aliased location is updated at a different type, reading from the alias produces a value of unexpected type. Calculus $\lambda^{\mathscr{A}}$'s type system prevents this, but adding $\lambda_{\mathscr{C}}$ means that a $\lambda^{\mathscr{A}}$ value may be aliased outside $\lambda^{\mathscr{A}}$. Our soundness criterion is that no program that gets stuck is assigned a type. In particular, all aliasing of affine values is either prevented by $\lambda^{\mathscr{A}}$'s type system or detected by a contract at run time.

In order to prove a Wright-Felleisen–style type soundness theorem [21], we identify precisely what property is preserved by subject reduction. We use an internal type system to track which portions of the store are reachable from $\lambda^{\mathscr{A}}$ values that have flowed into $\lambda_{\mathscr{C}}$. Under this type system, configurations enjoy standard progress and preservation, which allows us to state and prove a syntactic type soundness theorem using the internal type system's configuration typing judgment $\rhd^{M}\ C : \tau$:

Theorem (Type Soundness). *If* $\vdash M\,\mathbf{e} : \boldsymbol{\tau}$ *and* $(\{\}, \mathbf{e}) \longmapsto_{M}^{*} C$ *such that configuration* C *cannot take another step, then* C *is an answer with* $\rhd^{M}\ C : \boldsymbol{\tau}$*.*

Our full formalization, including complete definitions of the calculi and proofs, is available at `http://www.ccs.neu.edu/~tov/pubs/affine-contracts/`.

5 Conclusion

Our work is part of an ongoing program to investigate practical aspects of substructural type systems, and this paper describes one step in that program. Here, we have focused on the problem of interaction between substructural and non-substructural code, each governed by its own type system, and explored the use of higher-order contracts to prevent the conventional language from breaking the substructural language's invariants. Our answer to the problem at hand naturally raises 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. In our prototype, *try-with* expressions are 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 calculus. Terminating the program is problematic because of the implicit discarding of linear values, but catching an exception once part of a continuation containing linear values has been discarded seems even worse. Exceptions in linear languages remain an 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, unlike affine guarantees, which are safety properties, relevance guarantees—that a value is used at some point in the future—are a form of liveness property.

One approximation is to consider a contract representing a relevance guarantee 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 in general, but tracing garbage collection approximates a liveness property very close to the one we desire. 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 discarding. If we detect a violation, we probably could do nothing to prevent it, but at worst we could file a bug report.

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

Acknowledgments. We wish to thank Daniel Brown, Ryan Culpepper, Jed Davis, Matthias Felleisen, Alec Heller, Sam Tobin-Hochstadt, Aaron Turon, and the anonymous referees for their helpful comments, discussion, and corrections. This research was supported in part by AFOSR grant FA9550-09-1-0110.

References

1. Ahmed, A., Fluet, M., Morrisett, G.: $\mathbf{L}^3$: A linear language with locations. Tech. Rep. TR-24-04, Harvard University (2004)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science 6(6) (1996)
3. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: CSL'94. pp. 121–135. No. 933 in LNCS, Springer, Heidelberg (1995)
4. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP'02. pp. 48–59. ACM, New York (2002)
5. Flanagan, C.: Hybrid type checking. In: POPL'06. vol. 41, pp. 245–256. ACM (2006)
6. Gay, S.J., Hole, M.J.: Types and subtypes for client-server interactions. In: ESOP'09. LNCS, vol. 1576, pp. 74–90. Springer (1999)
7. Girard, J.Y.: Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VI (1972)
8. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: Proc. USENIX Annual Technical Conference (2002)
9. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. In: POPL'07. vol. 42, pp. 3–10. ACM, New York (2007)
10. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML. MIT, Cambridge, revised edn. (1997)
11. Plotkin, G.: Type theory and recursion. In: LICS'93. p. 374. IEEE Computer Society (1993)
12. Reynolds, J.C.: Towards a theory of type structure. In: Symposium on Programming, LNCS, vol. 19, pp. 408–423. Springer, Heidelberg (1974)
13. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Workshop on Scheme and Functional Programming. pp. 81–92. ACM, New York (2006)
14. Stevens, W.R.: UNIX Network programming. Prentice-Hall, New Jersey (1990)
15. Strom, R., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering 12(1) (1986)
16. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: OOPSLA'06. pp. 964–974. ACM, New York (2006)
17. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: POPL'07. pp. 395–406. ACM, New York (2008)
18. Turner, D.N., Wadler, P., Mossin, C.: Once upon a type. In: FPCA'95. pp. 1–11. ACM, New York (1995)
19. Wadler, P.: Linear types can change the world. In: Programming Concepts and Methods. pp. 347–359. North Holland, Amsterdam (1990)
20. Walker, D.: Substructural type systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, chap. 1, pp. 3–44. MIT, Cambridge (2005)
21. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)