

SAFARI: A Meta-Tooling Platform for Creating Language-Specific IDEs

Robert M. Fuhrer, Philippe Charles, Stanley M. Sutton Jr.
(IBM T. J. Watson Research Center)

Chris Laffra
(IBM Rational)

Outline

- **Introduction**
- SAFARI IDE Development Process Walk-through
- SAFARI Architecture
- Status & Future Work

Motivation: Easier IDE Creation

- New programming languages are being developed all the time
 - “Pure” language research – X10, Fortress, SQLJ, XJ, Linq, PolyJ,...
 - Languages to support new architectures, environments, ...
 - Domain specific languages
 - Scripting languages
- Evaluation of language design requires analysis of prolonged use on significant code bases
- IDE support is **critical** to adoption and substantial use of new language
- Many existing languages still don't enjoy support in mainstream IDEs

SAFARI Target: Desired IDE Functionality

syntax highlighting, compiler annotations, hover help, source folding, formatting...

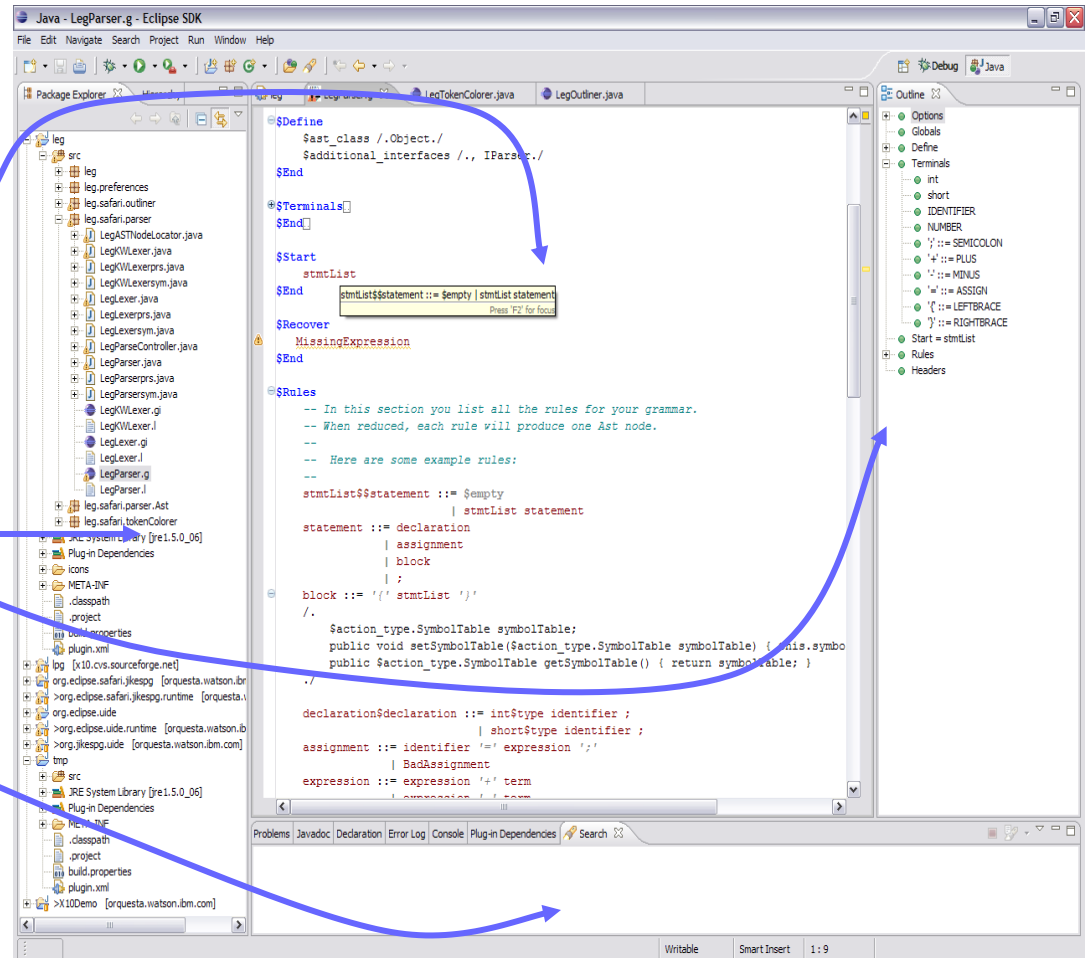
navigation (hyperlinks, "Open Type", ...)

content assist, quick fixes

structural views

compiler w/ incremental build, automatic dependency tracking

- New Project/Type/... creation wizards
- refactoring
- launch & debug: launch configs, breakpoints, backtraces, values, evaluation



JDT sets a very high bar!

SAFARI Approach

- Take advantage of common themes, structures, semantics
 - Encapsulate common IDE & language idioms
 - Language inheritance:
 - Δ in language structure/semantics \Rightarrow Δ in implementation
- Meta-tooling for language-specific IDEs
 - Language-definition support for syntax, auto-generated ASTs, analyses
 - Framework classes for IDE components
 - DSL's to more easily implement language services
 - Extensible multi-language static analysis framework (WALA)
 - Refactoring support
- Guide developer and direct focus to relevant APIs & customization sites
 - Cheat sheets, wizards, default implementations, example IDEs, ...

Enable IDE developers to get on with the interesting work!

Outline

- Introduction
- SAFARI IDE Development Process Walk-through
- SAFARI Architecture
- Status & Future Work

SAFARI Development Process: Overview

- Start with a plugin project (duh!)
- Define language descriptor
 - Identify base language (if any), file name extensions, ...
 - In the future: use standard Eclipse “content types”
- Define lexical and grammar specifications
 - Using LPG: create grammar skeleton; complete it; parser and AST types automatically generated
 - In the future: interoperate with other parser generators
 - Or do it all yourself
- Define various language services
 - Mostly in any order, though a few constraints (e.g., reference resolver before content assistance)
 - Customize each selected service as necessary

Demo, part 1: Basic Services

SAFARI Development Process: Adding a Builder/Compiler

- Create skeleton using wizard and SAFARI class library
- Flesh out skeleton:
 - Call out to an existing compiler
 - direct compiler messages to **IMessageHandler**
 - Write a new compiler starting from AST
 - If using Polyglot: implement standard analyses (type checking, reachability...)
 - Implement dependency visitor
- If compiler generates Java™ source: line breakpoint support by adding SMAP (JSR-44) attributes to generated Java class files
 - compiler inserts “**//#line**” comments to indicate original source location

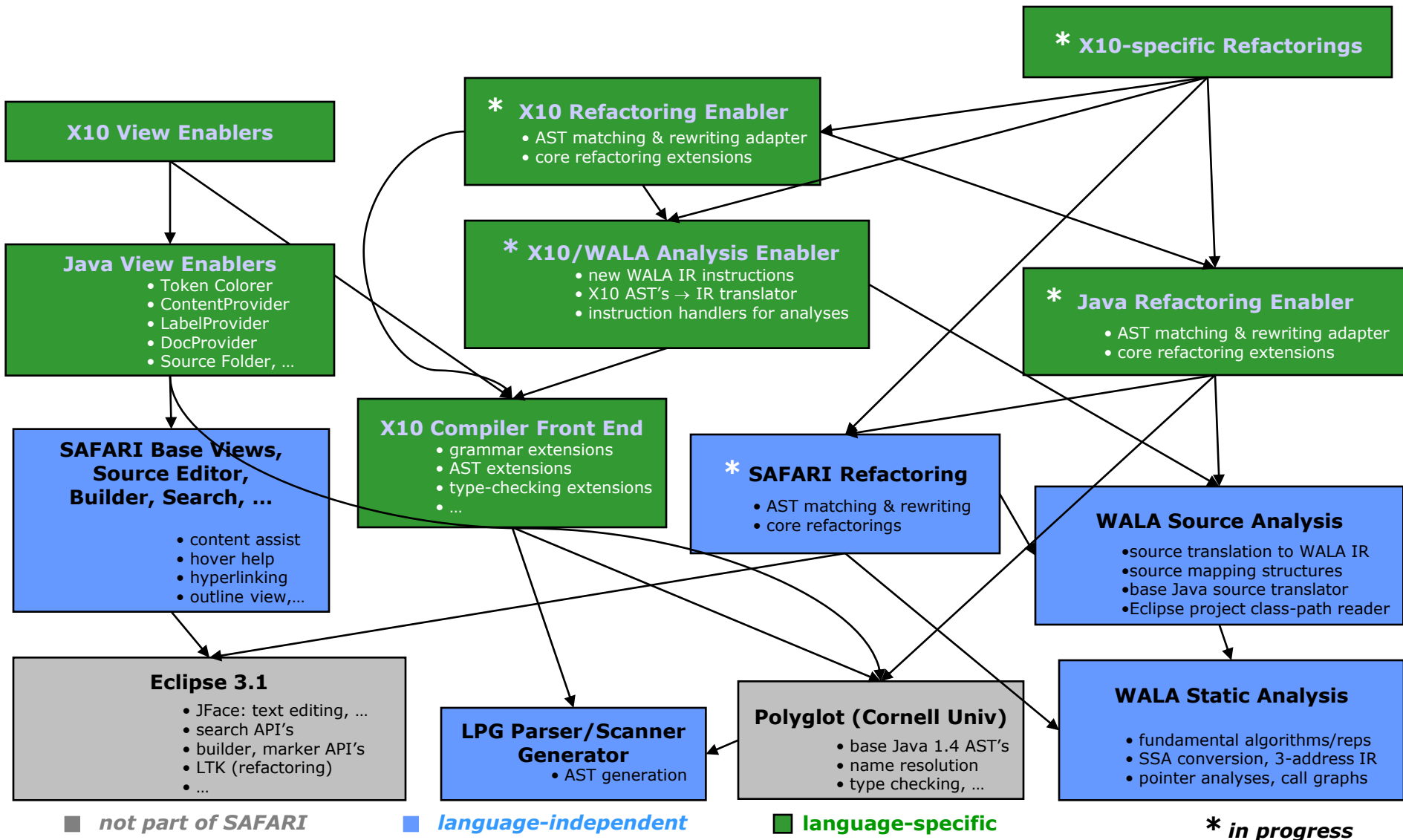
Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Demo, part 3: Building and Execution

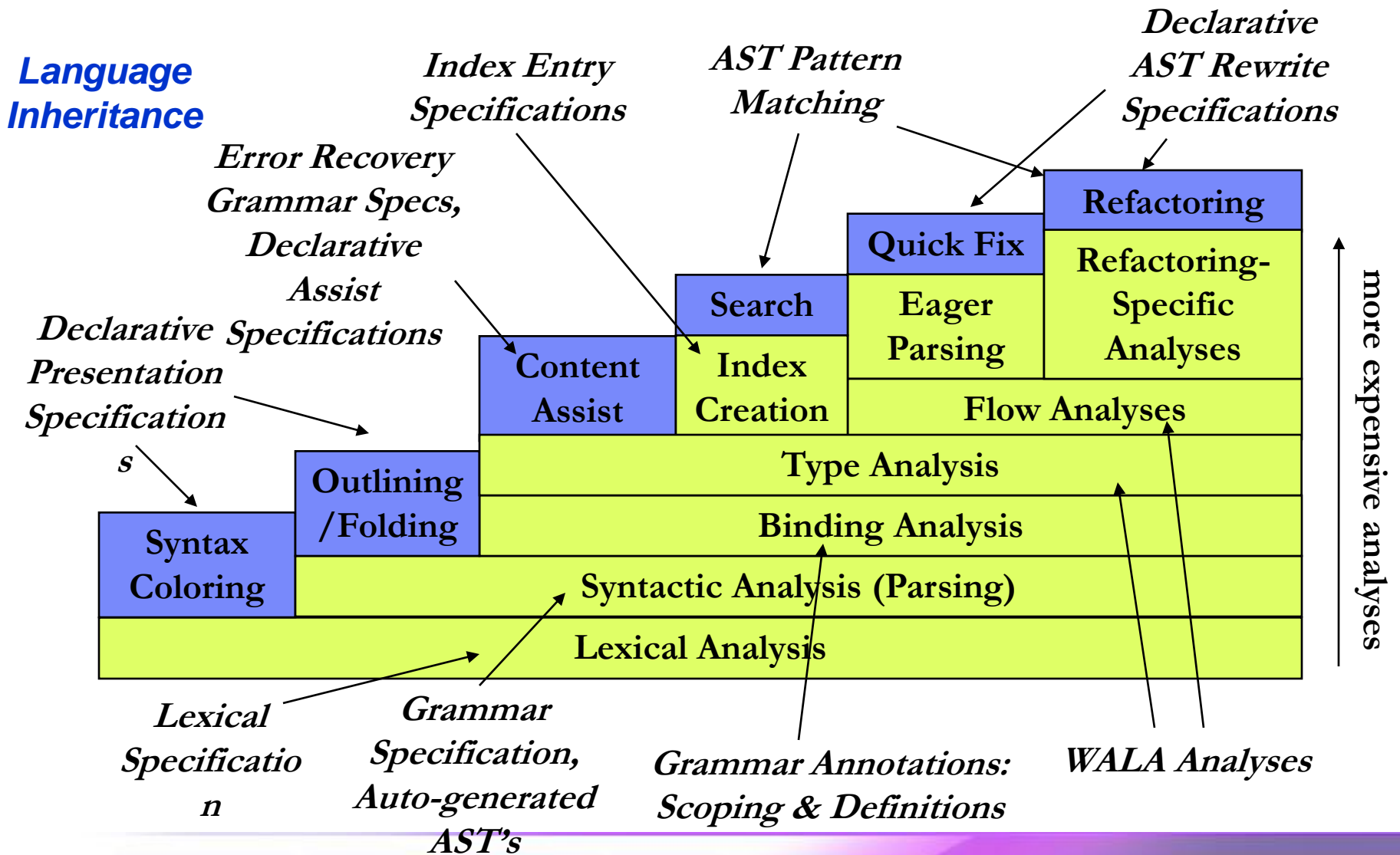
Outline

- Introduction
- SAFARI IDE Development Process Walk-through
- **SAFARI Architecture**
- Status & Future Work

Architecture of SAFARI-based IDEs



SAFARI Support for Language Services & Analysis



SAFARI (LPG) Scanner Specifications & Inheritance


Base Scanner:

```
ident ::= [a-zA-Z][a-zA-Z0-9_]+  
digit ::= [0-9]  
integer ::= digit+  
fixed ::= digit+ \. digit+  
...
```

Derived Scanner:

```
inherit base;  
drop float, fixed;  
  
httpProto ::= 'http'  
mailProto ::= 'mailto'  
ftpProto ::= 'ftp'  
hostname ::= ident ( \. ident )+  
hostIP ::= digit+ ( \. digit+ )+
```

incremental additions
to base scanner spec



new terminals



SAFARI (LPG) Grammar Specifications & Inheritance

Base Grammar:

```
start A;  
A ::= B | C  
B$$$ ::= b | B b  
mods[|mod|] ::= $empty | mods mod  
mod### ::= static | public | ...  
...
```

AST for B is an array of b's

mods is an OR of mod's

AST for mod is an enum

Derived Grammar:

```
inherit base;  
drop C;  
A ::= D  
D ::= ...
```

new production for
existing non-terminal

new non-terminal

SAFARI Presentation Specification

*associate token
types w/ text attributes*

```
package java.ui.views: associate ACT icons TextPresentation KeywordPresentation {  
language X10 extends Java {  
  icon nullableIcon = "icons/nullable.gif"; .name();  
  icon valueIcon    = "icons/value.gif";   fierIcons (Type);  
  
  set modifierIcons (Declaration decl) = {  
    super.modifierIcons (decl).  
    decl.modifiers() => nullableIcon +  
    decl.modifiers() => valueIcon +  
  
    icon staticIcon    = "icons/static.gif";  
    icon finalIcon     = "icons/final.gif";  
    icon publicIcon    = "icons/public.gif";  
    icon privateIcon   = "icons/private.gif";  
    icon protectIcon   = "icons/protected.gif";  
    icon packageIcon   = "icons/package.gif";  
  
    set modifierIcons (Declaration decl) = {  
      decl.modifiers().isStatic() => staticIcon +  
      decl.modifiers().isFinal()  => finalIcon +  
  
      decl.modifiers().isPublic()  => publicIcon +  
      decl.modifiers().isPrivate() => privateIcon +  
      decl.modifiers().isProtected()=>protectIcon +  
      decl.modifiers().isPackage() => packageIcon  
    }  
  }  
};  
  
Outline extends Java {  
  node async;  
}
```

*generate label &
image providers*

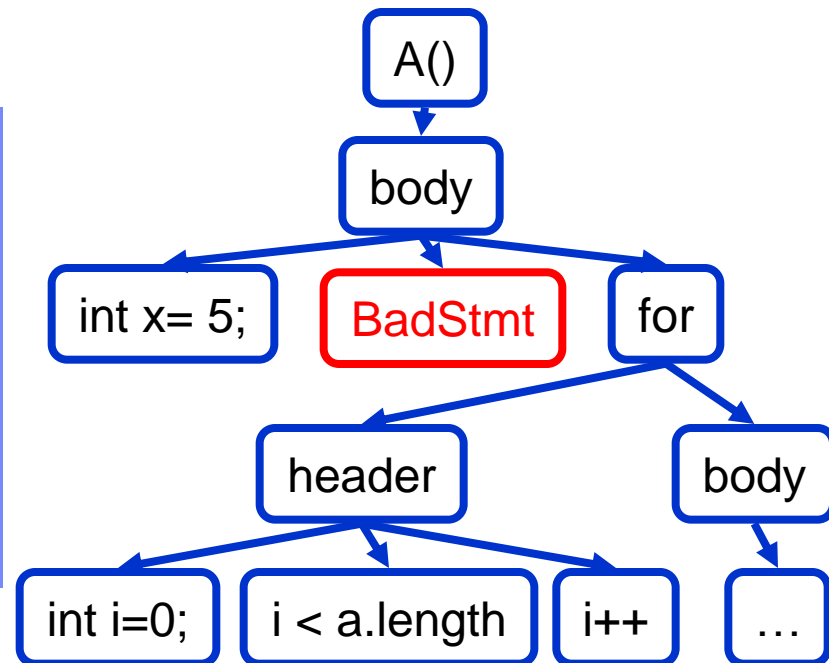
Error Handling

- Errors are the norm! \Rightarrow must not cripple the IDE!
- All analyses must produce something reasonable wherever possible

mangled statement

```
void A() {  
    int x= 5;  
    foo blah;  
    for(int i=0; i < a.length; i++) {  
        int y= a[i] * a[j];  
        x += y;  
    }  
}
```

dangling ref



- SAFARI/LPG: systematic, semi-automatic error recovery for parsing & creating “prosthetic” AST nodes

Code Generation in SAFARI

- Presently: two very simple approaches
- Template-based w/ substitutions for initial skeletons of user-modifiable code
 - substitution targets: package, folder, class names, etc.
 - information taken from several sources
 - wizard dialog fields
 - properties of existing code/meta-data gleaned by reflection
- Java code generation by syntax-directed translation for declarative specifications
 - Domain-specific specification languages designed to interoperate with existing Java/Eclipse APIs
 - N.B.: Some specification languages appear to be purely declarative, but actually extend imperative Java with declarative syntax
 - So: not constrained by power of declarative language; mixture of declarative and imperative specification possible

Code Generation (cont.)

- Two shortcomings of current template-based solution:
 - round-tripping (of course)
 - granularity
- Observation: some service implementations require incremental additions to existing code structures
 - e.g., add registration call to startup code

```
class LPGRefactoringContributor implements IRefactoringContributor {
    public IAction[] getRefactoringActions(UniversalEditor ed) {
        return new IAction[] {
            new FooRefactoringAction(editor),
            new BarRefactoringAction(editor)
        };
    }
}
```

- In fact, all services require incremental additions to existing meta-data
 - i.e., add one or more extension definitions to plugin.xml

Code Generation (cont.)

- Partial solution (under development): structural Java code manipulation via AST's and code templates
 - Specify what's being done (e.g. add method foo() to class Bar)
 - Prevents overwriting of entities irrelevant to transformation
 - Arbitrarily fine-grained (e.g. "add value to array initializer")
 - Builds on SAFARI AST transformation toolkit (declarative rewrite language)
 - Handles most common cases, but not a complete solution
- In fact, similar problems for generating code by syntax translation (except for granularity): solve the same way
- Also: need better reflection mechanisms to expose current state/structure of code + meta-data
 - Eclipse plugins consist of Java code + XML meta-data describing extensions
 - Eclipse Plugin Development Environment (PDE) provides some meta-data reflection, but not particularly convenient
 - Eclipse Java Development Toolkit (JDT) provides considerable help in representing Java code to be manipulated (find precise spot to modify)

Outline

- Introduction
- SAFARI IDE Development Process Walk-through
- SAFARI Architecture
- **Status & Future Work**

Status and Future Work

- Implementation used @ IBM for ongoing IDE & language development
- Installation via IBM-internal Eclipse update site
- Current SAFARI-based IDE implementations:
 - LPG, Java, X10 (IBM Watson Research)
 - JavaScript (IBM Tokyo Research)
- Eclipse.org Technology Project proposal and initial open-source release planned for 2Q07
- Support for
 - Source formatting
 - Language embedding
 - Language inheritance
 - Refactoring and transformation
- Refinements and extensions to static analysis infrastructure

The End

Questions?

- SAFARI Meta-Tooling Platform
 - <http://www.research.ibm.com/safari/>
- LPG (formerly JikesPG) Scanner/Parser Generator *slides online here*
 - <http://sourceforge.net/project/lpg>
- The X10 Concurrent Programming Language
 - <http://x10.sourceforge.net/>
- WALA (formerly DOMO) Static Analysis Framework
 - <http://wala.sourceforge.net/>
- Polyglot Extensible Compiler Framework
 - <http://www.cs.cornell.edu/projects/polyglot/>

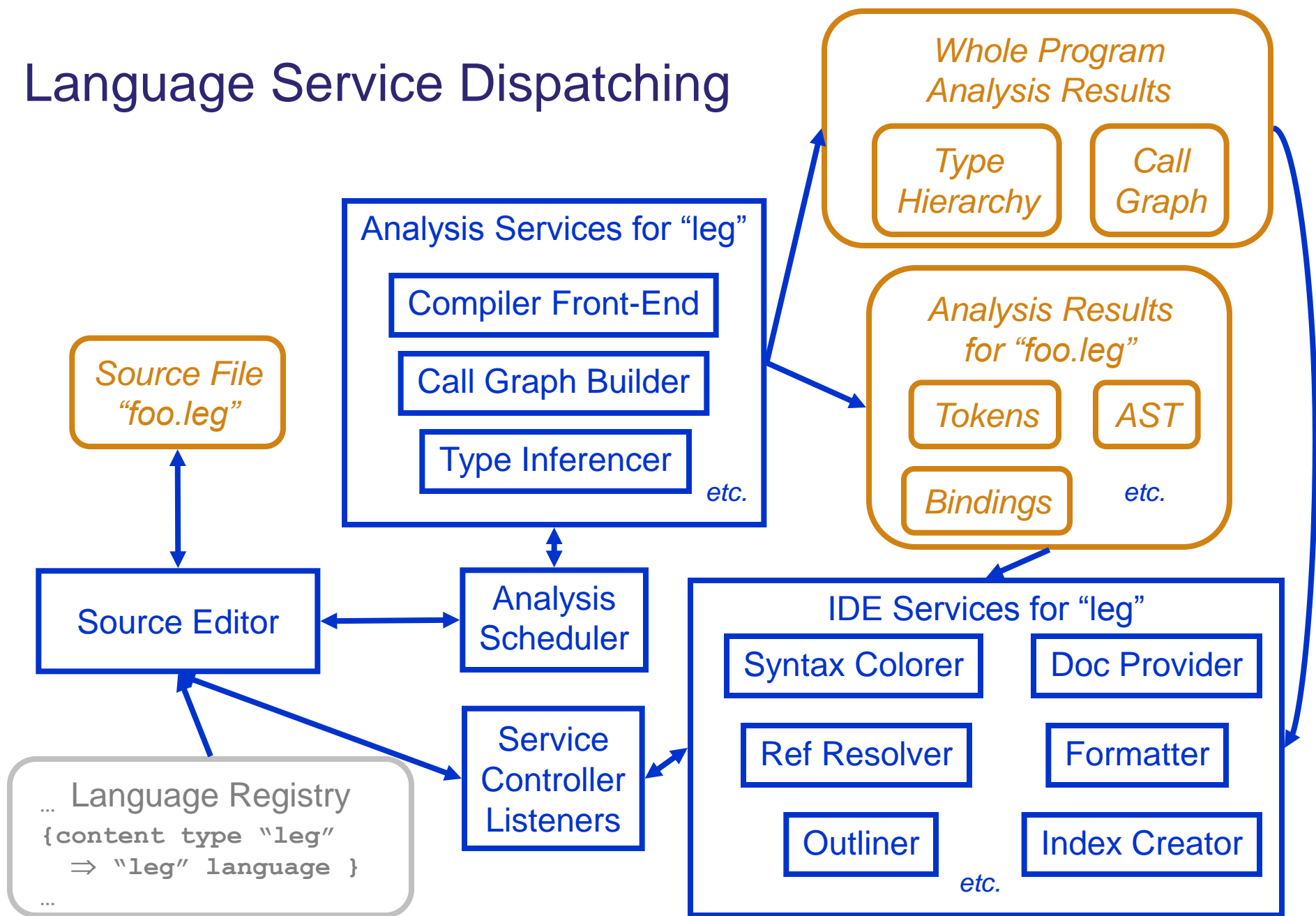
Backup Slides

User-Visible IDE Services

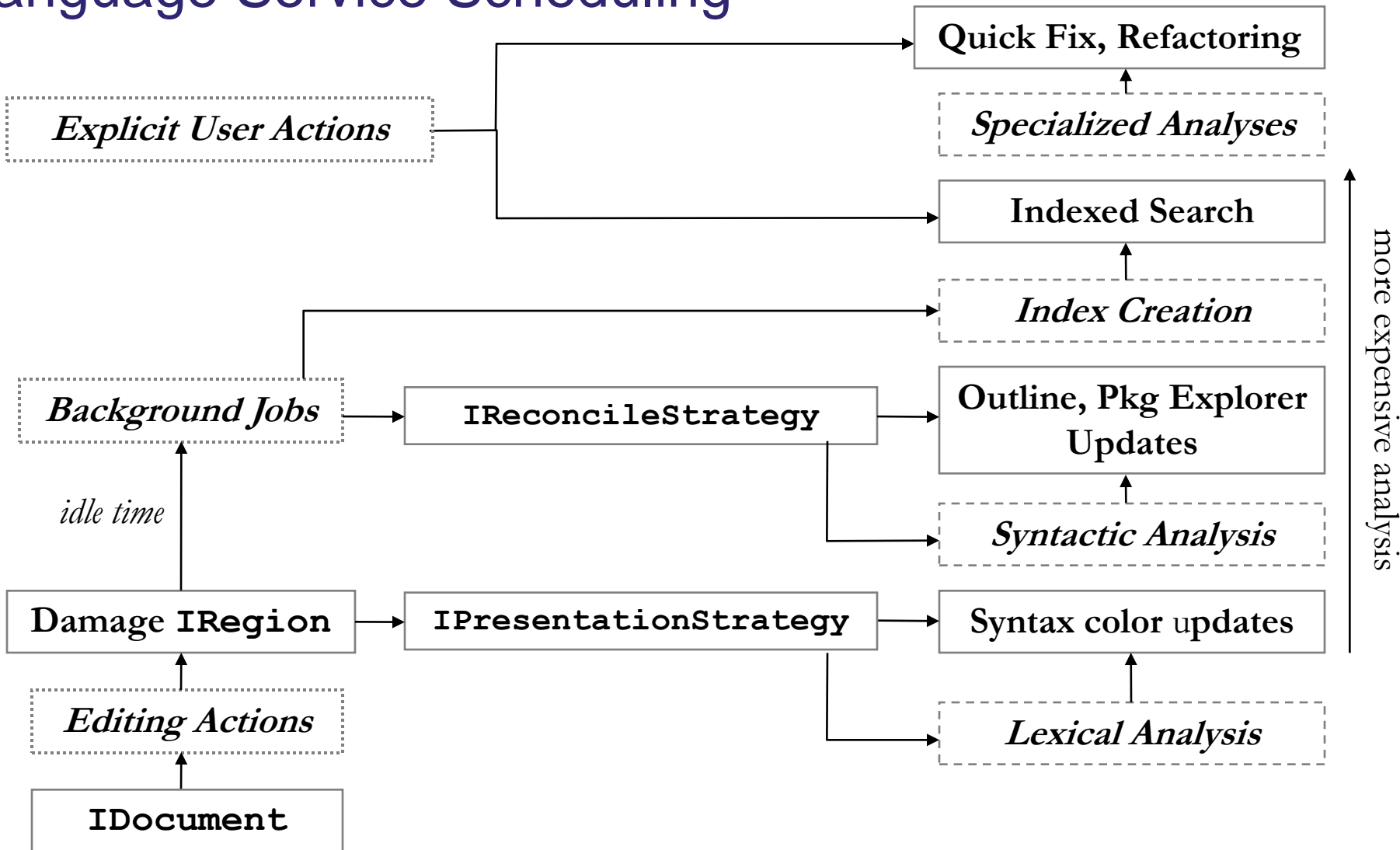
IDE Developer Responsibilities

-
- | | | |
|--------------------------------|---|----------------------------------|
| ▪ Source editor | ← | ▪ Language description |
| ▪ Compiler annotations | ← | ▪ Parser; message handling |
| ▪ Annotation hover | ← | ▪ Token colorer |
| ▪ Resource markers | ← | ▪ Reference resolver |
| ▪ Token coloring | ← | ▪ Documentation provider |
| ▪ Hyperlinked navigation | ← | ▪ Outline content provider |
| ▪ Hover help | ← | ▪ Label provider |
| ▪ Outline view | ← | ▪ Image decorator |
| ▪ Quick outline | ← | ▪ Content proposer |
| ▪ Content assistance | ← | ▪ Index contributor |
| ▪ Indexed search | ← | ▪ Folding updater |
| ▪ Source folding | ← | ▪ Auto-edit strategy |
| ▪ Auto-editing | ← | ▪ Formatter |
| ▪ Formatting | ← | ▪ Dependency scanner |
| ▪ Incremental compilation | ← | ▪ Compiler |
| ▪ Call graph | ← | ▪ Nature enabler |
| ▪ Type hierarchy | ← | ▪ Type analysis, IR construction |
| ▪ Refactoring contributions | ← | ▪ Refactoring contributions |
| ▪ Preference service and pages | ← | ▪ Preference service and pages |

Language Service Dispatching



Language Service Scheduling

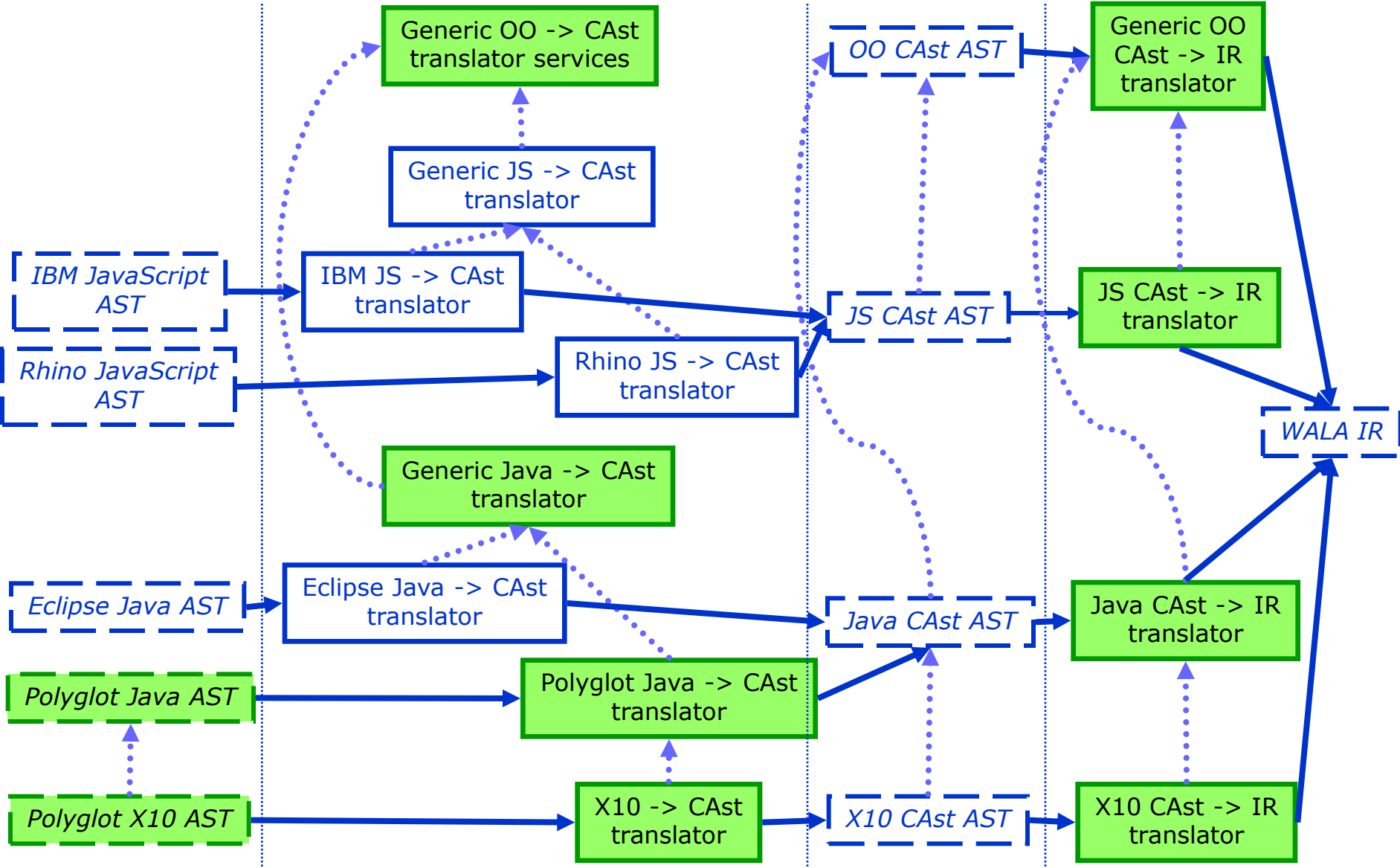


SAFARI Static Analysis Support

- Uses WALA open-source extensible static analysis framework
 - General framework encompassing many classic analyses
 - pointer, type, escape & effects analysis, call graph construction, ...
 - multiple precisions (CHA, RTA, 0-CFA, 1-CFA, etc.)
 - General iterative solver framework for expressing new analyses
 - Robust, highly scalable (capable of analyzing MLOC programs)
 - Handles static and dynamic languages
 - Currently supports Java, JavaScript, PHP, X10
- Adding support to WALA for a new language:
 - Implement translator from source AST's into WALA AST's
 - Define new instruction types for WALA IR as needed (~10 for X10)
 - Implement constraint handlers for new IR instructions to enable existing analyses (e.g. pointer analysis, effects analysis, escape analysis)

WALA IR Generation from Source Code

↑ inheritance
→ data flow



Related Work

- GUIDE (Laffra/IBM Rational):
 - inspiration, foundation for early SAFARI prototype
- Eclipse Language Development Toolkit (LDT):
 - Eclipse Technology Project proposal, vaguely similar goals to SAFARI, withdrawn
- Eclipse Web Standard Tools (WTP):
 - Focus on multi-language support
 - Structured Source Editor (SSE) offers similar editing infrastructure
 - API's, no meta-tooling (?)
 - May be possible to build parts of SAFARI on top of WTP/SSE (TBD)
- Eclipse Dynamic Languages Toolkit (Technology Project)
 - Focuses on dynamic languages
 - Uses single generic language model for program representation; SAFARI permits custom ASTs, and can use your existing compiler front-end as is
 - Not based on meta-tooling
 - Aims for language interoperability, SAFARI for IDE and language extensibility