An imperative calculus for unique access and immutability

(extended abstract)

Paola Giannini[1], Marco Servetto[2], and Elena Zucca[3]

[1] Università del Piemonte Orientale, Italy
giannini@di.unipmn.it
[2] Victoria University of Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz
[3] DIBRIS, Università di Genova, Italy
elena.zucca@unige.it

We present a typed imperative calculus where it is possible to express and check aliasing and immutability property directly on source terms, without introducing invariants on an auxiliary structure which mimics physical memory. Indeed, continuing previous work [1, 5], we adopt an innovative model for imperative languages which, differently from traditional models, is a *pure calculus*. That is, execution is modeled by just rewriting source code terms, in the same way lambda calculus models functional languages. Formally, this is achieved by a non standard semantics of local variable declarations. When the expression defining a local variable x is evaluated, x is not replaced by its value but, rather, the association from x to the value is kept[1], and plays the role of an association from a reference to a (right) value in the store.

For instance, in the following code, where we assume a class `B` with a field of type `B`:

```
mut B x = new B(y)  mut B y = new B(x)  y
```

the two declarations can be seen as a store where `x` denotes an object of class `B` whose field is `y`, and conversely. Moreover, as shown in the example, variables (references) can be tagged by *modifiers* which specify constraints on their behaviour. Indeed, in the recent years a massive amount of research, see, e.g., [3, 4, 2] has been devoted to make programming with side-effects easier to maintain and understand, notably using type modifiers to control state access. Here, we consider two properties of references: (1) *no mutation*, that is, the reachable object graph denoted by the reference cannot be modified, (2) *no aliasing*, that is, the reachable object graph denoted by the reference cannot be saved as part of another object. We will use four modifiers corresponding to the four combinations, that is: mutation and aliasing (`mut`), no mutation (`imm` for immutable), no aliasing (`lent`), no mutation and no aliasing (`read`). In addition, we also use a `capsule` type modifier, a subtype of both the mutable and immutable modifiers, which allows mutable data to be passed and stored as internal state for an object without allowing other access to the same data.

Store is not flat, as it usually happens in models of imperative languages. For instance in the following example, where we assume a class `D` with an integer field, and a class `A` with two fields of type `B` and `D`, respectively:

```
imm D z = new D(0)
imm A w = {
  mut B x = new B(y)
  mut B y = new B(x)
    new A(x, z)
  }
w
```

[1] As it happens, with different aims and technical problems, in cyclic lambda calculi.

the store associates to `w` a block introducing local declarations, that is, in turn a store. In this representation, the fact that an object is not referenced from outside some enclosing object is directly modeled by the block construct: for instance, the object denoted by `y` can only be reached through `w`. Conversely, references from an object to the outside are directly modeled by free variables: for instance, the object denoted by `w` refers to the external object `z`.[2] In other words, our calculus smoothly integrates memory representation with shadowing and α-conversion.

We outline now the type system. A type T consists in a class name C decorated by a modifier μ.

The subtyping relation is the reflexive and transitive relation on types induced by

$$\mu\, C \leq \mu'\, C \text{ if } \mu \leq \mu'$$
$$\texttt{capsule} \leq \texttt{mut} \leq \texttt{lent} \leq \texttt{read}$$
$$\texttt{capsule} \leq \texttt{imm} \leq \texttt{read}$$

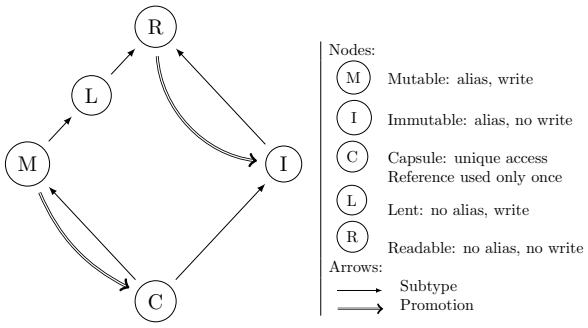

Figure 1: Type modifiers and their relationships

However, *promotion* rules can be used to move the type of an expression against the subtype hierarchy. More precisely: (1) Mutable expressions can be promoted to capsule, if mutable references are *weakly locked*, that is, can only be used as lent. (2) Readable expressions can be promoted to immutable, if lent, readable, and mutable references are *strongly locked*, that is, cannot be used at all. The situation is graphically depicted in Figure 1.

Our contribution includes the definition of the calculus with its type system, and proof of *soundness*. In addition, we formally stated and proved that modifiers imply their expected properties.

As a long term goal, we also plan to investigate (a form of) Hoare logic on top of our model. We believe that the hierarchical structure of our memory representation should help local reasoning.

References

[1] Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. *ICTCS 2015*.

[2] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. *AGERE! 2015*, pages 1–12. ACM Press.

[3] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. *OOPSLA 2012*, pages 21–40. ACM Press.

[4] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *POPL 2012*, pages 557–570. ACM Press.

[5] Marco Servetto and Elena Zucca. Aliasing control in an imperative pure calculus. *APLAS 2015*, *LNCS* 9458, pages 208–228. Springer.

[2] Note that the object denoted by `w` is a *capsule*, since its only external reference is `imm`, whereas its mutable state is encapsulated.