A new programming language: what for?

Xavier Leroy

INRIA Rocquencourt

Introduction

I wasn't sure what this meeting is about: improving Web programming, or just talented P.L. people looking for what to do next?

Will look for problems and application areas where a new programming language could be warranted.

- Web programming (imposed).
- Programming with formal methods in mind.
- Programming with testing in mind.

Will not cover: our current work plans for Caml and its future.

Web programming

Web programming

Current practice for server-side and client-side Web programming is messy. A big gap between the problem and the tools commonly used (general-purpose languages, scripting languages, RDBMs).

Could a new programming language help here?

Possible areas for improvement

More transparent database interfaces.

- Support for data persistence in general.

More "functional" user interactions.

- Less reliance on session state during interactions.

Better handling of XML data.

- Static typing of XML transformations.
- XML navigation, queries.

Better support for two-level programming.

- Server-side program generates bits of client-side scripts on the fly.

The Hyper-Learning proposal

Web navigation in large "conceptual networks" of literary documents (critical editions, manuscripts, commentaries, etc.)

Original prototype (Hyper-Nietzche) using conventional tools was a mess. Developers expressed need for functional languages.

A small C.S. part focused on:

- Stateless user interaction (Wash).
- XML transformation languages.
- Adding data persistence to functional languages.

Call "Hyper-learning and access to cultural heritage". Failed with honors. The use of F.P. to improve Web applications was favorably received.

The Actiweb proposal

Originally centered on:

- XML transformation languages.
- Semi-structured databases.
- Distributed Web programming
  (from Web services to process calculi).

Call "Global Computing". Failed with some strong criticisms ("not very ambitious in proposing innovative paradigms beyond an incremental progress in the state of the art").

My feelings

Web programming raises some interesting language issues, notably at the frontier with databases.

Can we do better than "an incremental progress in the state of the art"?

Huge amounts of grunt work (libraries, frameworks) before potential users will just contemplate possibly using it.

The community of potential users is large but fragmented in many independent developers. Who are the heavyweight industrial partners with R&D departments?

Is the problem serious enough to justify heavy involvement?

Programming with formal methods in mind

The need for formal methods

The highest levels of software certification now require formal methods. This is a huge challenge for the safety-critical industry – one of the few areas where they absolutely need help from academia.

Domain-specific languages such as Scade or Esterel were very successful in this area: they are restrictive enough to enable full verification by model checking.

More complex programs require program proof, and this is a big challenge – but also an academic area that has made big progresses lately.

Pure functional programming to the rescue

Provers such as HOL and Coq support executable specifications written in pure functional style.

This is functional programming at its purest:

- No side-effects (mutable state, exceptions, etc).
- All programs must be strongly normalizing (no lazyness).

I am currently writing and proving correct an optimizing compiler from C to PowerPC assembly code, written in Coq.

Uses most F.P. tricks in the books: persistent data structures, state and error monads, etc.

What can P.L. people contribute?

Programming in Coq is difficult, but still a breeze compared with proving.

Most improvements will come from the "proof" side.

We P.L. people could contribute:

- Generating efficient and provably correct code from the functional specifications.
- Our languages should contain a well-identified pure functional subset that can be imported directly as functional specifications.
- Writing libraries of (proved) persistent data structures.
- Proof principles and tactics for monadic programming.
- Software-proof co-design.

My feelings

A narrow area, incredibly difficult, highly rewarding.

If pure functional programming has a future, this is it.

Well-identified, interesting industrial partners with strong R&D departments: transportation, aerospace, security.

Programming with testing in mind

Testing: the dark side of programming

Even when certification at high levels is not requested, rigorous testing is a necessity for industrial-quality software.

Development of tests usually takes more time (and costs more) than programming.

We claim that our languages make programs safer and easier to write. But does this translate into less testing work?

Even if a Haskell or Caml program generally has much fewer bugs than the equivalent C program, it is not significantly simpler to test.

One interesting attempt: Quick Check. But random testing is no substitute for a test suite.

The challenge

Is it possible to design a programming language that takes testability into account?

- Support for pre- and post-conditions and data structure invariants.

  (Eiffel, Java Modeling Language, Spec#).

- Simultaneous design of the programming language and the specification language.

- Ability to enumerate data structures.

  (Datatypes are great; datatypes with observable sharing, less so.)

- High-level description of decision trees.

  (Pattern-matching is better than cascades of `if...then...else`. What would be better than pattern-matching?)

- Ability to "invert" computations: what inputs, if any, lead to a given outcome?

- And probably much more.

My feelings

Could have major impact if successful.

Need to understand what these testing methodologies are all about.

Not clear yet whether the language design can make a big difference.