Safe journeys into the unknown

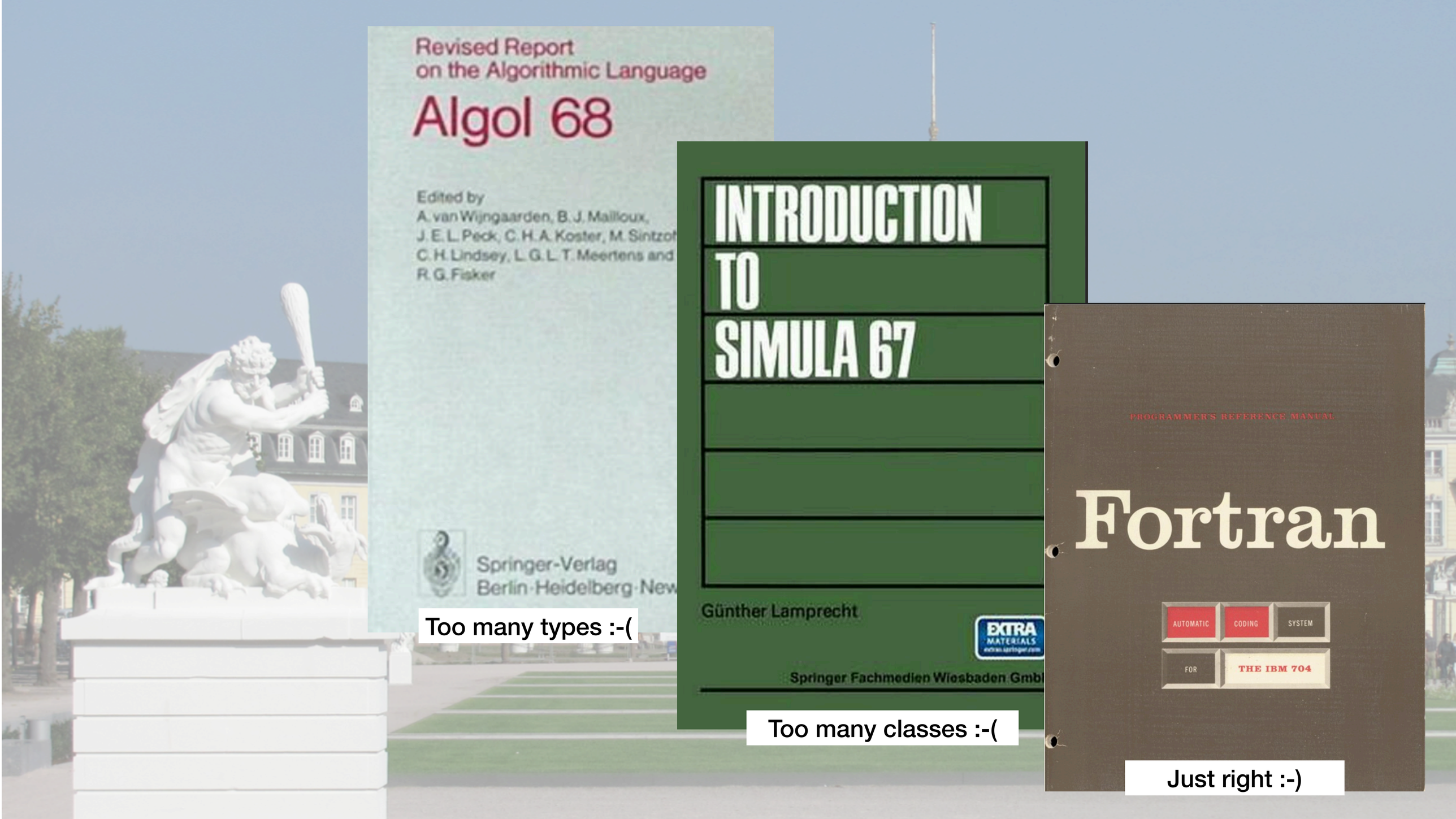
Object Capabilities

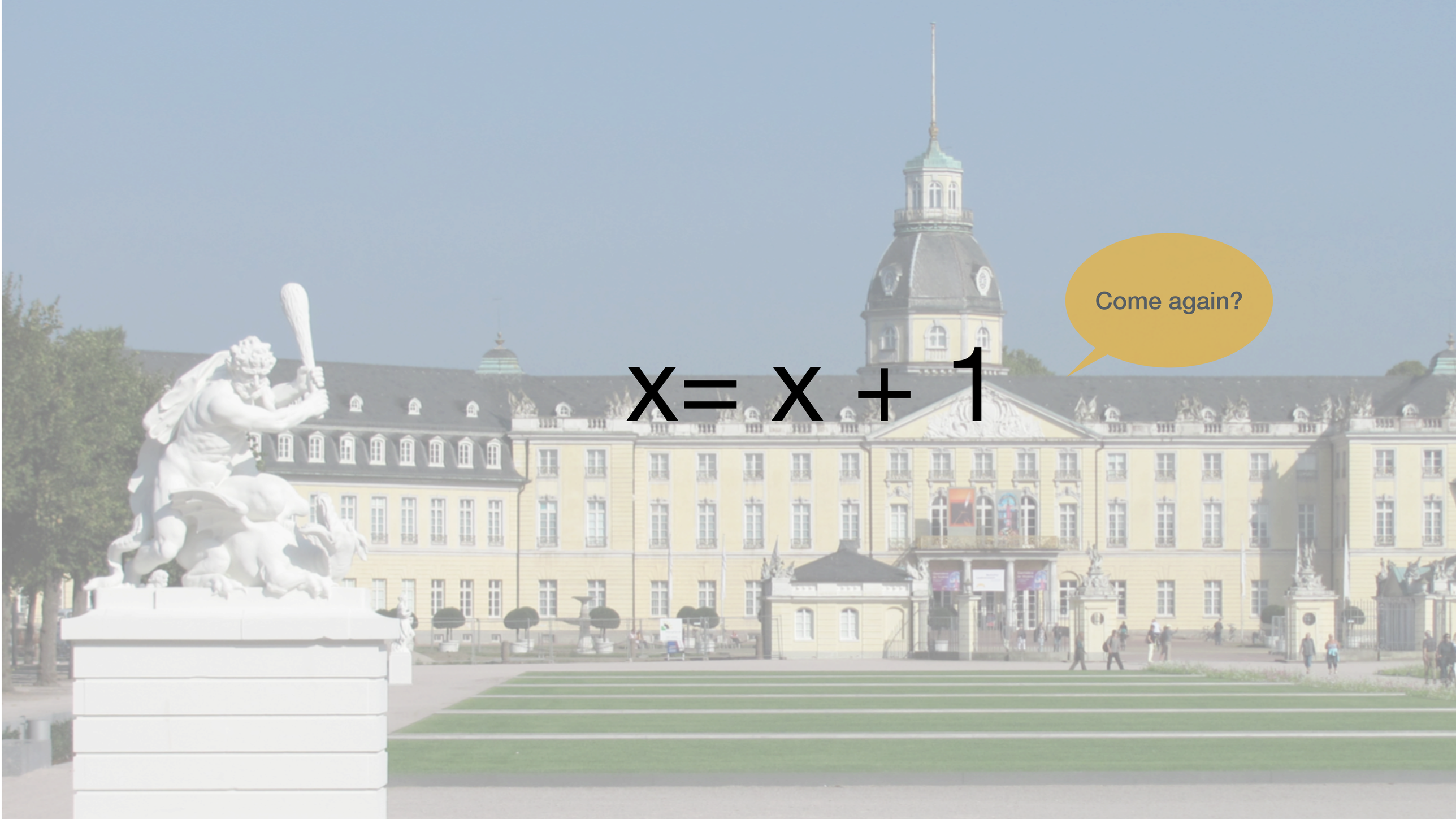
Sophia Drossopoulou, July 2023

My journey started at the Technishe Universitaet Karlsruhe — KIT

- 6 Informatik (CS) Professors, Prof Gerhard Goos, Prof Peter Deussen
- mostly no CS training;
  2 weeks' lags between learn and teach
- 400 Maths students; 100 CS students
- 40% women
- Women to excel in CS

SKEL


Revised Report
on the Algorithmic Language
Algol 68
Edited by
A. van Wijngaarden, B. J. Mailloux,
J. E. L. Peck, C. H. A. Koster, M. Sintzoff
C. H. Lindsey, L. G. L. T. Meertens and
R. G. Fisker
Springer-Verlag
Berlin·Heidelberg·New
Too many types :-(
INTRODUCTION
TO
SIMULA 67
Günther Lamprecht
EXTRA
MATERIALS
Springer Fachmedien Wiesbaden GmbH
Too many classes :-(
PROGRAMMER'S REFERENCE MANUAL
Fortran
AUTOMATIC
CODING
SYSTEM
FOR
THE IBM 704
Just right :-)


x= x + 1
Come again?

```
gcd(m,n){
    while (m = n){
        if m>n then
            m=m-n
        else
            n=n-m
    }
    return m
}
//gcd(m,n)=GreatestCommonDivisor(m,n)
```

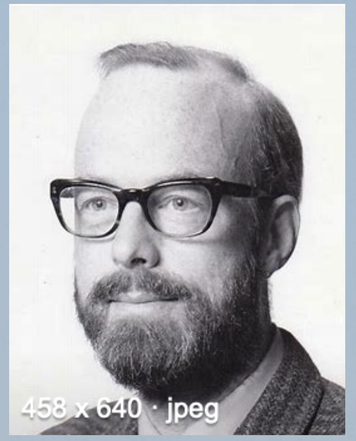
Yes ???

Should I drop CS??

```
gcd(m,n){
    invariant
    GreatestCommonDivisor(m, n) =
```

$$\frac{\{\ B \wedge Inv\ \}\quad S\quad \{\ Inv\ \}}{\{\ Inv\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \{\ \neg B \wedge Inv\ \}}$$

```
        n=n-m
    }
    return m
}
```

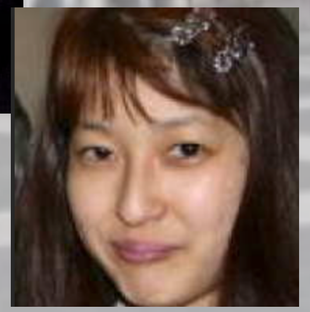


That makes sense!

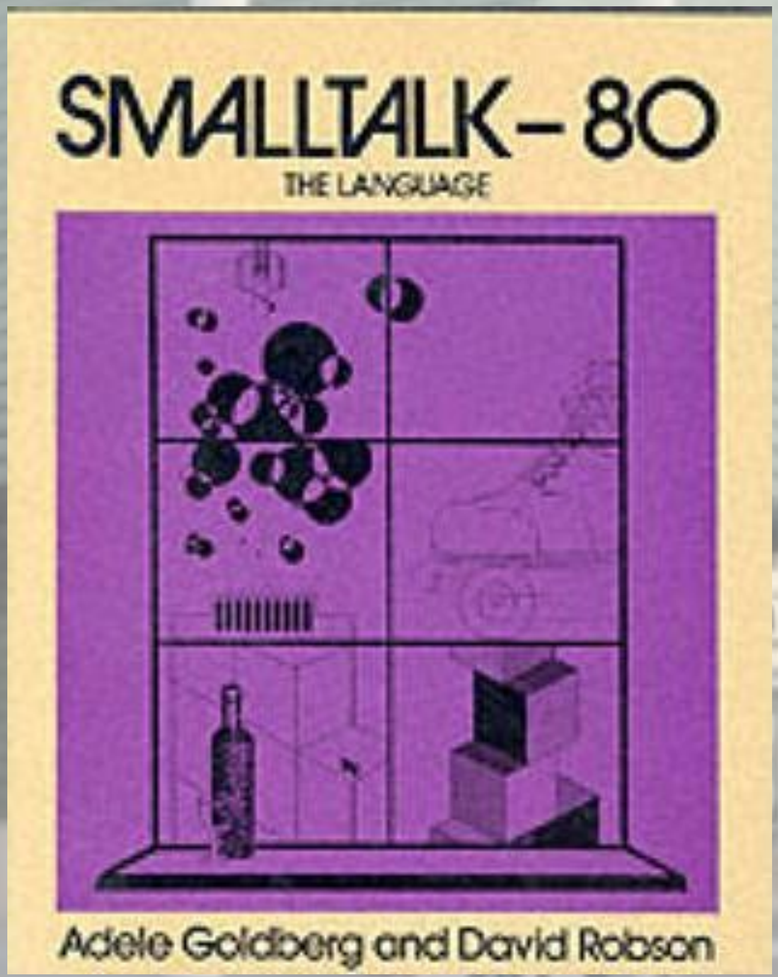
My journey continued at Imperial College London

… combining family life and career not trivial

My journey continued at Imperial College London



Taught a course on PLs, and discovered …



That makes lots of sense!

So, perhaps not that many classes in Simula, after all?

… teaching to research

… revise opinions

We worked about 1 year on …

Parameterized Interfaces are Interfaces – AIAS

Sophia Drossopoulou Ross Paterson* Susan Eisenbach

Department of Computing
Imperial College
London SW7, UK

Abstract

We introduce AIAS, a proposal for expressing higher order parameterization and abstraction for modules. A module consists of an interface and an implementation. Both interfaces and implementations can be parameterized, and they are first class citizens, *i.e.* any order of parameterization is possible. The interface of the application of a parameterized implementation may depend on the interface of its arguments, *i.e.* we have a weaker form of dependent product types.

AIAS is conceptually very simple because, in contrast to other systems, it does not distinguish between an interface of a parameterized implementation and a parameterized interface. Consequently, fewer entities need to be declared using AIAS than would

Is it not beautiful?

Perhaps, but where is the model?

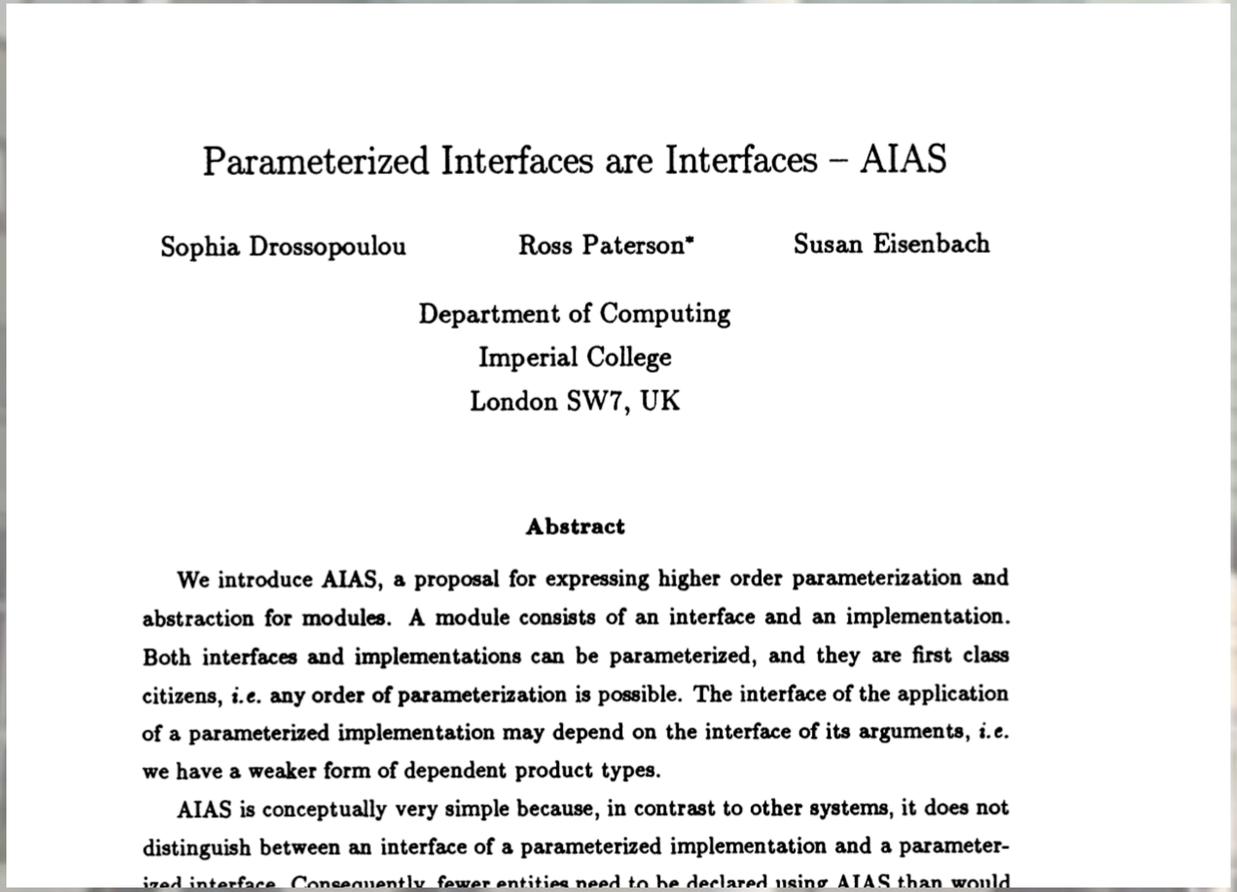
…

Parameterised modules are push outs

Ah, I understand!

… use the right tools

And then, for the next 5 (FIVE) years …



Parameterized Interfaces are Interfaces – AIAS

Sophia Drossopoulou Ross Paterson* Susan Eisenbach

Department of Computing
Imperial College
London SW7, UK

Abstract

We introduce AIAS, a proposal for expressing higher order parameterization and abstraction for modules. A module consists of an interface and an implementation. Both interfaces and implementations can be parameterized, and they are first class citizens, *i.e.* any order of parameterization is possible. The interface of the application of a parameterized implementation may depend on the interface of its arguments, *i.e.* we have a weaker form of dependent product types.

AIAS is conceptually very simple because, in contrast to other systems, it does not distinguish between an interface of a parameterized implementation and a parameterized interface. Consequently, fewer entities need to be declared using AIAS than would

Here is an algorithmic version

… and a proof …

… error in proof …

… a new proof…

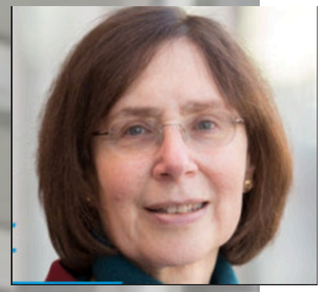
… a new error …

… another proof …

… another error …

… give up

Models of widely-used things …



Java is Type Safe — Probably

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Abstract. Amidst rocketing numbers of enthusiastic Java programmers and internet applet users, there is growing concern about the security of executing Java code produced by external, unknown sources. Rather than waiting to find out empirically what damage Java programs do, we aim to examine first the language and then the environment looking for points of weakness. A proof of the soundness of the Java type system is a first, necessary step towards demonstrating which Java programs won't compromise computer security.
We consider a type safe subset of Java describing primitive types, classes, inheritance, instance variables and methods, interfaces, shadowing, dynamic method binding, object creation, null and arrays. We argue that for this subset the type system is sound, by proving that program exe-

Lecture Notes in Computer Science

Mehmet Akşit Satoshi Ma

ECOOP'97 – Object-Oriented Programming

11th European Conference
Jyväskylä, Finland, June 1997
Proceedings

No denotational semantics

No category theory

No …

… simple is good

Java is Type Safe — Probably

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Abstract. Amidst rocketing numbers of enthusiastic Java programmers and internet applet users, there is growing concern about the security of executing Java code produced by external, unknown sources. Rather
aiting t damage Java programs do, we
environment looking for
he Java type system is a
ecessary
ating which Java programs won't
mise computer security.
sider a type safe subset of Java describing primitive types, classes,
nce, instance variables and methods, interfaces, shadowing, dy-
method binding, object creation, null and arrays. We argue that
subset the type system is sound, by proving that program exe-

The essence private/protected?

What next?

Full Parametricity

Overloading and Inheritance

Davide Ancona[1], Sophia Drossopoulou[2], Elena Zucca[1]

[1] DISI, University of Genova, [2] Department of Computing, Imperial College
E-mail: {zucca, davide}@disi.unige.it sd@doc.ic.ac.uk

Abstract

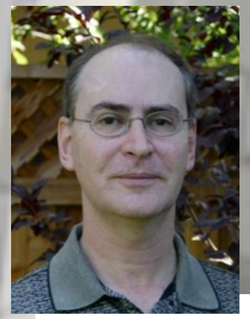
Overloading allows several function definitions for the same name, distinguished primarily through different argument types; it is typically resolved at compile-time. Inheritance allows subclasses to define more special versions of the same function; it is typically resolved at run-time.

Modern object-oriented languages incorporate both features, usually in a type-safe manner. However, the combination of these features sometimes turns out to have surprising, and even counterintuitive, effects.

the appropriate body for a function call is selected at compile time, that is, *overloading resolution* is static (*early* or *static* binding). This form of overloading can be reduced to a syntactic abbreviation.

In languages where types may be computed at run-time, indeed, the appropriate body for a function can be selected depending on the *dynamic type* of the arguments; hence we have *late* or *dynamic binding*. This happens typically with *inheritance* in object-oriented languages; usually run-time selection is adopted only for the receiver (*i.e.*, the first, implicit, parameter) of a method call. The same policy can

… identify the surprise



Systemic Overload

Type-based method overloading is a (mis)feature of mainstream statically typed languages. I am very much against it - I've spoken about it more than once in the past. I decided I should put my views on the matter in writing, just to get it out of my system. I'll start with the better known problems, and work my way up to more exotic issues.

Throughout this post, I'll use the term overloading to denote type based overloading. Arity based overloading, as in e.g., Erlang, is pretty harmless.

When introducing a feature into a language, one has to consider the cost/benefit ratio. You might argue that overloading lets you separate the handling of different types into different methods, instead of doing the type dispatch inside one method, which is tiresome and costly. A

Overloading undesirable

???

Programs
should have type erasure

Ahh, agree …

Ahhh … what about our overloading studies …

… work fast, publish fast

From metaphysics to applications

I was wondering what it might mean for an object to change class

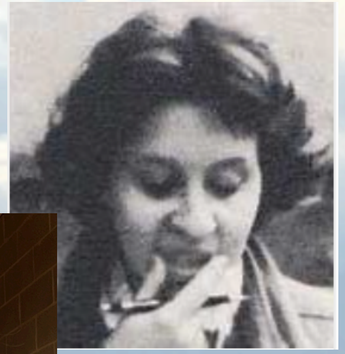
And then I convinced myself that it had no meaning

???

… the power of wonder

Some of the journey went through Torino

Some of the journey went through Torino



$\mathcal{F}ickle$: Dynamic Object Re-classification *

Sophia Drossopoulou[1], Ferruccio Damiani[2], Mariangiola Dezani-Ciancaglini[2], and Paola Giannini[3]

[1] Department of Computing, Imperial College
[2] Dipartimento di Informatica, Università di Torino
[3] DISTA, Università del Piemonte Orientale

Abstract. *Re-classification* changes at run-time the class membership of an object while retaining its identity. We suggest language features for object re-classification, which could extend an imperative, typed, class-based, object-oriented language.
We present our proposal through the language $\mathcal{F}ickle$. The imperative features combined with the requirement for a static and safe type system provided the main challenges. We develop a type and effect system for $\mathcal{F}ickle$ and prove its soundness with respect to the operational semantics. In particular, even though objects may be re-classified across classes with different members, they will never attempt to access non-existing members.

1 Introduction

In class-based, object-oriented programming, an object's behaviour is determined by its class. Case or conditional statements should be avoided when differences can be expressed through different classes. Thus, students paying reduced and employees paying full conference fees are best described through distinct classes Stdt and Empl with different methods fee().

However, this elegant approach does not scale when objects change classification. For example, how can we represent that mary, who was a Stdt, became an

Oper semantics took 1 month

Type system IDEA took one year

echos in typestate

… hard work

… the need for models

Fun and new language features

What *is* Java Binary Compatibility?

Sophia Drossopoulou, David Wragg, Susan Eisenbach
Department of Computing
Imperial College
sd{dpw,se}@doc.ac.ac.uk

Abstract

Separate compilation allows the decomposition of programs into units that may be compiled separately, and linked into an executable. Traditionally, separate compilation was equivalent to the compilation of all units together, and modification and re-compilation of one unit required re-compilation of all importing units.

Java suggests a more flexible framework, in which the linker che
bined. Certai

these units to be compiled one at a time using only the signature (*i.e.* type) information from imported units. The object code of such separately compiled units would be combined by a linker into an executable. If each unit were compiled after any unit it imported, each unit compiled successfully, and all units were present, then linking would be successful. The compiler had to check that units respected imported units' signatures, whereas the linker had to reconcile external references, and to check

A Fragment Calculus
– towards a model of Separate Compilation, Linking and Binary Compatibility

Sophia Drossopoulou, Susan Eisenbach and David Wragg
Department of Computing, Imperial College, 180 Queen's Gate, LONDON SW7 2BZ, England
email: sd,se,dpw@doc.ic.ac.uk

Abstract

We propose a calculus describing compilation and linking in terms of operations on fragments, *i.e. compilation*

tended: not only does it have to resolve external references, it also has to ensure that binaries (the compiled units) are structurally correct and that they respect the types of entities they import from other binaries; the order of compila-

Model for Java Dynamic Linking and Verification

This is my best paper

How can it be rejected?

The reviewers were envious!

I am shattered!

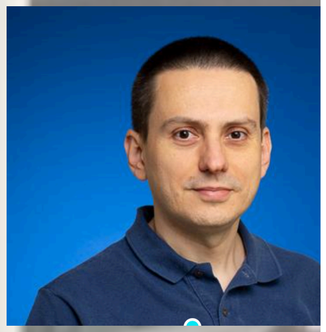
... fun, luck

... delusions of perfection

Fun and new language features

Model for Java Dynamic Linking and Verification

It was not such a great paper after all!



This one is MUCH better!

Flexible Models for Dynamic Linking

Sophia Drossopoulou[1], Giovanni Lagorio[2], Susan Eisenbach[1] *

[1] Department of Computing at Imperial College, [2] DISI at the University of Genova

Abstract Dynamic linking supports flexible code deployment: partially linked code links further code on the fly, as needed; and thus, end-users receive updates automatically. On the down side, each program run may link different versions of the same code, possibly causing subtle errors which mystify end-users.

... don't give up

From prototypes to classes

URL: http://www.elsevier.nl/locate/entcs/volume82.html 29 pages

BabyJ: From Object Based to Class Based Programming via Types

Christopher Anderson [1,2]

Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, U.K.

Sophia Drossopoulou [1,3]

Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, U.K.

Abstract

Object oriented programming can be classified into the object based, and the class based paradigm. Object based languages typically are weakly typed and interpreted, allow member addition and removal, and thus they support flexibility and prototyping. Class based languages are usually strongly typed and compiled, require a rigid class structure and class membership, and thus they support more robust, type safe programs.

Towards Type Inference for JavaScript*

Christopher Anderson[1], Paola Giannini[2], and Sophia Drossopoulou[1]

[1] Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, U.K
[2] Dipartimento di Informatica,
Università del Piemonte Orientale,
Via Bellini 25/G, Alessandria, Italy

Abstract. Object-oriented scripting languages like JavaScript and Python are popular partly because of their dynamic features. These in-

lead to gradual types

Ah!
Microsoft does not fund Javascript
:-(

... lost heart too soon

Making sense of Complex Features

A Model for Java with Wildcards

Nicholas Cameron[1], Sophia Drossopoulou[1], and Erik Ernst[2*]

[1] Imperial College London
{ncameron, scd}@doc.ic.ac.uk
[2] University of Aarhus
eernst@daimi.au.dk

Abstract. Wildcards are a complex and subtle part of the Java type system, present since version 5.0. Although there have been various formalisations and partial type soundness results concerning wildcards, to the best of our knowledge, no system that includes all the key aspects of Java wildcards has been proven type sound. This paper establishes

First model for Wildcards

Java and Scala's Type Systems are Unsound *

The Existential Crisis of Null Pointers

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Ross Tate
Cornell University, USA
ross@cs.cornell.edu

act

sent short programs that demonstrate the unsound-
Java and Scala's current type systems. In partic-
ese programs provide parametrically polymorphic
s that can turn any type into any type without
casting. Fortunately, parametric polymorphism was
grated into the Java Virtual Machine (JVM), so these
es do not demonstrate any unsoundness of the JVM.
eless, we discuss broader implications of these find-
the field of programming languages.

ture, we often develop a minimal calculus employing t
feature and then verify key properties of that calculus. I
these results provide no guarantees about how the feature
question will interact with the many other common featu
one might expect for a full language. The unsoundness
identify results from such an interaction of features. Th
in addition to valuing the development and verification
minimal calculi, our community should explore more w
to improve our chances of identifying abnormal interacti
of features within reasonable time but without unreasona

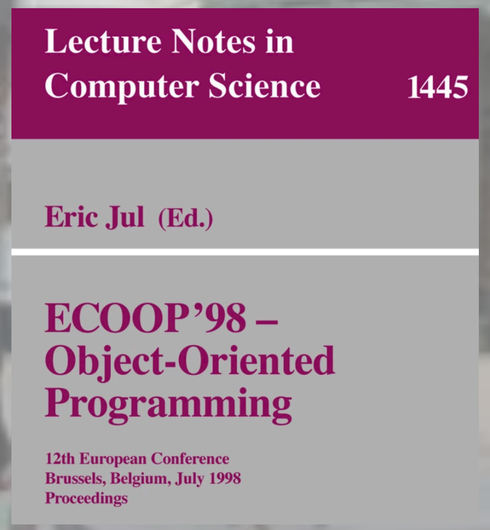
... simple not always good

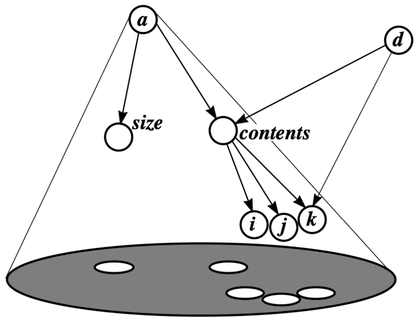
John Potter visiting Imperial

Flexible Alias Protection

James Noble[1], Jan Vitek[2], and John Potter[1]

[1] Microsoft Research Institute, Macquarie University, Sydney
kjx,potter@mri.mq.edu.au
[2] Object Systems Group, Université de Genève, Geneva.
Jan.Vitek@cui.unige.ch





WOW

Can this be used for reasoning about programs?

... the power of wonder

Starting a collaboration

Ownership, Encapsulation and the Disjointness of Type and Effect

Dave Clarke
Institute of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
dave@cs.uu.nl

Sophia Drossopoulou
Department of Computing
Imperial College
London, Great Britain
sd@doc.ic.ac.uk

ABSTRACT

Ownership types provide a statically enforceable notion of object-level encapsulation. We extend ownership types with computational effects to support reasoning about object-oriented programs. The ensuing system provides both access control and effects reporting. Based on this type system, we codify two formal systems for reasoning about aliasing and the disjointness of computational effects. The first can be

[8]. Security, such as required for the safe deployment of applets, depends on certain confinement properties [9, 16]. A concrete notion of object aggregate or component can really only exist when the integrity of the encapsulated objects is maintained [33, 22]. Modular reasoning is only possible when some control/knowledge of the aliasing and effects is possible [58, 56, 38, 43].

These tasks require knowledge about potential aliasing and potential accesses to parts of the state. If aliasing and

Can this be used for reasoning about programs?

YES, We can reason about preservation of properties

FIRST OOPSLA PAPER :-)

... don't be shy!

Starting a collaboration

Ownership, Encapsulation and the Disjointness of Type and Effect

Dave Clarke
Institute of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
dave@cs.uu.nl

Sophia Drossopoulou
Department of Computing
Imperial College
London, Great Britain
sd@doc.ic.ac.uk

ABSTRACT

Ownership types provide a statically enforceable notion of object-level encapsulation. We extend ownership types with computational effects to support reasoning about object-oriented programs. The ensuing system provides both access control and effects reporting. Based on this type system, we codify two formal systems for reasoning about aliasing and the disjointness of computational effects. The first can be

[8]. Security, such as required for the safe deployment of applets, depends on certain confinement properties [9, 16]. A concrete notion of object aggregate or component can really only exist when the integrity of the encapsulated objects is maintained [33, 22]. Modular reasoning is only possible when some control/knowledge of the aliasing and effects is possible [58, 56, 38, 43].

These tasks require knowledge about potential aliasing and potential accesses to parts of the state. If aliasing and

But I missed family trip to France :-(

… everything has got its price

25 Years' work and fun with Types for Heap Topology …

… the strength of types (A'68?)

… the joy of communities

Some other things I did

What lies ahead?

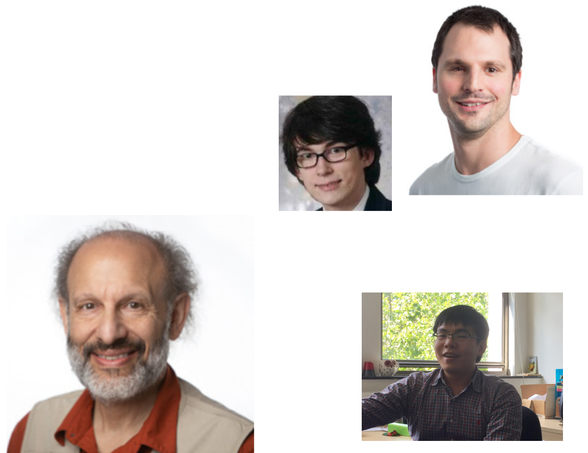
What lies ahead in my journey?

- ……
- Verona Types and Concurrency
- Distributed Garbage Collection
- **Specification/Verification of Object Capabilities**

Safe Journey into the Unknown
— Object Capabilities —

Safe Journey into the Unknown
— Object Capabilities —

Safe Journeys into the Unknown
— Object Capabilities —



… and at Dagsthul …

The essence private/protected?

How does encapsulation help with program verification

It does NOT.

Encapsulation indispensable for proof of holistic properties

… 10 years ago …

need new encapsulation features … membranes?

Ownership types?

… WHAT should these features guarantee?

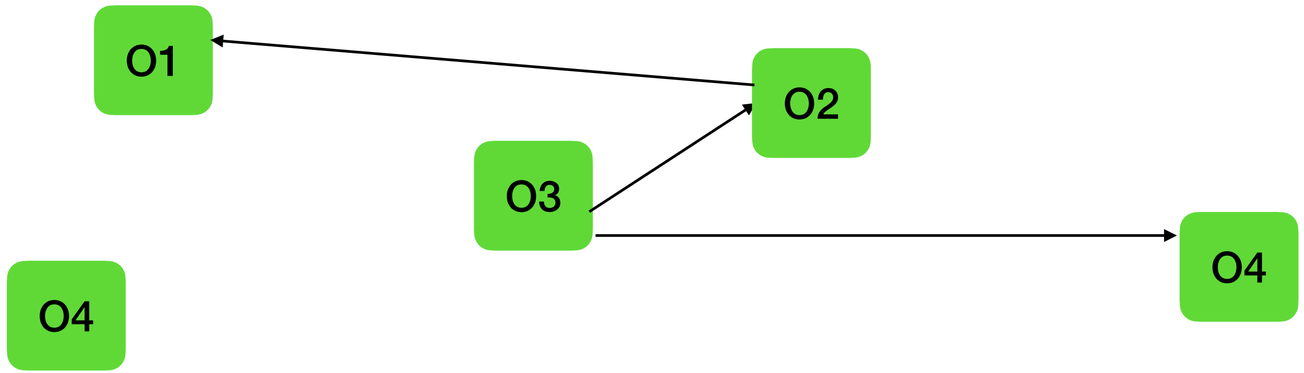
Encapsulation features are *mechanisms f*or robustness, but what are the guarantees?

Object Capabilities

The **object-capability model** is a computer security model. A capability describes a transferable right to perform one (or more) operations on a given object. It can be obtained by the following combination:

- An unforgeable reference (in the sense of object references or protected pointers) that can be sent in messages.
- A message that specifies the operation to be performed.

The security model relies on not being able to forge references.



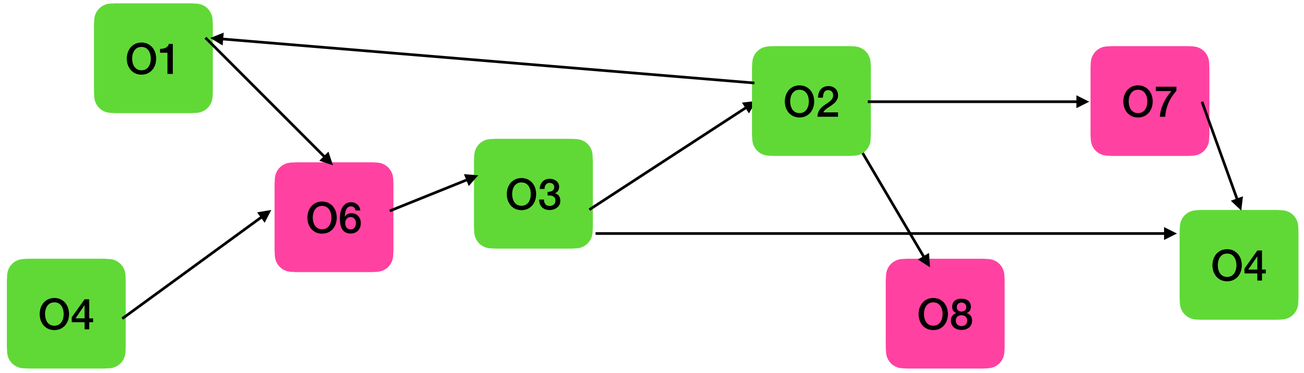
Object Capabilities

The **object-capability model** is a computer security model. A capability describes a transferable right to perform one (or more) operations on a given object. It can be obtained by the following combination:

- An unforgeable reference (in the sense of object references or protected pointers) that can be sent in messages.
- A message that specifies the operation to be performed.

The security model relies on not being able to forge references.

It is about how our, internal, trusted objects will interact securely with the external, untrusted, potentially malicious, objects.



Motivation by Example
and
Four Challenges

Three Modules

```
module Mod_good
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Account, pwd':Password) -> void
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
     method init(pwd':Password) -> void
      if this.pwd==null
        this.pwd=pwd'
  class Password
```

```
module Mod_bad
  class Account
    field balance:int
    field pwd: Password
    method transfer(..) ...
      ... as earlier ...
    method init(...) ...
       ... as earlier ...
    method set(pwd': Password)
      this.pwd=pwd'

  class Password
```

```
module Mod_better
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...

    method set(pwd',pwd'': Password)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

```
module Mod_good
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Account, pwd':Password) ->
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
     method init(pwd':Password) -> void
      if this.pwd==null
        this.pwd=pwd'
  class Password
```

Challenge_1: A module spec S, such that

$M_{good} \models S$

$M_{bad} \not\models S$

$M_{better} \models S$

```
module Mod_bad
  class Account
    field balance:int
    field pwd: Password
    method transfer(..) ...
      ... as earlier ...
    method init(...) ...
       ... as earlier ...
    method set(pwd': Password)
      this.pwd=pwd'

  class Password
```

```
module Mod_better
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...

    method set(pwd',pwd'': Password)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

```
module Mod_good
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Account, pwd':Password) ->
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
     method init(pwd':Password) -> void
      if this.pwd==null
        this.pwd=pwd'
  class Password
```

Challenge_2: An inference system, such that

$M_{good} \vdash S$

$M_{bad} \nvdash S$

$M_{better} \vdash S$

```
module Mod_bad
  class Account
    field balance:int
    field pwd: Password
    method transfer(..) ...
      ... as earlier ...
    method init(...) ...
      ... as earlier ...
    method set(pwd': Password)
      this.pwd=pwd'

  class Password
```

```
module Mod_better
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...

    method set(pwd',pwd'': Password)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

```
module Mod_good
  class Account
    field balance:int
    field pwd: Password
    method transfer(dest:Account, pwd':Password) ->
      if this.pwd==pwd'
        this.balance-=100
        dest.balance+=100
     method init(pwd':Password) -> void
      if this.pwd==null
        this.pwd=pwd'
  class Password
```

Challenge_3: The inference system should be algorithmic

```
module Mod_bad
  class Account
    field balance:int
    field pwd: Password
    method transfer(..) ...
      ... as earlier ...
    method init(...) ...
       ... as earlier ...
    method set(pwd': Password)
      this.pwd=pwd'

  class Password
```

```
module Mod_better
  class Account
    field balance:int
    field pwd: Password
    method transfer(..)
      ... as earlier ...

    method set(pwd',pwd'': Password)
      if (this.pwd==pwd')
        this.pwd=pwd''
  class Password
```

Does it matter?

```
1  module Mod1
2       ...
3      method cautious(untrusted:Object)
4          a = new Account
5          p = new Password
6          a.set(null,p)
7          ...
8          untrusted.make_payment(a)
9          ...
```

If `Account` comes from a “good” module,
and line 7 does not leak the password to `untrusted`,
then
the balance of `a` does not decrease by the call `untrusted` (line 8).

Challenge_4: An inference system, such that we can prove external calls.

Earlier Work

Holistic Specifications for Robust Programs

Sophia Drossopoulou[13][0000−0002−1993−1142], James Noble[2][0000−0001−9036−5692], Julian Mackay[2][0000−0003−3098−3901], and Susan Eisenbach[1][0000−0001−9072−6689]

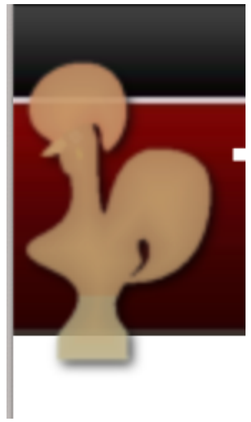
[1] Imperial College London, United Kingdom
{scd,susan}@imperial.ac.uk
[2] Victoria University of Wellington, New Zealand
{julian.mackay,kjx}@ecs.vuw.ac.nz
[3] Microsoft Research Cambridge

Abstract Functional specifications describe what program components *can* do:

Challenge_1: A module spec S, such that

$M_{good} \models S$
$M_{bad} \not\models S$
$M_{better} \models S$

$$\forall a.\forall S : \mathtt{Set}.\ [\ a : \mathtt{Account} \wedge \langle \textbf{will}\langle\ \textbf{changes}\langle a.\mathtt{balance}\rangle\rangle\ \textbf{in}\ S\ \rangle \longrightarrow \exists o.\ [o \in S \wedge \textbf{external}\langle o\rangle \wedge \langle o\ \textbf{access}\ a\rangle]\ \ ]$$



Holistic Specifications for Robust Programs

Sophia Drossopoulou[1 3 [0000−0002−1993−1142]], James Noble[2 [0000−0001−9036−5692]], Julian Mackay[2 [0000−0003−3098−3901]], and Susan Eisenbach[1 [0000−0001−9072−6689]]

[1] Imperial College London, United Kingdom
{scd,susan}@imperial.ac.uk
[2] Victoria University of Wellington, New Zealand
{julian.mackay,kjx}@ecs.vuw.ac.nz
[3] Microsoft Research Cambridge

Abstract Functional specifications describe what program components *can* do:

Challenge_2
Challenge_3
Challenge_4

$$\forall a.\forall S : \mathtt{Set}.\ [\ a : \mathtt{Account} \wedge \langle \textbf{will}\langle\ \textbf{changes}\langle a.\mathtt{balance}\rangle\rangle\ \textbf{in}\ S\ \rangle \longrightarrow \exists o.\ [o \in S \wedge \textbf{external}\langle o\rangle \wedge \langle o\ \textbf{access}\ a\rangle]\ \ ]$$

Necessity Specifications for Robustness

JULIAN MACKAY, Victoria University of Wellington, New Zealand
SUSAN EISENBACH, Imperial College London, United Kingdom
JAMES NOBLE, Creative Research & Programming, New Zealand
SOPHIA DROSSOPOULOU, Meta, and Imperial College London, United Kingdom

Robust modules guarantee to do *only* what they are supposed to do – even in the presence o
malicious clients, and considering not just the direct behaviour of individual methods, but also th
behaviour from calls to more than one method. *Necessity* is a language for specifying robustnes

Challenge_1: A module spec S, such that

$M_{good} \models S$
$M_{bad} \not\models S$
$M_{better} \models S$

```
 from    a:Account ∧ a.balance==bal    to    a.balance < bal
onlyIf  ∃ o.[⟨o external⟩ ∧ ⟨o access a.pwd⟩]
```

Necessity Specifications for Robustnes

JULIAN MACKAY, Victoria University of Wellington, New Zeala
SUSAN EISENBACH, Imperial College London, United Kingdor
JAMES NOBLE, Creative Research & Programming, New Zealan
SOPHIA DROSSOPOULOU, Meta, and Imperial College Lond

Robust modules guarantee to do *only* what they are supposed to d
malicious clients, and considering not just the direct behaviour of i

$$\frac{M \vdash \texttt{from}\ A\ \texttt{next}\ \neg A\ \texttt{onlyIf}\ A'}{M \vdash \texttt{from}\ A\ \texttt{to}\ \neg A\ \texttt{onlyThrough}\ A'}$$

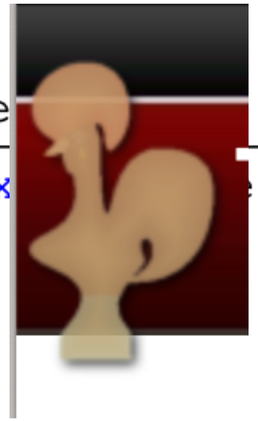
$$\frac{\begin{array}{c} M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_2\ \texttt{onlyThrough}\ A_3 \\ M \vdash \texttt{from}\ A_1\ \texttt{to}\ A_3\ \texttt{onlyThrough}\ A \end{array}}{}$$

$$\frac{\begin{array}{c} \left[\begin{array}{c} \textit{for all}\ C \in \textit{dom}(M)\ \ \textit{and}\ m \in M(C).\texttt{mths}, \\ [M \vdash \texttt{from}\ A_1 \wedge x : C \wedge \langle _\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3] \end{array}\right] \\ M \vdash A_1 \longrightarrow \neg A_2 \qquad M \vdash A_1 \Rightarrow \textit{Enc}(A_2) \end{array}}{M \vdash \texttt{from}\ A_1\ \texttt{next}\ A_2\ \texttt{onlyIf}\ A_3}$$

Challenge_2: An inference system, such

$M_{good} \vdash S$
$M_{bad} \nvdash S$
$M_{better} \vdash S$

$$\frac{M \vdash \{x : C \wedge \neg P\}\ \texttt{res} = x.m(\overline{z})\ \{\texttt{re}}{M \vdash \texttt{from}\ \texttt{inside}(y) \wedge x : C \wedge \langle _\ \texttt{calls}\ x.m(\overline{z})\rangle\ \texttt{nex}\ \ (y)\ \texttt{onlyI}}$$



Proven soundness of $\vdash$.

Proven that good module satisfied spec..

Proven that module with ledger satisfied spec..

Necessity Specifications for Robustness

JULIAN MACKAY, Victoria University of Wellington, New Zealand
SUSAN EISENBACH, Imperial College London, United Kingdom
JAMES NOBLE, Creative Research & Programming, New Zealand
SOPHIA DROSSOPOULOU, Meta, and Imperial College London, United Kingdom

Robust modules guarantee to do *only* what they are supposed to do – even in the presence o
malicious clients, and considering not just the direct behaviour of individual methods, but also th
ifying robustnes

Challenge_3
Challenge_4

Newer Work

Challenge_1: A module spec S, such that

$$M_{good} \models S$$
$$M_{bad} \not\models S$$
$$M_{better} \models S$$

Challenge_1: A module spec S, such that

$M_{good} \models S$
$M_{bad} \not\models S$
$M_{better} \models S$

Remember: A capability represents a transferable right to perform one or more operations on a given object

So: "The password enables withdrawal from the account"?

Or: "Without the password no withdrawal from the account"?

Or: "Without the password no reduction of the balance of the account"?

So: $\forall$ s:Statment. {without a.password $\wedge$ a.balance=b} s { a.balance >= b }

Challenge_1: A module spec S, such that …

So: $\forall$ s. {without a.password $\wedge$ a.balance=b} s { a.balance >= b }

So: $\forall$ a: Account. $\forall$ n: Num. (| **prt** a.password $\wedge$ a.balance=b |) (| a.balance>=b |)

In general: $\forall$ x1:C1,x2:C2… (| A |) (| A' |)

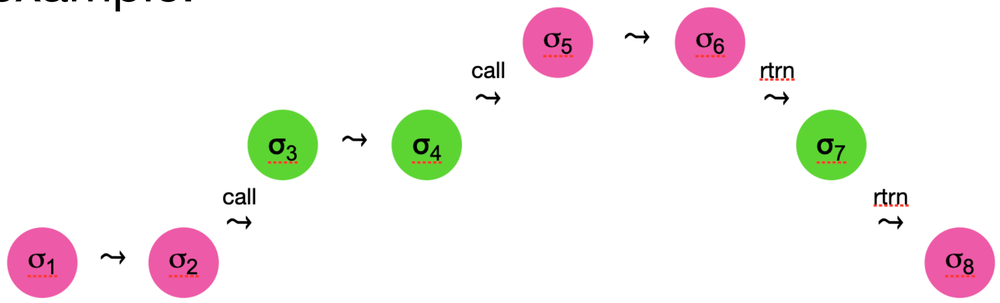
Challenge_1_a : Meaning of **prt** x

Challenge_1_b : Meaning of $\forall$ x1:C1,x2:C1,C2… (| A |) (| A' |)

Challenge_1_b : Meaning of ∀ x1:C1,x2:C1,C2… (| A |) (| A' |)

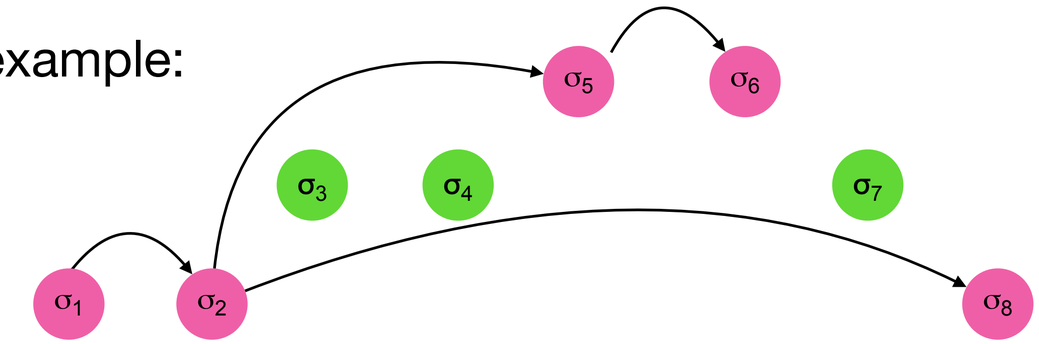
In execution, we pair our module M with an external module, M_{ext}, and distinguish internal stas (whose receiver is internal) from external states (whose receiver is external)

For example:



Def consider only states whose receivers are external, and define a weaker successor relation , where the successor state has not returned from the current method - may have called further methods.

For example:



Challenge_1_b : Meaning of ∀ x1:C1,x2:C1,C2… (| A |) (| A' |)

$$M \models \forall \overline{x : C}.(\!| A |\!)(\!| A' |\!) \quad \text{iff} \quad \begin{array}{l} \text{for all } M_{ext}, \sigma, \overline{x}, \overline{o} \text{ such that } \overline{x} \text{ are free in } \sigma \\ Arising(\sigma, M_{ext}, M) \ \wedge \ Glb_Relvnt(\overline{o}, \sigma) \wedge \\ M, \sigma[\overline{x \mapsto o}] \models \overline{x : C} \ \wedge \ M, \sigma[\overline{x \mapsto o}] \models A \ \wedge \\ M_{ext}; M, \ \sigma \ \curvearrowright^* \ \sigma' \\ \Longrightarrow \\ M, \sigma'[\overline{x \mapsto o}] \models A' \end{array}$$

[14]

For example:

∀ x:Object, (| A |) (| A' |)

Then

$\sigma_1 \models$ y:Object ∧ A

implies

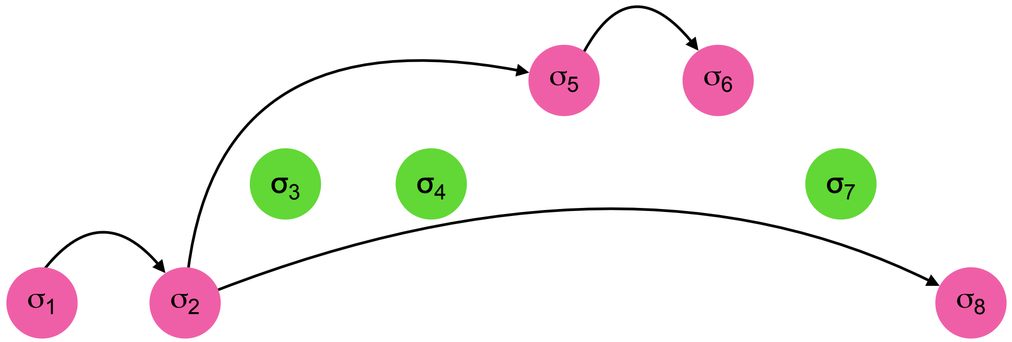
$\sigma_2 \models$ A', $\sigma_5 \models$ A', $\sigma_6 \models$ A', $\sigma_8 \models$ A'.

Also,

$\sigma_5 \models$ y:Object ∧ A

implies

$\sigma_6 \models$ A'.



Challenge_1_a : Meaning of **prt** x

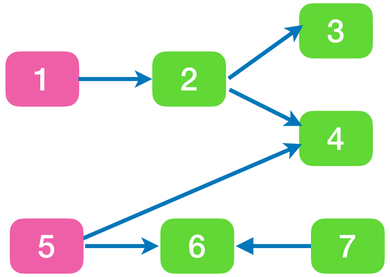
Remember:

…how our, internal, trusted objects will interact securely with the external, untrusted, potentially malicious, objects.

… key who has access to the capability

Def: o **prt-frm o'**, if **extr** o' and any path from o to o' goes through an internal object.

For example:



o3 **prt-frm** o1

o4 **prt-frm** o1

¬ (o4 **prt-frm** o5)

Challenge_1_a : Meaning of **prt** x

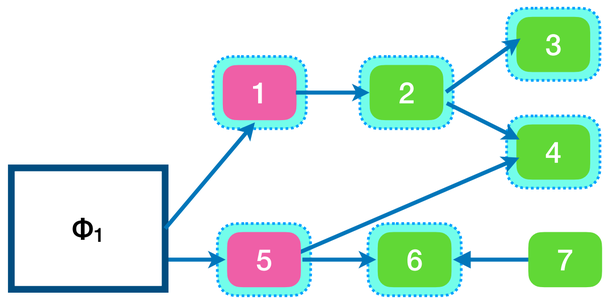
Def: o **prt-frm o'**, if **extr** o' and any path from o to o' goes through an internal object.

Def: Glb_Relevant the set of objects transitively accessible from any frame

Def: Loc_Relevant the set of objects transitively accessible from top frame

Def: o **prt** , if o prt-from and locally relevant object

For example:



Challenge_1_a : Meaning of **prt** x

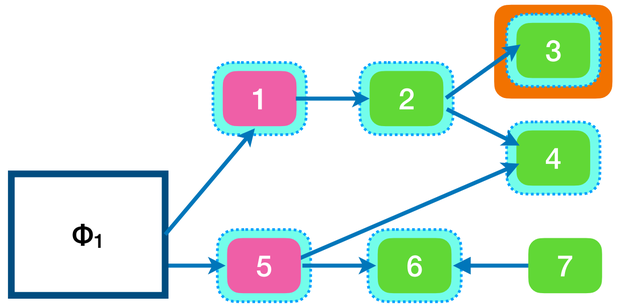
Def: o **prt-frm o'**, if **extr** o' and any path from o to o' goes through an internal object.

Def: Glb_Relevant the set of objects transitively accessible from any frame

Def: Loc_Relevant the set of objects transitively accessible from top frame

Def: o **prt** , if o prt-from and locally relevant object

For example:



Challenge_1_a : Meaning of **prt** x

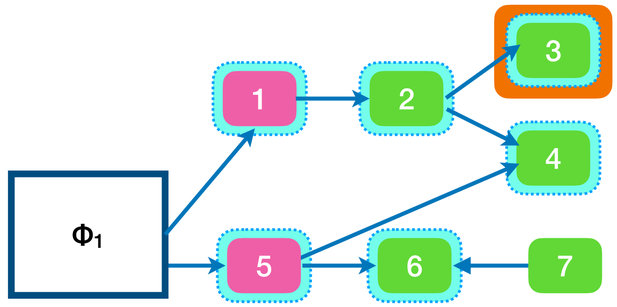
Def: o **prt-frm o'**, if **extr** o' and any path from o to o' goes through an internal object.

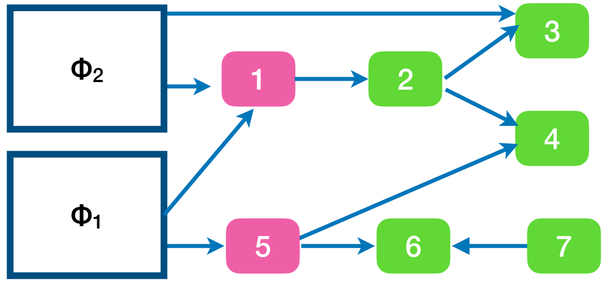
Def: Glb_Relevant the set of objects transitively accessible from any frame

Def: Loc_Relevant the set of objects transitively accessible from top frame

Def: o **prt** , if o prt-from and locally relevant object

For example:





Challenge_1_a : Meaning of **prt** x

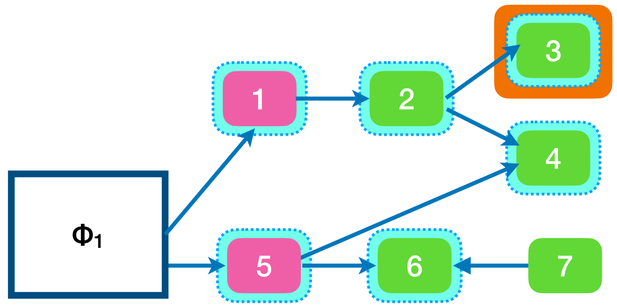
Def: o **prt-frm o'**, if **extr** o' and any path from o to o' goes through an internal object.

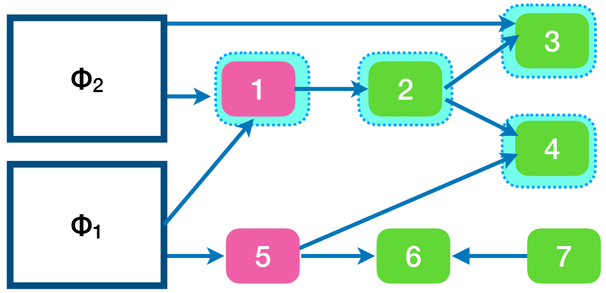
Def: Glb_Relevant the set of objects transitively accessible from any frame

Def: Loc_Relevant the set of objects transitively accessible from top frame

Def: o **prt** , if o prt-from and locally relevant object

For example:





Challenge_1_a : Meaning of **prt** x

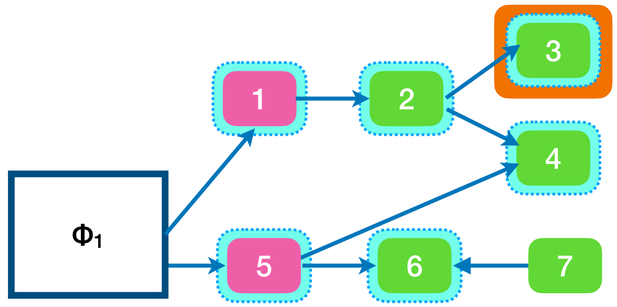
Def: o **prt-frm o'**, if **extr** o' and any path from o to o' goes through an internal object.

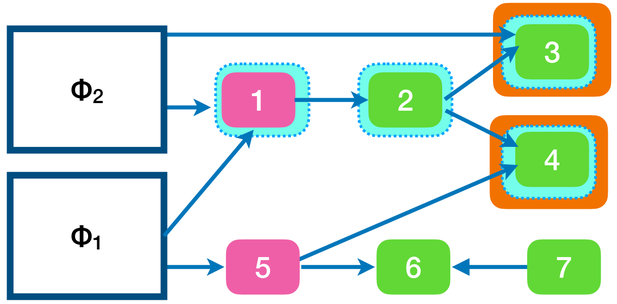
Def: Glb_Relevant the set of objects transitively accessible from any frame

Def: Loc_Relevant the set of objects transitively accessible from top frame

Def: o **prt** , if o prt-from and locally relevant object

For example:



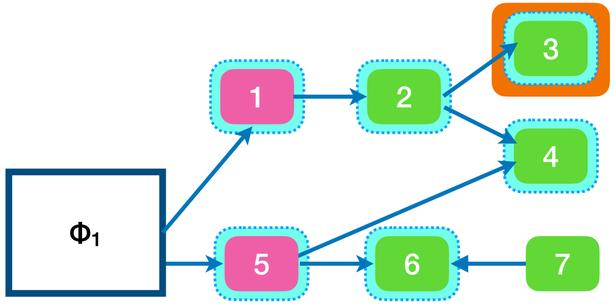


Putting it together

Therefore,

$\forall$ x: Object (| **prt** x $\wedge$ A(x) |) (| A(x) |)

guarantees that if we start below,



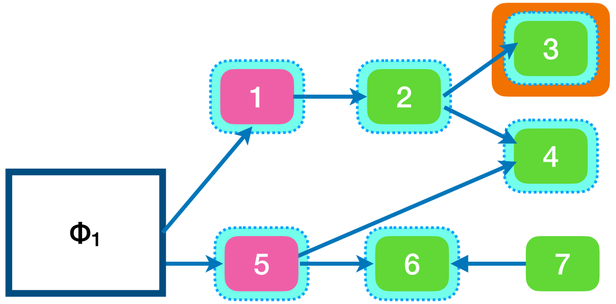
then, A(o3) will be preserved

Putting it together - 2

Therefore,

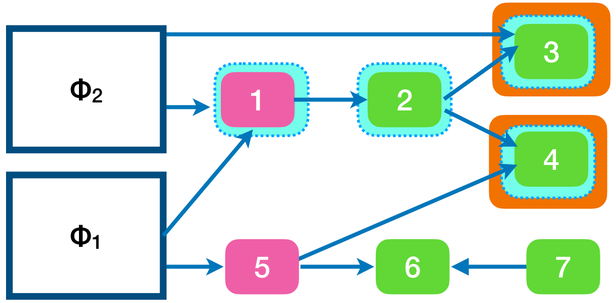
$\forall$ x: Object (| **prt** x $\wedge$ A(x) |) (| A(x) |)

guarantees that if we start below,



then, A(o3) will be preserved

Also guarantees that if we start below,



then, A(o3) and also A(o4)will be preserved

Challenge_1: A module spec S, such that

$M_{good} \models S$

$M_{bad} \not\models S$

$M_{better} \models S$

API - agnostic:
a.balance, a.password can be ghost

Talks about effects

$$
\begin{aligned}
S = \; & \forall\, a{:}Account\ (|\ \mathbf{prt}\ a\ |)\ (|\ \mathbf{prt}\ a\ |) \\
& \wedge \\
& \forall\, a{:}Account\ (|\ \mathbf{prt}\ a.password\ |)\ (|\ \mathbf{prt}\ a.password\ |) \\
& \wedge \\
& \forall\, a{:}Account.\ \forall\, b{:}Num\ (|\ \mathbf{prt}\ a \wedge a.balance = b\ |)\ (|\ a.balance = b\ |) \\
& \wedge \\
& \forall\, a{:}Account.\ \forall\, b{:}Num\ (|\ \mathbf{prt}\ a.password \wedge a.balance = b\ |)\ (|\ \mathbf{prt}\ a.balance \geq b\ |)
\end{aligned}
$$

Talks about emergent behaviour

Challenge_1: A module spec S, such that

$M_{good} \models S$

$M_{bad} \not\models S$

$M_{better} \models S$

Does not hold!

Does not help!

$\forall$ a:Account (| **prt** a |) (| **prt** a |)

$\wedge$

$\forall$ a:Account (| **prt** a.password |) (| **prt** a.password |)

$\wedge$

$\forall$ a:Account. $\forall$ b:Num (| **prt** a $\wedge$ a.balance = b |) (| a.balance = b |)

$\wedge$

$\forall$ a:Account. $\forall$ b:Num (| **prt** a.password $\wedge$ a.balance = b |) (| **prt** a.balance $\geq$ b |)

~~$\wedge$ $\forall$ a:Account (| a.password = p $\wedge$ p=null |) (| a.password = p |)~~

~~$\wedge$ $\forall$ a:Account. $\forall$ p:Password (| a.password = p $\wedge$ **prt** p |) (| **prt** p |)~~

Challenge_2: An inference system, such that

$$M_{good} \vdash S$$
$$M_{bad} \nvdash S$$
$$M_{better} \vdash S$$

Challenge_2: An inference system, such that …

An assertion A is **encapsulated** by module M, if
it can only be invalidated through calls to methods from M

Definition 4.1 (Assertion Encapsulation). An assertion A is *encapsulated* by module M and assertion A', written as

- $M \vDash Encaps(\, A \,)$

if, for all external modules M_{ext}, and all states σ, σ' such that $Arising(M_{ext}, M, \sigma)$, and variables $\overline{x}$ and objects $\overline{o}$:

- $M, \sigma \models A, \quad \overline{x} = Free(A), \quad \overline{o} = \sigma(\overline{x}), \quad M_{ext}; M, \ \sigma \curvearrowright \sigma', \quad M, \sigma'[\overline{y} \mapsto \overline{o}] \nvDash A$

implies

- $\exists y,\ m,\ \overline{y}.[\ \sigma.\mathtt{cont} = y.m(\overline{y}) \ \wedge\ M, \sigma \models \mathbf{intl}\, y\ ]$

Lemma 4.2. *For any module M, and variables x and y:*

(1) $M \vDash Encaps(\ \mathbf{prt}\, x\ )$
(2) $M \nvDash Encaps(\ \neg\, \mathbf{prt}\, x\ )$
(3) $M \nvDash Encaps(\ x\ \mathbf{prt\text{–}frm}\, z\ )$
(4) $M \vDash A \rightarrow A' \ \wedge\ M \vDash Encaps(\, A\,)$ *implies* $M \vDash Encaps(\, A'\,)$.

Proof. Sketches

Challenge_2: An inference system, such that …

Two-State

$$\frac{\begin{array}{c} M \vdash \mathit{Encaps}(\overline{x : C} \wedge A) \\ \textit{for all public methods from } D, \textit{ with } \mathtt{mBody}(m, D, M) = \overline{y : D}\{\, s \,\} \\ M \vdash \{\overline{x : C} \wedge A \wedge \mathtt{this} : \mathtt{D} \wedge \overline{y : D}\}\, s\, \{\, A \,\} \parallel A \end{array}}{\cdots}$$

External-Safe

$$\frac{\begin{array}{c} A_{ois} = \overline{\forall \overline{y : C}.(\!| A'' |\!)} \text{ all object invariants in } HS(M) \qquad A_{strng} = A_{ois} \wedge A' \\ M \vdash \{A_{strng}\}\, s\, \{\, A \,\} \\ \forall s', z, m.[\ (\, s = s'; z.m(_); _ \wedge M \vdash \{A_{strng}\}\, s'\, \{\ \textbf{extl}\, z\ \} \implies M \vdash \{A_{strng}\}\, s'\, \{\, A \,\}\ ] \end{array}}{M \vdash \{A'\}\, s\, \{\, A \,\} \parallel A}$$

Weaken

$$\frac{M \vdash S \qquad M \vdash S \sqsubseteq S'}{M \vdash S'}$$

Multi

$$\frac{M \vdash S \qquad M \vdash S'}{M \vdash S \wedge S'}$$

Challenge_4: An inference system, such that we can prove the “no money lost from a” property in `cautious`.

Challenge_4: An inference system, such we can prove external calls

ExtCall

$$\frac{\begin{array}{c} M \vdash A \;\rightarrow\; (\;\mathbf{extl}\, z \;\wedge\; \mathbf{extl}\, \overline{u} \;\wedge\; \mathbf{intl}\, \overline{y}\;) \qquad \overline{v} \text{ is a permutation of } \overline{u}, \overline{y} \\ \vdash\; M \;:\; \forall \overline{x : C}.(\!| A_1 |\!)(\!| A_2 |\!) \\ M \vdash Lift(A, \{z, \overline{u}\}, \overline{y}) \;\rightarrow\; (\,\overline{x : C} \wedge A_1\,) \end{array}}{M \;\vdash\; \{\; A \;\}\; z.m(\overline{v})\; \{\; Lower(A_2) \;\wedge\; Prsv(A, \{z, \overline{u}\}, \overline{y}, M) \;\}}$$

$$\begin{array}{lcll}
Lift(v = v', \overline{z}, \overline{y}) & = & v = v' & \\
Lift(x.f = v, \overline{z}, \overline{y}) & = & x.f = v & \\
Lift(\,\mathbf{prt}\, x\,, \overline{z}, \overline{y}) & = & \mathbf{prt}\, x & \\
Lift(\,x\, \mathbf{prt\text{–}frm}\, \overline{u}, \overline{z}, \overline{y}) & = & \mathbf{prt}\, x & \text{if } \overline{z} \subseteq \overline{u}, \text{ and } x \notin \overline{y} \\
 & = & \texttt{true} & \text{otherwise} \\
Lift(A_1 \wedge A_2, \overline{z}, \overline{y}) & = & Lift(A_1, \overline{u}, M) \;\wedge Lift(A_2, \overline{u}, M) &
\end{array}$$

$$\begin{array}{lcl}
Lower(v = v') & = & v = v' \\
Lower(x.f = v) & = & x.f = v \\
Lower(\,\mathbf{prt}\, x\,) & = & \texttt{true} \\
Lower(\,x\, \mathbf{prt\text{–}frm}\, \overline{u}) & = & x\, \mathbf{prt\text{–}frm}\, \overline{u} \\
Lower(A_1 \wedge A_2) & = & Lower(A_1) \;\wedge\; Lower(A_2)
\end{array}$$

$$\begin{array}{lcll}
Prsv(v = v', \overline{z}, \overline{y}, M) & = & v = v' & \\
Prsv(x.f = v, \overline{z}, \overline{y}, M) & = & x.f = v & \text{if } \vdash\; M \;:\; \forall \overline{x' : C}, x : D.(\!| x.f = v \wedge A_1 |\!)(\!| x.f = v |\!) \\
 & & & \text{and } M \vdash Lift(A, \overline{z}, \overline{y}) \;\rightarrow\; (\,\overline{x : D} \wedge A_1\,) \\
 & = & \texttt{true} & \text{otherwise} \\
Prsv(\,\mathbf{prt}\, x\,, \overline{z}, \overline{y}, M) & = & \mathbf{prt}\, x & \\
Prsv(\,x\, \mathbf{prt\text{–}frm}\, \overline{u}, \overline{z}, \overline{y}, M) & = & x\, \mathbf{prt\text{–}frm}\, \overline{u} & \text{if } \vdash\; M \;:\; \forall \overline{x' : C}, x : D.(\!| x.f = v \wedge A_1 |\!)(\!| x.f = v |\!) \\
 & & & \text{and } M \vdash Lift(A, \overline{z}, \overline{y}) \;\rightarrow\; (\,\overline{x : D} \wedge A_1\,) \\
 & = & \texttt{true} & \text{otherwise}
\end{array}$$

PROT-1

$$\frac{}{M \vdash \{\ x\ \textbf{prt–frm}\ z\ \wedge\ \textbf{intl}\ v\ \}\ v = v'\ \{\ x\ \textbf{prt–frm}\ z\ \}}$$

PROT-2

$$\frac{}{M \vdash \{\ x\ \textbf{prt–frm}\ z\ \wedge\ \textbf{extl}\ v\ \wedge\ z \neq v\ \}\ v = v'\ \{\ x\ \textbf{prt–frm}\ z\ \}}$$

PROT-3

$$\frac{}{M \vdash \{\ x\ \textbf{prt–frm}\ \{v\ \wedge\ \textbf{extl}\ z\ \}\ z = v\ \{\ x\ \textbf{prt–frm}\ z\ \}}$$

PROT-4

$$\frac{}{M \vdash \{\ x\ \textbf{prt–frm}\ z\ \}\ y.f = v\ \{\ x\ \textbf{prt–frm}\ z\ \}}$$

Challenge_3: The inference system should be algorithmic

Summary

- Specifications talk about necessary conditions for effect:

  $\forall$ x: Object (| **denial of capability** $\wedge$ A |) (| A |)

-