Using RSCRIPT for Software Analysis

Paul Klint

Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 AB Amsterdam, The Netherlands
`http://www.cwi.nl/~paulk`

Abstract

RSCRIPT *is a concept language that explores the design space of relation-based languages for software analysis. We briefly sketch the* RSCRIPT *language by way of a standard example, summarize our experience, and point at future developments.*

1. Introduction

Car designers make concept cars to illustrate the direction in which their designs are evolving and to provoke responses from future customers. In this same sense, RSCRIPT [4, 5] is a concept language that explores the design space of relation-based languages for software analysis. In this short paper we will sketch the RSCRIPT language by way of a standard example (Section 2), summarize our experience (Section 3), and point at future developments (Section 4). Before embarking on this, we sketch the context of this research (Section 1.1) and give a quick introduction to RSCRIPT (Section 1.2). For related work, we refer the reader to [4, 5].

1.1. Context

The overall context of our design is software refactoring and transformation as occurs during software improvement and renovation processes, see for instance [9]. We focus here on the following steps in such a process:

1. Extract facts from the source code. Examples are call relations, control flow, and use/def relations of variables. RSCRIPT does not consider fact extraction *per se* so we assume that facts have been extracted from the software by some other tool.

2. After the extraction phase, we try to understand the extracted facts by writing queries to explore their properties. For instance, we may want to know *how many calls* there are, or *how many procedures*. We may also want to enrich these facts, for instance, by computing who calls who in more than one step. Computing dominator trees, dataflow properties or program slices are other examples.

3. Finally, we produce a simple textual report giving answers to the questions we are interested in or sufficient relational data to drive a visualisation tool.

Steps 2 is the primary domain of RSCRIPT.

1.2. RSCRIPT at a glance

RSCRIPT is a typed language based on relational calculus. It has some standard elementary datatypes (booleans, integers, strings) and a non-standard one: source code locations that contain a file name and text coordinates to uniquely describe a source text fragment. As composite datatypes RSCRIPT provides sets, tuples (with optionally named elements), and relations. Functions may have type parameters to make them more generic and reusable. A comprehensive set of operators and library functions is available on the built-in datatypes ranging from the standard set operations and subset generation to the manipulation of relations by taking transitive closure, inversion, domain and range restrictions and the like. The library also provide various functions (e.g., conditional reachability) that enable the manipulation of relations as graphs.

Suppose the following facts have been extracted from given source code and are represented by the relation `Calls`:

```
type proc = str
rel[proc , proc] Calls = {<"a", "b">,
  <"b", "c">, <"b", "d">, <"d", "c">,
  <"d","e">, <"f", "e">, <"f", "g">,
  <"g", "e">}.
```

The user-defined type `proc` is an abbreviation for strings and improves both readability and modifiability of the RSCRIPT code. Each tuple represents a call between two procedures.

The *top* of a relation contains those left-hand sides of tuples in a relation that do not occur in any right-hand side.

When a relation is viewed as a graph, its top corresponds to the root nodes of that graph. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

```
set[proc] entryPoints = top(Calls)
```

In this case, `entryPoints` is equal to `{"a", "f"}`. In other words, procedures `"a"` and `"f"` are the entry points of this application.

We can also determine the *indirect calls* between procedures, by taking the transitive closure of the `Calls` relation:

```
rel[proc, proc] closureCalls = Calls+
```

We know now the entry points for this application (`"a"` and `"f"`) and the indirect call relations. Combining this information, we can determine which procedures are called from each entry point. This is done by taking the *right image* of `closureCalls`. The right image operator determines all right-hand sides of tuples that have a given value as left-hand side:

```
set[proc] calledFromA = closureCalls["a"]
```

yields `{"b", "c", "d", "e"}` and

```
set[proc] calledFromF = closureCalls["f"]
```

yields `{"e", "g"}`. Applying this simple computation to a realistic call graph makes a good case for the expressive power and conciseness achieved in this description.

2. Reaching definitions

We illustrate the calculation of reaching definitions using the text book example in Figure 1 which was inspired by [1, Example 10.15].

We assume the following basic relations PRED (the predecessor relation between program points), DEFS (program points where the value of a variable is defined) and USES (uses of variables) about the program:

```
type stat = int
type var = str
rel[stat,stat] PRED = { <1,2>, <2,3>,
  <3,4>, <4,5>, <5,6>, <5,7>, <6,7>,<7,4>}
rel[stat, var] DEFS = { <1, "i">, <2, "j">,
  <3, "a">, <4, "i">, <5, "j">, <6, "a">,
  <7, "i">}
rel[stat, var] USES = { <1, "m">, <2, "n">,
  <3, "u1">,<4, "i">, <5, "j">, <6, "u2">,
  <7, "u3">}
```

For convenience, we introduce a notion `def` that describes that a certain statement defines some variable and we revamp the basic relations into a more convenient format using this new type:

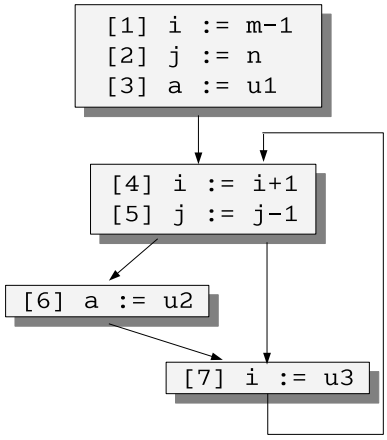

Figure 1. Flow graph for reaching definitions

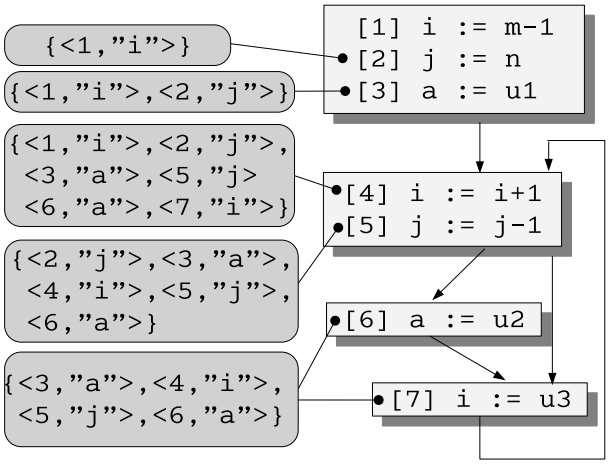

Figure 2. Reaching definitions for example

```
type def  = <stat theStat, var theVar>

rel[stat, def] DEF =
  {<S, <S, V>> | <stat S, var V> : DEFS}

rel[stat, def] USE =
  {<S, <S, V>> | <stat S, var V> : USES}
```

The new DEF relation gets as value:

```
{<1, <1, "i">>, <2, <2, "j">>,
 <3, <3, "a">>, <4, <4, "i">>,
 <5, <5, "j">>, <6, <6, "a">>,
 <7, <7, "i">>}
```

and USE gets as value:

```
{<1, <1, "m">>, <2, <2, "n">>,
 <3, <3, "u1">>, <4, <4, "i">>,
 <5, <5, "j">>, <6, <6, "u2">>,
 <7, <7, "u3">>}
```

Now we are ready to define a new relation KILL that defines which variable definitions are undone (killed) at each statement and is defined as follows:

```
rel[stat, def] KILL =
   {<S1, <S2, V>> | <stat S1, var V> : DEFS,
                    <stat S2, V> : DEFS,
                    S1 != S2}
```

In this definition, all variable definitions are compared with each other, and for each variable definition all *other* definitions of the same variable are placed in its kill set. In the example, KILL gets the value

```
{<1, <4, "i">>, <1, <7, "i">>,
 <2, <5, "j">>, <3, <6, "a">>,
 <4, <1, "i">>, <4, <7, "i">>,
 <5, <2, "j">>, <6, <3, "a">>,
 <7, <1, "i">>, <7, <4, "i">>}
```

and, for instance, the definition of variable i in statement 1 kills the definitions of i in statements 4 and 7. Next, we introduce the collection of statements

```
set[stat] STATEMENTS = carrier(PRED)
```

which gets as value {1, 2, 3, 4, 5, 6, 7} and two convenience functions to obtain the predecessor, respectively, the successor of a statement:

```
set[stat] predecessor(stat S) = PRED[-,S]
set[stat] successor(stat S) = PRED[S,-]
```

After these preparations, we are ready to formulate the reaching definitions problem in terms of two relations IN and OUT. IN captures all the variable definitions that are valid at the entry of each statement and OUT captures the definitions that are still valid after execution of each statement. Intuitively, for each statement S, IN[S] is equal to the union of the OUT of all the predecessors of S. OUT[S], on the other hand, is equal to the definitions generated by S to which we add IN[S] minus the definitions that are killed in S. Mathematically, the following set of equations captures this idea for each statement:

$$IN[S] = \bigcup_{P \in predecessor of S} OUT[P]$$

$$OUT[S] = DEF[S] \cup (IN[S] - KILL[S])$$

This idea can be expressed in RSCRIPT quite literally:

```
equations
 initial
  rel[stat,def] IN init {}
  rel[stat,def] OUT init DEF
 satisfy
  IN  = {<S, D> | stat S : STATEMENTS,
                  stat P : predecessor(S),
                  def D : OUT[P]}
  OUT = {<S, D> | stat S : STATEMENTS,
                  def D : DEF[S] union
                          (IN[S]\KILL[S])}
end equations
```

First, the relations IN and OUT are declared and initialized. Next, two equations are given that very much resemble the ones given above. They are solved by repeatedly computing the values of IN and OUT until their values become stable. For our running example (Figure 2) the results are as follows. Relation IN has as value:

```
{<2, <1, "i">>, <3, <2, "j">>,
 <3, <1, "i">>, <4, <3, "a">>,
 <4, <2, "j">>, <4, <1, "i">>,
 <4, <7, "i">>, <4, <5, "j">>,
 <4, <6, "a">>, <5, <4, "i">>,
 <5, <3, "a">>, <5, <2, "j">>,
 <5, <5, "j">>, <5, <6, "a">>,
 <6, <5, "j">>, <6, <4, "i">>,
 <6, <3, "a">>, <6, <6, "a">>,
 <7, <5, "j">>, <7, <4, "i">>,
 <7, <3, "a">>, <7, <6, "a">>}
```

If we consider statement 3, then the definitions of i and j from the preceding two statements are still valid. A more interesting case are the definitions that can reach statement 4:

- The definitions of variables a, j and i from, respectively, statements 3, 2 and 1.
- The definition of variable i from statement 7 (via the backward control flow path from 7 to 4).
- The definition of variable j from statement 5 (via the path 5, 7, 4).
- The definition of variable a from statement 6 (via the path 6, 7, 4).

For relation OUT a similar analysis can be given. The above definitions can be easily used to formulate a function reaching-definitions that can be used to define, for instance, other dataflow functions and program slicers.

3. Experience

RSCRIPT has been used in various case studies, including static analysis of syntax definitions [2], identification of dead code in Java [7], generic program slicing [5, 8], and prototyping of configuration integration of software components [6]. In addition to this, RSCRIPT turns out to be a very nice didactic tool to explain source code analysis.

The RSCRIPT implementation is written in ASF+SDF and does not address efficiency issues, although a preliminary study of implementation issues has been made in [3].

For instance, sets and relations are represented as linear lists. Nonetheless, the above applications could be assessed with ease. In addition to this a comprehensive framework of visualisations has been added to the ASF+SDF Meta-Environment[1] to visualize relational data.

The overall conclusions of these projects are:

- Extracting facts from source code is cumbersome and very language-dependant. Language-parametric fact extraction forms a major research challenge.
- Once, the facts are available, writing the RSCRIPT solution is easy.
- Generic visualizations for relations help writing the script and understanding the results. Tailoring these visualisations towards source code analysis tasks is an interesting and rewarding challenge.
- The completely unoptimized implementation becomes, not unexpectedly, a bottleneck when projects are larger (say over 100 KLOC).

4. Future developments

In addition to the observations made above, there are other concerns of interest:

- In many cases, it is convenient to include syntax tree fragments in sets and relations. We want to provide a type-safe framework and this implies including all types generated by a grammar in the RSCRIPT type system.
- The inclusion of syntax trees also implies the need to traverse (and transform, see next point) them in a type-safe manner.
- A more fundamental question is how to integrate the functionality provided by RSCRIPT with rewriting-based transformations. It is desirable that the facts that come available through a relational computation can drive the transformations.
- An even more fundamental question is how incrementality can be achieved when relational analysis and transformation interact. Since a transformation may invalidate relational facts, in the worst case, the transformed program should be the subject of a completely new fact extraction phase and subsequent relational analysis.

[1] See `http://www.meta-environment.org` for downloads and documentation.

Clearly, these and other questions will further influence the choice of implementation techniques. Together with Jurgen Vinju and Tijs van der Storm we are now exploring the design of a language that addresses most of the issues mentioned above: a completely new design that reuses the best features of ASF+SDF and RSCRIPT and supports them in a modern IDE. This will turn the concept language RSCRIPT into a versatile and efficient language for software analysis and transformation.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] J. Arnoldus. Grammaticacontrole met behulp van Rscript (in Dutch). Master's thesis, University of Amsterdam, 2005.

[3] M. Bredenoord. How to optimize Rscript comprehensions? Master's thesis, University of Amsterdam, 2006.

[4] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[5] P. Klint. Rscript—A Relational Approach to Software Analysis, 2007.

[6] T. van der Storm. Continuous release and upgrade of component-based software. In Jim Whitehead and Annita Persson Dahlqvist, editors, *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, 2005.

[7] J. van Willegen. Extractie van dode code in een heterogeen systeem: statische analyse in combinatie met dynamische analyse (in Dutch). Master's thesis, University of Amsterdam, 2006.

[8] I. Vankov. Relational approach to program slicing. Master's thesis, University of Amsterdam, 2005.

[9] J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, University of Amsterdam, 2005.

Note

References [2, 3, 7, 8] are directly available from the author's home page at `http://www.cwi.nl/~paulk`. Reference [5] is part of the documentation of the ASF+SDF Meta-Environment and can be found at `http://homepages.cwi.nl/~daybuild/daily-books/analysis/rscript/rscript.pdf`.