Rscript: a Relational Approach to Program and System Understanding

Paul Klint
Structure of Presentation

- Background and context
- About program understanding
- Roadmap: Rscript
# Background

**Application areas**

- Domain-specific languages
  - **Software renovation**
    - System understanding
    - System transformation

**Technology**

- ASF+SDF Meta-Environment
- Generalized LR parsing
- (Compiled) term rewriting
- ToolBus coordination architecture
- Code Generators

**Foundations**

- Formal languages
- Term rewriting
- Relational calculus
- Process Algebra Module algebra

This talk focuses on the **Relational Approach to Program and System Understanding**.
Compilation is a mature area

- Some new developments
  - just-in-time compilation
  - energy-aware code generation
- Many research results are not yet used widely
  - interprocedural pointer analysis
  - slicing
- Why don't we just apply all these techniques to understanding and restructuring?
Compilation is a mature area

• ... of course, we do just that, but ...
• there is a mismatch between
  – standard compilation techniques and
  – the needs for understanding and restructuring
Compilation is ...

- A well-defined process with well-defined input, output and constraints
- Input: source program in a fixed language with well-defined syntax and semantics
- Output: a fixed target language with well-defined syntax and semantics
- Constraints are known (correctness, performance)
- A batch-like process
Compilation is ...

Single, well defined, source

Single, well defined, target

Source

A batch-like process with clear constraints

Target
Understanding is ...

- An exploration process with as input
  - system artifacts (source, documentation, tests, ...)
  - implicit knowledge of its designers or maintainers
- There is no clear target language
- An interactive process:
  - Extract elementary facts
  - Abstract to get derived facts needed for analysis
  - View derived facts through visualization or browsing
Examples of understanding problems

- Which programs call each others?
- Which programs use which databases?
- If we change this database record, which programs are affected?
- Which programs are more complex than others?
- How much code clones exist in the code?
Examples of the results of understanding

- Textual reports indicating properties of system parts (complexity, use of certain utilities, ...)
- Same, but in hyperlinked format
- Graphs (call graphs, use def graphs for databases)
- More sophisticated visualizations
Other aspects of Understanding

- Systems consist of several source languages
- Analysis techniques over multiple language => a language-independent analysis framework is needed
- A very close link to the source text is needed
Related approaches

• Generic dataflow frameworks exist but are not used widely

• Relations have been used for querying of software (Rigi, GROK, RPA, ...)
  – All based on untyped, binary, relation algebra
  – Mostly used for architectural, coarse grain, queries
Relation-based analysis

- What happens if we use relations for fine grain software analysis (ex: find uninitialized variables)?
- What happens if we use a relational calculus (as opposed to the relational algebra approaches)?
- What happens if we use term rewriting as basic computational mechanism?
  - relations can represent graphs in the rewriting world
- Could yield a unifying framework for analysis and transformation
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Rscript in a Nutshell

- Basic types: `bool`, `int`, `str`, `loc` (text location in specific file with comparison operators)
- Sets, relations and associated operations (domain, range, inverse, projection, ...)
- Comprehensions
- User-defined types
- Fully typed
- Functions and sets of equations over the above
Rscript: examples

- **Set**: \{3, 5, 7\}  
  - type: set[int]
- **Set**: \{"y", "x","z"\}  
  - type: set[str]
- **Relation**: \{<"y",3>, <"x",3>, <"z", 5>\}  
  - type: rel[str,int]
Rscript: examples

• `rel[str, int] U = {"y", 3}, {"x", 3}, {"z", 5}`

• `int Usize = #U`
  - 3

• `rel[int, str] Uinv = inv(U)`
  - `{<3, "y">, <3, "x">, <5, "z">}`

• `set[str] Udom = domain(U)`
  - {"y", "x", "z"}
Comprehensions

- Comprehensions: \( \{ \text{Exp} \mid \text{Gen1, Gen2, ...} \} \)
  - A generator is an enumerator or a test
  - Enumerators: \( V : \text{SetExp} \) or \( <V1,V2> : \text{RelExp} \)
  - Tests: any predicate
  - consider all combinations of values in \( \text{Gen1, Gen2,...} \)
  - if some \( \text{Gen}_i \) is false, reject that combination
  - compute \( \text{Exp} \) for all legal combinations
Comprehensions

- \{X \mid \text{int } X : \{1,2,3,4,5\}\}
  - yields \{1,2,3,4,5\}

- \{X \mid \text{int } X : \{1,2,3,4,5\}, X > 3\}
  - yields \{4,5\}

- \{<Y, X> \mid <\text{int } X, \text{int } Y> : \{<1,10>,<2,20>\}\}
  - yields \{<10,1>,<20,2>\}
Functions

- \text{rel}[\text{int}, \text{int}] \ \text{inv}(\text{rel}[\text{int}, \text{int}] \ R) =
  \{ \langle Y, X \rangle \mid \langle \text{int} \ X, \text{int} \ Y \rangle : R \} \\
  \text{inv}({\langle 1,10\rangle, \langle 2,20\rangle}) \ \text{yields} \ \{\langle 10,1\rangle,\langle 20,2\rangle\}

- \text{rel}[^{\&B}, ^{\&A}] \ \text{inv}(\text{rel}[^{\&A}, ^{\&B}] \ R) =
  \{ \langle Y, X \rangle \mid \langle ^{\&A} \ X, ^{\&B} \ Y \rangle : R \} \\
  \text{inv}({\langle 1,"a"\rangle, \langle 2,"b"\rangle}) \ \text{yields} \ \{\langle "a",1\rangle,\langle "b",2\rangle\}

^{\&A}, ^{\&B} \ \text{indicate any type and are used to define polymorphic functions}
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Roadmap

• Rscript in a nutshell
• Example 1: call graph analysis
• Example 2: component structure
• Example 3: Java analysis
• Example 4: a toy language
• A visualization experiment
Analyzing the call structure of an application

rel[\text{str}, \text{str}] \text{ calls} = \{<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">, <"d","e">, <"f", "e">, <"f", "g">, <"g", "e">\}
Some questions

- How many calls are there?
  - `int ncalls = # calls`
  - 8

- How many procedures are there?
  - `int nprocs = # carrier(calls)`
  - 7
Some questions

- What are the entry points?
  - `set[str] entryPoints = top(calls)`
  - `{“a”, “f”}`
- What are the leaves?
  - `set[str] bottomCalls = bottom(calls)`
  - `{“c”, “e”}`
Intermezzo: Top

• The **roots** of a relation viewed as a graph

• \( \text{top}\{\{1,2\},\{1,3\},\{2,4\},\{3,4\}\} \) yields \{1\}

• **Consists** of all elements that occur on the **lhs but not on the rhs** of a tuple

• \( \text{set[\&T] top(rel[\&T, \&T] R) = domain(R) \setminus range(R)} \)
Intermezzo: Bottom

- The leaves of a relation viewed as a graph
- \( \text{bottom}(\{(1,2),(1,3),(2,4),(3,4)\}) \) yields \( \{4\} \)
- Consists of all elements that occur on the rhs but not on the lhs of a tuple
- \( \text{set[&T]} \text{ bottom}(\text{rel[&T, &T]} R) = \text{range}(R) \setminus \text{domain}(R) \)
Some questions

- What are the indirect calls between procedures?
  - rel[str, str] closureCalls = calls+
  - \{<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">,
    <"d", "e">, <"f", "e">, <"f", "g">, <"g", "e">, <"a",
    "c">, <"a", "d">, <"b", "e">, <"a", "e">\}

- What are the calls from entry point a?
  - set[str] calledFromA = closureCalls["a"]
  - \{"b", "c", "d", "e"\}
Intermezzo: right image

- Right-image of a relation: all elements that have a given value as left element (resembles array access)
- Notation: relation followed by $[\text{Value}]$
- Ex. $\text{Rel} = \{<1,10>,<2,20>,<1,11>,<3,30>,<2,21>\}$
- $\text{Rel}[1]$ yields $\{10,11\}$
- $\text{Rel}[\{1,2\}]$ yields $\{10,11,20,21\}$
Intermezzo: left image

- Left-image of a relation: all elements that have a given value as right element

- Notation: relation followed by \([-,\text{Value}]\)

- Ex. \(\text{Rel} = \{<1,10>,<2,20>,<1,11>,<3,30>,<2,21>\}\)

- \(\text{Rel}[-,10] \) yields \{1\}

- \(\text{Rel}[-,{10,20}] \) yields \{1,2\}
Some questions

- What are the calls to procedure e?
  - `set[str] callsToE = closureCalls[-,"e"]`
  - `{"a", "b", "d", "f", "g"}`
Some questions

- What are the calls from entry point \texttt{f}?
  - \texttt{set[str] calledFromF = closureCalls["f"]}
  - \{"e", "g"\}

- What are the common procedures?
  - \texttt{set[str] commonProcs = calledFromA inter calledFromF}
  - \{"e"\}

Intersection
Rscript: a Relational Approach to Program and System Understanding

Running Rscript using rscript-meta
Script -> Open...
File calls has been opened

Right click -> Edit script
Editing `calls.rscript`

type proc = str

rel[proc, proc] Calls = {"a", "b"}, {"b", "c"},
{"b", "d"}, {"d", "c"},
{"d", "e"}, {"f", "e"},
{"f", "g"}, {"g", "e"}

int nCalls = # Calls

set[proc] procs = carrier(Calls)

int nprocs = # carrier(Calls)

set[str] dCalls = domain(Calls)

set[str] rCalls = range(Calls)

set[proc] entryPoints = top(Calls)

set[proc] bottomCalls = bottom(Calls)

rel[proc, proc] closureCalls = Calls+
Making errors ...
Script -> Run

```
Rscript: a Relational Approach to Program and System Understanding

$ type proc = str

$ rel[proc, proc] Calls = {"a", "b", "c"},
                          {
                          "b", "d", "e"},
                          {
                          "d", "e"},
                          {
                          "f", "e"},
                          {
                          "f", "g"},
                          {
                          "g", "e"} }

$ int nCalls = # Calls

$ set[proc] procs = carrier(Calls)

$ int nprocs = # carrier(Calls)

$ set[str] dCalls = domain(Calls)

$ set[str] rCalls = range(Calls)

$ set[proc] entryPoints = top(Calls)

$ set[proc] bottomCalls = bottom(Calls)

$ rel[proc,proc] closureCalls = Calls+
```
Unfolding the rstore ...
Unfolding closureCalls
closureCalls as Text
closureCalls as Table
closureCalls as Graph
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- **Example 2: component structure**
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Component Structure of Application

- Suppose, we know:
  - the call relation between procedures (Calls)
  - the component of each procedure (PartOf)

- Question:
  - Can we lift the relation between procedures to a relation between components (ComponentCalls)?

- This is useful for checking that real code conforms to architectural constraints
type proc = str
type comp = str
rel[proc,proc] Calls = {<"main", "a">, <"main", "b">, <"a", "b">, <"a", "c">, <"a", "d">, <"b", "d">}
set[comp] Components = {"Appl", "DB", "Lib"}

rel[proc, comp] PartOf = 
    {"main", "Appl"}, "a", "Appl"}, "b", "DB"}, 
        "c", "Lib"}, "d", "Lib"}
rel[comp,comp] lift(rel[proc,proc] aCalls, rel[proc,comp] aPartOf) =
\{ <C1, C2> | <proc P1, proc P2> : aCalls,

rel[comp,comp] ComponentCalls = lift(Calls2, PartOf)

Result: {"DB", "Lib"}, {"Appl", "Lib"}, {"Appl", "DB"}, {"Appl", "Appl"}
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Cyclic Dependencies

- A class uses (directly or indirectly) itself
- Use = methods calls, inheritance, containment

```java
class ContainedClass {}
class SuperClass {}
class SubClass extends SuperClass {
    ContainedClass C;
}
```

Motivation: cyclic class dependencies are difficult to understand/maintain
Cyclic Dependencies: Examples

```java
class A {  B B1; ... }
class B extends A { ... }

class A {  C C1; ... }
class B extends A{ ... }
class C { B B1; ...}
```
Java analysis: classes in cycles

- Assume the following extracted information:
  - rel[str,str] CALL
    - method call from first class to the second
  - rel[str,str] INHERITANCE
    - extends and implements
  - rel[str,str] CONTAINMENT
    - attribute of first class is of the type of the second class
- Question: which classes occur in a cyclic dependency?
Java analysis: cycles in classes

- Define the USE relation between two classes:
  - $\text{rel}[	ext{str,str}] \text{ USE} = \text{CALL union CONTAINMENT union INHERITANCE}$
  - $\text{set}[	ext{str}] \text{ ClassesInCycle} = \{C1 | <\text{str }C1, \text{str }C2>: \text{USE+}, C1 == C2\}$
- In this way we get a set of classes that occur in a cyclic dependency, but ...
- ... which classes are in the cycle?
Java analysis: cyclic classes

rel[str,str] USE = CALL union CONTAINMENT
union INHERITANCE

set↑[str] CLASSES = carrier(USE)

rel[str,str] USETRANS = USE+

rel[str,set[str]] = {<C, USETRANS[C]> |
    str C : CLASSES,
    <C, C> in USETRANS}

Each cyclic class is associated with a set of classes that form a cycle
Applications of this approach

- Search for “similar” classes
- Search for design patterns (as characterized by specific relations between the classes in the pattern)
- …
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A vizualization experiment
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Toy program

begin declare x : natural, y : natural, z : natural;
    x := 3;
    if 3 then
        z := y + x
    else
        x := 4
    fi
    y := z
end

y is undefined

z may be undefined
Toy program

begin declare x : natural, y : natural, 
z : natural;

[1] x := 3;

if [2] 3 then
   [3] z := y + x
else
   [4] x := 4
fi

[5] y := z
end

rel[int,str] DEFS = \{<1, "x">, <3, "z">, <4, "x">, <5, "y">\}

rel[int,str] USES = \{<3, "y">, <3, "x">, <5, "z">\}

rel[int,int] PRED = \{<0, 1>, <1, 2>, <2, 3>, <2, 4>, <3, 5>, <4, 5>\}
Finding uninitialized variables

Start of program

Def 1 of x

Def 2 of x

Use of x

Along these path, we encounter a definition

Along this path, we can reach a use without passing a definition

Value of x may be undefined here
Intermezzo: reachX

- Reachability with exclusion of certain elements

- set[&T] reachX(
  set[&T] Start,
  set[&T] Excl,
  rel[&T,&T] Rel)

- reachX({1}, {2}, {{1,2},{1,3},{2,4},{3,4}})
  yields {<3,4>}

![Graph diagram]
The undefined query

\[
\text{rel}[\text{int}, \text{str}] \text{ DEF}S = \ldots
\]
\[
\text{rel}[\text{int}, \text{str}] \text{ USES} = \ldots
\]
\[
\text{rel}[\text{int}, \text{int}] \text{ PRED} = \ldots
\]
\[
\text{rel}[\text{int}, \text{str}] \text{ UNINIT} = \\
\{ <\text{int}\ N, \text{str}\ V> : \text{USES}, \ N \text{ in reachX}(\{0\}, \text{DEF}S[-,V],\text{PRED}) \}
\]

There is a path from the root to \( N \): \( V \) is not initialized

Start from the root

Exclude all definitions of \( V \)

Use the \text{PRED} relation

Reach exclude
Applying the undefined query

```plaintext
begin declare x : natural, y : natural, z : natural;

[1] x := 3;
       if[2] 3 then
       [3]  z := y + x

else
    [4]  x := 4
fi

[5]  y := z

end

y is undefined

z may be undefined

Result:

{<5,"z">, <3,"y">}
```
Some Questions

- There are several additional questions:
  - In the example so far we have worked with statement numbers but how do we make a connection with the source text? (Discussed now)
  - How do we extract relations like PRED and USE from the source text? (Discussed later)
Use locations to connect with the source text

Variable occurrence in a statement

rel[int,str] DEFS = ...
rel[int,str] USES = ...
rel[int,int] PRED = ...

Use location instead of number

rel[loc,str] DEFS
rel[loc,str] USES
rel[loc,loc] PRED
rel[str, loc] OCCURS
Example Rstore

rstore(
  <PRED, rel[loc,loc],
    {<area-in-file("/home/paulk/.../example.pico", area(4, 2, 4, 8, 84, 6)),
     area-in-file("/home/paulk/.../example.pico", area(5, 2, 5, 8, 94, 6))>,
     <area-in-file("/home/paulk/.../example.pico", area(5, 2, 5, 8, 94, 6)),
     area-in-file("/home/paulk/.../example.pico", area(6, 2, 10, 4, 104, 56))>,
     ... }>,

  <DEFS, {
    <OCCURS, rel[str,loc],
      {"y", area-in-file("/home/paulk/.../example.pico", area(11, 2, 11, 3, 164, 1))>,
       "z", area-in-file("/home/paulk/.../example.pico", area(11, 7, 11, 8, 169, 1))>,
      ... }
  })


Extracting Facts

• Goal: extract facts from source code and use as input for queries
• How should fact extraction be organized?
• How to write a fact extractor?
Workflow Fact Extraction

Grammar
- Obtain sources of SUI
- Obtain grammar for source language of SUI
- Validate grammar
- Improve

Facts
- Determine needed facts
- Obtain fact extractor
- Validate extracted facts
- Improve

Queries
- Write queries
- Execute queries
- Validate answers
- Use answers
- Improve

SUI = System Under Investigation
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Roadmap

- Rscript in a nutshell
- Example 1: call graph analysis
- Example 2: component structure
- Example 3: Java analysis
- Example 4: a toy language
- A visualization experiment
Issues in Program Visualization

- Small graphs are nice, large graphs are a disaster

(Courtesy: Arie van Deursen)
Issues in Program Visualization

- How to display information related to source text?
- Approach (Steven Eick): use a pixel-based image of the source text
- Over 100,000 LOC on one screen!
- Experiment: visualize an Rstore for JHotDRaw (15,000 LOC)

Extraction by Hayco de Jong and Taeke Kooiker (using ASF+SDF)
Rscript: a Relational Approach to Program and System Understanding

Relations

Categories of names

Rectangle per file
Hovering over file shows full name
Selecting a category displays all names in that category.
Uses of class URL are here

Click here for textual view ...

Select class URL
private void readFromStorableInput(String filename) {
    try {
        URL url = new URL(getCodeBase(), filename);
        InputStream stream = url.openStream();
        StorableInput input = new StorableInput(stream);
        Drawing.release();
    }
}

private void readFromObjectInput(String filename) {
    try {
        URL url = new URL(getCodeBase(), filename);
        InputStream stream = url.openStream();
        ObjectInput input = new ObjectInputStream(stream);
        Drawing.release();
        Drawing = (Drawing)input.readObject();
        View.setDrawing(Drawing);
    }
}

private String guessType(String file) {
    if (file.endsWith(".draw"))
        return "storable";
    if (file.endsWith(".ser"))
        return "serialized";
}
Wrap up: Rscript

- A simple, language-independent, relational calculus
- Fully typed
- Equation solver (=> dataflow equations)
- Areas allow close link with source text
- Implementation: ASF+SDF
- **IDE:** `rscript-meta`
  - an instance of The Meta-Environment
Wrap up: Rscript

- Calls analysis
- Lifting of procedure calls to component relations
- Uninitialized/unused variables
- McCabe & friends
- Clones in C code

- Dataflow analysis
  - reaching definitions
  - live variables
- Program slicing
- Java & ToolBus analysis
- Feature Descriptions/package dependencies
Wrap up: visualization

- A lot of work to do but promising start
- Alternative pixel representations?
- Treemaps for directory structure of files?
- Colormaps for displaying metrics?
- Implementation: Tcl/Tk but may change to Swing
- Some simple visualizations are included in rscript-meta
Further reading

- P. Klint, How understanding and restructuring differ from compiling: a rewriting approach, IWPC03
- P. Klint, A tutorial introduction to Rscript on www.meta-environment.org
- www.cwi.nl/~paulk/publications/all.html