Aliasing control with view-based typestate

Filipe Militão[1,2]
filipe.militao@cs.cmu.edu

Jonathan Aldrich[1]
jonathan.aldrich@cs.cmu.edu

Luís Caires[2]
luis.caires@di.fct.unl.pt

[1] Carnegie Mellon University

[2] CITI / Dep. de Informática
Univ. Nova de Lisboa, Portugal

ABSTRACT

Tracking the state of an object (in the sense of how a `File` can be in an `Open` or `Closed` state) is difficult not just because of the problem of managing state transitions but also due to the complexity introduced by aliasing. Unchecked duplication of object references makes local reasoning impossible by allowing situations where transitions can be triggered unexpectedly (for instance, passing aliased parameters to a method that expects unaliased parameters, or calling a method that has a side effect through an alias deeply nested in a data structure).

We propose a generalization of *access permissions* that goes beyond a fixed set of permissions to an object. In this paper we present a new aliasing control mechanism that uses a small set of permissions as building block for the creation of *views* that capture a projection of an object with specific access constraints to its fields and/or methods. This makes permission tracking more fine grained while also making the designer's intent more explicit.

We present a few meaningful examples of how these *views* handle situations such as: separating different sections of an object for safe initialization; and access with either an unbounded number of readers or a single writer (multiple readers or unique writer). Finally, we show a type system for checking correctness of state use in the presence of this kind of controlled aliasing.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation—*View Typestate*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*View Typestate*

General Terms

Languages, Theory, Verification

Keywords

Aliasing control, view typestate

FTfJP '10, June 22, 2010, Maribor, Slovenia

1. INTRODUCTION

Typestate [14] and other techniques [6, 8] are often used to typecheck object-oriented programs that define *protocols*: ordering constraints on the calls to an object that must be obeyed by all clients. Thus, conceptually, the type of an object can potentially transition from one type to another on each call in order to model the restrictions imposed at each individual state. The usual example is a `File` class: a call to the `close` method only makes sense after the `File` has been opened and not before.

How exactly such restrictions are modeled in the type system varies and depends on whether the focus is centered on the externally observable sequence of calls (behavior) or the internal states of the object (typestate). The latter approach has lead to a typestate oriented programming language [1] that is the basis for the grammar that we will use in this paper. In it the states are made explicit. Therefore, in the previous example, the `File` transitions to an `OpenFile` and then to a `ClosedFile` state.

However, doing this kind of tracking quickly becomes a hard task not due to the transitions themselves but because of aliasing. When multiple references point to the same object, the type system needs to check the correctness of state use in situations where it can be non-obvious how such interferences may occur. For instance, a call may use arguments assuming that they are not aliased, but such a specification needs to be reflected in the signature so that a type system can reason locally about its correctness. Nonetheless, defining a format for aliasing specification is itself a tricky problem since restricting it too much (for instance, making all types linear [15] - where only a single reference to an object can exist at any time) simplifies reasoning but compromises flexibility; while more complex solutions may make the system harder to use.

Our work expands on *access permissions* [2], that combines typestate with a set of 5 kinds of permission annotations each one of them modeling certain types of object aliasing. Thus, permissions express alias information on what alias situations may exist and how they are allowed to interact ranging all the way from a linear **unique** permission to a more complex **shared** permission that allows the shared object to be mutated within some state space (the state guarantee).

We aim to make this mechanism more generic by, instead of offering a fixed set permissions, allowing the creation of an arbitrary number of *views*: projections of an object, each modeling specific permissions to its content (both on fields and methods). Thus, with this model, aliasing control is

centered on managing how these *views* interact. Our contributions include:

- A new abstraction to handle aliasing: *views*, a projection of a (complete) object defined based on a small set of permissions and equations that restrict what can be accessed (methods and fields) and when. These views allow for improved clarity by more tightly modeling the designer's intent. They also support more fine grained control by not limiting the developer to a fixed set of permissions.
- A type system that combines views and typestate (together with fractions [3]) to check the use of state in the presence of several user defined aliasing patterns, including single writer/multiple reader.

2. OVERVIEW

In this section we illustrate our approach with a couple of simple examples: initialization of a pair data structure (where each element of the pair is handled separately); and a single-writer or multiple-readers scenario (where an immutable view of an object can co-exist with an unbounded number of copies of that view, but where write access requires all of them to be collected back to ensure no interferences can occur).

2.1 Simple Linear Views

We now show how to define a pair datatype where we track the non-initialized state (`EmptyPair`) and the completely initialized state (`Pair`), while allowing the left and right fields to be separately initialized. This will require aliasing of the pair object but, by pushing each field into a distinct view, the type system can ensure that they will not interfere.

```
1  class EmptyPair {
2    /* view declarations */
3    view EmptyLeft  { none l; }
4    view EmptyRight { none r; }
5    view Pair  { L l; R r; }
6    view Right { R r; }
7    view Left  { L l; }
```

In this chunk of code of the `EmptyPair` class we declare all the views, each defining a set of usable fields. All views starting with `Empty` have fields with the none type (void content). Consequently, a constructor does need to not take arguments since all fields are initialized by default to null. Therefore, the class itself is a view where all fields are typed with none. This makes field declarations outside a view redundant and consequently we do not allow them. The full set of fields in a class is formed by the union of all fields of all views. In this example, all fields have read and write permissions to the field variable, in the next example we will show how we can go further and define views with only a read permission.

Each *view* is conceptually an isolated chunk of an object that has no immediate relation with other *views*. Therefore, we use *view equations* to specify how several views can be safely split and merged. Splitting implies breaking one single *view* into others while merging is just the reverse operation.

```
8    // ( more of EmptyPair class )
9    EmptyPair = EmptyLeft * EmptyRight
10   Pair = Left * Right
```

The * operator in the first equation (cf. separation logic [12]) specifies that `EmptyPair` can be decomposed into views `EmptyLeft` and `EmptyRight` which can be used independently. The converse (merging both views into `EmptyPair`) is also true since equations are symmetric.

Unlike view declarations which contain private information (fields), these view equations are public (i.e. visible to the outside of a class). Consequently, there also needs to be a verification of well-formeeness between what the equation declares and the internal contents of the views to ensure that there are no conflicts. In this example, the verification would immediately succeed as the views use a disjoint set of fields and therefore there is no risk of interferences.

Lastly, we introduce the method declarations, which must account for the use of views, and are of the form:

$$T\ m(\overline{T \gg T\ x})\ [T \gg T]\ \{\ e\ \}$$

Our syntax is inspired by the Plaid language [1]. The (possibly empty) list of arguments is formed by annotations $T_{in} \gg T_{out}$ that indicates the type (i.e. view) of each argument at the beginning and at end of the call, respectively (thus, T_{in} is the in-type and T_{out} the out-type). The annotation inside the trailing square brackets also express view change, but for the receiver (this). It is also important to note that these define traditional pre/post conditions that are only valid at the boundaries of the method and do not (by themselves) impose constraints between the two events. We show the annotated code for three methods:

```
11   none setLeft(L>>none x)[EmptyLeft>>Left]{
12     this.l = x;
13   }
14   none init()[EmptyPair>>Pair]{
15     // (this:EmptyPair)
16     // (this:EmptyLeft*EmptyRight) [by equation]
17     this.auto_init(this,this);
18     // (this:Left*Right)
19     // (this:Pair) [by equation]
20   }
21   none auto_init(EmptyLeft>>Left l,
22                  EmptyRight>>Right r)[none>>none]{
23     // (l:EmptyLeft,r:EmptyRight)
24     l.setLeft( new L() );
25     // (l:Left,r:EmptyRight)
26     r.setRight( new R() );
27     // (l:Left,r:Right)
28   }
29 }
```

Comments in the code show the typing environment at each step, and its splitting modulo * using the view equations. Such manipulations are captured by the subtyping relation described in Section 4. Notice that `this` is safely aliased no less than three times in `this.auto_init(this,this);`

We use a linear ownership model for tracking individual references. Thus, a method signature such as ($\mathsf{T} \gg$ none) indicates that the method's body may store the argument and consequently the caller will lose ownership of it. This is in opposition to borrowing ($T \gg U$), where ownership is returned at the end (even if at a different view type U).

The `setLeft` method signature illustrates view state transitions where starting from view `EmptyLeft`, by assigning the correct type to the field it satisfies the conditions to be packed back into a `Left` view. Packing and unpacking will be explained in more detail in Section 4.

2.2 Unbounded Sharing of Views

So far we have illustrated fully disjoint (non-overlapping / linear) views. In this section we will show how unbounded sharing is accommodated in our framework, allowing views to be aliased an unlimited number of times. The key issue is how to check that such a view is indeed safely replicable and tracking whether the entire initial view is back together or not (i.e. if it is still a fraction or the full view).

To exemplify, consider the iterator problem: in Java, an `Iterator` over a `Collection` is only correct as long as the underlying `Collection` remains unchanged. Once a modification is made the iterator should become invalid, a situation that is usually flagged with an exception. We can model such constraint using the view equation:

```
Collection = Iterable! * UnderIteration
```

A view that may be unboundedly split is marked with a ! as in `Iterable!` (which is equivalent to an infinite equation of the form `Iterable = Iterable * Iterable * ...`). Thus, to obtain an `Iterable` view we decompose `Collection` into both `Iterable!` and `UnderIteration`. This last view models an immutable `Collection` that cannot interfere with the several `Iterable` view used by iterators (such check between the view declaration and the view equations to ensure consistency between what both specify is done statically). To regain modification access, all pieces of `Iterable` must be joined back. Our type system keeps track of such pieces using fractions (cf. [3]), which are kept hidden from the programmer. Fractions track the number of copies of a single view. Thus, a unique view will have a full fraction (1) while a partial fraction will have some fraction of 1 of that view ($\frac{1}{k}$). Since our splitting operation is binary, we decided to use a multiplication of a fixed fraction ($\frac{1}{2}$) using the / symbol as will be explain further below.

This modeling assumes that all iterators must be read-only. An alternative, also present in the Java API, would use the single writer multiple readers scheme for iterators, that is:

```
RW_Iterator = RO_Iterator!
Collection = RW_Iterator * UnderIteration
```

And therefore in this situation we could have one iterator causing changes in the Collection only when all other RO_Iterators have been collected (by the type system, statically).

We illustrate unbounded aliasing using a more detailed example, involving a `Cell` object, which contains a `Lamp` object. The Lamp object goes through linear and unbounded shared views. In particular, it may be unlimitedly split into many views, providing only read operations. Once all view splits get merged back into the full type, a unique view enabling the write operation is recovered. The example also illustrates nested replication since sharing of the outer `Cell` implicitly also shares its `Lamp` field. All pure immutable values (such as `Integer`) will never need any sort of aliasing control since they can always be safely copied, e.g, `Integer=Integer!`.

```
1 class Lamp {
2   view LampOn { Integer bulb; }
3   view LampOff { none bulb; }
4   view StaticLamp { const Integer bulb; }
5   Lamp = LampOff
6   LampOn = StaticLamp!
```

The view declarations are similar to our previous example except for the const modifier in the `StaticLamp` view. This indicates that field `bulb` in view `StaticLamp` is read only. It also restrains the type for that field to be immutable as well, but since `Integer` is pure already, such condition is automatically met.

The second equation allows the type system to switch between `LampOn` or a full replicable view `StaticLamp!`. Getting back to `LampOn` requires *all* the previously split `StaticLamp` fractions of the full replicable view to be collected. To simplify the use of fractions, we do not allow specific fractions to be declared in the source code. Instead, we only have two notations: the full replicable view (`T!`) or a type annotation that abstracts over the actual fraction number the view is in at the moment of the call (`T?`). This is illustrated in the following method declarations:

```
7   none turnOn() [ LampOff >> LampOn  ] { ... }
8   none turnOff()[ LampOn  >> LampOff ] { ... }
9   Integer getLightIntensity()
10      [ StaticLamp? >> StaticLamp? ] { bulb }
11 }
```

The `getLightIntensity` method uses the latter annotation to mean that the receiver is parametric on the view's fraction. On type checking, ? gets instantiated by a concrete fraction so that borrowing is properly chained for that call (and the final type carries the correct fraction count). Fraction annotations (! or ?) are only allowed (and required) for replicable views to differentiate between the two possibilities (full or partial), non-replicable views do not suffer from this ambiguity and therefore are not allowed to have those annotations.

We now proceed to show the code for the class that wraps `Lamp`: the `EmptyCell` class.

```
1 class EmptyCell {
2   view ReadOnly { const StaticLamp! lamp; }
3   view FilledCellOff { LampOff lamp; }
4   view FilledCellOn { LampOn lamp; }
5   FilledCellOn = ReadOnly!
```

Notice that for a view to be safely replicable all its content field types must also be replicable. Thus, `ReadOnly` contains not only a const modifier but also a `StaticLamp!` as the field type. The remaining methods are mostly straightforward and therefore, we will only show their signatures:

```
6   none setLamp(Lamp>>none lamp)
7      [EmptyCell >> FilledCellOff]
8   Integer readIntensity()[ReadOnly? >> ReadOnly?]
9   none writeLampOn()[FilledCellOff >> FilledCellOn]
10  none writeLampOff()[FilledCellOn >> FilledCellOff]
11  Lamp getLamp()[FilledCellOn >> EmptyCell]
12 }
```

We show how our type system uses fractional views, in typing the following client code:

```
none m0(FilledCellOff>>FilledCellOn cell )
   [ none >> none ] {
  // (cell : FilledCellOff)
  cell.writeLampOn();
  // (cell : FilledCellOn)
  // (cell : ReadOnly!)   [by equation]
  cell.readIntensity();
  // (cell : ReadOnly!)
```

```
  // (cell : FilledCellOn) [by equation]
}
```

Notice how the instantiation of ? in the method signature to the correct fraction (in this case the full fraction of the replicable view) is carried on for the borrowing. Thus, the method type checks. Now consider the signature:

```
none n( ReadOnly? >> ReadOnly? a,
  ReadOnly? >> ReadOnly? b,
  ReadOnly? >> ReadOnly? c )[ none >> none ]
```

and typed code, showing the intermediate type environments

```
none m1(ReadOnly! >> ReadOnly! x)[none >> none]{
  // ( x : ReadOnly! )
  // ( x : ReadOnly/ * ReadOnly/ )
  // ( x : ReadOnly/ * ReadOnly// * ReadOnly// )
  n(x,x,x);
  //  ReadOnly/  >> ReadOnly/  a
  //  ReadOnly// >> ReadOnly// b
  //  ReadOnly// >> ReadOnly// c
  // ( x : ReadOnly! )
}
```

The three lines below `n(x,x,x)` show the correct instantiation of the parameter annotations of method `n`. Note that the ? instantiation is local to each parameter. Our system uses $T/$ to mean a concrete fraction (one half, cf. $1/2$) of the type T, such concrete fractions are introduced by primitive subtyping equations (e.g., $T! = T/ * T/$). The abstract fraction notation avoids the use of concrete rational numbers in the type system, and is expressive enough for our purposes.

3. PROGRAMMING LANGUAGE

The complete abstract syntax of our core programming language is shown in Fig. 1; we follow a let-normal form [13] syntax for simplicity. The semantics is the expected one for a typical core object-oriented languages [9], as view annotations play no role at runtime (we don't consider method overloading or inheritance for now). To simplify the presentation we adopt a few conventions.

- Concrete fractions never appear in source code, and are only used internally by the type system;
- As a consequence of the previous point, we treat captured method arguments conservatively: if a method captures any non-empty fraction of an argument (e.g. by storing it in a field) then the residual argument type is forced to be none.

  This design choice has the benefit of making clear from the signature of a method which arguments may be captured in whole or in part, without requiring the developer to use explicit fractions. In situations where a well-known, finite fraction of an object is captured, however, it potentailly loses precision relative to a solution based on explicit fractions. Note, however, that recovering a complete fraction from well-known, finite parts would also require a way to reason about the equality of two pointers so their fraction values can be merged; we leave this as a topic to future work.
- Parametric fractions $T?$ can only be used in in/out-types of a method signature and not elsewhere. In fact, $T?$ types were introduced to allow methods to work uniformly over all concrete fractional views of the argument type.

Notice that we extended our core language with the standard abbreviation: $e_0; e_1 \triangleq \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1$ (x not free in e_1)

4. SUBTYPING

Our system relies on a subtyping relation that plays a crucial role in expressing view splitting and merging. The full definition may be found in Fig. 2. Our rules include a method subtyping rule, which is interesting in that all inputs to the method are contravariant, but permission outputs–including permissions returned to arguments–are covariant. The other rules describe subtyping for classes, subtyping axioms for conjunction (*), and the semantics of $T!$ and $T/$. These rules allow a full fraction (!) to be split into sequences of / and merged back together as described in the examples above.

In Fig. 3, subtyping is extended beyond a single type to include subtyping for typing environments. We use a *linear* typing environment (Δ) that contains an unordered set of declarations of the form $(x : T, z : J, \ldots)$, where e.g., x is a variable and T its declared type. This linearity is needed to track the progress of type mutations caused by state transitions throughout the program that cannot be correctly tracked by a normal (lexical) typing environment.

The definition of subtyping for typing environments is simply applying the subtype relation to each of the variables in a subset of that environment. This implies that we can go beyond separating a type into separating pieces of an environment (as in $\Delta_0 <:> \Delta_1 * \Delta_2$). For instance:

$$(\Delta, (x : U * T)) <:> (\Delta, (x : U)) * (x : T)$$

Note the difference between both sides: the comma separated lists are for single environments (thus assumes no duplication - just a unique set of labels) while the star separates two such environments.

Besides the expected split rules, there is the need for a more subtle rule, to take into account the issue of packing/unpacking this: we cannot allow both access to the this pointer and its fields. Otherwise, using a field as an argument in a (self) call could cause an alias if we also allow it to be accessible in the method's body. Therefore, we use packing/unpacking [7] to exchange the external view of an object for access to its fields by enforcing that:

- At any given point in a program, either there is access to this or to the fields of this ($\overline{\mathsf{this}.x}$) but never to both at the same time (for some view T);
- Packing to ($\mathsf{this} : T$) requires all fields to be at the correct state required for that view T.
- Crucially, unpacking a replicable view requires the view's current fraction to be assigned to the fields, because these are implicitly replicated as a consequence of the outer view replication. For instance, in the previous example were we used the (replicable) `ReadOnly` view: when a fractional view such as `ReadOnly///` is unpacked its field must be exposed as the type `StaticLamp///` and not as the plain type `StaticLamp!` declared in the view.

  This is only valid because we impose that fields containing replicable views must be full (!). Therefore, there is no risk of messing the fraction count (of what part of the fraction belongs to this and what part is

$$
\begin{array}{rcll}
x & \in & \textsc{IdentifierNames} & \\
m & \in & \textsc{MethodNames} & \\
c & \in & \textsc{ClassNames} & \\
\\
P & ::= & \langle \overline{C}, e \rangle & \text{(program)} \\
C & ::= & \mathsf{class}\ c\ \{\ \overline{V}\ \overline{E}\ \overline{M}\ \} & \text{(class)} \\
V & ::= & \mathsf{view}\ c\ \{\ \overline{F}\ \} & \text{(view)} \\
F & ::= & [\mathsf{const}]^{opt}\ T\ x; & \text{(field)} \\
E & ::= & c = T & \text{(equation)} \\
M & ::= & T\ m(\overline{T \gg T\ x})\ [T \gg T]\ \{\ e\ \} & \text{(method)}
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & v \mid \mathsf{new}\ c() & \text{(expressions)} \\
 & \mid & \mathsf{this}.x = v & \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid v.m(\overline{v}) & \\
v, u & ::= & x \mid \mathsf{this} \mid \mathsf{this}.x_{[\mathsf{const}]^{opt}} \mid \mathsf{null} & \text{(values)} \\
N & ::= & c \mid \mathsf{none} & \text{(type names)} \\
T, U, J, W, K & ::= & T * T \mid N\ [f]^{opt} & \text{(types)} \\
f & ::= & k & \text{(fractions)} \\
 & \mid & ! & \text{(full fraction)} \\
k & ::= & [\ ?\ ]^{opt}\ \{/\}^* & \text{(partial fraction)}
\end{array}
$$

Figure 1: Programming language syntax.

$$\frac{\forall m.(m \in methods(U) : mtype(m,T) <: mtype(m,U))}{T <: U}(\text{class}) \qquad \frac{}{T <:> T * \mathsf{none}}(\text{id}) \qquad \frac{(c = T) \in E}{c <:> T}(\text{equation})$$

$$\frac{W_0 <: W_1 \quad \overline{T_1 <: T_0} \quad \overline{U_0 <: U_1} \quad K_1 <: K_0 \quad J_0 <: J_1}{m(\overline{T_0 \gg U_0\ x})\ [K_0 \gg J_0] \to W_0\ <:\ m(\overline{T_1 \gg U_1\ x})\ [K_1 \gg J_1] \to W_1}(\text{method}) \qquad \frac{}{T <: \mathsf{none}}(\text{top})$$

$$\frac{}{(T * J) * U <: T * (J * U)}(\text{assoc}) \qquad \frac{}{T * U <: U * T}(\text{comm}) \qquad \frac{T <: J \quad J <: U}{T <: U}(\text{trans}) \qquad \frac{T <: J}{U * T <: U * J}(*\text{cong})$$

$$\frac{}{N! <:> N/ * N/}(\text{replication}) \qquad \frac{}{N\ k <:> N\ k/ * N\ k/}(\text{replication - fractions})$$

Figure 2: Subtyping on types.

from the field) when we pack/unpack as would happen if replicable fields could have a generic (?) fraction.

- Each field must be packed/unpacked with the appropriate const value of the view. When unpacked, this is tracked by the index on the label such as $\mathsf{this}.x_{\mathsf{const}}$ or without the indexed const if it is writable since otherwise this information is only present on some view declaration.

These conditions imply that a view not containing fields can always be packed/unpacked at any time; being stateless, they may be freely aliased. We exemplify packing (exchanging access to fields to access to this) and unpacking (the opposite operation) by getting back to the `Pair` example:

```
none doSomethingR(R>>R x)[Left>>Left]
none outsideRight(Right>>Right x)[Left>>Left]
none pair-method()[Pair>>Pair]{
  // ( this : Pair )
  // ( this : Left * Right )
  // ( this : Left ), ( this : Right )
  // ( this : Left ), ( this.b : R ) [ by unpack ]
  this.doSomethingR(this.b);
  // ( this : Left ), ( this.b : R )
  // ( this : Left * Right ) [ by pack ]
  this.outsideRight(this); // ( this : Pair )
}
```

5. TYPE SYSTEM

We now present our type system, that combines views and typestate with fractions, to check the use of state in the presence of aliasing patterns controlled by user defined equations. Typing judgements have the form:

$$\Delta_0 \vdash e : T \dashv \Delta_1$$

Such a judgment states that in typing environment Δ_0, the expression (e) has type T, with effects resulting in the environment Δ_1. The typing rules are shown in Fig. 4.

In the (read) rule, we separate a view of v from Δ and return it. This includes the case where v is this or $\mathsf{this}.x$ and thus may require previous pack/unpack steps using the (subtyping) rule.

For (field assign), the old content is returned while the new one (v) is extracted from the environment as in the previous rule. Note that it requires the field to be non-const (not of the form $\mathsf{this}.x_{\mathsf{const}}$) so that it must be the unpacked field of a view with write permissions. There is no restrictions on the type of the new value, which eases the transitions between views as they just need to be packable to the correct type at the end of the call.

The (new) just returns a type of the instantiated class, since there are no arguments to be checked (they are all initialized to null and typed with none).

In (let), the result of e_0 (that will be substituted into x at runtime) is forwarded to e_1. Therefore, the typing environments follow the same flow. The rule does not return the type of x (the variable that will fall out of scope) to the environment at the end. For that, we could define an additional construct let-borrow that models such behavior by creating an intermediate auxiliary method where all arguments are borrows (thus, it is just syntax sugar).

In the (call) rule, any possibly T? views in the argument types need to be instantiated to the appropriate fraction. This is done using a notation similar to standard substitution but that gets applied solely to the type. After this instantiation, one checks that the arguments' type are correct for the call and proceeds with the out-types of the signature (adjusted to the appropriate fraction instantiation). Checking the correctness of the method's body is done through the (method check) rule and the correctness of a class with the (class check) one. Notice that the (subtyping) rule incorporates all the view split and merge principles, as discussed above.

Finally, we need rules to check the well-formedness between views and equations: views can only be set as repli-

$$\frac{\Delta_0 <:> \Delta_1 * \Delta_2}{\Delta_0, (v : T * U) <:> (\Delta_1, (v : T)) * (\Delta_2, (v : U))} \qquad \frac{T <: U}{(\Delta, (v : T)) <: (\Delta, (v : U))}$$

$$\frac{}{(\Delta_1, \Delta_2) <:> \Delta_1 * \Delta_2} \qquad \frac{fields(T) = \overline{[\mathsf{const}]^{opt}\ U\ x} \qquad \overline{[frac(T)/!]U = J}}{(\mathsf{this} : T) <:> (\overline{\mathsf{this}.x_{[\mathsf{const}]^{opt}} : J})}\,(\text{pack/unpack})$$

Figure 3: Subtyping on environments.

$$\frac{\Delta_0 <: \Delta_1 \quad \Delta_1 \vdash e : U \dashv \Delta_2 \quad \Delta_2 <: \Delta_3 \quad U <: T}{\Delta_0 \vdash e : T \dashv \Delta_3}\,(\text{subtyping}) \qquad \frac{\mathsf{class}\ c\ \{\ \overline{V}\ \overline{E}\ \overline{M}\ \} \in C}{\Delta \vdash \mathsf{new}\ c() : c \dashv \Delta}\,(\text{new}) \qquad \frac{}{\Delta \vdash \mathsf{null} : \mathsf{none} \dashv \Delta}\,(\text{null})$$

$$\frac{\begin{array}{c} mtype(m, K) = \overline{T' \gg U'}\ [K' \gg J'] \to W \\ \overline{[frac(K)/?]\ (K' \gg J') = (K'' \gg J)} \quad \overline{K <: K''} \\ \overline{[frac(T)/?]\ (T' \gg U') = (T'' \gg U)} \quad \overline{T <: T''} \end{array}}{\Delta * (v : K, \overline{u : T}) \vdash v.m(\overline{u}) : W \dashv \Delta * (v : J, \overline{u : U})}\,(\text{call})$$

$$\frac{\mathsf{this}.x \neq \mathsf{this}.x_{\mathsf{const}}}{(\Delta,\ \mathsf{this}.x : K) * (v : T)\ \vdash\ \mathsf{this}.x = v : K\ \dashv\ (\Delta,\ \mathsf{this}.x : T)}\,(\text{assign})$$

$$\frac{\begin{array}{c} \Delta_0 \vdash e_0 : U \dashv \Delta_1 \\ \Delta_1, (x : U) \vdash e_1 : T \dashv \Delta_2, (x : K) \end{array}}{\Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 : T \dashv \Delta_2}\,(\text{let}) \qquad \frac{}{\Delta * (v : T)\ \vdash\ v : T\ \dashv\ \Delta}\,(\text{read})$$

$$\frac{\begin{array}{c} mtype(m, c) = \overline{T \gg U}\ [K \gg J] \to W \\ \overline{x : T}, \mathsf{this} : K\ \vdash\ e : W\ \dashv\ \overline{x : U}, \mathsf{this} : J \end{array}}{W\ m(\overline{T \gg U\ x})\ [K \gg J]\ \{\ e\ \}\quad \mathtt{OK}}\,(\text{method check}) \qquad \frac{\overline{M}\quad \mathtt{OK}\quad \overline{E}\quad \mathtt{OK}}{\mathsf{class}\ c\ \{\ \overline{V}\ \overline{E}\ \overline{M}\ \}\quad \mathtt{OK}}\,(\text{class check})$$

$$\begin{array}{rclrcl} fields(T) & = & \text{“fields of type } T\text{”} & methods(T) & = & \text{“methods of type } T\text{”} \\ frac(T) & = & \text{“fraction of } T\text{”} & mtype(m, T) & = & \text{“type of method } m \text{ in type } T\text{”} \end{array}$$

Figure 4: Typing rules.

cable in an equation if all its fields are both constant and with a replicable type (so that the replication cannot cause unwanted interferences); and that the set of writable fields is disjoint (non-overlapping) when splitting a view (to ensure a unique writer for each field). Thus, on a split, the set of fields of a view is either disjoint, or non-disjoint fields are const and replicable.

6. RELATED WORK

In [2] Bierhoff and Aldrich define an aliasing control mechanism to modularly check state use in Java based on a set of 5 different kinds of *access permissions*. Our work has its roots in an attempt to unify both the state and aliasing control under the same abstraction (that we call *views*). Although our model is generally cleaner, we have not yet reached the same level of expressiveness of aliasing modalities (such as allowing for multiple writers to share an object with limited guarantees on what the type system can assume about its current state). Similarly, our approach was also influenced by the Plaid language [1]. However, our focus on views means that instead of changing the set of fields at each state, we model this by changing the types accordingly (the field is still “there” but without a permission to use it). In general, a view is identical to a state except that it can be recombined (merged and split using the same mechanism as Boyland’s fractional permissions but applied to views), and therefore we are approaching a separation algebra [5] over views, as accountable “parts” of typestates. The replicable view types are inspired in the *shared types* of [4]. Our research is also related to *masked types* [11], were a type system statically ensures that uninitialized fields are never read. In our case, we can use views to model the same situation: a class that has not yet been properly initialized is represented by giving all its fields a none type in the initial view. However, *masked types* handle inheritance which we do not. *Data groups* [10] are also similar to our *views* since they allow to partition fields into separate groups with restricted access to an object’s fields but does not include a mechanism similar to our *view equations* for complex recombination (merges and splits).

7. CONCLUSION AND FUTURE WORK

We have presented a new abstraction to control aliasing: *views*. They create a limited projection of an object that is then managed by the type system to provide safe use of state in the presence of aliasing. By allowing the programmer to specify these views as well as the way in which they can be mixed, we showed that this increased flexibility not only helps understanding the code but also provides a more fine grained permission granularity. So far, our approach is capable of handling alias of the form of a single-writer / multiple-readers by using fractions to track whether all readers have been collected into the single writer.

In future work, we plan to explore how to model more complex forms of interaction between views to handle other cases of aliasing (such as coordination between different ends of a pipe) and improve the expressiveness of our system.

Acknowledgements.

This work was partially supported by Fundação para a Ciência e Tecnologia and the Information and Communication Technology Institute at CMU, INTERFACES NGN44-2009-2012, and the Æminium project SE38-2009-2012. The Plaid group is acknowledged for many insightful discussions. We also thank the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proc. Onward!*, pages 1015–1022, 2009.

[2] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Proc. Object-Oriented Programming Systems, Languages, and Applications*, pages 301–320, 2007.

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

[4] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.

[5] 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.

[6] F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in java-like languages. *Acta Inf.*, 45(7-8):479–536, 2008.

[7] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

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

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[10] K. R. M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Not.*, 33(10):144–153, 1998.

[11] X. Qi and A. C. Myers. Masked types for sound object initialization. In *Proc. Principles of Programming Languages*, pages 53–65, 2009.

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

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

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

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