2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation

RASCAL: a Domain Specific Language for Source Code Analysis and Manipulation

Paul Klint Tijs van der Storm Jurgen Vinju

Centrum Wiskunde & Informatica *and* Informatics Institute, University of Amsterdam

Abstract

Many automated software engineering tools require tight integration of techniques for source code analysis and manipulation. State-of-the-art tools exist for both, but the domains have remained notoriously separate because different computational paradigms fit each domain best. This impedance mismatch hampers the development of new solutions because the desired functionality and scalability can only be achieved by repeated and ad hoc integration of different techniques.

RASCAL *is a domain-specific language that takes away most of this boilerplate by integrating source code analysis and manipulation at the conceptual, syntactic, semantic and technical level. We give an overview of the language and assess its merits by implementing a complex refactoring.*

1 The SCAM Domain

Source code analysis and manipulation (SCAM) is a large and diverse area both conceptually and technologically. There are plentiful libraries, tools and languages available but integrated facilities that combine both domains are scarce [19]. Both domains depend on a wide range of concepts such as grammars and parsing, abstract syntax trees, pattern matching, generalized tree traversal, constraint solving, type inference, high fidelity transformations, slicing, abstract interpretation, model checking, and abstract state machines. Examples of tools that implement some of these concepts are ANTLR [15], ASF+SDF [18], CodeSurfer [1], Crocopat [4], DMS [3], Grok [11], Stratego [5], TOM [2] and TXL [7]. These tools either specialize in analysis or in transformation, but not in both. As a result, combinations of analysis and transformation tools are used to get the job done. For instance, ASF+SDF [18] relies on RSCRIPT [13] for querying and TXL [7] interfaces with databases or query tools. In other approaches, analysis and transformation are implemented from scratch, as done in the Eclipse JDT. The TOM [2] tool adds transformation primitives to Java, such that libraries for analysis can be used directly. In either approach, the job of integrating analysis with transformation has to be done over and over again for each application and this requires a significant investment.

We propose a more radical solution by completely merging the set of concepts for analysis and transformation of source code into a single language called RASCAL. This language covers the range of applications from pure analyses to pure transformations and everything inbetween. Our contribution does not consist of new concepts or language features *per se*, but rather the careful collaboration, integration and cross-fertilization of existing concepts and language features.

The goals of RASCAL are: (a) to remove the cognitive and computational overhead of integrating analysis and transformation tools, (b) to provide a safe and interactive environment for constructing and experimenting with large and complicated source code analyses and transformations such as, for instance, needed for refactorings, and (c) to be easily understandable by a large group of computer programming experts. RASCAL is not limited to one particular object programming language, but is generically applicable. Reusable, language specific, functionality is realized as libraries.

We informally present a first version of RASCAL (Section 2) and its application to a complicated refactoring called "Infer Generic Type Arguments" on Featherweight Generic Java (Section 3). This example is used for an early assessment of RASCAL (Section 4).

2 The Rascal Design

RASCAL takes inspiration from many languages and systems. RASCAL's syntactic features are directly based on SDF [10]. Its analysis features take most from relational calculus, relation algebra and logic programming systems such as Crocopat [4], Grok [11] and RSCRIPT [13]. We also acknowledge the analysis and viewing facilities of CodeSurfer [1]. RASCAL has strongly simplified backtracking and fixed point computation features which remind of constraint programming and logic programming systems like Moreau's Choice Point Library [14], Prolog

978-0-7695-3793-1/09 $26.00 © 2009 IEEE
DOI 10.1109/SCAM.2009.28

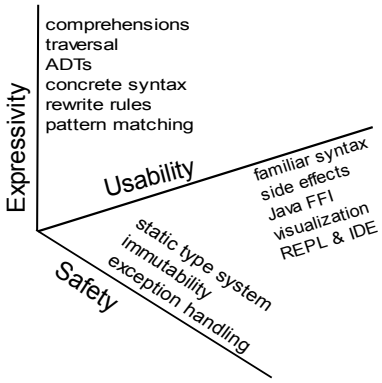

Figure 1. Dimensions of requirements

and Datalog. Its transformation and manipulation features are most directly inspired by term rewriting/functional languages such as ASF+SDF [18], Stratego [5], TOM [2], and TXL [7]. Syntactically, RASCAL takes from ASF+SDF, TXL and TOM, while semantics and implementation details are very much like ASF+SDF. The ATerm library [16], inspired RASCAL's immutable values. The ANTLR toolset [15], Eclipse IMP [6] and TOM [2] have been an inspiration because they integrate well with a mainstream programming environment. Their tractable and debuggable behaviour is very attractive. We also picked some cherries from general purpose languages such as Haskell, Java, and Ruby.

2.1 Requirements

RASCAL has been designed from a software engineering perspective rather than from a formal, mathematical, perspective. We have profited from our experience in building source code analysis and transformation solutions using ASF+SDF and RSCRIPT to formulate RASCAL's requirements. We have focussed on three dimensions of requirements: *expressiveness*, *safety* and *usability*. Figure 1 shows these dimensions together with some of the design decisions that are motivated by them. Additionally, sufficient performance for a wide range of SCAM applications is another key requirement. Below we describe each dimension in more detail.

Expressiveness Excellent means for expressing SCAM solutions is our most important requirement. We can subdivide it along the analysis/transformation line. *Analysis* requires suitable primitives for syntax analysis, pattern matching and collection, aggregation, projection, comprehension and combination of (relational) analysis results. *Transformation* requires powerful forms of pattern matching and traversal for high-fidelity source-to-source transformations. The use of concrete syntax as opposed to abstract syntax in the definition of transformation rules is essential.

Our goal is to cover the whole spectrum of SCAM. The language should scale up sufficiently to tackle large, complex problems like, for instance, legacy renovation or refactoring. It is preferable to solve these problems completely in RASCAL without having to resort to ad hoc coding of custom data-structures and/or algorithms in a general purpose language. However, we also want it to scale down, so that simple things remain simple. Computing the McCabe complexity of all methods in a large Java project should be close to a one-liner. Furthermore, problems usually solved with simple tools like GREP or AWK, should be easily solvable in RASCAL too, and preferably have the same usability characteristics.

Safety Source code analysis and transformation is a complex domain where solutions are error-prone. Many applications are both deep (conceptually hard) and wide (many details to consider). A modular language that facilitates encapsulation and reuse helps to deal with such complexity.

A static type system that offers safety features such as immutability and well-formedness will also help managing this complexity. We require that this type system integrates both the analysis and the transformation domain. This means that analysis results can be easily (re)used during transformation and that conversion, encoding and serialization of data between analysis and transformation phases is avoided. This also implies that syntax trees are fully typed; an essential prerequisite to ensure syntax safety for high-fidelity source-to-source transformations.

Usability Usability includes learnability, readability, debuggability, traceability, deployability and extensibility. We like the *principle of least surprise* and take stock in the fact that source code analysis and transformation is a form of programming. Staying close to ordinary, main stream programming languages will lower the barrier to entry for RASCAL. We also favour the *what you see is what you get* paradigm: most forms of implicitness or heuristics will eventually present usability problems.

No matter how good our domain analysis is, we cannot anticipate everything. Advanced users of RASCAL should therefore be able to extend the language with additional primitive functions in order to cater for new interfacing needs, faster implementation, or dedicated domain specific functionality. We advocate an open design that enables easy interoperability and integration with third-party components such as databases, parsers, SAT solvers, model checkers, visualization tool kits, and IDEs. Finally, we require RASCAL to have good encapsulation mechanisms that enable users to build reuseable components. Libraries of

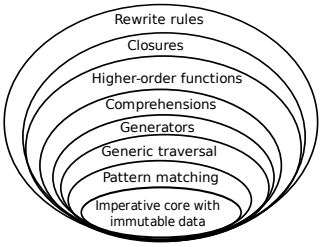

Figure 2. Layers in the Rascal design

reusable solutions for specific programming languages or programming paradigms directly increase usability.

Performance In addition to the main requirements dimensions show in Figure 1, performance requirements depend on the actual SCAM application: batch wise upgrading of a software portfolio may be less demanding than an interactive refactoring that is activated from an IDE. The results of source code analysis are often huge and the RASCAL implementation should be fast and lean enough to support such applications. Since different use cases may dictate different performance requirements, users must be able to supply different implementations of the core RASCAL data-structures if needed.

2.2 Rascal Language Design

The design of RASCAL has a layered structure (Figure 2), a desirable property from an educational point of view. RASCAL is an imperative language with a statically checked type system that prevents type errors and uninitialized variables. There are no run-time type casts as in Java or C# and there are therefore less opportunities for run-time errors. The type system features parametric polymorphism to facilitate the definition of generic functions. Functions (both defined and anonymous) are first-class values and can be passed to other functions as closures.

The types in RASCAL are distributed over a lattice according to a subtype relation with **value** at the top and **void** at the bottom. The subtype relation is co-variant for parametrized data-types such as sets and relations because all data is immutable. Sub-typing allows the programmer to express generic solutions with different levels of static checking. For instance, it is possible to write a function to process a parse tree typed over a given grammar. It is, however, also possible to write less strictly typed functions that can process parse trees over any grammar. Similarly, heterogeneous collections can be represented using the **value** type; pattern matching is used for type distinction (see paragraph 2.2.1).

Type	**Example literal**
bool	true, false
int	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
str	”abc”, ”first\nnext”
loc	!file:///etc/passwd
tuple[$t_1,\ldots,t_n$]	$\langle 1,2\rangle$, $\langle$”john”$,43,true\rangle$
list[t]	[], [1], [1,2,3], [true, 2, ”abc”]
set[t]	$\{\}$, $\{1,2,3,5,7\}$, {”john”$,4.0\}$
rel[$t_1,\ldots,t_n$]	$\{\langle 1,2\rangle,\langle 2,3\rangle,\langle 1,3\rangle\}$, $\{\langle 1,10,100\rangle,\langle 2,20,200\rangle\}$
map[t,u]	$()$, $(1:\text{true},2:\text{true})$, $(6:\{1,2,3,6\},7:\{1,7\})$
node	f, add(x, y), g(”abc”, [2, 3, 4])

Table 1. Basic Rascal Types

2.2.1 Concepts

Data-types and Types RASCAL provides a rich set of data-types. From booleans, infinite precision integers and reals to strings and source code locations. From lists, (optionally labeled) tuples and sets to maps and relations. From untyped tree structures to fully typed algebraic data-types (ADTs). The basic data-types are summarized in Table 1 together with their literal notations. A wealth of built-in operators is provided on these standard data-types. Many operators are heavily overloaded to allow for maximum flexibility.

Since source-to-source transformation requires concrete syntax patterns, the types of (parse) trees generated by a given grammar are first class RASCAL types. This includes all the non-terminals of the grammar, as well as (regular) grammar symbols, such as $S*$, $S+$ and $S?$. Parse trees can be processed as concrete syntax patterns or as instances of the ADT that is automatically derived from the grammar, or they can be analysed using a generic ADT for parse trees.

A type aliasing mechanism allows documenting specific uses of an existing type. The following example is from RASCAL's standard library:

```
alias Graph[&T] = rel[&T from, &T to];
```

The Graph data-type is equivalent to all binary relation types that have the same domain and range. Note how the type parameter &T makes this definition polymorphic and how the domain and range are labeled to allow projections on columns g.from and g.to for a graph g.

Users can extend the language with arbitrary ADTs, which could, for instance, be used to define the abstract syntax of a programming language. Here is a fragment of the abstract syntax for statements in a simple programming language:

```
data Stat =
```

```
  asgStat (Id name, Exp exp)
| ifStat (Exp cond, list [ Stat ] thenPart , list [ Stat ] elsePart )
| whileStat (Exp cond, list [ Stat ] body);
```

Values of the Stat type are constructed using familiar term syntax. For instance an assignment statement could be constructed as follows: asgStat(id("x"), nat(3)) where id("x") and nat(3) are constructors of Id and Exp respectively.

ADT values can be annotated with arbitrary values. For instance, expressions in the AST of a programming language could be annotated with type information. Annotations are declared so that their correct use can be enforced.

Pattern matching Pattern matching is *the* mechanism for case distinction in RASCAL. We provide string matching based on regular expressions, list (A) and set (ACI) matching, matching of abstract data-types, and matching of concrete syntax patterns. Patterns may contain variables, which will be bound when the match is successful. Anonymous (don't care) positions are indicated by an underscore (_). Patterns are used in switch statements, tree traversal, comprehensions, rewrite rules, exception handlers. The explicit match operator := allows patterns to be used in boolean expressions. For instance, the following match expression can be used to match a while statement as defined above:

```
whileStat (Exp cond, list [ Stat ] body) := stat
```

Pattern variables like cond and body can either be declared in-line—as in this example—or they may be declared in the context in which the pattern occurs. Since a pattern match may have more than one solution, local backtracking over the alternatives of a match is provided[1].

The pattern matching primitives clearly illustrate our effort to allow RASCAL both to scale up and to scale down. We provide sophisticated forms of matching but we also have regular expression patterns similar to those found in languages like Perl and Ruby.

Comprehensions and Control structures Many software analyses are relational in nature. Set comprehensions, such as found in RSCRIPT [13] provide a powerful, concise way of expressing a variety of analysis tasks, such as aggregation and projection. RASCAL has inherited comprehensions from RSCRIPT and generalizes them in various ways. Comprehensions exist for lists, sets and maps. A comprehension consists of an expression that determines the successive elements to be included in the result and a list of enumerators and tests. The enumerators (indicated by ←) produce values and the tests are boolean expressions that filter them. A standard example is

```
{ x * x | int x ← [1..10], x % 3 == 0 }
```

which returns the set {9, 36, 81}, i.e. the squares of the integers in the range [1..10] that are divisible by 3. A more intriguing example is

```
{ name | asgStat (Id name, _) ← P }
```

which returns a set of all identifiers that occur on the left-hand side of assignment statements in program P. The enumerator traverses the complete program P (that is assumed to have a Program as value) and only yields statements that match the assignment pattern.

Combinations of enumerators and tests also drive control structures. For instance,

```
for ( asgStat (Id name, _) ← P, size(name) > 10)
    println (name);
```

prints all identifiers in assignment statements that consist of more than 10 characters.

Switching and Visiting The *switch statement* as known from C and Java is generalized: the subject value to switch on may be an arbitrary value and the cases are arbitrary patterns. When a match fails, all its side-effects are undone and when it succeeds the statements associated with that case are executed.

Visiting the elements of a data-structure is one of the most common operations in our domain and we give it first class support by way of *visit expressions* that resemble the switch statement. A visit expression consists of an expression that may yield an arbitrarily complex subject value (e.g., a parse tree) and a number of cases. All the elements of the subject (e.g., all sub-trees) are visited and when one of the cases matches the statements associated with that case are executed. These cases may: (a) cause some side effect; (b) execute an insert statement that replaces the current element; (c) execute a fail statement that causes the match for the current case to fail. The value of a visit expression is the original subject value with all replacements made as dictated by matching cases. The traversal order in a visit expressions can be explicitly chosen by the programmer using strategy annotations like top-down and bottom-up.

Exception handling SCAM solutions are no different from other software solutions in that exceptional situations may occur. RASCAL features a try-catch exception handling mechanism similar to that found in Java or C#.

Functions and Rewrite Rules Functions are explicitly declared and are fully typed. Here is an example of a function that computes the cyclomatic complexity in a program:

```
int cyclomaticComplexity(Program p) {
  n = 1;
  visit (p) {
    case ifStat (_, _, _):      n += 1;
    case whileStat (_, _, _): n += 1;
```

[1] For safety, variable assignments are undone if backtracking occurs.

```
  }
  return n;
}
```

Note how this function simulates an accumulating traversal [17] using a (local) side-effect. The types of local variables may optionally be declared and local type inference is used otherwise. This is illustrated by the local variable n which has the inferred type int.

Rewrite rules are the only implicit control mechanism in the language and are used to maintain invariants during computations. For example, the following rule transforms if-statements of the example programming language.

```
rule
  ifStat (neg(Exp cond), list [ Stat ] thenPart , list [ Stat ] elsePart )
  ⇒ ifStat(cond, elsePart , thenPart );
```

If the condition of an if-statement contains a negated expression, the negation is removed and the branches are swapped. This rule will fire every time an if-statement is constructed that matches the left-hand side of this rule. The rule feature of RASCAL subsumes all features of ASF+SDF conditional equations.

2.2.2 Implementation and Tooling

It is important to be able to introduce the language in small steps. Such a piecemeal introduction requires lightweight tooling that further lowers the barrier to entry. For this, the RASCAL implementation features a command line Read-Eval-Print-Loop (REPL) in which the user can interactively enter RASCAL declarations and statements thus encouraging experimentation with small examples.

For professional use we have developed an Eclipse-based IDE, which currently features syntax highlighting, an outliner and a module browser. This IDE includes the RASCAL REPL so that it is still easy to prototype or test snippets of RASCAL code. The IDE also includes a visualization component which can be used to display complex data. This component can be compared to the visualization features found in the ASF+SDF Meta-Environment [18] and Semmle's .QL environment [8].

The basic data-structures of RASCAL are implemented by the Program Database (PDB) that is part of the Eclipse IMP framework. Since we were heavily involved in the design and implementation of the PDB, it will come as no surprise that there is a seamless match between RASCAL datatypes and PDB data-types. The design of the PDB follows the AbstractFactory design pattern so that RASCAL can be made to work with different underlying implementations; currently there are three such implementations.

RASCAL is accompanied by an elaborate standard library providing functions operating on the standard datatypes. The library also provides functions for reading and writing data in various formats (binary PDB values, XML, RSF). This is another way of enabling RASCAL to interface with existing data and/or tooling. In the near future we expect to extend the standard library with predefined datatypes and functions from the SCAM domain: libraries for metrics, control-flow analyses, slicing, etc.

For functionality that is not (easily) expressible in RASCAL itself the user can implement RASCAL functions with Java method bodies directly in RASCAL source code. This is implemented by runtime compilation of the Java bodies and linking them to the interpreter.

RASCAL is a modular language: source code is divided over a number of modules that may or may not (the default) export functions and global variables. Next to importing a module (which is similar to Java package importing), modules can be *extended*. This effectively creates a copy of a module (an instance) with possibly overridden functions. Module extension is intended for reuse with variation.

3 Featherweight Refactoring

Let's demonstrate RASCAL with a small but—we must warn—non-trivial example. The *infer generic type arguments* refactoring (IGTA) for Java [9] is interesting since it needs extensive analysis before simple source transformations can be applied. This refactoring automatically binds type parameters in code that uses generic classes but does not instantiate their type parameters (i.e., it uses *raw* types). After that the refactoring removes all casts that have become unnecessary. It guarantees to preserve type correctness of the code as well as run-time behaviour, such as method dispatch and casts. The snippet below illustrates how the refactoring works:

```
List l = new LinkedList();                          List<Integer> l =
l.add(new Integer (0));                      ⇒        new LinkedList<Integer>();
Integer i = (Integer )l.get (0);                    l.add(new Integer (0));
                                                    Integer i = l.get (0);
```

To keep the example small we present the RASCAL code that implements this refactoring for Featherweight Java with generics—a.k.a. FGJ. This is a micro language based on a number of core constructs in Java [12]. This example does not yet use RASCAL's concrete syntax feature because the implementation is unfinished at the time of writing. The current section highlights part of the example refactoring. We evaluate the results in Section 4.

Assuming the input program is type correct, the refactoring algorithm can be outlined in four steps. (1) For each class extract a set of type constraints that the initial program satisfies and a refactored program must still satisfy. (2) For each variable in the original program derive an initial set of estimated types. (3) Iteratively apply the extracted constraints to the estimates to obtain the new types. Finally (4), rewrite each declaration and remove superfluous casts. For Java this refactoring quickly runs into scalability problems.

Listing 1 Abstract syntax of Featherweight Generic Java.

```
1  module AbstractSyntax
2
3  alias Name = str;
4  data Type = typeVar(Name varName)
5            | typeLit(Name className, list[Type] actuals);
6  alias FormalTypes = tuple[ list[Type] vars, list[Type] bounds];
7  alias FormalVars = tuple[ list[Type] types, list[Name] names];
8
9  data Class  = class(Name className, FormalTypes formals,
10                      Type extends, FormalVars fields,
11                      Constr constr, set[Method] methods);
12 data Constr = cons(FormalVars args, Super super, list[Init] inits);
13 data Super = super( list[Name] fields );
14 data Init  = this(Name field);
15 data Method = method(FormalTypes formalTypes, Type returnType,
16                      Name name, FormalVars formals, Expr expr);
17 data Expr = var(Name varName)
18           | access(Expr receiver, Name fieldName)
19           | call  (Expr receiver, Name methodName,
20             list[Type] actualTypes, list[Expr] actuals)
21           | new(Type class, list[Expr] actuals)
22           | cast(Type class, Expr expr);
```

The sheer number of constraints extracted is huge for average systems. Our demonstration already incorporates some of the required optimizations presented in [9].

3.1 Abstract Syntax of FGJ

Listing 1 shows a module defining the abstract syntax of FGJ. Intuitively, it is a substitution calculus with objects. Classes and methods may introduce type parameters and **new** and **call** expressions can instantiate them. We assume that when **new** is used with an empty type parameter list the intention is to use the *raw* type.

Note the use of a **set** of methods in the definition of classes, which encodes the fact that the order of method declarations is irrelevant.

3.2 Querying Types

The extraction phase needs to know the static types of expressions. Listings 2 and 3 contain snippets of the RASCAL module that implements type queries directly on FGJ abstract syntax trees. This code implements the definition of FGJ from [12] and its size approaches the size of that definition; it is almost a one-to-one mapping. However, the implementation needs to be more precise in how and when to apply substitutions while transitively closing the subtype relations.

Let's highlight the bindings function from Listing 2, lines 10–14. It implements the binding of actual parameters to formal parameters in a concise way. A **map** is generated using a map comprehension. The formal and actual parameters need to be paired to produce a map. The comprehension iterates over the possible indices of the list of formals and looks up the actual type for each of them. The ? operator ensures that if an actual parameter does not exist for a certain index the type parameter is bound to Object. The result is short code, but it is precise and functional. Similarly, on line 33 a generator is used in the subtypes function to quantify over the elements of two lists. It looks for a counter example where two types are not sub-types at any particular index in the two lists.

Listing 2 Querying FGJ types (1/2)

```
1  module Types
2  import AbstractSyntax; import List;
3
4  public Type Object                    = typeLit("Object",[]);
5  public map[Name,Class] ClassTable = ("Object":ObjectClass);
6
7  alias Bounds     = map[Type var, Type bound];
8  alias Env        = map[Name var, Type varType];
9
10 public map[Type,Type]
11 bindings(list[Type] formals, list[Type] actuals) {
12   return (  formals[i] : actuals[i] ? Object |
13             int i <- domain(formals));
14 }
15 public &T inst(&T arg, list[Type] formals, list[Type] actuals) {
16   map[Type,Type] subs = bindings(formals, actuals);
17   return visit (arg) { case Type t => subs[t] ? t };
18 }
19 public rel[Name sub, Name sup] subclasses() {
20   return { <c, ClassTable[c].extends.className> |
21            Name c <- ClassTable }*;
22 }
23 public bool subtype(Bounds bounds, Type sub, Type sup) {
24   if (sub == sup || sup == Object) return true;
25   if (sub == Object) return false;
26   if (typeVar(name) := sub) return subtype(bounds[name], sup);
27   if (typeLit(name, actuals) := sub) {
28     Class d = ClassTable[name];
29     return subtype(inst(d.extends, d.formals.vars, actuals), sup);
30   }
31 }
32 public bool subtypes(Bounds env, list[Type] t1, list[Type] t2) {
33   return !(( int i <- domain(t1)) && !subtype(env, t1[i], t2[i]));
34 }
```

The etype function (Listing 3) computes the type of an expression. It uses pattern matching in a **switch** statement to dispatch over different types of expressions. The reason this code is very similar to the constraint inference rules from [12] is that their implicit universal quantification can be implemented easily using comprehensions (lines 20 and 30). Also de-structuring bind via matching in cases (lines 3, 4, 5, 11–12, 27, and 35) results in concise code. In this function we use the **if** conditional to merge the handling of several inference rules into a single case. We could have used overlapping case patterns that fail if one of the conditions of a rule fails, but this was shorter and clearer.

We have left out similar functions such as ftype and mtype

Listing 3 Querying FGJ types (2/2)

```
1  public Type etype(Env env, Bounds bounds, Expr expr) {
2    switch (expr) {
3      case this : return env["this"];
4      case var(Name v): return env[v];
5      case access(Expr rec, Name field) : {
6        Type Trec = etype(env, bounds, rec);
7        <types, fields > = fields (bound(bounds, Trec));
8        if (int i ←domain(types) && fields[i] == field)
9          return types[i];
10     }
11     case call(Expr rec, Name methodName,
12               list [Type] actualTypes, list [Expr] params) : {
13       Type Trec = etype(env, bounds, rec);
14       <<vars,varBounds>, returnType, formals> =
15           mtype(methodName, bound(bounds, Trec));
16
17       if (subtypes(bounds, actualTypes,
18                    inst(varBounds, vars, actualTypes))) {
19         paramTypes =
20           [ etype(env, bounds, param) | param ←params];
21         if (subtypes(bounds, paramTypes,
22                      inst(formals, vars, actualTypes))) {
23           return inst(returnType, vars, actualTypes);
24         }
25       }
26     }
27     case new(Type t, list [Expr] params) : {
28       <types, fields > = fields (t);
29       paramTypes =
30         [ etype(env, bounds, param) | param ←params];
31       if (subtypes(bounds, paramTypes, types)) {
32         return t;
33       }
34     }
35     case cast(Type t, Expr sup) : {
36       Tsup = etype(env, bounds, sup);
37       Bsup = bound(bounds, Tsup);
38
39       if (subtype(Bsup, t)) return t;
40       if (subtype(t, Bsup) && dcast(t, Bsup)) return t;
41     }
42   }
43   throw NoType(expr);
44 }
```

which compute the types of fields and methods. Also note that error handling, which is not specified in [12] at all, is implemented using RASCAL's exception handling in the form of the **throw** statement (Listing 3, line 43). RASCAL also throws IndexOutOfBounds exceptions for array indexers such as in Listing 2 lines 12–13 and 33. Without exceptions, error handling is typically an "implementation detail" that may require a lot of boilerplate code.

Note that we could have annotated every expression with a type to cache the result of these queries. For simplicity's sake we have not, but a more optimized version of this code should certainly do that. RASCAL's mechanism for declared and type safe annotations would be useful in that case.

Listing 4 Constraint variables, constraints and solutions.

```
1  module TypeConstraints import AbstractSyntax; import Types;
2  data TypeOf       = typeof(Expr expr) | typeof(Method method)
3                    | typeof(Name fieldName) | typeof(Type typeId)
4                    | typeof(Type var, Expr expr);
5  data Constraint   = eq(TypeOf a, TypeOf b)
6                    | subtype(TypeOf a, TypeOf b)
7                    | subtype(TypeOf a, set[TypeOf] alts);
8  data TypeSet      = Universe | EmptySet | Root | Single(Type T)
9                    | Set(set[Type] Ts) | Subtypes(TypeSet subs)
10                   | Union(set[TypeSet] args)
11                   | Intersection(set[TypeSet] args);
12 rule Set({Object})                    ⇒ Root;
13 rule Set({})                          ⇒ EmptySet;
14 rule Single(Type T)                   ⇒ Set({T});
15 rule Subtypes(Root)                   ⇒ Universe;
16 rule Subtypes(EmptySet)               ⇒ EmptySet;
17 rule Subtypes(Universe)               ⇒ Universe;
18 rule Subtypes(Subtypes(TypeSet x))    ⇒ Subtypes(x);
19 rule Intersection({Subtypes(TypeSet x), x, set[TypeSet] rest}) ⇒
20      Intersection(Subtypes(x), rest);
21 rule Intersection({EmptySet, set[TypeSet] _}) ⇒ EmptySet;
22 rule Intersection({Universe, set[TypeSet] x}) ⇒ Intersection({x});
23 rule Intersection({Set(set[Type] t1), Set(set[Type] t2),
24       set[TypeSet] rest}) ⇒ Intersection({Set(t1 & t2), rest});
25 rule Union({Universe,set[TypeSet] _})  ⇒ Universe;
26 rule Union({EmptySet,set[TypeSet] x})  ⇒ Union({x});
27 rule Union({Set(set[Type] t1), Set(set[Type] t2),
28            set[TypeSet] rest}) ⇒ Union({Set(t1 + t2), rest});
```

3.3 Defining and Extracting Constraints

Constraint extraction should be complete, so that any alternative type assignment of the original program *P* that satisfies all constraints is guaranteed to preserve static and dynamic semantics of the original program. Using this information the refactoring can then choose a type assignment that binds type parameters (if it exists) and continue to modify the code.

RASCAL does not have a built-in constraint solver but has the right primitives to implement a constraint solving algorithm efficiently and without much boilerplate code. Listing 4 defines the representation of constraint variables and constraints; the rules are described in 3.4. Existing constraint solvers such as the one presented in [9] are specialized for particular sets of problems. Hand-crafted data and computation specializations are an important tool for making source code analyses scale.

Listing 5 shows an excerpt of the RASCAL code that extracts type constraints (defined by Listing 4) from a FGJ program. It traverses the AST using the **visit** statement and matches each statement or expression that may contribute to the set of constraints. The set of constraints is incrementally constructed using simple additions of tuples or set comprehensions.

The cGen function (Listing 5, lines 38–45) is interesting. It is a bit simpler than the definition in [9] because FGJ is

Listing 5 Extracting type constraints

```
1  module Extract
2    import AbstractSyntax; import TypeConstraints; import Types; import List;
3
4  set[Constraint] extract(Bounds bounds, Class def, Method method) {
5    set[Constraint] result = {};
6    bounds += (method.formalTypes.vars[i]:method.formalTypes.bounds[i] | i ← domain(method.formalTypes.vars));
7    env = ("this": typeLit(def.className, []));
8
9    visit (method.expr) {
10     case x:access(Expr erec, Name fieldName): {
11       Trec = etype(env, bounds, erec);
12       fieldType = ftype(Trec, fieldName);
13       if (! isLibraryClass(def.className))
14         result += { eq(typeof(method), typeof(fieldType)), subtype(typeof(erec), typeof(fdecl(Trec, fieldName)))}; }
15     case x:new(Type new, list[Expr] args): {
16       result += {eq(typeof(x), typeof(new))};
17       if (! isLibraryClass(new))
18         result += { subtype(typeof(args[i]), typeof(constructorTypes(new)[i])) | int i ← domain(args) }; }
19     case x:call(Expr rec, Name methodName, list[Type] actuals, list[Expr] args): {
20       Trec = etype(env, bounds, rec);
21       result += {subtype(typeof(x), typeof(Trec))};
22       if (! isLibraryClass(Trec)) {
23         methodType = mtype(methodName, Trec);
24         result += eq(typeof(x), typeof(methodType.resultType));
25         result += { subtype(typeof(args[i]), typeof(methodType.formals[i])) | int i ← domain(args) }; }
26       else {
27         methodType = mtype(methodName, Trec);
28         result += cGen(typeof(etype(env, bounds, x)), methodType.returnType, rec, #eq);
29         result += { c | i ← domain(args), Ei := args[i], c ← cGen(Ei, methodType.formals[i], rec, #subtype)}; } }
30     case x:cast(Type to, Expr expr):
31       result += {eq(typeof(x), typeof(to)), subtype(typeof(expr), typeof(to))};
32     case x:var("this"):
33       result += {eq(typeof(x), typeof(typeLit(def.className,def.formals.bounds)))};
34   }
35   return result;
36 }
37
38 set[Constraint] cGen(Type a, Type T, Expr E, Constraint (TypeOf t1, TypeOf t2) op) {
39   if (T in etype((),(), E).actuals)
40     return {#op(typeof(a), typeof(T, E))};
41   else if (typelit(name, actuals) := T) {
42     Wi = ClassTable[name].formals.vars;
43     return { c | i ← domain(Wi), Wia := a.actuals[i], c ← cGen(Wia, Wi[i], E, #eq)}
44          + { #op(typeof(a), typeof(T)) }; }
45 }
```

simpler than Java, otherwise it is very similar. It even uses a higher order data constructors (#eq) as function parameters (lines 28, 43).

3.4 Constraint Evaluation

The constraint evaluation implementation in Listing 6 is straightforward. An initial estimate is computed for each constraint variable. For most variables this set will be the Universe. Then, in a fixed point computation implemented by the **solve** statement (Listing 6, lines 8–15), using Intersections implied by the extracted constraints all estimates are reduced to smaller sets.

We implemented the optimization from [9] to never fully enumerate the subtypes of any type during constraint solving using algebraic simplification. The (anonymous) rewrite rules from Listing 4 perform constraint simplification in an automatic and innermost fashion, for instance by eliminating Intersections. After constraint solving, the **visit** statement on lines 18–22, will expand all nested Subtypes nodes after which the rewrite rules will reduce each estimate to a final set of type literals.

Note that set matching in Listing 4 is used here to simulate matching modulo associativity, commutativity and idempotence of the binary set intersection operator.

Listing 6 Solving constraints.

```
1  module ConstraintEvaluation
2  import TypeConstraints; import Types;
3  import AbstractSyntax; import Extract;
4  public map[TypeOf var, TypeSet possibles] solveConstraints () {
5    constraints = {c | name ← ClassTable, c ← extract(name)};
6    with
7      estimates = initialEstimates ( constraints );
8    solve
9      for (TypeOf v                          ← estimates,
10          subtype(v, typeof(Type t))  ← constraints) {
11       estimates[v] = Intersection ({estimates[v],
12                                     Subtypes(Single(t))});
13   }
14
15   types = {}; visit ( constraints ) {case Type t : types += {t};};
16   subtypes = {<u,t> | t ← types, u ← types, subtype ((), t, u)};
17
18   estimates = innermost visit ( estimates ) {
19     case Subtypes(Set({s, set[Type] rest})) ⇒
20          Union({Single(s), Set(subtypes[s]),
21                 Subtypes(Set({rest}))}) };
22   return estimates;
23 }
24 public map[TypeOf, TypeSet]
25  initialEstimates (set[Constraint] constraints ) {
26   map[TypeOf, TypeSet] result = ();
27    visit ( constraints ) {
28     case eq(TypeOf t, typeof(Type o)) : result[t]=Single(o);
29     case t:typeof(typeVar(x), expr)   : result[t]=Single(Object);
30     case t:typeof(u:typeLit(x,y))     : ;
31     case TypeOf t                     : result[t]=Universe;
32   };
33   return result;
34 }
```

3.5 Source Manipulation

Finally, the resulting estimates for the constraint variables can be used to modify the code. This code is so trivial we will not show it here. The **visit** statement is used to find instances of expressions that can now be typed more precisely and the **insert** statement is used to replace them.

4 Assessment

Expressiveness. Table 2 shows size comparisons of the definitions of the FGJ type system and the IGTA refactoring functionality and their implementation in Rascal (including functions that we omitted from this paper). As measures we use Lines of Print (LOP) and Lines of Code (LOC). Lines of print of inference rules is counted as if rules are printed in a single column, but premises share single lines exactly as they are printed in the respective papers. Otherwise LOP is simply the lines of text in the two respective single-column papers. Lines of code is counted as number of non-empty, non-single-bracket, non-comment lines that fit on a 80 column page, but are otherwise formatted for understandability.

This comparison shows that the RASCAL implementation competes with the abstract mathematical and natural language explanation in terms of size. Unavoidably, the comparison is unfair to both representations. First, the (in)formal definitions use concrete syntax patterns, while the implementation in RASCAL uses abstract syntax and—this may come as a surprise to some readers—abstract syntax is more verbose.[2] Second, the (in)formal definitions use single character variables, while the implementation in RASCAL uses full identifier names. Third, the English explanations have gaps of imprecision and ambiguity, while the implementation is complete and non-ambiguous. In [9] some inference rules even share conditions to save space. On the one hand, our typing rules assume that the input program is valid, which saves a number of conditions to implement. On the other hand, the inference rules for constraint extraction assume type analysis and name binding to have been completed, which our implementation does on-the-fly. Finally, the extraction rules from [9] have two rules for static methods that FGJ does not implement, which are good for 4 LOP and 4 shared premises.

With these provisions in mind, we conclude from Table 2 that the (in)formal definition and the actual implementation in RASCAL are very close in size, that there is apparently hardly any boilerplate code and that RASCAL offers the right domain abstractions.

Safety The IGTA refactoring on FGJ represents a significant amount of work. Both the language and the refactoring are far from trivial. Therefore, as in every software project, the implementation changed gradually from an initial, explorative prototype to a final "product". We started with a completely different definition of the abstract syntax which was shorter but less like the original definition in [12]. Also we have had different representations for the constraints and different implementations of the solver.

The abstract data definitions served as *contracts* for the code which the RASCAL type checker could check for obvious mistakes while code was migrated. Also, the types of functions serve to keep things working. However, we frequently used the local type inference for variables in functions, just to be able to ignore thinking about specific details about intermediate variables while coding. We noticed that such type inference sometimes leads to "stupid" mistakes, but since their influence is always local to a function they are easy to trace and fix by adding the missing type declarations.

Usability. The refactoring code we demonstrated contains many design choices. Many different styles of implemen-

[2] Recall that we do not use RASCAL's concrete syntax feature.

Feature	(In)formal definition			Implementation in RASCAL		
	Inf. rules	Premises	LOP	Functions	Cases+cond's	LOC
Typing [12]	28	66	62	16	8+22	101
Constraint Extraction [9]	25	41	49	6	5+6	78
Constraint Evaluation [9]	*English explanation of 1200 words*		*85 lines*	2	27+0	56
Rewriting [9]	*English explanation of 76 words*		*6 lines*	1	4+0	15

Table 2. Definition versus Implementation in Rascal: LOC (250) on par with lines of print (202).

tations would have been possible in RASCAL, all on the same level of abstraction, but with different characteristics. It means that RASCAL is not closed to a specific way of solving SCAM problems, but allows experimentation with different algorithms and data structures on a high level of abstraction.

Performance We have yet to evaluate the design and implementation of RASCAL in terms of performance. There are obvious ways of improving performance however, by using existing optimization techniques from term rewriting engines, such as ASF+SDF, Tom and Stratego, and relational calculators such as Grok and Crocopat. Additionally, we expect that just-in-time compilation to Java byte-code will pay off. One data point we can provide, is that we can currently compute the transitive closure of the method call graph of the complete Eclipse JDT source in 16 seconds on a 2GHz dual core machine.

Acknowledgements. We thank Bob Fuhrer (IBM Research) for his inspiration, co-authoring the PDB and explaining the IGTA refactoring. We thank Arnold Lankamp for implementing the faster implementations of the PDB API.

References

[1] P. Anderson and M. Zarins. The CodeSurfer software understanding platform. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, pages 147–148. IEEE, 2005.

[2] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[3] I. Baxter, P. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'04)*, pages 625–634. IEEE, 2004.

[4] D. Beyer. Relational programming with CrocoPat. In *Proceedings of the 28th international conference on Software engineering (ICSE'06)*, pages 807–810, New York, NY, USA, 2006. ACM.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[6] P. Charles, R. M. Fuhrer, and S. M. Sutton Jr. IMP: a meta-tooling platform for creating language-specific IDEs in eclipse. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 485–488. ACM, 2007.

[7] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[8] O. de Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .QL: Object-oriented queries made easy. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 78–133. Springer, 2008.

[9] R. M. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 71–96, Glasgow, Scotland, July 27–29, 2005.

[10] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[11] R. C. Holt. Grokking software architecture. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 5–14. IEEE, 2008. Most influential paper.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[13] P. Klint. Using Rscript for software analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.

[14] P.-E. Moreau. A choice-point library for backtrack programming. JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic, 1998.

[15] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[16] M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[17] M. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[18] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[19] J. J. Vinju and J. R. Cordy. How to make a bridge between transformation and analysis technologies? In *Dagstuhl Seminar on Transformation Techniques in Software Engineering*, 2005. http://drops.dagstuhl.de/opus/volltexte/2006/426.