See the Pet in the Beast: How to Limit Effects of Aliasing

Franz Puntigam
Technische Universität Wien, Institut für Computersprachen
franz@complang.tuwien.ac.at

Abstract

Aliasing is quite powerful, but difficult to control. Often clients need exclusive access to objects for some concerns, and sometimes we see no other way than to ensure this by controlling aliasing. Instead, we propose to restrict what clients can do when accessing objects. To invoke methods in an object clients need tokens issued by this object. Static type checking enforces the tokens to be available and ensures exclusive access for specific concerns without avoiding aliasing. We show by examples how this concept works and discuss several possibilities to improve its flexibility.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Object-Oriented Programming, Aliasing

Keywords Types, Tokens

1. Motivation and Overview

Aliasing is like a beast causing troubles. It shows up where we do not expect it and perverts our statements. It is slippery and escapes when we think we have caught it.

Aliasing is also like a pet. Object-oriented programmers love it. It opens doors to objects that seem far far away. Science fiction authors would be surprised if they knew how easy we walk from one object to another at a completely different part of the world.

The beast and the pet are actually the same animal. Aliasing gives programming languages much expressive power, so much that we easily lose control. Programming systems become weaker whenever we cage or tame the beast. We must be careful not to destroy flexibility: Although it is possible to develop nearly every kind of system without the undesirable properties of aliasing (for example, in a referentially transparent functional language) we have to do so with considerably less flexibility in structuring the code. Such flexibility is essential in object-oriented programming to achieve good factorization.

In this paper we discuss an approach to annotate object references with constraints on how to access referenced objects. This approach fully supports aliasing; there is no cage for it as in many other approaches [2, 3, 12, 15, 39]. However, we limit what the beast can do by ensuring that constraints on references are preserved when introducing new aliases. For example, if we want a specific method in an object to be invokable at most once, then we annotate the only reference existing on object creation with a corresponding constraint. After introducing further references to the object we still have this property: There is only one reference through which the message can be sent although we usually do not know where to find this reference. We need not restrict aliasing by itself; we just limit effects of aliasing.

We express what can be done through an object reference by a set of tokens (or just names) associated with the reference. Method specifications give semantics to tokens: A method can require specific tokens to be associated with the reference through which it is invoked; these tokens are removed from the reference on invocation, and further tokens (specified by the method) can be added on return. We express constraints by tokens because they are easily understood by both programmers and tools like compilers.

This approach was introduced as part of a type system expressing synchronization to ensure linearity at the presence of aliasing [29]. Applications of this technique are usually related to synchronization and coordination. In the above example of a single invocation, all clients of an object must be coordinated such that at most one of them invokes the specific method. Such kinds of coordination are inevitably connected with constraints on single references (as opposed to the whole referenced object) at the presence of aliasing.

The goal of this paper is to survey how this approach works and show by examples what can be done with it and where its limits are. Thereby, the focus is on limiting effects of aliasing, not synchronization and coordination. In Section 2 we give the basics of our token-based approach, and in Section 3 we show how to distribute tokens within a system. In Section 4 we briefly describe static type checking. In the remaining sections we use various concepts to add flexibility – dependences between tokens in Section 5, relationships between values and tokens in Section 6, type parameters in Section 7, and a dynamic concept in Section 8 – before we discuss related work in Section 9 and give concluding remarks in Section 10.

2. Tokens to Ensure Limited Access

We show how to specify constraints by examples in a Java-like pseudo-language. The first example gives a simplified interface of a window, where method invocations depend on tokens (in square brackets – tokens removed on invocation to the left and those added on return to the right of arrows):

```
interface Window {
   [init -> shown,ready] void initialize (...);
   [ready -> ready] void update (...);
   [shown -> icon] void iconify ();
   [icon -> shown] void uniconify ();
   [shown,ready ->] void close ();
   int getCreationTime ();
}
```

In our pseudo-language, brackets denote token sets associated with types and methods, they do not denote arrays. Let us assume that new windows are of type `Window[init]`, this is, we have a reference to an instance of `Window` associated with a token `init`. Through this reference we can invoke only `initialize` and `getCreationTime`. All other methods require tokens (as specified to the left of `->`) not available in the reference; they are not invokable. When we invoke `initialize` the type of the reference changes first to `Window[]` (or equivalently just `Window`) and on return to `Window[shown,ready]`. Further methods become invokable, but `initialize` cannot be invoked again. We can distribute references to the window all over the system (see Section 3 how to do that in a type-save way). The type checker prevents tokens from being duplicated thereby. Maybe, we get a reference of type `Window[ready]` in a client feeding the window with new data, a reference of type `Window[shown]` in a window control panel, and any number of references of type `Window` through which only `getCreationTime` can be invoked. However, we cannot get two references to the same window, both of type `Window[ready]`, because there exists always at most one token `ready`. In the control panel we can invoke `iconify` and `uniconify` only in alternation: Invoking `iconify` changes the type associated with the window reference to `Window[icon]`, and invoking `uniconify` again to `Window[shown]`. The method `close` becomes invokable only if we rearrange the system such that all available tokens (`ready` and `shown`) occur again in the type of a single reference. After invoking `close` all tokens are gone, and only `getCreationTime` remains invokable.

The example shows how we can easily specify nontrivial constraints on method invocations and how clients are forced to satisfy them. This technique is expressive: We can specify all prefix-closed trace sets [30]. Clients that hold tokens have got partial control over the corresponding object: The client holding `ready` is the only one being able to invoke `update`, and the client holding `icon` or `shown` completely controls whether the window is iconified. Clients cannot influence each other in this respect. Moreover, a client who holds any token of a window can prevent `close` ever to be invoked. This kind of "separation of concerns" works without any knowledge about aliases in the system. We need not know which client has control over a concern. This client can even change dynamically.

To ensure limited access we usually want to have at most one token of each name per object. Separation would be weaker if we had several tokens `ready` and possibly several clients invoking `update` in the same window object. As proposed in [29] this approach supports several tokens of the same name in order to express limited resources (that are not necessarily limited to one; for example, buffer sizes) for the purpose of synchronization. For the purpose of aliasing control we need no limited resources of this kind. In this paper we implicitly assume tokens to occur at most once per object (this is, in the types of all references to the same object).

Explicit result types of constructors play an important role in specifying initial object states:

```
class MyWindow implements Window {
   MyWindow[init] () {};
   ...
}
```

An invocation of `new MyWindow()` returns a new instance with a single token `init`. Based on this information we can compute the maximum of tokens for this object available in the whole system (see Section 4). Since `MyWindow` does not add tokens to those inherited from `Window`, there can always be at most an `init`, or a `ready` and either a `shown` or an `icon` in the types of all references to an instance of `MyWindow`.

3. How to Distribute Tokens

In the next example we show how to handle tokens in types of parameters and variables:

```
class Test {
   void play (Window[ready -> ready] w) {
      w.update(...);
      w.update(...);
   }
   void blink (Window[icon -> shown] w) {
      w.uniconify();
      w.iconify();
      w.uniconify();
   }
   Window[ready] win;
   [unique -> unique]
    Window[ready] swap (Window[ready ->] w) {
      Window[ready] old = win;
      win = w;
      return old;
   }
   [unique -> unique] void condUpdate() {
      if (win != null) { win.update(...); }
   }
   Test[unique] () { win = null; }
}
```

Let `y` be a variable of type `Window[ready,icon]` and `x` one of type `Test[unique]`. We can invoke `x.play(y)` since `y` has the token `ready` as required in the formal parameter type to the left of `->`. On invocation the type of `y` changes to `Window[icon]` (this is, `ready` is removed) and on return from `play` again to `Window[ready,icon]` (this is, the token to the right of `->` in the formal parameter type is added). In the body of `play` the parameter `w` has a token `ready` on invocation as well as on return; we can invoke `update` as often as we want to do so.

Invocations of `blink` change argument types: On return from `x.blink(y)` variable `y` will be of type `Window[ready,shown]`. In the body of `blink` we must invoke `uniconify` at least once to ensure `w` to have the appropriate token on return. We can invoke `x.play(y)` and `x.blink(y)` in any ordering and even concurrently because the token sets required from `y` (as well as the empty token sets required from `x`) do not overlap.

Parameter passing does not produce or consume tokens. Tokens just move from the argument type to the parameter type on invocation and vice versa on return.

Whenever we introduce an alias (in this case by binding a formal parameter to an argument) we perform *type splitting:* The tokens specified in the argument type are split into two groups. Tokens specified in the formal parameter type (to the left of the arrow) move to the formal parameter while all other tokens remain in the argument's type. After return the formal parameter is no longer in use. We combine the previously split types again; thereby tokens (specified to the right of the arrow) move from the formal parameter to the argument.

Assignment resembles parameter passing on method invocation: When assigning a reference to a variable where the variable type specifies tokens, these tokens are removed from the reference; this is, the tokens move from the assigned value to the variable. In the body of `swap` the token `ready` moves from the parameter `w` to the instance variable `win`. Since the token finally belongs to `win`, it cannot move back to the argument `y` on return from `x.swap(y)`.

Types specifying tokens in square brackets frequently change. For example, `w` in the body of `blink` is of type `Window[icon]` before invoking `uniconify` and of type `Window[shown]` afterwards. There is no difficulty for a type checker and usually also for a human reader to determine what is the current type of a local variable at some position in the program. However, such type changes cause troubles on instance variables: There can be independent accesses of the same variable through concurrent threads as well as through aliases. If one of the clients accessing the variable causes tokens to be removed from the variable, others do not know about this change and can assume the tokens still to be available; there can be an unexpected and undesired duplication of tokens. To avoid such problems we require tokens of instance variables to be visible only if we can exclude simultaneous accesses through concurrent threads and aliases. In our example we get uniqueness by requiring the token `unique` on `x` when executing `x.swap(y)` and `x.condUpdate()`.[1] Static type checking ensures such variable accesses to be actually unique (see Section 4). Furthermore, such variables must not be accessed from outside (except though getter and setter methods). Of course, on return from methods all such variables must hold their declared tokens.

As in all Java-like languages `null` is an appropriate instance of each reference type. Since no method is invokable through `null`, we can assume this special value to be associated with any token. Tokens do not compromise the use of `null`.

If we have two references of the types `Window[ready]` and `Window[shown]`, then it is in general not possible to invoke `close` through any of them. However, the following method allows us to combine the token sets:

```
Window[ready,shown] comb (Window[ready ->] x,
                          Window[shown ->] y) {
   if (x == y) { return x; }
   else { return null; }
}
```

The method is correct because in this case `x` and `y` are known to be aliases with the common type `Window[ready,shown]`. Whenever we know two variables (or parameters) to refer to the same object (after comparing identity) we assume all tokens belonging to any of the two variables to belong to both of them. The essential part is just the conditional statement with an identity comparison as condition; the rest of this example just gives us a setting where this statement may be useful.

Our approach supports subtyping considering tokens. We give just a raw idea of it (see [29, 30, 33] for more complete descriptions): Subtypes specify all (relevant) tokens specified by supertypes. Hence, `MyWindow[ready,shown]` is a subtype of `Window[ready]`, but is not related to `Window[icon]` by subtyping. Methods declared in subtypes have to the left of `->` at most and to the right at least those tokens that occur to the left or right of the arrow in the corresponding method declaration in the supertype. Irrelevant tokens (these are tokens no method depends upon) need not be considered. As a consequence we can invoke at least each sequence of methods through a reference to an instance of a subtype that we can invoke through a reference to an instance of a corresponding supertype. Supertypes are more restrictive than (or equal to) subtypes.

4. Static Type Checking

Static type checking in our approach is rather simple and can be performed at a class by class basis (separate compilation). Programmers give all information the checker needs by specifying tokens in types and together with methods. The type checker must ensure all specified types and tokens to be consistent (which is much simpler than inferring information about aliasing or synchronization from a program). It can do so by a single walk through the code of a class. In detail, the checker has to ensure the following properties:

1. *At any time there cannot be several tokens of the same name for the same object.* To ensure this property we apply a simple fixed-point algorithm to compute for each class an upper bound of token sets that can become available: Initially we have the sets of tokens specified in constructors (one set per constructor). We construct further token sets by updating each token set according to each method where the token set contains all tokens occurring to the left of `->` in the method; tokens to the left of the arrow are removed and tokens to the right are added. The algorithm terminates if no new token sets can be constructed this way. Type checking fails if a token set contains the same token twice. Usually the fixed point is reached quickly because there are only few different tokens in a class. Since new tokens can be introduced only by method invocations (as ensured by the properties mentioned below) this fixed-point construction is sufficient to ensure that two tokens of the same name can never exist for any object.
2. *Methods are invoked only through references associated with all needed tokens.* Initially we assume types of variables to carry tokens as in the variable declarations, and types of parameters as to the left of arrows in parameter declarations. While walking through the code according to the control flow we ensure for each method invocation that the type of the reference the method is invoked through which contains all tokens occurring to the left of `->` in the declaration of this method. Furthermore, we update the type of the reference by removing all tokens occurring to the left and adding all tokens occurring to the right of the arrow in the method declaration. Whenever the control flow is split (for example, in a conditional statement) we perform these checks for each path separately. At joins of several paths we remove all tokens that do not occur in all corresponding types constructed independently in the paths to be joined.
3. *Tokens are not duplicated when introducing aliases.* While walking through the code according to the control flow we ensure for each method invocation that types of arguments have all tokens occurring to the left of the arrow in the corresponding formal parameter type. These types are updated by removing all tokens occurring to the left and adding all tokens occurring to the right of the arrow in the formal parameter type. For each assignment of a value to a variable we ensure that the value has all tokens specified in the type of the variable and remove these tokens from the value's type. At the end of the control flow of each method and constructor we ensure that
   - each parameter has all tokens that occur to the right of the arrow in the parameter declaration,
   - and each instance variable has at least all tokens that occur in the variable declaration.
4. *Always at most one method can make use of tokens associated with an instance variable.* Such variables are not directly accessible from outside the object they belong to which. To ensure the absence of simultaneous accesses to each such variable within an object we use the set of methods accessing the variable and the upper bound of token sets constructed while checking property 1: If there is no token set in the upper bound that contains all tokens occurring to the left of the arrows of any pair of methods in the method set, then these methods cannot be invoked simultaneously and the variable access is unique.

   For example, in class `Test` (in the previous section) only `swap` and `condUpdate` accesses the instance variable `win`. Both methods have `unique` to the left of the arrow. Each of the four possible method pairs has two tokens `unique` to the left of the arrows. The upper bound constructed from `Test` contains only a single token set with a single token `unique`. Since no token set in the upper bound contains two tokens `unique`, several concurrent or overlapping invocations are impossible. In this case (and in many similar cases) we do not need the upper bound to show this property because we know that no token set

[1] Declaring `swap` and `condUpdate` as synchronized is not sufficient because there is still the possibility of a simultaneous access through aliasing. Requiring a unique token is a stronger condition. It ensures the absence of any other client also invoking one of these methods.

in the upper bound contains the same token twice. Sometimes the use of upper bounds increases accuracy. For example, in MyWindow the methods iconify and uniconify cannot be invoked simultaneously because no token set in the upper bound contains both shown and icon.

The type system is strong and sound in the sense that methods can be invoked only when objects are in appropriate states as specified by tokens. Essential parts of a corresponding proof can be found in [29, 30]. To get this result we need not restrict aliasing, and we need no knowledge of aliases (except of local information about statements possibly introducing new aliases to ensure property 3). This is an important difference to many seemingly similar approaches like the Fugue protocol checker [10].

There is a (still incomplete) implementation of the type checker for a simple language similar to the language we use in this paper. From early experiences with this checker we see that the type system is quite good in detecting errors where programmers get tokens wrong. Wrong tokens in method declarations usually show up as diverging upper bounds (as constructed to ensure property 1) or cause methods not to be invocable. Wrong tokens in types cause methods not to be invocable or references not to be usable as method arguments. The type checker complains about such errors.

Concerning type safety it does not matter if tokens are lost or hidden in the type of unused references. In such cases, clients just do not make use of services offered by objects. To enforce clients to make use of services we can extend the type checker as proposed in [34] at the cost of flexibility.

5. Dependent Tokens

In this and the following sections we discuss a number of approaches to improve the expressiveness and flexibility of our technique. An important step in this direction is to make use of known relationships between tokens that belong to different objects.

In the following example we show a possibility to specify tokens belonging to an instance variable in dependence of tokens of the object that contains the variable [33]:

```
class IconButtons {
   Window[icon for down][shown for up] window;
   [down -> up] void pressUp() {
      window.uniconify();
   }
   [up -> down] void pressDown() {
      window.iconify();
   }
   IconButtons[up] (Window[shown] w) {
      window = w;
   }
}
```

We think of IconButtons as a wrapper for the part of Window dealing with icons. The variable window has one token icon for each token down known to occur in the corresponding instance of IconButtons and one token shown for each up in the instance. In general, we regard a set of tokens to the left of for as available it there exists the set of tokens to the right of for. In the body of pressUp we know down to be available at method invocation and up on return because of [down -> up]. Hence, we assume window to have a token icon on invocation, and we must ensure that window has a token shown on return. An invocation of uniconify changes the token appropriately. Because of [up -> down] specified for pressDown we assume window to have a shown on invocation of this method, and we have to ensure the variable to have an icon on return. On object creation we must initialize window with a reference having a shown because the new instance of IconButtons is associated with an up.

Checking for-clauses in instance variable specifications is straightforward because type safety follows from the construction of this language concept. There is only a small difference to type checking as proposed in Section 4: To ensure property 3 we have to compute the tokens carried by variables from token specifications in methods instead of having them declared directly.

Using class IconButtons we control both buttons in a single class. Distributing a concern (like controlling the state of iconification) over several classes is a much more difficult topic that occurs in practice. In the next example we show an alternative solution to IconButtons based on separate classes for each button:

```
class ButtonA {
  Window[shown for activeA] window;
  ButtonB[passiveB for activeA] button;
  [passiveA -> activeA] void activate() {...}
  [activeA -> passiveA] void press() {
     window.iconify();
     button.activate();
  }
  [initA -> activeA]
   void init (Window[shown for activeA ->] w,
              ButtonB[passiveB for activeA->] b) {
     window = w;
     button = b;
  }
  ButtonA[initA] () {}
}

class ButtonB {
  Window[icon for activeB] window;
  ButtonA[passiveA for activeB] button;
  [passiveB -> activeB] void activate() {...}
  [activeB -> passiveB] void press() {
     window.uniconify();
     button.activate();
  }
  [initB -> passiveB]
   void init (Window[icon for activeB ->] w,
              ButtonA[passiveA for activeB->] b) {
     window = w;
     button = b;
  }
  ButtonB[initB] () {}
}
```

The variable window carries a token shown in ButtonA (and icon in ButtonB) when the button in active. Otherwise we do not know any token of window. After pressing the active button we activate the other button, and the pressed button becomes passive. These classes work essentially in the same way as IconButtons once the objects have been initialized. On invocation of activate the variables window in the two objects implicitly exchange the only available token. The initialization is the tricky part: We have to tell the two objects that they can safely assume to have the only token shown or icon of window when they are active. In ButtonA the parameter type Window[shown for activeA ->] specifies that w refers to a window carrying shown only while the button is active (and has no tokens on return from init); this parameter is assigned to window of essentially the same type. To initialize the objects we may use the following piece of code:

```
w = new MyWindow();
a = new ButtonA();
b = newButtonB();
```

```
    a.initialize(w,b);
    b.initialize(w,a);
```

The variable w occurs in both invocations of initialize and gives away its only token to both objects depending on the states of the objects as specified by the for clause in the formal parameter type. To ensure this initialization to be correct we have to show the following properties:

- The two objects never have the tokens activeA and activeB at the same time. Because of two different objects this property is not obvious. Using least bounds of token sets constructed for both classes as in Section 4 we can show this property: No token set constructed from ButtonA contains both activeA and passiveA, and no token set constructed from ButtonB contains activeB and passiveB. Since button in ButtonA carries passiveB if there is a token activeA, there cannot exist a token activeB at the same time, and analogously for button in ButtonB. Hence, activeA and activeB cannot exist at the same time.
- When the other object becomes active, there exists the token on window needed by the other object. This means, window must carry a token icon (or shown) at the end of each method where activeA (or activeB) occurs to the left of the arrow and does not also occur to the right.

Because these checks are ad hoc and compromise separate compilation, it is an open question whether for clauses in formal parameter types shall be supported or not.

6. Values as Tokens and Tokens as Values

Dependent tokens are safe and (without for clauses in parameter types) easy to handle where they are appropriate. However, in many situations we need more freedom. Especially, we want to relate the availability of tokens to values in variables. In the next example we show how to establish such relationships:

```
class SwapButton {
   int state;
   Window[icon if state < 0]
         [shown if state > 0] window;
   [unique -> unique] void press() {
      if (state < 0) {
         window.uniconify();
         state = 1;
      }
      else if (state > 0) {
         window.iconify();
         state = -1;
      }
   }
   SwapButton[unique] (Window[shown] w) {
      window = w;
      state = w == null ? 0 : 1;
   }
}
```

The variable window is associated with a token icon if state holds an integer value below zero, and with shown if the value is larger than zero. There is no token for zero. Before we can make use of these tokens we have to ensure corresponding conditions (considering the value of state) to be satisfied. After changing tokens associated with window we must update state.

This approach to relate tokens with values raises a large number of problems:

- Tokens are allowed to depend only on side-effect-free conditions that read only instance variables of the object. Such variables like state must not be written from outside, and there must not exist aliases of them. Otherwise it would be impossible to keep results of evaluating the conditions synchronized with the available tokens. In the programming language Ada we have similar requirements on conditions in when clauses belonging to protected types (Ada's notion for monitors) [16].
- The compiler must be able to determine whether conditions specified in square brackets correspond to other occurrences of the same conditions in conditional statements. Usually the compiler can determine only structural equivalence. The use of named conditions (based on name equivalence) can be helpful in this respect. For example, we define a parameterless boolean function that implements the condition and invoke this function instead of using the condition directly. This way it is easy to determine equivalence of conditions.
- On return from a method that changes tokens of variables or assigns new values to variables like state we have to ensure tokens and variable values to correspond to each other. We can do so by checking the conditions. In general, we can perform these checks only at run time and thereby lose static type safety. To avoid this problem we restrict values assigned to variables like state (where conditions depend upon) to be constant. In this case we can perform the checks at compilation time and keep static type safety. This restriction reduces the expressiveness, but tokens depending on values are still quite expressive.

Each of these problems can be solved (although the first and the last one are serious) and dependence of tokens on values does not compromise static type checking. However, since we need rather heavy machinery, we may prefer to use another approach that allows us to express more directly what we want to have:

```
class SwapButton2 {
   Window[?] window;
   [unique -> unique] void press() {
      if ([icon]window) {
         window.uniconify(); }
      else if ([shown]window) {
         window.iconify(); }
   }
   SwapButton2[unique] (Window[shown] w) {
      window = w;
   }
}
```

The question mark in the declaration of window states that we do not know statically which tokens will be associated with the variable. The tokens associated with window are stored in an implicit variable. An expression of the form [...]window returns true if this implicit variable contains all tokens specified in the square brackets. In the body of press we dynamically check if window is associated with icon or shown and make use of the found token. On return from the method (as well as from the constructor) the tokens of window are automatically stored in the implicit variable.

Up to now we regarded tokens to be a purely static language concept. The approach taken in SwapButton2 handles tokens dynamically. Nonetheless we can ensure static type safety without any difficulty because types are split and updated in the same way as in the purely static concept. By storing tokens in implicit variables (not directly modifiable by the programmer) we avoid the difficulties we have to address in the approach taken in SwapButton.

The implicit variable in SwapButton2 corresponds essentially to state in SwapButton. These two classes differ mainly in the syntax. In the approach of SwapButton we can use state information also for purposes not related to tokens, while the approach of SwapButton2 requires less program code and is simpler to check.

Using tokens as values as well as letting tokens depend on values adds much flexibility to the whole language concept.

7. Type Parameters

Tokens encoded into types and changes of types cause a difficulty together with homogeneous genericity as in Java: Each use of a type parameter refers to the same type while we often want to refer to types with different token sets, and we have to consider tokens to avoid unexpected token duplication. We need some notation to express tokens for type parameters. In the next example we show our first approach where we use essentially the same notation as for types:

```
class IconList<W extends Window> {
   [i -> i] void add (String s, W[icon] w) {...}
   [i -> s] void uniconifyAll () {...}
   [s -> s] W[shown] delete (String s) {...}
   [s -> s] W get (String s) {...}
   IconList[i] () {}
}
```

An instance of `IconList<MyWindow>` can be used as expected: We can add objects of type `MyWindow[icon]`, cause all added windows to become uniconified, and delete uniconified windows and thereby get back instances of `MyWindow[shown]`. An invocation of `get` cannot return any token because the returned instance of `MyWindow` remains in the list where the token still is needed. The compiler would complain if we tried to return the token and at the same time keep it in the list.

However, for types like `IconList<MyWindow[ready]>` this approach is inappropriate. The type parameter `W` must not carry tokens because `get` cannot return any reference associated with tokens as explained above. Otherwise we would implicitly duplicate tokens and destroy type safety.

If we need type parameters carrying tokens, we must declare the parameters with a question mark to make our intention clear:

```
class IconList<W[?] extends Window> {...}
```

In this variant the compiler complains about possible token duplication in `get`.

In Java we have no access to types substituting type parameters at run time. Therefore, it is most natural to keep also tokens in these types invisible. In languages with run-time support of genericity (like C#) we regard tokens associated with type parameters as being stored in an implicit variable. Then, we can use the boolean expression `[ready]W` to dynamically determine if each instance of `W` is associated with a token `ready` in a similar way as we did in `SwapButton2`. As a special case we can use `[]W` to ensure in methods like `get` no token to be associated with `W`.

8. Dynamic Tokens

A simple and seemingly still powerful approach to further increase flexibility introduces a dynamic pool of tokens into each object. We differentiate between static tokens (used so far) and dynamic tokens stored in dynamic pools. Dynamic tokens required on invocation (this is, dynamic tokens to the left of `->` in brackets associated with methods) are taken from the dynamic pools of the objects the methods belongs to (not from references to them). On return dynamic tokens are added to the pools, not to references. If a required dynamic token is not available on invocation, then the invocation is delayed until the token becomes available. The main purpose of dynamic tokens is synchronization [31, 33].

In this paper we prefix dynamic tokens with `$` to distinguish them syntactically from static tokens. By replacing `unique` in our `SwapButton` example with a dynamic token we get:

```
[$unique -> $unique] void press () {...}
SwapButton[$unique] (Window[shown] w) {...}
```

Each client can invoke `press` without needing a token. Several simultaneous invocations will be synchronized and executed in any sequential ordering. In this respect the use of dynamic tokens resembles that of "synchronized" in Java. However, we consider recursive invocations[2] of `press` as erroneous while recursive synchronized methods are supported. Unfortunately, there is no easy way to statically determine indirect recursive invocations especially together with separate compilation. We can detect erroneous recursive invocations practically only at run time as deadlocks.

Dynamic tokens are not as useful in controlling aliasing as they seem to be at a first glance. A client does not get unique access for some concern for a sequence of invocations – just for a single invocation. In simple cases (like ensuring unique access to a variable carrying tokens) dynamic tokens give us more flexibility at the cost of lost static safety and lost control over effects of aliasing. As we can see from dynamic tokens there is a fundamental difference between conventional synchronization and limiting the effects of aliasing although these concepts are related. Synchronization is a much weaker concept.

9. Related Work

The work presented in this paper is closely related to process types [27, 29, 30], a type concept where we express synchronization in types of active objects and in types of references to active objects. Process types were developed as abstractions over expressions in object-oriented process calculi like Actors [1] and build the formal basis of the present work. Static type checking ensures that only acceptable messages can be sent and thereby enforces required synchronization. Process types allow us to specify nearly arbitrary constraints on the acceptability of messages: We can specify all prefix-closed trace sets, type equivalence is based on trace-set equivalence, and subtyping on trace-set inclusion [28]. A notation based on tokens helps us to keep static type checking as well as deciding type equivalence and subtyping simple [29, 30]. The process type concept considers types to be partial behavior specifications [19, 20] especially useful in specifying the behavior of software components [4, 18, 25].

Recent work regards process types as a synchronization concept in Java-like object-oriented programming languages [31, 32, 33]. This work adds a further dynamic level of synchronization while keeping the completely static level of (required) synchronization. To control aliasing we need mainly the static level.

There are several approaches similar to process types. Some approaches ensure subtypes to show the same deadlock behavior as supertypes, but do not enforce message acceptability [24, 25]. Other approaches consider dynamic changes of message acceptability, but do not guarantee message acceptability in all cases [8, 9, 35]. Few approaches ensure all sent messages to be acceptable [17, 23]. There is essentially the same idea behind the well-known work on linear types [17] based on the π-calculus [21] and process types based on an Actor-like model. However, since there is no natural notion of message acceptability in the π-calculus as in the Actor model, static checking of linear types has to prevent deadlocks and (therefore) is much more restrictive than checking of process types that can ensure message acceptability without preventing deadlocks.

The Fugue protocol checker [10, 11] uses a different approach to specify client-server protocols: Rules for using interfaces are recorded as declarative specifications. These rules can limit the or-

[2] In general, this restriction applies to invocations of all methods that require the same dynamic tokens, not just recursive invocations of the same method.

der in which methods are called as well as specify pre- and post-conditions. Tokens in this protocol checker represent typestates. Other than in our approach, they can be used only for unique references. Since there is no concept resembling type splitting (as in our approach), the Fugue protocol checker cannot statically ensure all methods to be invoked in specified orders at the presence of aliasing. In these cases the checker introduces pre- and post-conditions to be executed at run time. Hence, our approach can statically ensure type safety in many cases where the Fugue protocol checker can perform only dynamic checks. There is a number of further similar approaches to express (abstract) object states in types and check protocol compatibility [6, 7, 36, 37, 38].

Several programming languages [5, 13, 26] were developed based on the Join calculus [14]. For example, in Polyphonic C# [5] we combine methods like put and get in a buffer to a chord to be executed as a single unit. Clients can see how methods in a chord are synchronized. Since only one method in a chord is executed synchronously and all other methods are asynchronous, only specific forms of synchronization are supported. Communication in Polyphonic C# and similar languages resembles that of the rendezvous concept while (dynamic versions of) process types extend monitors. There is no way to constrain method invocation sequences as with process types, and there is no obvious way to use chords in controlling aliasing.

Synchronization with tokens has a long tradition: Petri Nets have been explored for nearly half of a century as a basis of synchronization [22]. In general, expressing object states by abstract tokens often has clear (both practical and theoretical) advantages over expressing them more concretely by values in instance variables: Tokens are much easier tractable than concrete states especially when used in types. Many proposals use tokens to express abstract object states [6, 10, 37].

The major contribution of this paper is to explore process types from the perspective of aliasing control. Different from earlier work on process types we assume each token to occur at most once in a system. As a consequence we get clear separation of concerns, better error detection from static type checking, and more flexibility in specifying tokens associated with instance variables. Dependent tokens distributed over several classes as well as values used as tokens and tokens depending on values have not been considered so far in the context of process types.

10. Conclusions

The basic approach to limit effects of aliasing is simple: Objects issue tokens, and clients need tokens to interact with objects. A client holding a token gets exclusive access to the object that issued the token for the concern associated with the token because there exist only one token for this concern in the whole system. Static type checking ensures that methods can be invoked only in specified sequences by clients holding the required tokens. We apply a number of techniques to manage tokens in more or less complicated situations to increase the flexibility of this approach. For example, with dependent tokens we safely specify tokens to be available if other tokens are available, and with specific boolean expressions we get dynamic access to (otherwise static) tokens. We also detected some cases where this concept causes difficulties or reaches its limits: Access to instance variables carrying tokens must be exclusive, dependent tokens distributed over several objects are difficult to handle, and dynamic tokens (which are quite useful for synchronization) do not help much for our purpose. Nonetheless, we already have a number of techniques to avoid most cases of undesirable effects of aliasing while we need not restrict aliasing by itself.

References

[1] AGHA, G., MASON, I. A., SMITH, S., AND TALCOTT, C. Towards a theory of actor computation. In *Proceedings CONCUR'92* (1992), no. 630 in Lecture Notes in Computer Science, Springer-Verlag, pp. 565–579.

[2] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Nov. 2002).

[3] ALMEIDA, P. S. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP'97* (1997), no. 1241 in Lecture Notes in Computer Science, Springer-Verlag.

[4] ARBAB, F. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming 55*, 1–3 (2005), 3–52.

[5] BENTON, N., CARDELLI, L., AND FOURNET, C. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems 26*, 5 (Sept. 2004), 269–804.

[6] BIERHOFF, K., AND ALDRICH, J. Lightweight object specification with typestates. In *ESEC/FSE-13* (Lisbon, Portugal, Sept. 2005), ACM Press, pp. 217–226.

[7] BOYLAND, J. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium* (Berlin, Heidelberg, New York, 2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 55–72.

[8] CAROMEL, D. Toward a method of object-oriented concurrent programming. *Communications of the ACM 36*, 9 (Sept. 1993), 90–101.

[9] COLACO, J.-L., PANTEL, M., AND SALLE, P. A set-constraint-based analysis of actors. In *Proceedings FMOODS'97* (Canterbury, United Kingdom, July 1997), Chapman & Hall.

[10] DELINE, R., AND FÄHNDRICH, M. The fugue protocol checker: Is your software baroque? Tech. rep., Microsoft Research, 2004. http://www.research.microsoft.com.

[11] DELINE, R., AND FÄHNDRICH, M. Typestates for objects. In *ECOOP 2004 – Object-Oriented Programming* (Oslo, Norway, June 2004), no. 3086 in Lecture Notes in Computer Science, Springer-Verlag.

[12] DROSSOPOULOU, S., CLARKE, D., AND NOBLE, J. Types for hierarchic shapes. In *ESOP* (2006), pp. 1–6.

[13] DROSSOPOULOU, S., PETROUNIAS, A., BUCKLEY, A., AND EISENBACH, S. School: A small chorded object-oriented language. In *Proceedings of ICALP Workshop on Developments in Computational Models* (2005).

[14] FOURNET, C., AND GONTHIER, G. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* (1996), pp. 372–385.

[15] HOGG, J. Islands: Aliasing protection in object-oriented languages. *ACM SIGPLAN Notices 26*, 10 (Oct. 1991), 271–285. Proceedings OOPSLA'91.

[16] ISO/IEC 8652:1995. Annotated ada reference manual. Intermetrics, Inc., 1995.

[17] KOBAYASHI, N., PIERCE, B., AND TURNER, D. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems 21*, 5 (1999), 914–947.

[18] LEE, E. A., AND XIONG, Y. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing 16*, 3 (Aug. 2004), 210–237.

[19] LISKOV, B., AND WING, J. M. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), 16–28. Proceedings OOPSLA'93.

[20] MEYER, B. *Object-Oriented Software Construction*, second edition ed. Prentice Hall, 1997.

[21] MILNER, R. The polyadic π-calculus: A tutorial. In *Logic and*

Algebra of Specification (1992), Springer-Verlag, pp. 203–246.

[22] MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (Apr. 1989), 541–580.

[23] NAJM, E., AND NIMOUR, A. A calculus of object bindings. In *Proceedings FMOODS'97* (Canterbury, United Kingdom, July 1997), Chapman & Hall.

[24] NIELSON, F., AND NIELSON, H. R. From CML to process algebras. In *Proceedings CONCUR'93* (1993), no. 715 in Lecture Notes in Computer Science, Springer-Verlag, pp. 493–508.

[25] NIERSTRASZ, O. Regular types for active objects. *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), 1–15. Proceedings OOPSLA'93.

[26] ODERSKY, M. Functional nets. In *Proceedings of the European Symposium on Programming* (2000), Springer-Verlag.

[27] PUNTIGAM, F. Type specifications with processes. In *Proceedings FORTE'95* (Montreal, Canada, Oct. 1995), IFIP WG 6.1, Chapman & Hall.

[28] PUNTIGAM, F. Types for active objects based on trace semantics. In *Proceedings FMOODS'96* (Paris, France, Mar. 1996), E. Najm and J.-B. Stefani, Eds., Chapman & Hall, pp. 4–19.

[29] PUNTIGAM, F. Coordination requirements expressed in types for active objects. In *Proceedings ECOOP'97* (Jyväskylä, Finland, June 1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes in Computer Science, Springer-Verlag, pp. 367–388.

[30] PUNTIGAM, F. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.

[31] PUNTIGAM, F. Client and server synchronization expressed in types. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)* (San Diego, USA, Oct. 2005).

[32] PUNTIGAM, F. From static to dynamic process types. In *First International Conference on Software and Data Technologies* (Setubal, Portugal, Sept. 2006), INSTICC Press 2006, pp. 21–28.

[33] PUNTIGAM, F. Internal and external token-based synchronization in object-oriented languages. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006* (Oxford, UK, Sept. 2006), no. 4228 in Lecture Notes in Computer Science, Springer-Verlag, pp. 251–270.

[34] PUNTIGAM, F., AND PETER, C. Types for active objects with static deadlock prevention. *Fundamenta Informaticae 48*, 4 (2001), 315–341.

[35] RAVARA, A., AND VASCONCELOS, V. T. Behavioural types for a calculus of concurrent objects. In *Proceedings Euro-Par'97* (1997), no. 1300 in Lecture Notes in Computer Science, Springer-Verlag, pp. 554–561.

[36] STROM, R. E., AND YELLIN, D. M. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering 19*, 5 (May 1993), 478–485.

[37] STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering 12* (1986), 157–171.

[38] YELLIN, D. M., AND STROM, R. E. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems 19*, 2 (Mar. 1997), 292–333.

[39] ZHAO, T., PALSBERG, J., AND VITEK, J. Lightweight confinement for featherweight Java. In *OOPSLA 2003 Conference Proceedings* (Anaheim, California, Oct. 2003), ACM, pp. 135–148.