Advanced Issues in Object Oriented Programming *Languages*

Sophia Drossopoulou (scd@doc.ic.ac.uk)

Motivation: Understand design process for oo languages, esp wrt types.

Outline We discuss language features of oo languages, in terms of

- Rationale,
- Implementation,
- Formal description and formalization of properties (no proofs).

What we learn Study all aspects of oo programming language features

Approach partly formal, partly practical

Assessment written coursework (around week 5 – groups up to 5 students). No programming. Exams as usual (past papers available).

Material Slides with holes. Information *with* holes sufficient. Interaction during class. Tutorials at the lecture, sample answers on CATE later on. **Discussion** indicates non-examinable, interesting stuff.

Work lectures integrated with tutorials. Using Piazza and CATE.

Requirements

Necessary skills:

- programmed in an object oriented language
- basic logic (notions $\forall, \exists, \nexists, \in, \emptyset, \Rightarrow, \cap, \cup$)

Useful skills:

- Compilers
- Operational Semantics
- Type Systems

Why take this course?

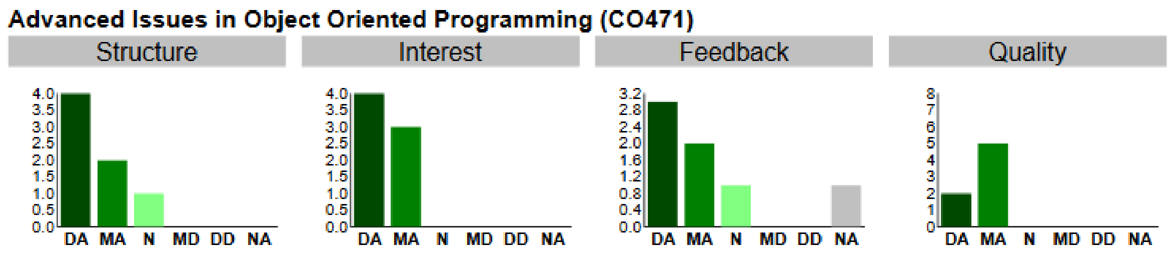
- Interesting stuff,
- Work hard, good marks,
- Good projects, good prospects.

Before moving on -- a word about SOLE

Please fill in. Your input is highly valued.

For example, for AIOOP, past students asked:

1. Use Panopto
2. Do not use Comic font
3. No holes in slides
4. More solved exercises
5. Do not start at 9:00 am; do not teach Wednesdays
6. Improve writing on board



Contents

1. Background and Motivation for formal descriptions
2. L1: Calculus for an imperative, class based language without inheritance (types, type safety)
3. L2: L1 + inheritance
4. Design and Implementation of C++: assignment, object representation, method call, multiple inheritance, static and dynamic binding
5. Implementation of Java: bytecode and verification
6. Dynamic linking and loading in Java
7. Ownership types
8. Verification/Concurrency of OO programs – if we have time

NOTE: 2, 3, 5, 7, 8 formal description. 1, 4, 5, 6, 8 practical approach.

This course is about oo languages, not oo programming!

Background –
The three main (sequential) programming paradigms

Imperative ----------------------→ we ask …

---- Functional -----→ we ask …

Declarative

----- Logic -----→ we ask …

Object Oriented -------------------→ we ask …

Background –
Characteristics of Object Oriented Programming

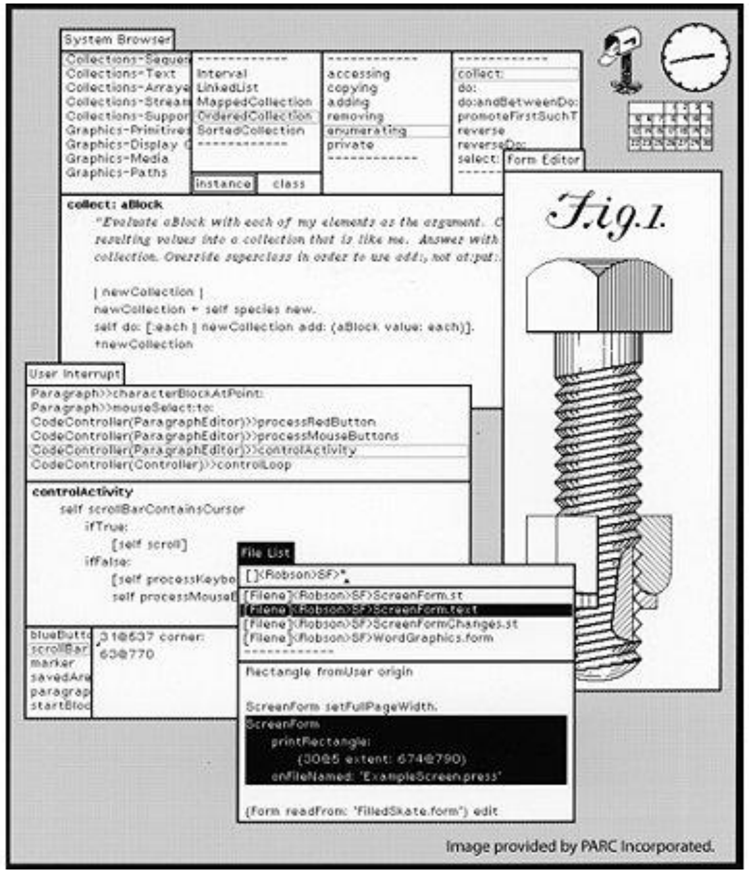
Object oriented programming was developed to support simulations (Simula'67), and the implementation of a novel, revolutionary, paradigm in HCI (Smalltalk'72).

There are many flavours of object oriented programming languages, but they share some characteristics:

- objects are the basic unit of decomposition,
- objects have state and processing ability,
- objects exchange messages,
- encapsulation,
- dynamic method binding,
- sharing of code.

60's: Simula'67 was developed to write simulations of the Oslo harbour, by Dahl and Nygaard.

70's-80's: Smalltalk '72 and Smalltalk'80 developed as the quintessential oo programming language, and user environment.



```
Collections-Sequen  ------------          ------------   ------------
Collections-Text    Interval              accessing      collect:
Collections-Arraye  LinkedList            copying        do:
Collections-Stream  MappedCollection      adding         do:andBetweenDo:
Collections-Suppor  OrderedCollection     removing       promoteFirstSuchT
Graphics-Primitives SortedCollection      enumerating    reverse
Graphics-Display C  ------------          private        reverseDo:
Graphics-Media                            ------------   select:
Graphics-Paths      instance   class
```

```
collect: aBlock
    "Evaluate aBlock with each of my elements as the argument. C
    resulting values into a collection that is like me. Answer with
    collection. Override superclass in order to use add:, not at:put:.

    | newCollection |
    newCollection ← self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ↑newCollection
```

```
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
```

```
controlActivity
    self scrollBarContainsCursor
        ifTrue:
            [self scroll]
        ifFalse:
            [self processKeybo
            self processMouse
```

```
blueButt   31@637 corner:
scrollBar  63@770
marker
savedAre
paragrap
startBloc
```

```
[]<Robson>SF>*
[Filene]<Robson>SF>ScreenForm.st
[Filene]<Robson>SF>ScreenForm.text
[Filene]<Robson>SF>ScreenFormChanges.st
[Filene]<Robson>SF>WordGraphics.form
------------
Rectangle fromUser origin

ScreenForm setFullPageWidth.
ScreenForm
    printRectangle:
        (30@5 extent: 674@790)
    onFileNamed: 'ExampleScreen.press'

(Form readFrom: 'FilledSkate.form') edit
```

Image provided by PARC Incorporated.

vs

Background – issues in OO languages

- class based vs object based languages
- static types vs dynamic types
- structural types vs name types
- static binding vs dynamic binding
- inheritance vs delegation
- polymorphism through inheritance, or parameterised abstract data types
- subtypes vs subclasses, interf. inheritance vs implementation inheritance
- information hiding and alias protection
- multiple inheritance
- separate compilation, open world vs closed world
- object reclassification, roles, and self-modifying code
- combination with other paradigms
- closures, traits
- *concurrency and distribution*
- *verification for oo programs*
- …

Many of above not exclusive to oo paradigm. Course discusses some of above.

Motivation for formal descriptions – in general

Formal descriptions

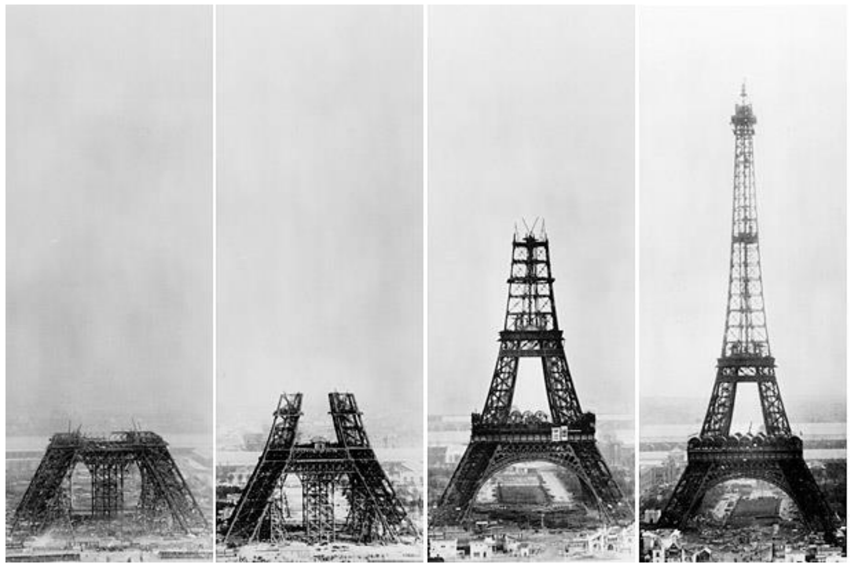
- are succinct,
- are precise,
- give an additional view
- uncover flaws, and can give certainty of correctness.

But, formal descriptions have the following limitations:

- How much to formalize?
- Hand-written proofs may contain errors.
- Tools (eg Coq, Isabelle, Dafny) are “heavy” – but there is progress.

Motivation for formal descriptions - Eiffel

We discuss a problem related to field specialization, which is tackled differently in various languages, with disastrous consequences in some cases.



We will first clarify the meaning of static/dynamic types, static/dynamic errors and open/closed world view.

Then we will discuss an example in Eiffel, show what the problems are, and present six different approaches.

Static/dynamic types/errors

Static types are determined at compile-time. Static types depend only on the declared types of entities, and are immutable.

Dynamic types are determined at runtime. An expression may have different dynamic type at different times of execution.

For example

```
    class Person{}; class Man extends Person{};

    Person john;
    john = new Person;
    // john has static type Person
    // john has dynamic type Person
    john = new Man;
   // john has static type Person
   // john has dynamic type Man
```

Static errors are detected at compile-time. Usually, a program with static errors is not executed.

Dynamic errors are detected at run-time. Usually, if a dynamic error is detected, execution stops. Some dynamic errors are type errors.

For example

```
    int i;
    Person john;
    john = i;              // static type error
    i = 10/0;              // dynamic error
    john = new Man;
    (Man)john;
    john = new Person;
    (Man)john;             // dynamic type error
```

We also use the terms

- *compile-time type* and *compile-time error* (static)
- *run-time type* and *run-time error* or *exception* (dynamic)

Open/Closed world

… describes amount of information necessary before certain task (e.g. compilation, checking, linking, or execution) can be performed.

Closed world view requires all the code of a project to be present before the task.

Open world view requires only some of the code of a project to be present before the task.

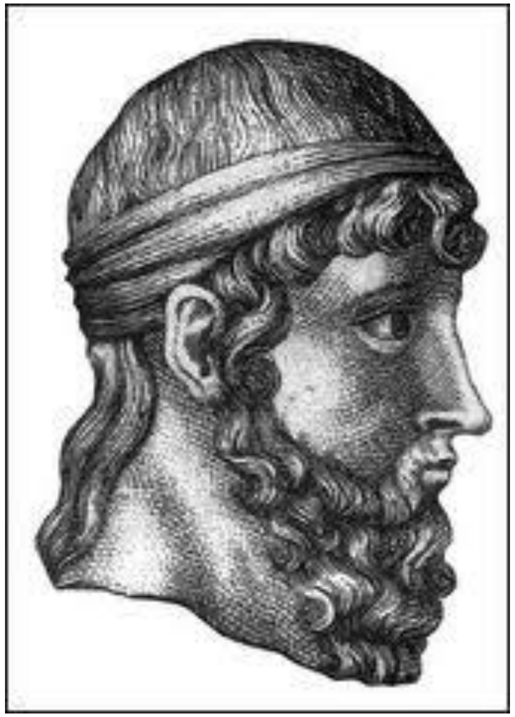
For example:

- Separate compilation in Java, C++ - closed world view.
- Linking in Modula-2 and C++ - closed world view.
- Linking in Java - open world view.
- Data flow analysis – usually closed world view.

Discussion: Advantages/disadvantages of open/closed, and static/dynamic. What are currently prevalent trends?

The problem: Field specialization and type soundness

- an example -



Assume that we have people and philosophers, and that people like books, while Philosophers like philosophical books.

Consider the following program:

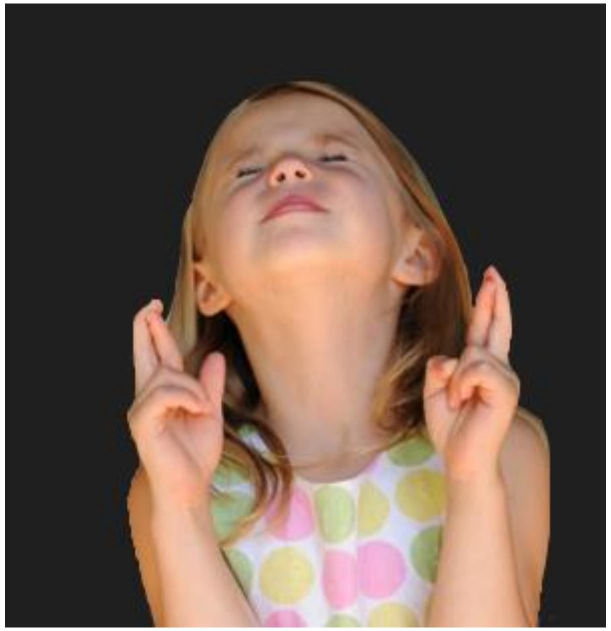
```
class Person{
    Book like;
}
class Philosopher extends Person{
    PhilBook like;
}

class Book{   }
class PhilBook  extends Book{
    void print(){…}
}
```

In Eiffel, the definition of the field `like` in class `Philosopher` "overrides" the definition of the field `like` in class `Person`. Therefore, `Person` objects have a field `like` of type `Book`, whereas `Philosopher` objects have a field `like` of type `PhilBook`.

Therefore, `Person` and `Philosopher` objects could look as follows:

This seems like a sensible set-up.



Now, consider the following:

```
    Person john; Philosopher plato;
    john=new Person;               // 1
    plato=new Philosopher;         // 2
    plato.like=new PhilBook;       // 3
```

Object store after 3 could look like:

Now, consider:

```
    john=plato;                    // 4
    john.like=new Book;            // 5
```

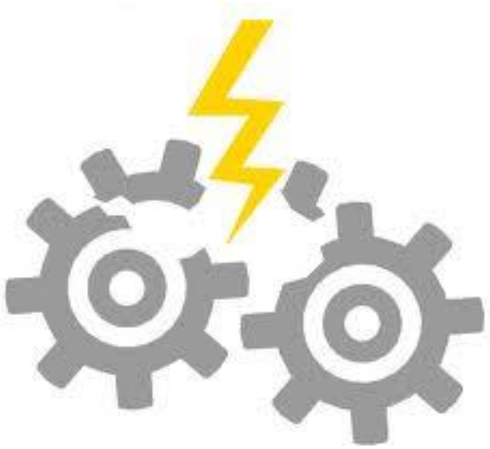
Object store after 5 could look like:

Now, execute:

```
    plato.like.print();        // static type OK
                               // dynamic error!!
```

In dynamically typed languages the program is “stuck”, because `Book` does not have a `print` method. But in statically typed languages, the behaviour is undefined.

The above program and lines 1-5 were type correct in the first release of Eiffel.



This means that the type system of the first release of Eiffel is **not sound**!

*A type system is **sound**, if*

execution of any term with (static) type `T` returns an object of type `T'` where `T'` is a subtype of `T`.

Soundness of type systems is a crucial property: Sound type systems protect memory from illegal (or malicious) misuse – most attacks on Java security were based on fooling the verifier, ie on breaking the type system of the Java bytecode (cf Drew Dean et al).

Bertand Meyer

Above issue already known in 1985 (Cardelli: A Semantics of Multiple Inheritance, Information and Computation, 1988). However, Eiffel was designed and implemented with that flaw (1988), and the problem was discovered and published (Cook, OOPSLA'1990) *after* the deployment of many Eiffel programs.

Luca Cardelli

At that stage, the treatment of the problem for Eiffel was very difficult, because of the requirement of *backwards compatibility*.

William Cook

Do you know of any current programming language with an unsound type system?

more next time!

Back to Eiffel.

What can be done about field specialization?

SEVEN approaches to field specialization

We will briefly mention seven approaches to the problem of field specialization. These have to address the following two questions:

Q1: What is the “shape” of objects of a subclass which “redefines” a field from a superclass?

Q2: What is the meaning of field access?

1st approach (the “purist” approach)

Q1: Subclasses may not redefine fields defined in superclasses.
Q2: Fields selection depends on field identifier, and nothing else.

2nd approach (the C++/Java approach)

Q1: A field defined in a subclass “hides” but does *not* override any fields with same name from superclasses.
Q2: Fields selected according to *static* type of object from which selected.

3rd approach (similar to Java arrays approach)

Q1: If a field has the same name as a field defined in a superclass, then it overrides that field, and must have a subtype of the type of the field in that superclass.

Q2: Assignment to field checks at run-time that value assigned has appropriate type for the run-time class of the object.

4th approach (the dataflow approach - Eiffel)

Q1: As in the 3rd approach.

Q2: Dataflow analysis estimates conservatively possible run-time types. Check at compile-time that value assigned to object's field has appropriate type for estimated run-time class of object.

5th approach ("exact" vs "loose" types – D. Yang's PhD 97)

Q1: As in the 3rd approach

Q2: *Loose* types, e.g. `<C>`, denote a class and all subclasses. Exact types, e.g. `[C]`, denote only the specific class. With these types, we conservatively check field assignments.

Discussion similar distinction can be found in C++ by distinguishing type from type .

6th approach (subclasses are not subtypes)

A subclass introduces code inheritance, but not specification inheritance (not subtypes)

Q1: If a field has the same name as a field defined in a superclass, then it overrides that field.

Q2: field access as normal.

7th approach (Michal Srb, MEng IC 2011-2015)

Q1: If a field has the same name as a field defined in a superclass, then it overrides that field, provided that the overridden field is constant, and the overriding type is a subtype of the type of the overridden field.

Q2: field read as normal, assignment forbidden for constant fields

Discussion

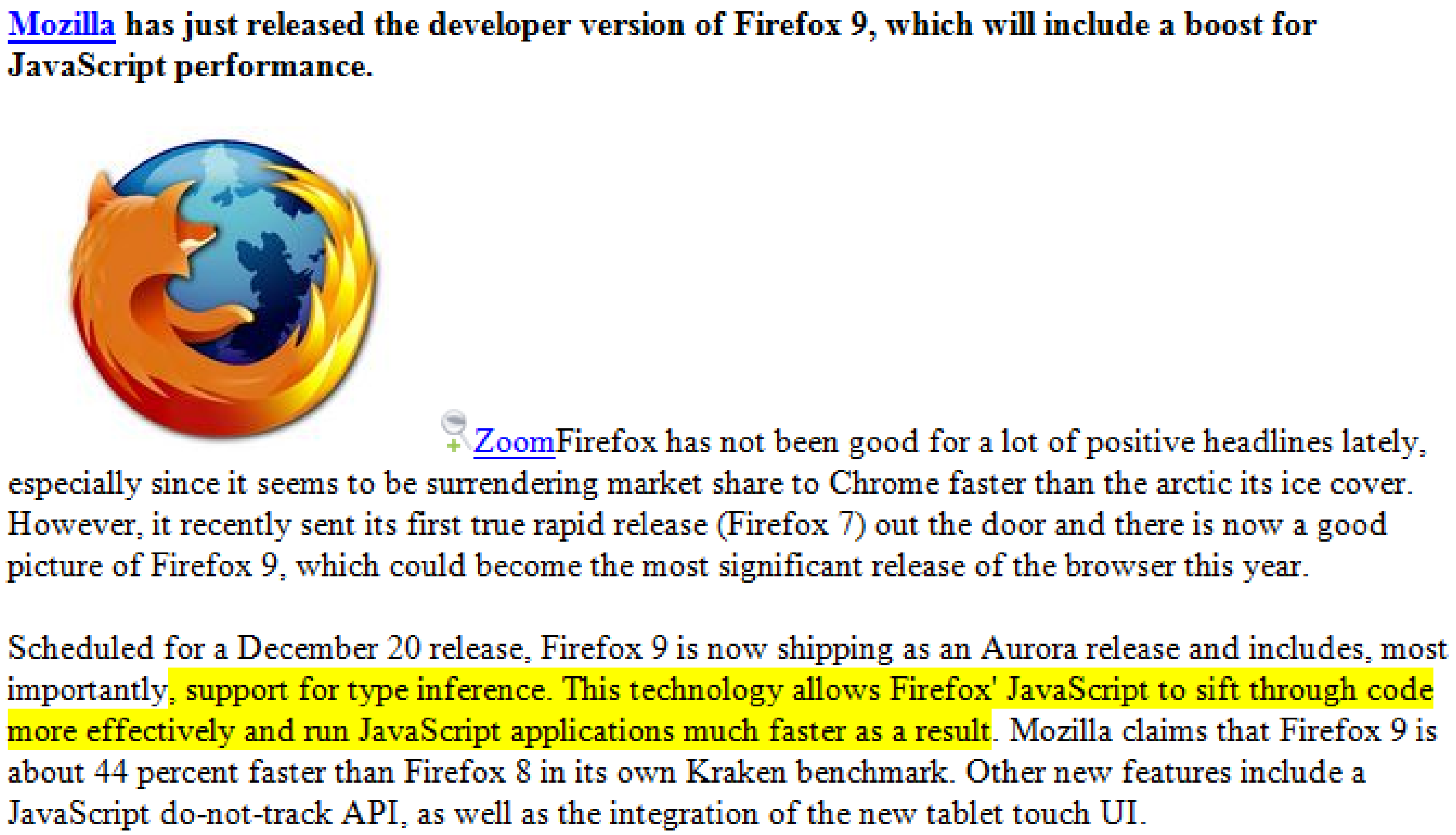
- Which approach would you chose for your own, new programming language and why?
- Which approach was chosen in Dart or TypeScript?

Formal Language Descriptions

- allow a succinct and precise description of language features,
- give the means to prove soundness (and other properties of a programming language).

As language features are becoming more sophisticated, the use of a formal description is indispensable (c.f. generics in Java and C#, database access facilities in C#)

ISOs and Individual projects - the past



Mozilla has just released the developer version of Firefox 9, which will include a boost for JavaScript performance.

ZoomFirefox has not been good for a lot of positive headlines lately, especially since it seems to be surrendering market share to Chrome faster than the arctic its ice cover. However, it recently sent its first true rapid release (Firefox 7) out the door and there is now a good picture of Firefox 9, which could become the most significant release of the browser this year.

Scheduled for a December 20 release, Firefox 9 is now shipping as an Aurora release and includes, most importantly, support for type inference. This technology allows Firefox' JavaScript to sift through code more effectively and run JavaScript applications much faster as a result. Mozilla claims that Firefox 9 is about 44 percent faster than Firefox 8 in its own Kraken benchmark. Other new features include a JavaScript do-not-track API, as well as the integration of the new tablet touch UI.

Christopher Anderson: Type Inference for Javascript, PhD, Imperial College, 2006

Some past ISOs and Individual projects

- Alex Buckley, Ownership Types, Distinguished Project, then PhD IC, then Sun/Oracle Java Language Designer
- Christopher Anderson, Implementing Fickle, Distinguished Project, 3 papers on that, then PhD IC, then TfL, now start up
- Alistair Wren, Type Inference for Ownership Types, Distinguished Project, then PhD Cam
- Charles Smith, Traits for Java, and paper at ECOOP'05
- Rok Strnisa, Modelling C++, then PhD Cam, now start up
- Azalea Raad, Multirole session types, Distinguished Project, then PhD IC, now Max Plank
- William Sonnex, Theorem Proving for Inductive Properties, MSR Prize, paper at TACAS, then PhD Cam
- Raoul Gabriel Urma, Relationships in Java, Qualcomm Prize, then PhD Cam and startup
- James Elford, Types for NonNullity and Immutability, JMC prize
- George Steed, A principled design of object capabilities for Pony, 2016, Centenary Prize, now ARM
- Luke Cheeseman, Value Dependent Types for Pony, 2016, Corporate Partnership Prize, now ARM
- Paul Lietar, Formalizing Generics in Pony, 2017, Corporate Partnership Prize, now Google and then MSR, and PhD IC (2019)
- Daniel Slocombe 2018 Reliable Garbage Collection in Pony, Distinguished Project and 3rd Prize at ACM Student Research Competition <Programming>, now Microsoft

ISOs and Individual projects - current

With Sylvan Clebsch and/or Juliana Franco

- Generics in Pony
- Borrowing in Pony
- Sound Garbage Collectors in Pony

[[With Mark S. Miller (Google) and James Noble (VUW)]]

- Object Capabilities for Safe Open Programs
- Robust Blockchain Contracts

[With Mark Wheelhouse]

- iProve

Eager to hear your ideas too ...