A Program Analysis Infrastructure for IMP
(a sketch of one)

Bob Fuhrer & Jurgen Vinju

IBM, Hawthorne

March 5th, 2007

Overview

1. Introduction
2. Vocabulary and Architecture
3. High level specification
4. Demo

This is all ongoing and future work

Source code analysis in IDE's

- Integrated Development Environments (IDE's):
  - help understanding code.
  - help manipulating code.
- That's why IDE's contain many analyses;
  - "Simple" — like parsing and name resolution.
  - "Advanced" — like call graphs and data dependencies.
  - "Hairy" — like dead code analysis and clone detection.
  - Static, dynamic, and hybrid
  - Cross programming language/formalism (Makefiles, Ant, CVS)
- Analysis results are used implicitly in the IDE:
  - Code search, Outlining, many forms of hyperlinking, pre-conditions of refactorings, source metrics, etc.

IDE Meta-tooling Platform (IMP)

- Help the IDE builder by providing framework infra-structure and code generators.
- What about source code analysis?
- The goal is to help the IDE builder to:
  - add analyses to an Eclipse-based IDE for any language.
  - analyze the code using different kinds of analysis methods
  - focus on the analysis itself, and not the boilerplate of integration.
  - express analyses in DSL's as well as directly in Java.
- Requirements: fast, small, composable, decoupled

Experience with The Meta-Environment

- An environment for constructing IDE's and meta programs
- ASF+SDF: extraction of facts and transformation
- RScript: relational calculus on facts
- RStores: common data-structure for fact representation
- Fact browser: pluggable visualization framework
- Applied and tested in an academic environment:
  - Java dead code detection (static and dynamic analysis)
  - C comment vs. code consistency checker
  - C call graph extraction, data dependencies
  - Grammar analyses for SDF
  - Connected 3 different kinds of viz. toolkits
  - Connected 2 different kinds of extracting front-ends
- Issues: size, speed and limited visibility.

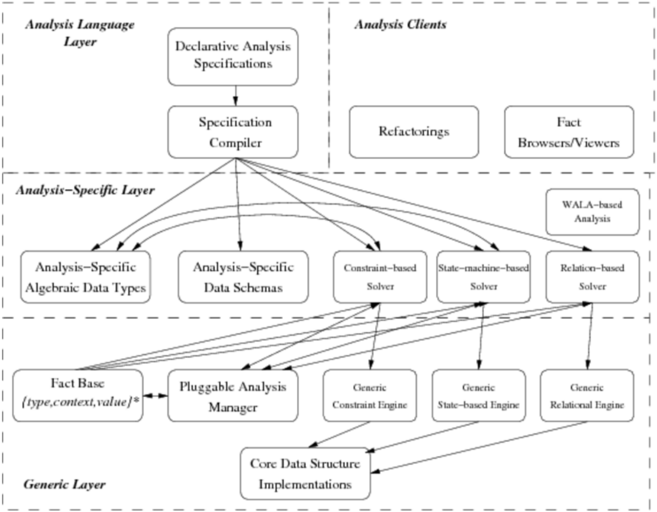
Source code representations

1. "Extract" basic undeniable facts from source code
2. "Analyze" the facts and elaborate on them
3. "Visualize" the resulting data in the IDE

Architecture and design

- Typed relations are the central data-structure
- Fact Generators produce facts and store them in the Fact Base
- Fact Generators may also consume facts from the Fact Base
- Visualizations consume facts from the Fact Base
- Loosely coupled, the Fact Base hides by who, how and when a fact is produced.

A nice picture



Type system for relations

- Goal 1: prevent/diagnose common programming and composition errors.
- Goal 2: allow automated discovery and composition of analyses and visualizations.
- A simple first-order type system for relations, sets, maps, and atomic values.
- Sub-typing with covariance ($t_1 < t_2 \Rightarrow set[t1] < set[t2]$)
- Mantra: maximize flexibility (reuse), without loosing safety.

- DSL's for analysis (and transformation)
- Compile down to Java that uses the FactBase and the Relations
- DSL's share the type system of relations
- Current project: Relational Calculus/Algebraic Data Types
  - Operators on sets and relations; comprehensions; fixed point operators
  - Typed term constructors and normalizing rewrite rules

Example — Infer generic type arguments

- Problem: transform to Java code that uses generics
  - The current Java program represents a set of constraints on the types of variables
  - First find these constraints using Java static semantics
  - Then find least general type parameters for variables, satisfying the constraints
- The type universe is very big
- While solving the constraint system, intermediate results (sets of types) can not be kept in main memory

ADT's for Infer Generic Type Arguments

```
atype Term = QualifiedTypeName(str)
| Expression(Expr)
| Method(Class, Method)
| Field(Class, Field)
| Param(Method, int)
| Decl(Method)
| Decl(Field)

type Type = str

atype TypeSet = set[Type]
| Union(TypeSet,TypeSet)
| Intersection(TypeSet,TypeSet)
| Subtypes(TypeSet)
| Supertypes(TypeSet)
| Subtypes(Type)
| Supertypes(Type)
```

Rules for Infer Generic Type Arguments

```
rules
Subtypes(root) => Universe
Subtypes(Universe) => Universe
Subtypes(Subtypes(x)) => Subtypes(x)

Intersection(EmptySet,_) => EmptySet
Intersection(Universe,x) => x
Intersection(x,Universe) => x
else: Intersection(set[Type] s1, set[Type] s2) => s1 intersect s2
...
```

Fixed point equations for Infer Generic Type Arguments

```
TypeSet getInitialEstimate(Term t) =
    case t = QualifiedTypeName(name): SingletonType(name)
    else: Universe

analysis typeInference {
    nodes {
        Term t := getInitialEstimate(t);
    }

    constraints {
        rel[Term lhs, Term rhs] simpleConstraints =
               equalConstraints union inv(equalConstraints) union subTypeConstraints;

        satisfy (simpleConstraints) {
            lhsEst := estimates[lhs];
            rhsEst := estimates[rhs];

            estimates'[lhs] := Intersection(lhsEst, Subtypes(rhsEst));
            estimates'[rhs] := Intersection(rhsEst, Supertypes(lhsEst));

            error if estimates'[lhs] is EmptySet
            error if estimates'[rhs] is EmptySet
        }
    }
}
```

What we have now

- A type system
- Full (but naive) implementation of relations
- Analysis Manager
- Fact Base
- Simple Fact Browser for debugging purposes
- Simple extractors for LPG and Java