Substructural Typestates

Filipe Militão
Carnegie Mellon University &
Universidade Nova de Lisboa
filipe.militao@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Luís Caires
Universidade Nova de Lisboa
luis.caires@di.fct.unl.pt

Abstract

Finding simple, yet expressive, verification techniques to reason about both aliasing and mutable state has been a major challenge for static program verification. One such approach, of practical relevance, is centered around a lightweight typing discipline where types denote abstract object states, known as *typestates*.

In this paper, we show how key typestate concepts can be precisely captured by a substructural type-and-effect system, exploiting ideas from linear and separation logic. Building on this foundation, we show how a small set of primitive concepts can be composed to express high-level idioms such as objects with multiple independent state dimensions, dynamic state tests, and behavior-oriented usage protocols that enforce strong information hiding. By exploring the relationship between two mainstream modularity concepts, state abstraction and hiding, we also provide new insights on how they naturally fit together and complement one another.

Technically, our results are based on a typed lambda calculus with mutable references, location-dependent types, and second-order polymorphism. The soundness of our type system is shown through progress and preservation theorems. We also describe a prototype implementation of a type checker for our system, which is available on the web and can be used to experiment with the examples in the paper.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Abstract Data Types; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Typestate; Aliasing; Linearity; Capabilities

1. Introduction

In typical typed programming languages, the use of mutable state is restricted by invariant types, so that a memory cell is constrained to hold a single type of content during its entire lifetime. As a consequence, mutable state variables are deliberately assigned overly conservative types, leading to excessive, error-prone reliance on defensive run-time tests to analyze the actual state of a cell at each relevant program point—instead of accurately tracking the *type of the state* as the program executes. This limitation is present in mainstream languages [5] where violations of state-sensitive usage protocols, of the kind captured by typestate systems [6, 16, 40, 41], are a considerable obstacle to producing reliable code. For instance, in Java a `FileOutputStream` type represents both the open and closed states of that abstraction by relying on dynamic tests to detect incorrect uses (such as writing to a closed stream) instead of statically tracking the changing properties of the type.

A diverse set of techniques (such as *alias types* [2, 13, 15, 39], *typestate* [6, 16, 34], *behavioral types* [10], and others [20, 26, 44, 45, 47]; discussed in more detail in Section 4) have been proposed to tackle this problem, with various degrees of expressiveness. In this work, we reconstruct several core concepts introduced in practical typestate systems (e.g., [6]), by modeling them in a substructural type theory. In particular, we show how key typestate concepts (namely state abstraction, dynamic state tests, and state dimensions) can be captured by a substructural type-and-effect system, exploiting ideas from linear [22] and separation [11, 36] logic.

Typestate approaches expose to clients of an abstract data type object a type-like specification of the *abstract state*, so that the internal representation of the state is not visible, providing information hiding and modularity. Note that the notion of abstract state *refines*, and should not be confused with, the notion of abstract type. While an abstract type hides a representation type, an abstract state goes beyond it by not being (necessarily) limited to abstracting a representation invariant, and being instead able to abstract a particular state of the representation type, or set of concrete states of the representation type, that are meaningful to express the dynamic behavior of the data type. For example, an ADT for `stack` objects could expose two abstract states: *empty* and *non-empty*.

Furthermore, if an object statically falls into an imprecise state, client code may inspect the statically known state using *dynamic state tests* [6] (or similar techniques, such as *keyed variants* [15]), which correspond to using observer ADT operations. For instance, each `stack` object offers `push` and `pop` operations, but the latter requires the non-empty state as a pre-condition. Therefore, when a `stack`'s actual representation state is unknown, clients may rely on an `isEmpty` function, which performs a run-time test on the representation, without exposing the representation to clients. It is well-known how existential quantification may be used to model abstract data types [29], abstracting the representation type. In our case, our type system is able to express the various abstract (type)states using existential quantification on the (substructural) state representation, thus abstracting over various concrete states of the representation type. It turns out that we may also express indirect dynamic tests of abstracted state by leveraging standard mechanisms of case analysis and sum types, rather than relying on ad-hoc methods.

Another interesting insight coming out of our development is a clarification of the relationship between typestate-based approaches, and approaches based on behavioral types [10] (we use here "behavioral types" in the sense of types that describe "tem-

PLPV '14, January 21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2541568.2541574

poral" usage protocols, not in the sense as Liskov and Wing used in [25]). As we will show in the paper, by exploiting linearity and higher-order functional types, our type structure is expressive enough to represent object usage protocols akin to behavioral types, which ensure soundness of object usage without revealing the abstract state, in fact completely hiding it from client code.

In this work, we do not address arbitrary sharing of state (which may lead to interference) since that topic is the subject of ongoing work, discussed in the conclusion section. However, we already show how independent abstract *views* of an object, manipulated by independent aliases (in the spirit of [17, 27, 46]), can be expressed in our framework, by imposing a monoidal structure on the substructural type structure by the separation operator $A * B$. The way this separation interacts with existential quantification is also interesting because it offers a simple way of expressing so-called *state dimensions* [6, 42], so that an object may be shared according to several different usage protocols separately as long as the internal part of the (abstracted) representation is orthogonal.

The main contribution of this paper is to show how key concepts introduced in lightweight typestate-based verification of stateful programs can be precisely reconstructed from lower-level standard programming language constructs. The concepts we reconstruct include the ability to model abstract states, expressing case analysis on those abstract states, combining separation with abstraction to represent orthogonal state dimensions; as well as a novel, yet simple, technique that enables the complete hiding of abstract state through behavioral-types like usage protocols. Our core language is based on a fairly canonical substructural type-and-effect system, based on the core calculus of $\mathbf{L}^3$ [2], but extending it with higher-order quantifiers, sum types, recursive types, and some key technical improvements, crucially the implicit threading of capabilities, which are not run-time values in our approach, but compile-time entities belonging to the static verification context. The harmonious combination of all these ingredients leads to a simple typed core language in which general mechanisms of state abstraction and controlled aliasing of stateful objects can be used and *mixed* to build high-level, flexible, stateful programming abstractions. Finally, the type system is firmly rooted in the core language's operational semantics, for which standard preservation and progress theorems are provided, leading to a soundness result for the approach. Although we do not discuss algorithmic issues in this presentation, we have already implemented a working prototype (available on the web [1]), in which all examples in this paper have been mechanically checked.

In the next section, we introduce the core aspects of the language and leave all technical details to Section 3. The final sections discuss related work, conclusions and future work. Some additional details, such as the unabridged proof of soundness, are relegated to the appendix [28].

2. Overview of Language and Type System

In this section, we introduce and motivate our language and type system in general terms, providing basic intuitions, and discussing several examples. The complete technical presentation of the system is developed further below in Section 3. Our language is based on the polymorphic λ-calculus with mutable references, immutable records (in the style of the language Reynolds used to investigate intersection types [35]), tagged sums and recursive types. In this presentation, we do not address sharing of state through statically disconnected aliases, and focus instead on issues of protocol conformance, abstraction and information hiding, and fully separate uses of stateful objects. We build on techniques developed by Ahmed, Fluet and Morrisett for the system $\mathbf{L}^3$ [2], more specifically, we make use of static knowledge about what references must alias (when such knowledge exists) to reason about aliasing in a particular static context, useful for dealing with strong updates, and in general with stateful objects subject to complex usage protocols. Therefore, and as in that work, by separating *references* (which may be freely duplicated) from *capabilities* (which are *linear* and resource-like), we can track the state of memory cells indirectly, enabling multiple aliases to one same heap location without additional mechanisms when this particular scenario holds. Separating references from capabilities allows the *permission* to use some state to be granted only when both the reference and its capability are present at hand in the typing context. However, our system threads capabilities in a more implicit way, as proof-time objects: the type discipline is flexible enough to support *stacking* of capabilities either on top of a value or to *thread* them instead. This contrasts with the explicit capability-passing approach explored in other substructural systems [2, 24, 44], which require capabilities to be explicitly manipulated as first class values.

We now present our language through a series of examples, all checkable in our type-checker prototype implementation [1].

2.1 A Datatype for Pairs

We consider a stateful pair object, using two private memory cells (`l` and `r`) to store the left and right components, defined using the familiar idiom of representing "objects" as records of closures. The components are private in the sense that references to them are never exposed outside the closures that represent the object's methods — only the *type* of the components is visible to clients. Abstracting the two components under the same typestate would be unnecessarily coarse-grained, since invoking functions that are limited to using a single cell individually would statically require a larger footprint than operationally necessary. Instead, our system supports individually abstracting multiple separate, orthogonal states enabling fine-grained abstractions of the object's state such that each component of the pair can be used in isolation if a function does not require access to the full typestate footprint.

```
1  let newPair = fun( _ : [] ).
2    open <pl,l> = new {} in // location variable 'pl', variable (reference) 'l'
3    open <pr,r> = new {} in // location variable 'pr', variable (reference) 'r'
4      { // RW capability to location variable 'pl' (that has contents of type '[]')
5       initL = fun( i : int :: rw pl [] ). l := i,
6       initR = fun( i : int :: rw pr [] ). r := i,
7       sum = fun( _ : [] :: ( rw pl int * rw pr int ) ). !l+!r,
8       destroy = fun ( _ : [] :: ( rw pl int * rw pr int ) ).
9        delete l; delete r
10     } // end of (labeled) record value
11   end end
```

The code above defines a function `newPair`, which creates a new pair object. Clients can only use the pair through the functions in the record that `newPair` returns. The two local memory cells are created using the `new` primitive. In our language newly created memory cells have an existential dependent type value packing a capability (linear) together with a location (pure value). For example, the new expression on line 2 has the type:

$$\exists t.(\ \mathbf{ref}\ t :: \mathbf{rw}\ t\ [\ ]\ )$$

which means that there exists some fresh location t, and the new expression evaluates to a reference to t ("**ref** t"). Locations can be freely passed around and duplicated, while capabilities are manipulated linearly, and are used as permissions for actually accessing the state. To statically relate run-time locations, our type system uses *location variables* (a static name, or key [15, 19]). Consequently, our types encode a "must-alias" relation where if two location variables are statically equal then they must alias the same memory cell. Therefore, the new construct is assigned a type that abstracts the concrete location just created. This reference comes together with a capability to access it, associated to the reference type by a stacking operator ::. In our example, the capability is **rw** t [], representing **r**ead and **w**rite capability to the location t, which currently

contains a value of type [] (an empty record, or unit type). We recall that capabilities are subject to a linear disciple, and only play a role at the level of typing.

Also in line 2, we then open the existential by giving it a location variable (*pl*) and a regular variable (`l`) to refer that reference value. The capability is automatically unstacked and from there on is implicitly threaded through the program as necessary. Thus, on line 3 the type system carries the assumptions:

$$\Gamma \;=\; \_ : [] \;,\; pl : \textbf{loc} \;,\; \texttt{l} : \textbf{ref}\ pl \qquad ; \qquad \Delta \;=\; \textbf{rw}\ pl\ []$$

where Γ is the lexical environment (of persistent/pure resources), and Δ is a linear typing environment that contains all linear capabilities. Each linear capability must either be used up or passed on through the program (e.g. by returning it from a function). The type of the content of the reference `l` is known statically by looking up the capability for the **loc**ation *pl* to which `l` refers.

Capabilities can also be stacked in a function's arguments, as is the case on line 5 with the `initL` function. An argument capability of this form does not need to be present when the function is defined, but rather must be provided by the caller when the function is invoked. Consequently, and since capabilities are threaded implicitly (i.e. the resulting type has capabilities automatically stacked on top), `initL` has type:

$$!(\ \text{int} :: \textbf{rw}\ pl\ []\ \multimap\ [] :: \textbf{rw}\ pl\ \text{int}\ )$$

where the enclosing ! signals that the linear function (⊸) is pure — its definition does not capture any enclosing linear resources (capabilities), and so the function is safe to be invoked multiple times. Instead, function `initL` *borrows* [7, 30] the linear resources it requires (the capability to *pl*), returning them together with the result of the function. Primitive values and the unit type ([]) are also pure, but we will later show why these can be left unbanged due to subtyping (see Section 3). Thus, the `newPair` function has type:

$$!(\ [] \multimap\ !\texttt{Result} ::\ \textbf{rw}\ pl\ []\ *\ \textbf{rw}\ pr\ []\ )$$

where `Result` is an abbreviation for the record type:

$$\begin{array}{ll}
[\ \texttt{initL}: & !(\ \text{int} :: \textbf{rw}\ pl\ [] \multimap [] :: \textbf{rw}\ pl\ \text{int}\ ),\\
\ \ \texttt{initR}: & !(\ \text{int} :: \textbf{rw}\ pr\ [] \multimap [] :: \textbf{rw}\ pr\ \text{int}\ ),\\
\ \ \texttt{sum}: & !(\ [] :: \textbf{rw}\ pl\ \text{int} * \textbf{rw}\ pr\ \text{int} \multimap \text{int} :: \textbf{rw}\ pl\ \text{int} * \textbf{rw}\ pr\ \text{int}\ ),\\
\ \ \texttt{destroy}: & !(\ [] :: \textbf{rw}\ pl\ \text{int} * \textbf{rw}\ pr\ \text{int} \multimap []\ )\ ]
\end{array}$$

When we need to stack multiple capabilities, the use of the :: type constructor becomes a technical burden since it does not allow for reordering of capabilities. With this in mind, we introduce ∗ to form an unordered *bundle* of separate capabilities. Such a group of disjoint state models the notion of *state dimensions* such that the global state of the pair object obeys several, orthogonal, usage protocols (or typestates) that can be used independently in certain functions (such as `initL` and `initR`) but are required together on others (such as `sum` and `destroy`).

Our design differentiates the two basic type checking operations of moving a capability on top of some other type (::) and grouping sets of capabilities together (∗), so that each operation is orthogonal to the other and, although frequently used together, they are modeling separate typing aspects. However, other systems [23] do not make such distinction.

The type above is technically not free to leave the scope of the open constructs, since it depends on local names for the *pl* and *pr* locations. A first attempt to fix this issue is to provide *location polymorphism* [39] to abstract those location variables once again. If we change the previous code so that we pack both locations using the following construct to wrap the previous record definition:

```
4   <pl,<pr, { /* same functions */ } > >
```

we now obtain the following version of the `newPair` function type (where *pl* is replaced by *l* and *pr* by *r*):

$$!(\ [] \multimap \exists l.\exists r.(\ !\texttt{Result'} ::\ \textbf{rw}\ l\ []\ *\ \textbf{rw}\ r\ []\ )\ )$$

where `Result'` is the record type where all location variables now refer the packed names (*l* and *r*):

$$\begin{array}{ll}
[\ \texttt{initL}: & !(\ \text{int} :: \textbf{rw}\ l\ [] \multimap [] :: \textbf{rw}\ l\ \text{int}\ ),\\
\ \ \texttt{initR}: & !(\ \text{int} :: \textbf{rw}\ r\ [] \multimap [] :: \textbf{rw}\ r\ \text{int}\ ),\\
\ \ \texttt{sum}: & !(\ [] :: \textbf{rw}\ l\ \text{int} * \textbf{rw}\ r\ \text{int} \multimap \text{int} :: \textbf{rw}\ l\ \text{int} * \textbf{rw}\ r\ \text{int}\ ),\\
\ \ \texttt{destroy}: & !(\ [] :: \textbf{rw}\ l\ \text{int} * \textbf{rw}\ r\ \text{int} \multimap []\ )\ ]
\end{array}$$

With this version of the `newPair` type, the result can now safely leave the scope of the function's definition. However, it still exposes the internal representation of the pair object's state to client code. To fully abstract that representation, we can use traditional type abstraction mechanisms, using existential types, but to pack the types of the capabilities by wrapping the record with the following constructs:

```
4   < rw pl [], // hides capability as "Empty Left" (EL)
5   < rw pr [], // "Empty Right" (ER)
6   < rw pl int, // "Left initialized" (L)
7   < rw pr int, // "Right initialized" (R)
8     { /* same functions */ } > > > >
```

This expression will produce a type that completely abstracts the representation of the pair object's state, exposing only the requirements for using the pair in terms of abstracted linear capabilities (i.e. typestates). To provide some intuition on the meaning of the abstracted types, we choose names such as EL to represent the abstracted capability for the **L**eft part of the pair when that part is **E**mpty, etc. Therefore, the final version of our `newPair` function is simply:

$$!(\ [] \multimap \exists \texttt{EL}.\exists \texttt{ER}.\exists \texttt{L}.\exists \texttt{R}.(\ !\texttt{Result''} :: \texttt{EL} * \texttt{ER}\ )\ )$$

where `Result''` is the next record type:

$$\begin{array}{rl}
[\ \texttt{initL}: & !(\ \text{int} :: \texttt{EL} \multimap [] :: \texttt{L}\ ),\\
\ \ \texttt{initR}: & !(\ \text{int} :: \texttt{ER} \multimap [] :: \texttt{R}\ ),\\
\ \ \texttt{sum}: & !(\ [] :: \texttt{L} * \texttt{R} \multimap \text{int} :: \texttt{L} * \texttt{R}\ ),\\
\ \ \texttt{destroy}: & !(\ [] :: \texttt{L} * \texttt{R} \multimap []\ )\ ]
\end{array}$$

The following client code exemplifies how calling `initL` and `initR` only affects the parts of the state that the respective functions require.

```
1  open < EL, ER, L, R, x > = newPair({}) in // sugared open   Δ = EL, ER
2    x.initL(12);                                              Δ = L, ER
3    x.initR(34);                                              Δ = L, R
4    x.sum({});                                                Δ = L, R
5    x.destroy({})                                             Δ = ·
```

where the typing environments at the beginning of line 2 contain:

$$\Gamma = \texttt{x} : [...]\ ,\ \texttt{EL} : \textbf{type}\ ,\ \texttt{ER} : \textbf{type}\ ,\ \texttt{L} : \textbf{type}\ ,\ \texttt{R} : \textbf{type}\ ;\ \Delta = \texttt{EL}\ ,\ \texttt{ER}$$

Observe how the left and right typestates of the pair operate over independent dimensions of the complete state of the pair making each change separate [11, 36] up until they are required together (such as for invoking `sum` or `destroy`). In those cases, the type system implicitly stacks both types on top of the {} value (grouped together as ∗) and unstacks them if returned.

Perhaps a clearer use of this separation is exemplified by the following (untyped) function:

```
1  fun( x , y ).( x.initL(12); y.initR(34) )
```

where `initL` and `initR` are called over seemingly unrelated names. In our system we can give that function the type:

$$\forall A.\forall B.\forall C.\forall D.(\ [\texttt{initL} : \text{int} :: A \multimap [] :: B] \multimap [\texttt{initR} : \text{int} :: C \multimap [] :: D] \multimap [] :: B * D\ )$$

Such that the two function calls work completely independently, regardless if the state involved is or not referring the same underlying pair object—thus exploiting the disjointness of the pair's state.

2.2 Modularity

The previous example motivates the issue of how to expose the *type constraints* to clients without compromising safety, but also without revealing implementation details. We now show how our model supports code modularity by focusing on two intuitive concepts of *abstracting* and *hiding* state.

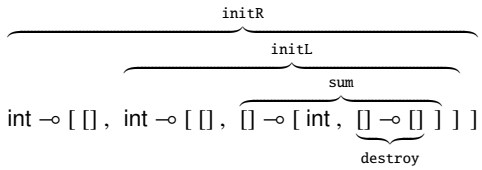
State abstraction With the last type representation of the pair object, we never exposed the internal details of the implementation and instead its type showed only the *type of the state* as in traditional *typestate* [6, 40, 41] approaches. Therefore, by adding traditional type abstraction mechanisms, we can abstract the actual (internal) type representation by only allowing client code to be aware of the *type* it must correctly use even though the actual state values are private — enabling implementation independence while ensuring that all stateful changes are respected and expressed in the types the client must obey. Since our capabilities are a purely static artifact, used by the type system to track type information on the state, the run-time of the client code remains isolated and completely decoupled from handling anything related to the internal state of objects (as would not be the case with manually threaded, value-based, capabilities).

Hiding state The alternative approach favors exposing the *behavior* of the state so that the state is completely *hidden* to clients—as if it were not there. Instead, stateful changes are expressed through usage constraints that enforce a specific *sequence of types* that will transparently thread the capabilities ("under the hood"), encapsulating their effects. Such perspective encodes a more "temporal" notion of usage, since there is a clear focus on what types will be available in future steps of a usage protocol. Therefore, behavioral types offer a complementary view to typestates since their main motivation is in offering less information about the state to clients so that it is only indirectly noticeable by the sequence of actions that are permitted to be invoked on a value. To exemplify how a type akin to a *behavioral type* [10] can be encoded in our model, consider the following sequential behavior of the pair object:

`initR` ; `initL` ; `sum` ; `destroy`

This behavioral type for the `pair`'s resulting record expresses that such a value initially (only) offers the field `initR` (we are omitting the function's type for brevity) and, after that call returns, it then (only) has the `initL` field available, and so on. After `destroy` is invoked, the value no longer has any remaining behavior meaning it finished its usage. Such a behavioral type completely hides the underlying capabilities of that state, favoring instead to expose a fixed sequence of function calls (accessible through the fields of the record) that threads the relevant capabilities completely transparently to clients.

The simplest way to express the behavior above is by using a type that returns the remainder of the usage protocol together with the return of a function (thus dodging the use of records), as in the following type:



The type uses immutable pairs (of the form $[A, A]$) to express the result of a function and the next behavior/type that must be respected (sidestepping the more verbose use of records, but retaining the same behavioral meaning). In contrast to the "typestate" style interface given above, this "behavioral type" does not just abstract, but rather completely hides the underlying capabilities of the stateful pair. The above type describes only one usage for using the object; others could have been assigned instead, for example initializing the left element of the pair before the right element. The implementation code is straightforward since it wraps the previous code to provide a result that also includes the following function of the described usage.

Mixing behavior and typestate Typestates and behavioral types offer complimentary views of the same phenomena. With typestates, the states are named, which can be a convenient abstraction, especially when there are multiple possible paths though the typestate/usage protocol. With behavioral types, the states are implicit, which simplifies the description of linear usages and makes it easy to provide structural equivalences. The two formalisms are interchangable and have equivalent expressiveness, so which one is preferred depends on the details of the particular situation.

Fortunately, the choice between typestate and behavioral types does not need to be fixed as, in our system, we can have a typestate object go through a behavioral phase and back. With such a scheme we can hide the state temporarily in a behavioral type and then later return to using abstract typestates — allowing the exchange or mixing of the two different code modularity approaches.

We illustrate this expressiveness by revisiting the pair example. Here we use an alternative encoding of behavior by storing the remaining behavior in a cell (similar to how the this pointer works in most object-oriented languages) so that the result of each function is unobstructed by the required continuation of behavior and more closely resembles object-based behavioral types. Likewise, for clarity, the usage of the object is encoded in a record so that the next behavior is accessible by field selection (also unlike in our previous example). In our illustration, the definition of the `newPair` function is unchanged, and returns an object with a state-based type. The client code—which knows nothing of the implementation details of the pair—then creates a behaviorally-typed wrapper for the object, as follows:

```
open < EL, ER, L, R, o > = newPair({}) in
  let behavioral = // the behavioral version of pair
    // 'this' reference to store the next behavior
    open <self,this> = new {} in
      // tags of record carefully picked to help readability
      this := {
        initLeft = fun( i : int :: rw self [] ).
          let result = o.initL(i) in
            // set the next behavior
            this := {
              initRight = fun( i : int :: rw self [] ).
                let result = o.initR(i) in
                  this := { // note that it returns the captured capability
                    addBoth = fun( _ : [] :: rw self [] ).
                      ( delete self ; o.sum({}) )
                  };
                  result
                end
            };
            result
          end
      };
      <self, this>
  end
in
... // client code that uses the behavior
```

Here the `behavioral` object has the type:

$$
\begin{array}{rcl}
\texttt{behavioral} & : & \exists t.(\ \textbf{ref}\ t\ ::\ \textbf{rw}\ t\ \texttt{initLType}\ ) \\
\texttt{initLType} & \triangleq & [\ \texttt{initLeft}\ :\ \mathsf{int} :: \textbf{rw}\ t\ []\ \multimap\ [] :: \textbf{rw}\ t\ \texttt{initRType}\ ] \\
\texttt{initRType} & \triangleq & [\ \texttt{initRight}\ :\ \mathsf{int} :: \textbf{rw}\ t\ []\ \multimap\ [] :: \textbf{rw}\ t\ \texttt{addType}\ ] \\
\texttt{addType} & \triangleq & [\ \texttt{addBoth}\ :\ [] :: \textbf{rw}\ t\ []\ \multimap\ \mathsf{int} :: \mathsf{L} * \mathsf{R}\ ]
\end{array}
$$

Note how the capabilities to the original pair typestate object are only returned after the specific sequence of behaviors is completed. Therefore, with this type, we support a client that is more specific in the kind of uses it makes of the state but oblivious to how those

uses are correlated with the object's capabilities. In fact, by adding a polymorphic type we can express a behavioral type entirely in terms of the functions that are called, so that the final resulting typestate (U) is an abstracted type:

```
Behavioral  ≜  ∃t.( ref t :: rw t initLType )
initLType   ≜  [ initLeft : int :: rw t [] ⊸ [] :: rw t initRType ]
initRType   ≜  [ initRight : int :: rw t [] ⊸ [] :: rw t addType ]
addType     ≜  [ addBoth : [] :: rw t [] ⊸ int :: U ]
```

This type allows clients to be abstract over the resulting capability U, while only needing to concern themselves with obeying the sequence of types that is encoded in such behavioral type. For instance, it enables the following client code:

```
<U>fun( o : Behavioral ). // polymorphic function on type U
  open <ind,ptr> = o in
    !ptr.initLeft(1);
    !ptr.initRight(2);
    !ptr.addBoth({})
  end
```

with maximum reuse since it only depends on that specific sequence of calls to be available, not on any particular number or kinds of typestates. Note that its return type would then be the capability U that is abstract, meaning it could be a set of capabilities, or even the result of combining several objects together to offer such behavior. Other practical applications of such behavioral generalization could be to model iterators that, once closed, return the (abstracted) capability to the collection from which they were extracted from without depending on specific kinds or numbers of abstract states of that collection.

In the remainder of the presentation, we focus on a typestate-based discussion since the conversion to behavioral types is generally straightforward to encode, even if potentially lengthy.

2.3 A Stack ADT

We now consider the stack example discussed in the introduction. The stack object is coded using a linear singly-linked list as representation type, the only private state that the stack uses. This example illustrates many prominent features of our approach namely how we are able to use case analysis on sums to indirectly test for capabilities to different abstract states—i.e. perform a dynamic test over the abstracted state of the object. Remember that capabilities are not values, and therefore if there are multiple possible alternative capabilities there is no direct way of distinguishing between them. However, a case analysis over tags in a value (using the syntax `tag#v`) can provide ways of deciding which of the different alternatives the abstraction's state must be in, and we can leverage this to learn about the capabilities that are available.

In the following code block we omit Γ and other parts of the typing environments when they are not relevant to express the intuition for how type checking proceeds.

```
1  let newStack = <T>fun( _ : [] ) .
   Γ = _ : [] , T : type  ;  Δ = ·
2    open <h,head> = new E#{} in //'head' contains tagged unit
   Γ = ... , h : loc , head : ref h  ;  Δ = rw h E#[]
3      {
4      push = fun( e : T :: EMPT[h] ⊕ ELEM[h] ).
   Γ = ...  ;  Δ = e : T , EMPT[h] ⊕ ELEM[h]
   [a] Δ = e : T , EMPT[h]    [b] Δ = e : T , ELEM[h]
5              open <n,next> = new !head in
   Γ = ... , n : loc , next : ref n
   [a] Δ = e : T , rw h [] , EMPT[n]    [b] Δ = e : T , rw h [] , ELEM[n]
   [a] [b] Δ = e : T , rw h [] , EMPT[n] ⊕ ELEM[n]
6                head := N#{ e ,  <n,next> } //tagged next node
   [a] [b] Δ = rw h N#[T, ∃p.( ref p :: EMPT[n] ⊕ ELEM[n] )]
   Δ = ELEM[h]
7              end,
8      pop = fun( _ : [] :: ELEM[h] ).
   Γ = ... , _ : []  ;  Δ = ELEM[h]
   Δ = rw h N#[T, ∃p.( ref p :: EMPT[n] ⊕ ELEM[n] )]
9              case !head of
10               N#[e,n] → // sugared pair open
   Δ = rw h [] , e : T , n : ∃p.( ref p :: EMPT[n] ⊕ ELEM[n] )
11                 open <t,ptr> = n in
   Γ = ... , t : loc , ptr : ref t  ;  Δ = rw h [] , e : T , EMPT[t] ⊕ ELEM[t]
   [a] Δ = rw h [] , e : T , EMPT[t]    [b] Δ = rw h [] , e : T , ELEM[t]
12                   head := !ptr;
   [a] Δ = rw t [] , e : T , EMPT[h]    [b] Δ = rw t [] , e : T , ELEM[h]
   Δ = rw t [] , e : T , EMPT[h] ⊕ ELEM[h]
13                   delete ptr;
   Δ = e : T , EMPT[h] ⊕ ELEM[h]
14                   e
   Δ = EMPT[h] ⊕ ELEM[h]
15                 end
16             end,
17     isEmpty = fun( _ : [] :: EMPT[h] ⊕ ELEM[h] ).
   Γ = ... , _ : []  ;  Δ = EMPT[h] ⊕ ELEM[h]
   [a] Δ = EMPT[h]    [b] Δ = ELEM[h]
18             case !head of // linear content (destructive read) thus
19               E#v → // requires (conservatively) reassigning the cell
20                 head := E#v;                                  [a] Δ = EMPT[h]
21                 Empty#{} : Empty#([] :: EMPT[h])              [a] Δ = ·
22             | N#n →
23                 head := N#n;                                  [b] Δ = ELEM[h]
24                 NonEmpty#{} : NonEmpty#([] :: ELEM[h])        [b] Δ = ·
25             end,                                                  Δ = ·
26     del = fun( _ : [] :: EMPT[h] ).     Γ = ... , _ : [] ; Δ = rw h E#[]
27             delete head                                           Δ = ·
28     }
29   end
```

In the code above, we used the following type definitions[1]:

```
EMPT  ≜  ∀p.( rw p E#[] )
ELEM  ≜  rec X.( ∀p.(rw p N#[T, ∃p'.( ref p' :: EMPT[p'] ⊕ X[p'] )] ) )
```

Thus, EMPT encodes an empty node, while ELEM is a non-empty node whose successor may or may not be empty (note the recursive definition on the type of the non-empty node). Alternatives (⊕) express the set of different capabilities that the following node may have. The necessity of such a type is directly linked to the fact that since capabilities are not values they also cannot be simply wrapped around a sum type to provide a distinctive tag that identifies each separate case, as otherwise we would fall into a system where capabilities must be manually threaded. Instead the type system is able to account for this uncertainty in the program state through the different alternatives listed in ⊕. The use of alternatives means that the type checker knows that we have one of several different capabilities, and consequently (to be safe) the expression must consider all those cases individually. Therefore, on line 5, to be able to use the different alternatives, we typecheck the expression considering each separate case individually (the cases are marked **[a]** and **[b]** in the typing environments). After analyzing each alternative, we can merge them back into a single program state that expresses the effects of both alternatives. Note that before line 6 we are weakening the typing environment (through subtyping as will be explained in Section 3) so that both alternatives then have the same typing context.

The newStack function has the non-abstracted type:

```
                    [ push :    T :: (EMPT[h] ⊕ ELEM[h]) ⊸ [] :: ELEM[h],
                      pop :     [] :: ELEM[h] ⊸ T :: (EMPT[h] ⊕ ELEM[h]),
∀T.[] ⊸      isEmpty :     [] :: (EMPT[h] ⊕ ELEM[h]) ⊸
                                 Empty#([] :: EMPT[h]) + NonEmpty#([] :: ELEM[h]),
                 del :     [] :: EMPT[h] ⊸ [] ] :: EMPT[h]
```

(Note that we are omitting ! for brevity, but all these functions, and the returned record, are pure since none of them captures capabilities in their scope.) By abstracting the capabilities we can

[1] $T \triangleq \forall p.A$ is a type definition/abbreviation, therefore we can apply it to a location variable, such as $T[q]$, without requiring ∀ to be a value.

construct a typestate abstraction for the stack object, with `E` (empty) and `NE` (non-empty) states, typed thus:

```
                          [ push :   T :: (E ⊕ NE) ⊸ [] :: NE,
                            pop :    [] :: NE ⊸ T :: (E ⊕ NE),
∀T.[] ⊸ ∃E.∃NE.      isEmpty :    [] :: (E ⊕ NE) ⊸
                                     Empty#([] :: E) + NonEmpty#([] :: NE),
                            del :    [] :: E ⊸ [] ] :: E
```

The most interesting aspect of using alternatives and sum types is shown in the `isEmpty` function. In it, we see that the result returns a sum type where the capabilities of the different alternatives are separated. Therefore, this function enables clients to test which case they are in, even though the state is abstracted and not immediately accessible—i.e. perform a *dynamic state test*—and with it tying the typing artifact with a value that can be tested. The implementation of this function distinguishes the alternatives indirectly based on the case branch that is taken as a result of the run-time value contained in the capability. Our system gains precision by *ignoring effects* of branches that are statically known to never be used. Therefore, on line 18, when the type checker is case analyzing the contents of `head` on alternative **[a]** it obtains the type `E#[]`. Instead of weakening such type to consider all the remaining branches of that case, we simply ignore the case branches that the type does not list (similar to ideas employed in [14, 18]). Consequently, for that alternative, type checking only takes into account the `E` tag and the respective branch.

Also note the subtle necessity in reassigning the `head` reference (lines 20 and 23) to restore its initial types after inspection. Since the contents of `head` include linear types, which cannot be duplicated, the de-reference of line 18 must leave the capability for *h* with the unit type so that the linear type can be bounded to the branch's variable without incurring in duplication. Therefore, the apparent redundant assignment operations are necessary to counter the destructive read that occurs at the type-level by refreshing the *h* cell with the same value as before.

The significance of our (slightly) non-standard case is perhaps clearer to understand in the following example where multiple alternatives also have different sets of available states. Consider the following code snippet:

```
1  Γ = x : ref l , l : loc , y : ref t , t : loc , z : ref u , u : loc
2  Δ = ( rw u HasX#[] ∗ rw l [] ) ⊕ ( rw u HasY#[] ∗ rw t [] )
3  [a] Δ = rw u HasX#[] , rw l []      [b] Δ = rw u HasY#[] , rw t []
5    case !z of                  [a] Δ = rw u [] , rw l []    [b] Δ = rw u [] , rw t []
7      HasX#_ →                                               [a] Δ = rw u [] , rw l []
9        delete x                                                   [a] Δ = rw u []
11   | HasY#_ →                                               [b] Δ = rw u [] , rw t []
13       delete y                                                   [b] Δ = rw u []
15   end                                                              Δ = rw u []
```

In the situation above, each branch deletes state that the other branch does not touch. This means that, although both branches know the same set of locations, their actions over the heap are distinct. The particularity of the static semantics of our case enables these seemingly incompatible alternative program states to be obeyed simultaneously by the same case expression as each program alternative does not need to respect those irreconcilable branches. Consequently, our type checking procedure is less conservative than traditional approaches that simply weaken the type to be case analyzed forcing it to use all available branches. However, our solution may leave "dead" branches since we do not ensure that each branch is used by at least one program alternative.

3. Technical Development

In this section, we carry out the full technical development of the system discussed. The appendix [28] includes the complete proof and a few additional constructs (such as pairs), encoded as abbreviations in our core language.

$\rho \in$ Location Constants (Addresses) $\quad t \in$ Location Variables $\quad p ::= \rho \mid t$

$\mathtt{l} \in$ Labels (Tags) $\quad \mathtt{f} \in$ Fields $\quad x \in$ Variables $\quad X \in$ Type Variables

$v \in$ Values	$::=$	ρ	(address)
		x	(variable)
		$\mathsf{fun}(x : A).e$	(function)
		$\langle t \rangle\, e$	(universal location)
		$\langle X \rangle\, e$	(universal type)
		$\langle p, v \rangle$	(pack location)
		$\langle A, v \rangle$	(pack type)
		$\{\overline{\mathtt{f} = v}\}$	(record)
		$\mathtt{l}\#v$	(tagged value)
$e \in$ Exprs.	$::=$	v	(value)
		$v[p]$	(location application)
		$v[A]$	(type application)
		$v.\mathtt{f}$	(field)
		$v\ v$	(application)
		$\mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}$	(let)
		$\mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open location)
		$\mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end}$	(open type)
		$\mathsf{new}\ v$	(cell creation)
		$\mathsf{delete}\ v$	(cell deletion)
		$!v$	(dereference)
		$v := v$	(assign)
		$\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{l}\#x \to e}\ \mathsf{end}$	(case)

Note that ρ is not source-level.

Figure 1. Expressions and values.

3.1 Core Language and Operational Semantics

In Figure 1 we introduce the syntax of our core language, which uses let-expanded form [38], for the sake of convenience. Note that capabilities are not present as values in the language, but just used at the level of types. Also note that we use p to range over positions, which include both location *variables* t and location *constants* ρ.

Our small step semantics (Figure 2) uses judgments of the form:

$$\langle\ H_0 \parallel e_0\ \rangle \mapsto \langle\ H_1 \parallel e_1\ \rangle$$

where a program execution is given by: $\langle\ \emptyset \parallel e\ \rangle \overset{\star}{\mapsto} \langle\ H \parallel v\ \rangle$, which states that starting from the empty heap ($\emptyset$) and an initial expression (e), we reach a final configuration of value v with heap H (after an arbitrary number of steps, $\mapsto$). The heap (H) binds addresses (ρ) to values (v) using the following format:

H	$::=$	$\emptyset$	(empty)
		$H\ ,\ \rho \hookrightarrow v$	(binding)

The semantics definition is fairly standard, except for a few small differences: the **(D:New)** and **(D:Delete)** reduction rules, as in [2], manipulate existential values that abstract the underlying location that was created or will be deleted, in order for the type system to properly handle such location abstractions (i.e. for the value to match the desired existential type).

3.2 Type System

The type structure, depicted in Figure 3, includes pure (!) types to express values that can be freely copied, linear functions (functions that can only be used once) and a few less familiar types. We use a *stacked* type (of the form $A_0 :: A_1$) to mean that type A_1 (actually, a capability) is stacked on top of A_0 (notice that the construct :: is not commutative). Similarly, we have the separation type ($A_0 * A_1$), which disjointly aggregates several types, in the spirit of separation logic (thus $*$ is commutative). We assume without explicitly stating that sum types are associative and commutative, and likewise for

$\langle H_0 \parallel e_0 \rangle \mapsto \langle H_1 \parallel e_1 \rangle$ **Dynamics, (D:*)**

(D:New)
$$\frac{\rho \textbf{ fresh}}{\langle H \parallel \mathsf{new}\ v \rangle \mapsto \langle H\ ,\ \rho \hookrightarrow v \parallel \langle \rho, \rho \rangle \rangle}$$

(D:Delete)
$$\frac{}{\langle H\ ,\ \rho \hookrightarrow v \parallel \mathsf{delete}\ \langle \rho, \rho \rangle \rangle \mapsto \langle H \parallel \langle \rho, v \rangle \rangle}$$

(D:Dereference)
$$\frac{}{\langle H\ ,\ \rho \hookrightarrow v \parallel !\rho \rangle \mapsto \langle H\ ,\ \rho \hookrightarrow v \parallel v \rangle}$$

(D:Assign)
$$\frac{}{\langle H\ ,\ \rho \hookrightarrow v_0 \parallel \rho := v_1 \rangle \mapsto \langle H\ ,\ \rho \hookrightarrow v_1 \parallel v_0 \rangle}$$

(D:Application)
$$\frac{}{\langle H \parallel (\mathsf{fun}(x : A).e)\ v \rangle \mapsto \langle H \parallel e\{v/x\} \rangle}$$

(D:Selection)
$$\frac{}{\left\langle H \parallel \{\overline{\mathtt{f} = v}\}.\mathtt{f}_i \right\rangle \mapsto \langle H \parallel v_i \rangle}$$

(D:LocApp)
$$\frac{}{\langle H \parallel (\langle t \rangle e)[\rho] \rangle \mapsto \langle H \parallel e\{\rho/t\} \rangle}$$

(D:TypeApp)
$$\frac{}{\langle H \parallel (\langle X \rangle e)[A] \rangle \mapsto \langle H \parallel e\{A/X\} \rangle}$$

(D:Case)
$$\frac{}{\left\langle H \parallel \mathsf{case}\ \mathtt{1}_i \# v_i\ \mathsf{of}\ \overline{\mathtt{1}\#x \to e}\ \mathsf{end} \right\rangle \mapsto \langle H \parallel e_i\{v_i/x_i\} \rangle}$$

(D:LocOpen)
$$\frac{}{\langle H \parallel \mathsf{open}\ \langle t, x \rangle = \langle \rho, v \rangle\ \mathsf{in}\ e\ \mathsf{end} \rangle \mapsto \langle H \parallel e\{v/x\}\{\rho/t\} \rangle}$$

(D:TypeOpen)
$$\frac{}{\langle H \parallel \mathsf{open}\ \langle X, x \rangle = \langle A, v \rangle\ \mathsf{in}\ e\ \mathsf{end} \rangle \mapsto \langle H \parallel e\{v/x\}\{A/X\} \rangle}$$

(D:Let)
$$\frac{}{\langle H \parallel \mathsf{let}\ x = v\ \mathsf{in}\ e\ \mathsf{end} \rangle \mapsto \langle H \parallel e\{v/x\} \rangle}$$

(D:LetCong)
$$\frac{\langle H_0 \parallel e_0 \rangle \mapsto \langle H_1 \parallel e_1 \rangle}{\langle H_0 \parallel \mathsf{let}\ x = e_0\ \mathsf{in}\ e_2\ \mathsf{end} \rangle \mapsto \langle H_1 \parallel \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \rangle}$$

$\{v/x\}$ is the (capture avoiding) substitution of variable x for value v, and analogous meaning for location and type variables variants.

Figure 2. Operational semantics.

A	::=	$!A$	(pure/persistent)
	|	$A \multimap A$	(linear function)
	|	$A :: A$	(stacking)
	|	$A * A$	(separation)
	|	X	(type variable)
	|	$\forall X.A$	(universal type quantification)
	|	$\exists X.A$	(existential type quantification)
	|	$[\overline{\mathtt{f} : A}]$	(record)
	|	$\forall t.A$	(universal location quantification)
	|	$\exists t.A$	(existential location quantification)
	|	**ref** p	(reference type)
	|	**rec** $X.A$	(recursive type)
	|	$\sum_i \mathtt{1}_i \# A_i$	(tagged sum)
	|	$A \oplus A$	(alternative)
	|	**rw** $p\ A$	(read-write capability to p)
	|	**none**	(empty capability)

Figure 3. Types (including capabilities) grammar.

alternatives (we only state that explicitly for the $*$ type). Our reference type only refers to the pure pointer, not to its usage capability, which is specified by a different construct. A capability describes the access kind (read-write, **rw**), the location it refers to (p) and the type of the value it currently holds (A); or is the empty capability (**none**). Finally, we include universal and existential types, both as location-dependent types and as a second order type quantifiers. Although our type structure presents capabilities and value inhabited types together, our type system ensures that those will be properly combined in complex type expressions, for instance, a **none** type is not inhabited by any value, and only capabilities will be stacked via ::. Alternatively, capabilities and value-inhabited types could also be presented separately. With our type grammar we simplify the syntax (by avoiding some redundancy in types that overlap as capabilities and as standard types) since such separation is technically not relevant because even if types that are not inhabited by a value are assumed (such as in a function's argument) they can never be created/introduced which effectively means that such value/type will never be usable anyway.

Our typing rules (Figure 4) use typing judgments of the form:

$$\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1$$

stating that with lexical environment Γ and linear resources Δ_0 (e.g., capabilities) we assign the expression e a type A and produce effects that result in Δ_1 (along the lines of a type and effect system [21]).

The typing environments are defined by:

Γ	::=	$\cdot$	(empty)
	|	$\Gamma, x : A$	(variable binding)
	|	$\Gamma, p :$ **loc**	(location variable assertion)
	|	$\Gamma, X :$ **type**	(type assertion)
Δ	::=	$\cdot$	(empty)
	|	$\Delta, x : A$	(linear binding)
	|	Δ, A	(capability)

We now discuss the main typing rules (shown in Figure 4).

(T:Ref) types any location constant as long as it refers a *known* location. Note that a location is more like a pointer or memory address, not a traditional reference since it still lacks the capability to actually access that location. **(T:Pure)** blesses as *pure* values that do not depend on any linear resources. **(T:Unit)** allows any value to be assigned a unit type since the unit type forbids any actual use of that value, so a unit usage is always safe. We support reading variables from the lexical environment **(T:Pure-Read)** requiring the type to be preceded by !. Destructive reads from the linear environment **(T:Linear-Read)** make a variable unavailable for further use. If a variable is of pure type then **(T:Pure-Elim)** allows it to be moved to the linear environment with its type explicitly tagged with !.

In **(T:New)** capabilities are threaded implicitly through the expression that will be assigned to the new cell. Since owning the **rw** capability implies uniqueness of access, deleting **(T:Delete)** is only allowed for a type that includes both the reference *and* the non-shared capability for that value. Our examples use an idiom for delete that avoids packing the location to be deleted (encoded as an idiom in the appendix), but for consistency with the new rule we use the packed reference version in here.

We allow two kinds of pointer dereference: a linear version **(T:Dereference-Linear)** that destroys the contents of the capability, and a pure version, **(T:Dereference-Pure)**, which leaves the same (pure) type behind. Note that although **(T:Dereference-Linear)** is destructive, operationally it will not destroy the contents of that cell. In this case, preservation of typing is ensured through the use of **(T:Unit)** so that the leftover value is effectively unusable "junk" from the type system's perspective when read from that

$\boxed{\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1}$ **Typing rules, (T:*)**

$$\text{(T:Ref)}\ \frac{}{\Gamma, \rho : \mathbf{loc}; \cdot \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot} \qquad \text{(T:Pure)}\ \frac{\Gamma; \cdot \vdash v : A \dashv \cdot}{\Gamma; \cdot \vdash v : !A \dashv \cdot} \qquad \text{(T:Unit)}\ \frac{}{\Gamma; \cdot \vdash v : [] \dashv \cdot} \qquad \text{(T:Pure-Read)}\ \frac{}{\Gamma, x : A; \cdot \vdash x : !A \dashv \cdot} \qquad \text{(T:Linear-Read)}\ \frac{}{\Gamma; x : A \vdash x : A \dashv \cdot} \qquad \text{(T:Pure-Elim)}\ \frac{\Gamma, x : A_0; \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma; \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}$$

$$\text{(T:New)}\ \frac{\Gamma; \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma; \Delta_0 \vdash \mathsf{new}\ v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1} \qquad \text{(T:Delete)}\ \frac{\Gamma; \Delta_0 \vdash v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}{\Gamma; \Delta_0 \vdash \mathsf{delete}\ v : \exists t.A \dashv \Delta_1} \qquad \text{(T:Assign)}\ \frac{\Gamma; \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \quad \Gamma; \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1}{\Gamma; \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0}$$

$$\text{(T:Dereference-Linear)}\ \frac{\Gamma; \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma; \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\ p\ []} \qquad \text{(T:Dereference-Pure)}\ \frac{\Gamma; \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ !A}{\Gamma; \Delta_0 \vdash !v : !A \dashv \Delta_1, \mathbf{rw}\ p\ !A} \qquad \text{(T:Record)}\ \frac{\overline{\Gamma; \Delta \vdash v : A \dashv \cdot}}{\Gamma; \Delta \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot} \qquad \text{(T:Selection)}\ \frac{\Gamma; \Delta_0 \vdash v : [\overline{\mathtt{f} : A}] \dashv \Delta_1}{\Gamma; \Delta_0 \vdash v.\mathtt{f}_i : A_i \dashv \Delta_1}$$

$$\text{(T:Function)}\ \frac{\Gamma; \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma; \Delta \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot} \qquad \text{(T:Application)}\ \frac{\Gamma; \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \quad \Gamma; \Delta_1 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_2}{\Gamma; \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2} \qquad \text{(T:Let)}\ \frac{\Gamma; \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \quad \Gamma; \Delta_1, x : A_0 \vdash e_1 : A_1 \dashv \Delta_2}{\Gamma; \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2}$$

$$\text{(T:Forall-Loc)}\ \frac{\Gamma, t : \mathbf{loc}; \Delta \vdash e : A \dashv \cdot}{\Gamma; \Delta \vdash \langle t \rangle\, e : \forall t.A \dashv \cdot} \qquad \text{(T:Loc-App)}\ \frac{p : \mathbf{loc} \in \Gamma \quad \Gamma; \Delta_0 \vdash v : \forall t.A \dashv \Delta_1}{\Gamma; \Delta_0 \vdash v[p] : A\{p/t\} \dashv \Delta_1} \qquad \text{(T:Loc-Pack)}\ \frac{\Gamma; \Delta \vdash v : A\{p/t\} \dashv \cdot}{\Gamma; \Delta \vdash \langle p, v \rangle : \exists t.A \dashv \cdot} \qquad \text{(T:Loc-Open)}\ \frac{\Gamma; \Delta_0 \vdash v : \exists t.A_0 \dashv \Delta_1 \quad \Gamma, t : \mathbf{loc}; \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma; \Delta_0 \vdash \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

$$\text{(T:Forall-Type)}\ \frac{\Gamma, X : \mathbf{type}; \Delta \vdash e : A \dashv \cdot}{\Gamma; \Delta \vdash \langle X \rangle\, e : \forall X.A \dashv \cdot} \qquad \text{(T:Type-App)}\ \frac{\Gamma \vdash A_1\ \mathbf{type} \quad \Gamma; \Delta_0 \vdash v : \forall X.A_0 \dashv \Delta_1}{\Gamma; \Delta_0 \vdash v[A_1] : A_0\{A_1/X\} \dashv \Delta_1} \qquad \text{(T:Type-Pack)}\ \frac{\Gamma; \Delta \vdash v : A_0\{A_1/X\} \dashv \cdot}{\Gamma; \Delta \vdash \langle A_1, v \rangle : \exists X.A_0 \dashv \cdot} \qquad \text{(T:Type-Open)}\ \frac{\Gamma; \Delta_0 \vdash v : \exists X.A_0 \dashv \Delta_1 \quad \Gamma, X : \mathbf{type}; \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma; \Delta_0 \vdash \mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

$$\text{(T:Cap-Elim)}\ \frac{\Gamma; \Delta_0, x : A_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma; \Delta_0, x : A_0 :: A_1 \vdash e : A_2 \dashv \Delta_1} \qquad \text{(T:Cap-Stack)}\ \frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1} \qquad \text{(T:Cap-Unstack)}\ \frac{\Gamma; \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1} \qquad \text{(T:Alternative-Left)}\ \frac{\Gamma; \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \quad \Gamma; \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma; \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1} \qquad \text{(T:Alternative-Right)}\ \frac{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma; \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \oplus A_2}$$

$$\text{(T:Subsumption)}\ \frac{\Delta_0 <: \Delta_1 \quad \Gamma; \Delta_1 \vdash e : A_0 \dashv \Delta_2 \quad A_0 <: A_1 \quad \Delta_2 <: \Delta_3}{\Gamma; \Delta_0 \vdash e : A_1 \dashv \Delta_3} \qquad \text{(T:Frame)}\ \frac{\Gamma; \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma; \Delta_0, \Delta_2 \vdash e : A \dashv \Delta_1, \Delta_2} \qquad \text{(T:Tag)}\ \frac{\Gamma; \Delta \vdash v : A \dashv \cdot}{\Gamma; \Delta \vdash \mathtt{1}\#v : \mathtt{1}\#A \dashv \cdot} \qquad \text{(T:Case)}\ \frac{\Gamma; \Delta_0 \vdash v : \sum_i \mathtt{1}_i \# A_i \dashv \Delta_1 \quad \overline{\Gamma; \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2} \quad i \leq j}{\Gamma; \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{1}_j \# x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2}$$

Note: all bound variables of a construct must be fresh in the respective rule's conclusion (i.e. x must be fresh in (T:Let) conclusion, etc.).

Figure 4. Static semantics.

capability after such dereference. Assigning **(T:Assign)** requires both the reference and the respective capability.

A record, **(T:Record)**, contains a set of labeled *choices*, its fields. Since selection, **(T:Selection)**, will pick one and discard the rest, we requires each field to produce the same effect. Therefore, even if such fields contain a linear type, they can be safely discarded as their effects would be equal to those produced by the selected field. Thus, a record type is akin to a linear (labeled) intersection type.

Since a function, **(T:Function)**, depends on the linear resources inside of Δ (which it captures), it must be linear (although it can be rendered exponential/pure (!), using **(T:Pure)** if that environment is empty). We rely on the combination with **(T:Pure-Elim)** to use non-linear arguments, so that they can all initially be assumed to be of linear kind. **(T:Application)** is the traditional rule. The fact that our basic function type is linear is not an actual restriction to the language expressiveness, since it may be combined with other type constructors which break linearity, namely !, in a fine-grained way. Observe how values (which includes functions, tagged values, etc.) have no resulting effects ($\cdot$) since they have no pending computations.

(T:Alternative-Left) encodes the uses of the alternative type. Note that it does not require the resulting type to distinguish between the different alternatives; we just require that each case is considered. This means that an alternative type is only usable when there exists an expression that satisfy all its cases, as otherwise such type can only be threaded and never inspected. Also note that **(T:Alternative-Right)** is derivable through subtyping, but is shown for consistency of the presentation.

(T:Forall-Loc), **(T:Loc-App)**, **(T:Loc-Open)** and **(T:Loc-pack)** provide location variable abstraction, while their -Type counterparts do the same for types. Existential quantification means that we can hide the underlying location (aliasing) relation, so that it can be renamed and reused in different contexts through the open construct. Note that p must either be in Γ if it occurs in A or, if it does not occur in A, then abstracting such location has no real consequence since the substitution has no impact in A (it remains well-formed regardless of whether p is valid or not). As with **(T:Loc-Pack)**, the absence of a $\Gamma \vdash A_1$ **type** premise in **(T:Type-Pack)** is related to the fact that, for A_1 to occur in A_0 it must be a proper **type**, or if it does not occur in A_0 then it is of no consequence whether A_1 is or not a **type**.

(T:Tag) and **(T:Case)** provide the introduction and elimination of tagged sum types. Carefully notice that the later rule is not completely standard. In order to rule out potential conservative inclusions of effects, we do not require the sum type to consider *all* tags listed in the case and instead allow the construct to have other "extra" branches that are ignored if they are not listed in the

sum type. Although they are statically known to never be executed, alternative program states may still need to consider them. The alternative and case typing rules interact in our language in a rather interesting way: by ignoring these "extra" branches we are able to have the same case expression obey seemingly incompatible alternatives that otherwise would not be possible to type check, adding extra flexibility.

The **(T:SUBSUMPTION)** rule allows an expression to rely on weaker assumptions while ensuring a stronger result than needed by its context (the subtyping rules are detailed in the next sub-section). Our **(T:FRAME)** accounts for simple disjoint separation [11, 36] so that parts of the heap that are not needed to type check an expression cannot be changed by that expression either.

(T:CAP-ELIM), **(T:CAP-STACK)** and **(T:CAP-UNSTACK)** manage our valueless capabilities through non-syntax-directed elimination, stacking, and unstacking of these non-indexed elements of the linear typing environment.

We impose the expected global constraint on all constructs with bound variables (such as let, functions, etc.): such variables must be fresh in the conclusion of the respective typing rule.

3.3 Subtyping

$\boxed{A_0 <: A_1}$ **Subtyping on types, (ST:*)**

$$\frac{}{A <: A}\ (\text{ST:SYMMETRY}) \qquad \frac{}{!A <: A}\ (\text{ST:TOLINEAR}) \qquad \frac{A_0 <: A_1}{!A_0 <: !A_1}\ (\text{ST:PURE}) \qquad \frac{}{!A <: ![]}\ (\text{ST:TOP}) \qquad \frac{}{\mathbf{ref}\ p <:\ !(\mathbf{ref}\ p)}\ (\text{ST:REF})$$

$$\frac{A_1 <: A_3 \qquad A_2 <: A_0}{A_0 \multimap A_1 <: A_2 \multimap A_3}\ (\text{ST:FUNCTION}) \qquad \frac{}{\sum_i 1_i \# A_i <: 1' \# A' + \sum_i 1_i \# A_i}\ (\text{ST:SUM})$$

$$\frac{A_0 <: A_1}{\exists t.A_0 <: \exists t.A_1}\ (\text{ST:LOC-EXISTS}) \qquad \frac{A_0 <: A_1}{\forall t.A_0 <: \forall t.A_1}\ (\text{ST:LOC-FORALL}) \qquad \frac{A_0 <: A_1}{\exists X.A_0 <: \exists X.A_1}\ (\text{ST:TYPE-EXISTS}) \qquad \frac{A_0 <: A_1}{\forall X.A_0 <: \forall X.A_1}\ (\text{ST:TYPE-FORALL})$$

$$\frac{A_i <: A_i'}{[\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i']}\ (\text{ST:RECORD}) \qquad \frac{i > 0}{[\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : A}]}\ (\text{ST:DISCARD})$$

$$\frac{}{[\overline{\mathtt{f} : !A}] <: ![\overline{\mathtt{f} : !A}]}\ (\text{ST:PURIFYREC}) \qquad \frac{A_0 <: A_1 \qquad A_2 <: A_3}{A_0 :: A_2 <: A_1 :: A_3}\ (\text{ST:STACK}) \qquad \frac{A_0 <: A_1}{\mathbf{rw}\ p\ A_0\ <:\ \mathbf{rw}\ p\ A_1}\ (\text{ST:CAP})$$

$$\frac{}{A_0 * A_1 <: A_1 * A_0}\ (\text{ST:COM}) \qquad \frac{A_1 <: A_2}{A_0 * A_1 <: A_0 * A_2}\ (\text{ST:CONG}) \qquad \frac{}{(A_1 * A_2) * A_3 <: A_1 * (A_2 * A_3)}\ (\text{ST:ASSOC})$$

$$\frac{}{\mathbf{rec}\ X.A <: A\{\mathbf{rec}\ X.A/X\}}\ (\text{ST:UNFOLD}) \qquad \frac{}{A\{X/\mathbf{rec}\ X.A\} <: \mathbf{rec}\ X.A}\ (\text{ST:FOLD}) \qquad \frac{A_0 <: A_1}{\mathbf{rec}\ X.A_0 <: \mathbf{rec}\ X.A_1}\ (\text{ST:REC})$$

$\boxed{\Delta_0 <: \Delta_1}$ **Subtyping on deltas, (SD:*)**

$$\frac{}{\Delta, A_0, A_1 <:> \Delta, A_0 * A_1}\ (\text{SD:STAR}) \qquad \frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, x : A_0 <: \Delta_1, x : A_1}\ (\text{SD:VAR}) \qquad \frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, A_0 <: \Delta_1, A_1}\ (\text{SD:TYPE})$$

$$\frac{}{\Delta <: \Delta}\ (\text{SD:SYMMETRY}) \qquad \frac{}{\Delta <:> \Delta, \mathbf{none}}\ (\text{SD:NONE}) \qquad \frac{}{\Delta, A_0 <: \Delta, A_0 \oplus A_1}\ (\text{SD:ALTERNATIVE-R}) \qquad \frac{\Delta_0, A_0 <: \Delta_1 \qquad \Delta_0, A_1 <: \Delta_1}{\Delta_0, A_0 \oplus A_1 <: \Delta_1}\ (\text{SD:ALTERNATIVE-L})$$

Figure 5. Subtyping rules.

Our subtyping rules are defined in Figure 5 using the *subtyping judgment* of the form $A_0 <: A_1$ that states A_0 is a subtype of A_1, meaning that A_0 can be used wherever A_1 is expected. Similarly, we also define subtyping on the linear typing environment Δ with an analogous judgment of equivalent meaning. We highlight the less obvious rules: unlike in traditional non-linear systems, our linear capabilities only need to be *read-consistent* (not write) which yields the additional flexibility shown in **(ST:CAP)** (i.e. due to their linearity they are covariant, as in other linear/affine systems [12]); **(ST:UNFOLD)**, **(ST:FOLD)** are used to fold and unfold a recursive type; **(SD:STAR)** allows to bundle several linear resources together using $*$, or break this type into its components (when the rule is read right to left). Also note **(ST:REF)** that enables us to not have to bang every reference since the type system can handle such through this subtyping rule, and similarly occurs for other primitive values.

$\boxed{\Gamma; \Delta \vdash H}$ **Store typing, (STR:*)**

$$\frac{}{\cdot; \cdot \vdash \cdot}\ (\text{STR:EMPTY}) \qquad \frac{\Gamma; \Delta \vdash H}{\Gamma, \rho : \mathbf{loc}; \Delta \vdash H}\ (\text{STR:LOC}) \qquad \frac{\Gamma; \Delta, A_0, A_1 \vdash H}{\Gamma; \Delta, A_0 * A_1 \vdash H}\ (\text{STR:STAR})$$

$$\frac{\Gamma; \Delta \vdash H}{\Gamma; \Delta, \mathbf{none} \vdash H}\ (\text{STR:NONE}) \qquad \frac{\Gamma; \Delta, A_0 \vdash H}{\Gamma; \Delta, A_0 \oplus A_1 \vdash H}\ (\text{STR:ALTERNATIVE}) \qquad \frac{\Gamma; \Delta, \Delta_v \vdash H \qquad \Gamma; \Delta_v \vdash v : A \dashv \cdot}{\Gamma; \Delta, \mathbf{rw}\ \rho\ A \vdash H, \rho \hookrightarrow v}\ (\text{STR:BINDING})$$

Figure 6. Store typing.

3.4 Technical Results

We proved our system sound through *progress* and *preservation* theorems (detailed proofs shown in [28]). These results rely on the definition of store typing (Figure 6) which relates well-formed environments with heaps. Store typing uses judgments of the form $\Gamma; \Delta \vdash H$ stating that the heap H conforms with the elements contained in Γ and Δ. Although typing rules such as the frame rule may appear to potentially extend our linear resources in arbitrary ways, our theorems show that when starting from a properly typed store we will never reach invalid store states. Note that (STR:ALTERNATIVE) assumes that $\oplus$ is commutative, so we introduce a single store typing rule. We now state our main theorems:

Theorem 1 (Progress)**.** *If e_0 is a closed expression (and where Γ and Δ_0 are also closed) such that:*

$$\Gamma; \Delta_0 \vdash e_0 : A \dashv \Delta_1$$

then either:

- *e_0 is a value, or;*
- *if exists H_0 such that $\Gamma; \Delta_0 \vdash H_0$ then $\langle H_0 \parallel e_0 \rangle \mapsto \langle H_1 \parallel e_1 \rangle$.*

The progress statement ensures that all well-typed expressions are either values or, if there is a heap that obeys the typing assumptions, the expression can step to some other program state — i.e. a well-typed programs never gets stuck.

Theorem 2 (Preservation)**.** *If e_0 is a closed expression such that:*

$$\Gamma_0; \Delta_0 \vdash e_0 : A \dashv \Delta \qquad \Gamma_0; \Delta_0 \vdash H_0 \qquad \langle H_0 \parallel e_0 \rangle \mapsto \langle H_1 \parallel e_1 \rangle$$

then, for some Δ_1, Γ_1:

$$\Gamma_0, \Gamma_1; \Delta_1 \vdash H_1 \qquad \Gamma_0, \Gamma_1; \Delta_1 \vdash e_1 : A \dashv \Delta$$

The theorem above requires the initial expression e_0 to be closed so that it is ready for evaluation. The preservation statement ensures that the resulting effects (Δ) and type (A) of the expression remains the same throughout the execution so that the initial typing is preserved by the dynamics of the language. Heap modifications may occur (such as on delete or new) but these preserve the previously known locations and are a consequence of the operational semantics (i.e. of the resulting H_1).

Note that no instrumentation of the operational semantics is necessary since in our system the presence of a memory cell in the

heap must also have its respective capability in Δ (our capabilities are linear). Any kind of wrong use of state can be reduced to a stuck condition in the language (for instance deleting a cell too early will cause the program to become stuck when an alias tries to access that location later on), and similar situations occur when accessing unique memory cells due to protocol violations, since the language supports strong updates. Therefore such two theorems are enough to ensure that state is properly used even if done through multiple aliases of the same underlying location. Through both theorems we ensure traditional type safety in the sense that correct programs do not go wrong since every well-typed program either terminates with a value of the expected type or will just run forever.

4. Related Work

We have shown how key concepts related to the general idea of typestate can be extended to a substructural setting and precisely captured by a fairly canonical substructural type-and-effect system, building on linearity, second order-polymorphism, and location-dependent types. We provide technical type safety results for our language, which give a solid foundation for somewhat ad-hoc mechanisms appearing in the typestate literature. Our type system is also distinctive in its use of implicitly threaded capabilities (in a completely substructural setting), in its characterization of typestates through existential abstraction, and in establishing a preliminary formal bridge between intuitively related approaches, such as state-based type systems (as in typestate) and behavior-based type systems (as in behavioral types). We now discuss some recent closely related work.

Permissions [7], a type mechanism to constrain the access to aliased mutable objects, have been actively explored, with systems based on linear logic [22] being the closest to our own. In [47] Wadler proposed the use of linear types to handle effects. Although such system still tied references with their content, it also supports a let! block to temporarily, but safely, relax linearity enabling multiple reads of the same cell in a scoped block where writes to that cell are forbidden until uniqueness is restored. More practical uses of typestates [40, 41] were pioneered by Fähndrich and DeLine, where they employed permissions to control aliasing through mechanisms such as adoption and focus [16, 19] to enable temporary breaks from linearity (a technique that was later further improved in [9]), and pack/unpack [15, 16] to distinguish when an object is or not consistent with its internal invariant, and where both inconsistency states should not be visible to other program contexts. Bierhoff and Aldrich further improved the practicality of such systems by combining it with *fractional permissions* [8] to defined *access permissions* [6], a flexible sharing mechanism centered around bucketing the kind of accesses an alias is allowed to perform into a fixed set of permission primitives, each modeling a specific type of interaction with the shared state. Subsequent work with Beckman [3] showed that this permission system is also adequate, and efficient [4], to express concurrent uses of shared state through atomic blocks. *Masked types* [34] adapts the typestate idea into a system that is specially targeted to solve the problem of object initialization, while allowing complex pointer topologies such as cyclic dependencies to remain safe.

From a technical perspective, we believe our system captures, in a uniform and integrated way, many of the somehow seemingly fragmentary features of the systems above, reconstructing them from a smaller set of fairly canonical type-theoretic primitives, based on a combination of linear location capabilities (introduced by [2]), second-order polymorphism, and implicit threading of capabilities via the stacking mechanism. Still, we omit from this work any sharing mechanism for statically disconnected aliases, meaning that many advanced uses of shared state that are possible in the works listed above are not feasible in ours. This limitation is being addressed in future work, building on the basic foundation shown here, and following the same design principles.

Our language also builds on the work of *alias types* [39, 48] in the way that it expresses aliasing information within the types; while our separation of pure reference types from linear capabilities was pioneered by $\mathbf{L}^3$ [2]. $\mathbf{L}^3$ targets a lower-level of abstraction, with manually-threaded capabilities and no mechanism for abstraction beyond location variables. It also lacks support for more practical types such as recursive and sum types. However, their core system goes beyond our work by reasoning about program termination, and also enabling invariant-based sharing mechanisms similar to adoption/focus (here called thaw/freeze/refreeze). Technically, our development expands on basic $\mathbf{L}^3$ concepts, extending them to the context of a type-and-effect system allowing for implicitly threaded capabilities, sum types and polymorphism. Our use of sum types, and the specific typing discipline for our case construct is also distinctive from [48] (although similar ideas also appear at least in [14, 18]) by adding extra flexibility in typing, and in particular by being expressive enough to lift, in a robust type safe way, dynamically checking variant values to dynamically checking abstract (type)states.

Krishnaswami *et al.* [24] develop a type system that is superficially substructural, since it employs a "fiction of disjointness" to allow sharing of mutable cells to occur underneath that layer. They also adapt $\mathbf{L}^3$, but in an affine variant. Our system simply does not handle the sharing problem in the generic terms that they describe, and our focus is instead in providing interfaces that represents abstract states and the problems that arise from it (such as supporting dynamic state tests). Although there are many similarities in the use of a substructural type system (although ours is technically a type-and-effect system with linear capabilities), our encoding of alternative program states is distinctive.

Alms [44] is an affine system with support for, among other things, manually threaded capabilities that can be used to "decorate" mutable state enabling a mechanism similar to the separation of pure references and linear capabilities that we use. However, capabilities do not express the contents of cell and instead are used as a token to manage access to abstract affine types. This distinction makes it not immediately obvious if their system could encode a more complex resource-aware abstraction as we provide with typestates. Regardless, our distinctive setting with valueless capabilities enables additional expressiveness that is not possible there.

In [20], Gay, *et al.* model (object) protocols through the use of *session types* [43], both locally and in a distributed environment, by generalizing the notion of channels to include method calls to an object. In such model all forms of aliasing are forbidden. Mazurak *et al.* [26] use linear types in an extension to System F that has no direct support for state. Yet, through the flexibility of their linear kinds they enable sufficient expressiveness to encode regular protocols. Caires and Seco [10], following ideas from process algebras, introduce the notion of behavioral separation where *behavioral types* are used to model complex usage protocols with the possibility of aliasing and where separation ensures safety in the use of aliased state. Our typestate focus means that this work emphasis state abstraction, and therefore we lack the expressiveness to encode all above mentioned kinds of usage protocols (such as the concurrency related type constructs of [10]) that express the behavior of hidden state, but our language does offer the distinctive feature of supporting both modularity approaches.

In [33] Parkinson and Bierman introduce the notion of *abstract predicates* which enriches a logical framework with predicates whose representation is only known inside a module, and use it to prove functional properties of programs. Abstract predicates encode a similar notion to typestates that are, however, not limited to a finite number of abstract states (since predicates can be

parametric on some variables). Consequently, typestates generally target a more lightweight verification. Beyond that distinction, the main technical difference resides in that we built our system out of type-theoretic primitives showing how these building blocks are enough to encode similar (albeit simpler) notions of abstraction.

Hoare Type Theory [31, 32] is a very expressive dependent type theory, enabling types to express specification details, and supporting reasoning about full functional correctness properties of a program. In our case, we investigate a much simpler and less expressive substructural system with the goal to provide lightweight type-like verification, focusing on the distillation of basic typing constructs to disciplined state usage, and potentially more amenable to automation.

In [37] Rondon *et al.* propose a static refinement type system to verify state in low-level C programs, where type inference significantly reduces the annotation burden. Their approach tackles pointer arithmetic and a series of issues related to their handling of low-level code, while supporting a form of predicate abstraction for their refinement types. By relying on typed heaps, this work constitutes an alternative, low-level, approach to handling state by fitting it into an existing languages. Our approach differs in that it is instead centered in rooting the type system directly on the core language, so that verification is a direct process, closely modeled by a substructural type theory, and backed by formal type soundness results.

5. Conclusions and Future Work

By focusing on a small set of primitive and type-theoretic concepts, we have developed a practical substructural type-and-effect system where valueless capabilities are threaded implicitly. Although close to classical type-theoretic concepts, our system is able to model fairly complex practical typestate aspects, namely state abstraction, dynamic state tests and state dimensions. Finally, we also showed how, by supporting mechanisms for abstracting and hiding state, our language is naturally able to combine in a uniform way some core concepts of typestates and behavioral types.

In this work, we only addressed aliasing in contexts where they either refer fully disjoint parts of an object's abstracted state space, or are "statically" connected (in the sense that the type system statically knows that they reference the same memory location). We are, however, investigating in ongoing work some very natural extensions to the current model which will allow much more flexible ways of sharing, exploiting state protocols at the shared (but coordinated) state level, rather than at the uniquely or disjointly owned level, as we have done in here.

A prototype implementation of this system, able to type check all the examples used in this paper (and others), is publicly available at [1]. It relies on a few minimal additional type annotations to direct type checking and make it decidable. The type checking algorithm for our language is not however to be seen as a contribution of this paper, and will be described in a separate publication. The implementation runs directly in a modern web browser with the intent to make it easy to use and facilitate experimentation (although Google Chrome is recommended, due to its efficient JavaScript engine).

Acknowledgments

This work was partially supported by Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under grant SFRH / BD / 33765 / 2009 and the Information and Communication Technology Institute at CMU, CITI PEst-OE / EEI / UI0527 / 2011, the U.S. National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages," and the U.S. Air Force Research Laboratory. We thank the anonymous reviewers for their helpful comments.

References

[1] Prototype. https://code.google.com/p/dead-parrot/.

[2] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundam. Inform.*, 77(4):397–449, 2007.

[3] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA'08*, pages 227–244. ACM, 2008.

[4] N. E. Beckman, Y. P. Kim, S. Stork, and J. Aldrich. Reducing stm overhead with access permissions. In *IWACO'09*, pages 2:1–2:10. ACM, 2009.

[5] N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP'11*, pages 2–26. Springer-Verlag, 2011.

[6] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA 2007*, pages 301–320, 2007.

[7] J. Boyland. Alias burying: unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, May 2001.

[8] J. Boyland. Checking interference with fractional permissions. In *Proc. Static Analysis Symposium*, pages 55–72, 2003.

[9] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *POPL '05*, pages 283–295. ACM, 2005.

[10] L. Caires and J. a. C. Seco. The type discipline of behavioral separation. In *POPL '13*, pages 275–286. ACM, 2013.

[11] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. Logic in Computer Science*, pages 366–378, 2007.

[12] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *ICFP '08*, pages 213–224. ACM, 2008.

[13] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL '99*, pages 262–275. ACM, 1999.

[14] R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP '00*, pages 198–208. ACM, 2000.

[15] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 2001*, pages 59–69. ACM, 2001.

[16] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490. Springer, 2004.

[17] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL '13*, pages 287–300. ACM, 2013.

[18] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *FOSSACS '03*, pages 250–266. Springer-Verlag LNCS 2620, 2003.

[19] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI 2002*, pages 13–24. ACM, 2002.

[20] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312, 2010.

[21] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *LFP '86*, pages 28–38. ACM, 1986.

[22] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[23] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP'10*, pages 354–378. Springer-Verlag, 2010.

[24] N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP 2012*, pages 41–54. ACM, 2012.

[25] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[26] K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight linear types in system f°. In *TLDI '10*, pages 77–88. ACM, 2010.

[27] F. Militão, J. Aldrich, and L. Caires. Aliasing control with view-based typestate. In *FTfJP*, pages 7:1–7:7. ACM, 2010.

[28] F. Militão, J. Aldrich, and L. Caires. Substructural typestates (technical appendix), 2013. http://www.cs.cmu.edu/~foliveir/papers/plpv14-appendix.pdf

[29] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988.

[30] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL 2012*, pages 557–570. ACM, 2012.

[31] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *ICFP '06*, pages 62–73. ACM, 2006.

[32] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *ESOP'07*, pages 189–204. Springer-Verlag, 2007.

[33] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.

[34] X. Qi and A. C. Myers. Masked types for sound object initialization. In *POPL*, pages 53–65, 2009.

[35] J. Reynolds. Syntactic control of interference part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer Berlin Heidelberg, 1989.

[36] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Logic in Computer Science*, pages 55–74, 2002.

[37] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL '10*, pages 131–144. ACM, 2010.

[38] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. LISP and Functional Programming*, pages 288–298, 1992.

[39] F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, pages 366–381. Springer-Verlag, 2000.

[40] R. E. Strom. Mechanisms for compile-time enforcement of security. In *POPL '83*, pages 276–284. ACM, 1983.

[41] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[42] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter. First-class state change in plaid. In *OOPSLA '11*, pages 713–732. ACM, 2011.

[43] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proc. of PARLE Conference on Parallel Architectures and Languages Europe*, pages 398–413. Springer-Verlag, 1994.

[44] J. A. Tov and R. Pucella. Practical affine types. In *POPL*, pages 447–458. ACM, 2011.

[45] J. Van Den Bos and C. Laffra. Procol: a parallel object language with protocols. In *OOPSLA '89*, pages 95–102. ACM, 1989.

[46] S. van Staden and C. Calcagno. Reasoning about multiple related abstractions with multistar. In *OOPSLA '10*, pages 504–519. ACM, 2010.

[47] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.

[48] D. Walker and J. G. Morrisett. Alias types for recursive data structures. In *Selected papers from the Third International Workshop on Types in Compilation*, TIC '00, pages 177–206. Springer-Verlag, 2001.