

Ownership For Modularity

Neel Krishnaswami

Department of Computer Science
Carnegie Mellon University

The Traditional (But Still Cogent) Question

How do we build large software systems?

The Traditional (But Still Cogent) Answer

Use information hiding to build a system modularly.
(Parnas)

- Specify interfaces between components, and write code that only depends on the interface.
- This will let you develop components independently.

An Amazing Fact

For a purely functional language, you can *prove* that a client that programs to an interface will have no dependencies on the implementation. This is Reynolds' famous parametricity theorem, aka the type abstraction theorem.

A Depressing Fact

However, this theorem *doesn't hold* for languages with state.

```
class Class {
    private Object signers[];

    public Object[] getSigners() {
        return signers;
    }
}
```

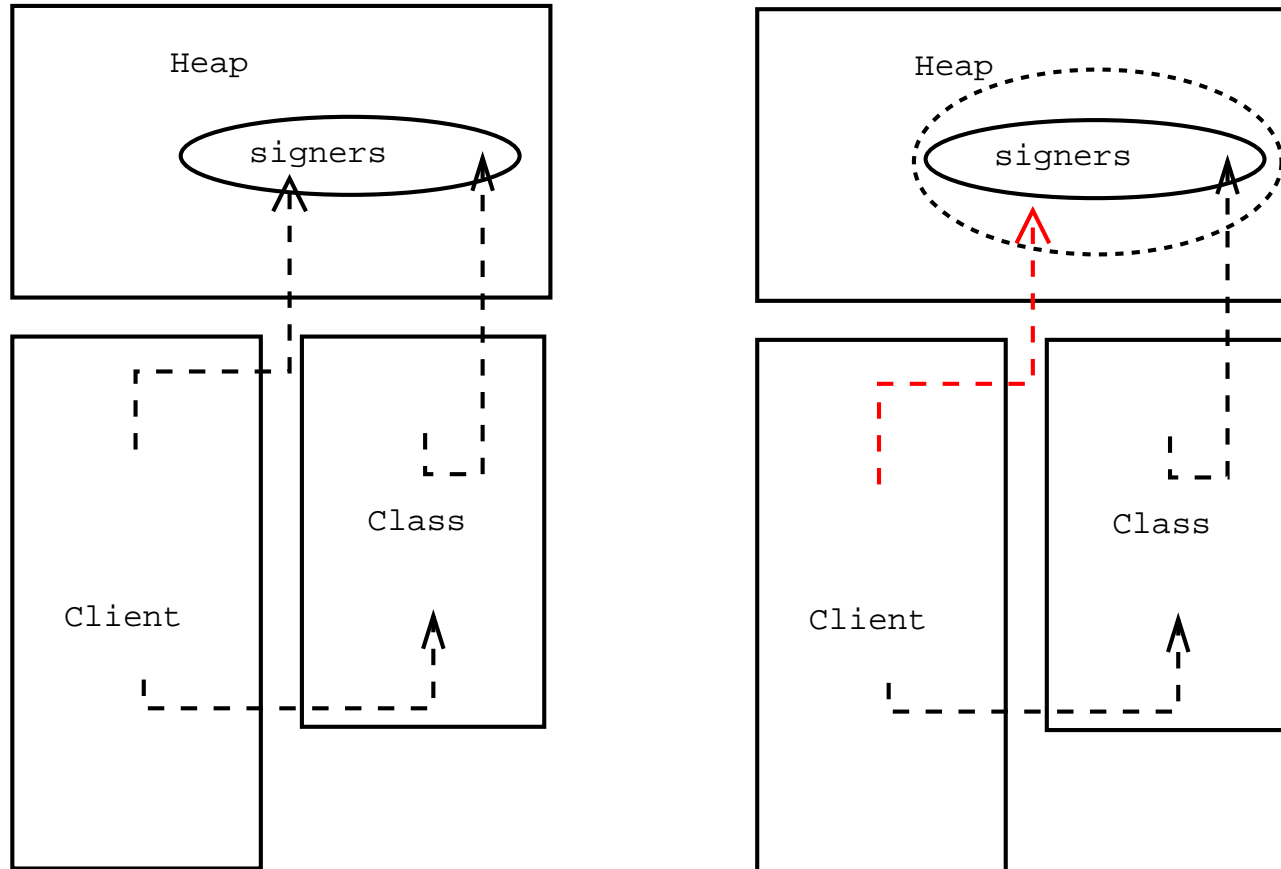
Note that a client can get a reference to the object's private state, and then modify it *without* calling methods in the object's interface. This was a real bug in the 1.1 Java JDK!

How Can We Fix This?

The basic problem is that mutable state lets a value's internals “leak”.

We use *Ownership types* to solve this problem; we annotate each reference and function with an *owning domain*, and only permit a client to modify some owned state if it has permission to access that domain.

A diagram



How domains work

```
module signature COUNTER:
  domain public
  type t

  val init : ref <public> int
  val create : () --<public>--> t
  val increment : t --<public>--> ()

module Counter implements COUNTER:
  domain public
  domain internal
  type t = ref<internal> int

  val init = ref<public> 0
  fun create<public> () = return ref<private>(!init)
  fun increment<public> (x :: ref<internal> int) = x := !x + 1
```

How domains work

- We have two permissions, creation and access.
- An access permission lets code in domain A access a reference or run code in domain B.
- A creation permission lets code in A create references or functions in domain B.
- Our type system checks to ensure that user code respects all the permissions, and that no one can create domains to subvert the access restrictions. The soundness theorem for the type system proves this is true.

Back to Type Abstraction

Do ownership permissions actually help us write more modular programs?

This is where the type abstraction theorem comes in. Our current plan is to prove a type abstraction result for a core language with ownership domains.

That is, we're trying to prove that abstract types work in a language with state, if all the state is properly tagged with ownership domains. That way, we can identify which parts of the heap are “inside” an abstraction, and keep clients from gaining access to it.

A Note on Research

Note that this is almost certainly not true. We are doing a proof to find the bugs in our definitions, and to learn what the idioms for modular programming with state actually are!