Program Analysis Scenarios in Rascal

Mark Hills[1], Paul Klint[1,2], and Jurgen J. Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. Rascal is a meta programming language focused on the implementation of domain-specific languages and on the rapid construction of tools for software analysis and software transformation. In this paper we focus on the use of Rascal for software analysis. We illustrate a range of scenarios for building new software analysis tools through a number of examples, including one showing integration with an existing Maude-based analysis. We then focus on ongoing work on alias analysis and type inference for PHP, showing how Rascal is being used, and sketching a hypothetical solution in Maude. We conclude with a high-level discussion on the commonalities and differences between Rascal and Maude when applied to program analysis.

1 Introduction

Rascal [32,33] is a meta programming language focused on the implementation of domain-specific languages and on the rapid construction of tools for software analysis and software transformation. Rascal is the successor to both ASF [4] and ASF+SDF [48,46], providing features for defining grammars, parsing programs, analyzing program code, generating new programs, interacting with external tools (through Java), and visualizing the results of these operations.

In this paper we focus on software analysis, exploring the design space of Rascal analysis solutions. We begin this in Section 2 by providing a brief introduction to Rascal, focusing on the design of the language and how this design is realized by Rascal's language features. We also show several small examples of Rascal code. We continue this in Section 3, presenting several analyses developed using Rascal: a hybrid Rascal/Maude [11] analysis for finding type and units of measurement errors; a Rascal analysis of Java code that uses information extracted from the Eclipse Java Development Tools; and the Rascal name resolver and type checker, which works as part of the Rascal development environment and is implemented completely in Rascal. These examples highlight different solution scenarios in the design space, including (at one extreme) using Rascal as a coordination language for existing analysis tools and (at the other) building an analysis solely in Rascal.

In Section 4 we then describe ongoing work, written mostly in Rascal but with some integration of external tools, on defining a set of analyses for the PHP language. We focus here on two specific analyses: alias analysis and type inference. We start this description by setting out a number of tasks that need to be performed for these analyses – for instance, parsing the source program, or defining an internal representation for storing analysis facts. We then show how Rascal is used to provide support for each of

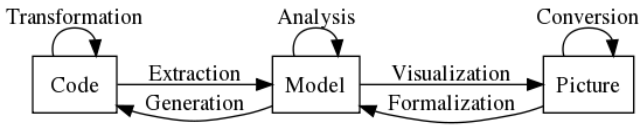

Fig. 1. Meta-programming domain: 3 layers of software representation with transitions.

these tasks. Another possibility for defining these analyses would be to use rewriting logic [34], specifically techniques developed as part of the rewriting logic semantics [35] project. Therefore, to end the section, we also show how these analysis tasks could be supported with a rewriting logic semantics-based analysis in Maude.

This paper touches on several areas with extensive related work. Section 5 focuses on existing research related directly to Rascal, the analysis of PHP programs, and program analysis using Maude. Section 6 then offers observations and discussion, highlighting what we believe to be the advantages and disadvantages of Rascal and Maude for developing program analysis tools.

2 Rascal

Rascal was designed to cover the entire domain of meta-programming, shown pictorially in Figure 1. The language itself is designed with unofficial "language layers". This allows Rascal developers to start with just the core language features, adding more advanced features as they become more comfortable with the language. This language core contains basic data-types (booleans, integers, reals, source locations, date-time, lists, sets, tuples, maps, relations), structured control flow (if, while, switch, for), and exception handling (try, catch). The syntax of these constructs is designed to be familiar to programmers: for instance, if statements and try/catch blocks look like those found in C and Java, respectively. All data in Rascal is immutable (i.e., no references are ever created or taken), and all code is statically typed. At this level, Rascal looks like a standard general purpose programming language with immutable data structures.

Rascal's type system is organized as a lattice, with bottom (`void`) and top (`value`) elements. The Rascal `node` type is the parent of all user-defined datatypes, including the types of concrete syntax elements (`Stmt`, `Expr`, etc). Numeric types also have a parent type, `num`, but are not themselves in a subtype relation: i.e., `real` is not a parent of `int`. The basic types available in Rascal, including examples, are shown in Table 1.

Beyond the type system and the language core, Rascal also includes a number of more advanced features. These features can be progressively added to create more complex programs, and are needed in Rascal to enable the full range of meta-programming capabilities. These more advanced features include:

- Algebraic data type definitions, with optional type parameters, allow the user to define new data types for use in the analysis. These data types are similar to sum types in functional languages like ML or (possibly parameterized) sort and operator definitions in algebraic systems like Maude.

Type	**Example literal**
bool	true, false
int	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
rat	1r4, 22r7, -3r8
str	”abc”, ”first\nnext”
loc	|file:///etc/passwd|
datetime	$2012-05-08T22:09:04.120+0200
tuple[$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.

- A built-in grammar formalism allows the definition of context-free grammars. These grammars are used to generate a scannerless generalized parser, which allows for modular syntax definitions (i.e., unions of defined grammars) and the parsing of programs in real programming languages. The syntax formalism is EBNF-like and includes disambiguation facilities, such as the ability to indicate associativity and precedence, add follow restrictions, and even provide arbitrary code to disallow specific parses.
- Pattern matching is provided over all Rascal data types: matches can be performed against numbers, strings, nodes, etc. A number of advanced pattern matching operators, such as deep match (`/`, matching values nested at an arbitrary depth inside other values), negative match (`!`), set matching, and list matching are also provided. Given the importance of concrete syntax for some meta-programming tasks, it is also possible to match against concrete syntax fragments, e.g., matching a while loop and binding variables to syntax fragments representing the loop condition and the loop body.
- Additionally, pattern matching is used in the formal parameters of functions, allowing function dispatch to be based on the pattern matching mechanism. This provides for more extensible code, since new constructors of a user-defined datatype can be handled by using new variants of an existing function, instead of requiring a single function with a large switch/case statement. As an equivalent to the switch/case `default` case, a default function provides the default behavior for the function when none of the other cases match.
- In cases where there are multiple matches for a pattern, backtracking happens from right to left in a pattern, enforcing lexical scope (names bind starting at the left, and can be used in the pattern to the right of the binding site) and providing a natural order on matches. A successful match can be explicitly discarded by the user with the `fail` keyword.

– List, set, and map comprehensions, in combination with pattern matching and other Rascal expressions, allow new lists, sets, and maps to be constructed based on complex conditions. For instance, one could use a deep match to find all while loops in a set of program files that contain a condition with a less than comparison. Also provided is the <− element generation operator, which can enumerate the elements of all container data-types, e.g. lists, sets, maps, and trees, and can be used inside comprehensions and in for loops.
– String templates with margins and an auto-indent feature provide a straightforward way to generate formatted code in multi-line source code templates.
– `visit` statements, with a syntax similar to that of `switch` statements, perform *structure-shy* traversals of Rascal data types, allowing one to match only those cases of interest. Visit cases can execute arbitrary code, for instance to keep track of statistics or analysis information, or can directly replace the matched node with one of the same type. Visits are parameterized by a traversal strategy (e.g., top-down) to allow different traversal orders.
– `solve` statements allow fixed-point computations to be expressed directly as a language construct. The statement continues to iterate as long as the result of the condition expression continues to change.

A number of Rascal features focus on the safety and modularity of Rascal code. While local variable types can be inferred, parameter and return types in functions must be provided. This allows better error messages to be generated, since errors detected by the inferencer can be localized within a function, and also provides documentation (through type annotations) on function signatures. Also, the only casting mechanism is pattern matching, which prevents the problems with casts found in C (lack of safety) and Java (runtime casting exceptions). Finally, the use of persistent data structures eliminates a number of standard problems with using references which can leak out of the current scope or be captured by other variables.

Example: As a simple example, imagine that we want to work with the Peano representation of natural numbers. In Maude, these could be defined as follows:

```
fmod PEANO is
  sort Nat .
  op z : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  vars N M : Nat .

  op plus : Nat Nat -> Nat .
  eq plus(s(N),M) = s(plus(N,M)) .
  eq plus(z,M) = M .
endfm
```

In Rascal, this same functionality would be defined as follows:

```
module Nat

data Nat = z() | s(Nat);

Nat plus(s(Nat n), Nat m) = s(plus(n,m));
Nat plus(z(), Nat m) = m;
```

Function `plus` could also be defined using a switch/case statement, as follows:

```
Nat plussc(Nat n, Nat m) {
  switch(n) {
    case s(j) : return s(plussc(j,m)) ;
    case z()  : return m;
  }
}
```

As a more complex example, take the case where we have colored binary trees: trees with an integer in the leaves, but with a color (given as a string) defined at each composite node. This would be defined as follows:

```
data ColoredTree
    = leaf(int n)
    | composite(str color, ColoredTree left, ColoredTree right);
```

Suppose we want to analyze a `ColoredTree`, computing how often each color appears at each node. The Rascal code is shown in Listing 1. In this code, we use a map, held in a local variable `counts` with inferred type `map[str, int]`, to maintain the counts. A visit statement is used to traverse the binary tree, matching only the composite nodes, and binding the color stored in the node to the string variable `color`. The statement `counts[color]?0 += 1` then increments the current frequency count for the given color if it exists, or it initializes this count to 0 first and then increments, assigning the result back into the map entry for the color.

Listing 1 Counting frequencies of colors in a ColoredTree.

```
public map[str, int] colorDistribution(ColoredTree t) {
   counts = ();
   visit(t) {
     case composite(str color, _, _): counts[color] ? 0 += 1;
   }
   return counts;
}
```

3 Scenarios for Program Analysis in Rascal

Scenarios for program analysis using Rascal can be viewed as a spectrum: at one end, Rascal acts just as a coordination language, with all the analysis work done using external

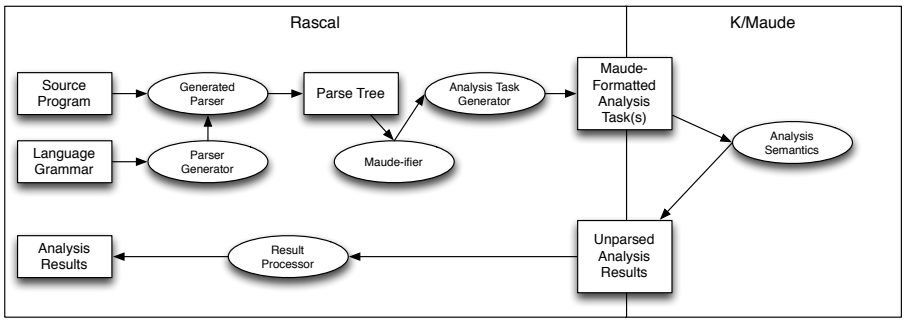

Fig. 2. Integrating Rascal and K.

tools; at the other, all work needed for the analysis, from parsing, through all the analysis tasks, to the display of the results, is done within Rascal. Many solutions are somewhere in between, with Rascal providing significant functionality while also interacting with existing tools. This section presents three examples exemplifying these alternatives.

3.1 Integrating Rascal with RLS-Based Analysis Tools

One approach to program analysis in Rascal, near the "coordination language" end of the spectrum, is to use a program analysis tool based on rewriting logic semantics (RLS) and Maude while leveraging the support for parsing and IDE integration provided by Rascal. This is supported in Rascal through the RLSRunner library [22].

RLSRunner provides components in both Rascal and Maude for linking Rascal language definitions with RLS analysis semantics. In Rascal, functions and data types are provided both to perform the analysis in Maude using the analysis semantics and to process the results to yield information about the error and warning messages to display in Eclipse. In Maude, sorts and operations are defined to model and use Rascal source locations, allowing the locations of errors to be tracked by the analysis and reported accurately to the user in Eclipse. These locations are added to an analysis semantics by extending abstract syntax sorts with new operations to represent located versions of terms, with additional equations provided to keep track of the locations in the configuration and to give back the original (unlocated) terms.

Figure 2 provides an overview of the RLSRunner process. The two initial inputs are a grammar for the language being analyzed, created using the Rascal grammar formalism, and a source program. The grammar is processed using the Rascal parser generator, generating a parser for the language under analysis. This parser is then used to parse the source program, as well as to provide features used by the Eclipse-based program user interface such as code folding, code outlining, etc. Using the parse tree emitted by the parser, a Rascal program dubbed the "Maude-ifier" is then run. In conjunction with the analysis task generator, this generates individual Maude terms from the parsed program, with one term per analysis task – in some cases a task represents the entire program, while in others tasks may be generated for smaller units, such as for individual functions. Each analysis task is then evaluated in the analysis semantics, yielding the

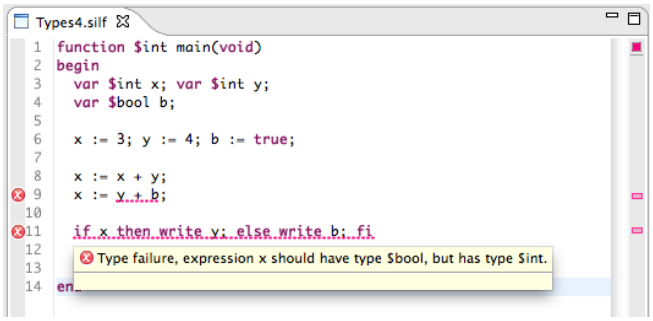

Fig. 3. Type Errors in SILF Programs shown in Rascal's Eclipse-based IDE.

analysis results, which are processed by the result processor to yield analysis information shown to the user in the IDE (e.g., error messages).

An example of the result of this process is shown in Figure 3, which shows type errors identified in a SILF [20] program. The type errors are identified using the types policy in the SILF Policy Framework [24], a rewriting logic semantics-based framework for defining program analyses. The SILF units policy generates similar messages, but based instead on annotations provided for units of measurement.

3.2 Refactoring Analysis with the Eclipse JDT

As an experiment in measuring the maintainability of large software systems, we decided to investigate the difference in maintenance complexity for an interpreter written using either the Visitor [17, page 331] or the Interpreter [17, page 243] design patterns. We did this by creating a refactoring [38,37,16], dubbed V2I [23], which transforms an interpreter written using the Visitor pattern into one written using the Interpreter pattern, holding all else constant. We then measured the difficulty of performing various maintenance scenarios on the two systems [21].

As part of the work in developing V2I, we had to develop an analysis that would identify the code to be refactored. Normally, this would require creating a grammar for Java, parsing the Java code used in the interpreter, and performing analyses to bind name and type declaration information to entities in the code (while also taking account of information provided by external libraries, which could be in binary form), all before developing the analysis needed to perform the refactoring. Instead of doing this in Rascal, we opted to instead reuse the information computed by the Eclipse Java Development Tools (JDT), which computes all of these facts as part of its IDE support for Java development. These facts are extracted from the JDT using the Rascal JDT library, which communicates with Eclipse to build Rascal representations of a number of Java program facts. Some of the facts used by the V2I analysis are shown in Table 2, with both entities (types, classes, fields, methods) and relationships between entities (the remaining four items) shown.

Using these entities, the analysis performs a number of computations to find all methods that must be refactored. This is done by computing a number of intermediary relations – for instance, from visitor interfaces to implementing classes – with the final

Extracted Fact	Description
types	classes, interfaces, enums
classes	classes
fields	fields
methods	methods
modifiers	modifiers on definitions (e.g., public, final)
implements	interface $\times$ implementer
extends	class or interface $\times$ extender
declaredMethods	class or interface $\times$ method declaration

Table 2. Rascal JDT Interface: Extracted Entities and Relationships.

relation containing all identified methods, the method locations, and the method source code. Additional relations indicate any dependencies of the refactored methods that must also be modified. For example, in cases where method code is relocated to a different class, uses of private fields in this code are changed first to uses of public getter and setter methods. As in the prior example, Rascal acts partly as a coordination language, but here also performs all the V2I-specific analysis using Rascal code.

3.3 Type-Checking Rascal in Rascal

The Rascal type checker (referred to hereafter as the RTC for conciseness) enforces the static typing rules of the Rascal language. It is fully implemented in Rascal and uses no external tools.

The parsing of Rascal code precedes RTC, naturally. Rascal includes a syntax definition formalism which is bootstrapped, and supports embedded concrete syntax fragments. When parsing Rascal modules with concrete syntax, such as in the implementation of RTC, an automatically generated parser for Rascal embedded in Rascal is used. The input for RTC are parse trees of any Rascal modules, which are also processed by similarly generated parsers.

Several key principles are used in the RTC:

- Checking occurs at the level of individual modules. Imported modules are assumed to be correct, and supply a module signature with type information for all declarations.
- Checking with a module is performed by checking each function individually. Function signatures must be given explicitly for each declared function, and are not inferred.
- Function bodies are checked by statically evaluating the code in the function body using a recursive interpreter which implements the type semantics. Values in the interpreter represent types and abstract entities, such as variables, functions, etc.
- Local inference is handled by checking for stabilization across multiple (static) evaluations of iterating constructs. For instance, loops that assign new values to variables are evaluated twice. Types that fail to stabilize are assumed to be `value`, the top of the type lattice.

The result of running the checker is an assignment of types (including error types) to all names and expressions in a Rascal module. This information is then used by

the Rascal IDE to provide type documentation (visible by hovering over a name or expression in the IDE), type error annotations, and documentation links allowing the user to jump to the definition of a name, including definitions of variables, constructors, functions, and user-defined data types.

3.4 Discussion

These three tools exemplify the diversity of solution strategies when using Rascal. In some cases we use external tools, while in others we do not. The RTC is self-contained and bootstrapped, the refactoring emphasizes reuse of the Eclipse JDT and the SILF checker reuses a version of K in Maude. These examples also use different intermediate data-structures and different types of analysis algorithms.

The design of Rascal is intended to leave many choices to the meta programmer. It provides a kaleidoscopic set of solution scenarios. Users are not forced to use the language for all components, and are actively helped by the system to connect to other systems. This is one reason Rascal has been designed as a programming language rather than as a specification formalism [45].

4 Analyzing PHP

In this section we describe analyses of PHP code using Rascal. We also describe the same analyses as they could be implemented in Maude. Our goal is to give the reader insight into how these two systems compare in terms of functionality and style.

PHP[1] is a dynamically-typed server-side scripting language. According to the TIOBE Index[2], as of May 2012 PHP is the 6th most popular programming language. PHP5[3] is an object-oriented language with an imperative core. The object system is based around a single-inheritance model with multiple inheritance of interfaces. The visibility mechanism uses the familiar keywords `public`, `private`, and `protected`, and works in conjunction with the inheritance mechanism (e.g., protected methods are visible in subclasses). As in C++, namespaces provide a mechanism for grouping user-defined names. Newer features include closures and traits [44]. PHP also includes extensive libraries, including standard functions for working with arrays, manipulating strings, and interacting with databases, and a number of third-party application frameworks and utility libraries are available.

We are interested in analyzing PHP code for several reasons. Since PHP has a weak typing model ("duck typing") it allows many common errors to go undetected. One question this raises is: even with this weak typing model, how many of these errors can be detected, and with what accuracy? Another goal of this research is to detect possible errors caused by changes in the PHP semantics, especially in cases where syntactically updated PHP4 code is running on a PHP5 engine. Tools which analyze PHP and detect such problems are typically in the application domain of Rascal, as well as in the application domain of Maude.

[1] `http://www.php.net`

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/`

[3] As of May 2012, the current version is 5.4.3.

First we describe some of the challenges, with possible solutions, in analyzing PHP. These include resolving includes, alias analysis and type inference.

4.1 Analyzing Includes, Aliases and Types for PHP

As part of our work on creating analysis tools for PHP, we are creating two analyses that will be used in many of the other tools: an alias analysis, and a type inferencer. These both work on individual PHP scripts, which can include other scripts as well as references to PHP library functions.

Includes The first challenge in our analysis is to actually get a script's complete source. Unlike in languages such as C, includes in PHP are resolved at runtime, with include paths that can be based on arbitrary expressions. In a worst-case scenario, this means that an include could refer to any other PHP file in the system. In practice, it is possible in many cases to statically resolve the include path using a number of techniques (constant propagation, algebraic simplification, path matching). Thus, for these analyses we assume we have a "fully inlined" script in which all includes have been merged in.

Type Inference The type inferencer is based on the Cartesian product algorithm [1]. This algorithm assigns a set of types from the universe of all possible program types (classes, interfaces, and built-in types) to the expressions and *access paths* (paths made up of names, field accesses, and array element accesses) in the script. This type assignment is initially seeded with those cases where an invariant type can be determined. For instance, the expression `new T` always yields type {`T`}, and the literal `5` is always given type {`int`}. Using this seed, other types are derived using typing rules. The algorithm gets its name through its treatment of method calls: given the type sets assigned to the invocation target and the actual parameters, the Cartesian product of these sets is formed. For each element of this Cartesian product, the proper method (based on the target type) is selected, types are bound to the formal parameters, and the method body is then typed. Overloaded expressions, such as the arithmetic expressions, can be treated as special versions of methods and typed similarly.

Alias Analysis The alias analysis is based on an interprocedural alias analysis algorithm that can handle function pointers, recursive calls, and references [25]. The alias analysis yields a relation between names at each program point, where the relation contains pairs of names which are may-aliases (i.e., which *may* refer to the same memory location). Aliases in PHP are created directly through reference assignments, the use of reference arguments and reference returns in functions and methods, and potentially through the use of the `global` statement. Aliases can also be created indirectly through the use of variable-variables, where (for instance) the name of a variable to access is stored as a string in another variable and "dereferenced" using a double dollar sign. Finally, indirect aliases are created through object references, which act as pointers to the referenced objects. Several of these cases are shown in Figure 4.

In order to get the correct results, the analyses need to be run in tandem, first running one, then the other, until a fixpoint is reached. This is because each analysis provides new

```
$g = 10;
function f1(&$p1) { $p1++; } // $p1 is a reference argument
function &f2() { global $g; return $g; } // reference return
class C1 { public $v1 = 5; }

$a = 3;
f1($a); // $a is now 4, $p1 aliases $a in f1
$b =& f2(); // $b now aliases $g
$c = new C1();
$d = $c; // $d and $c are not aliases, $c->v1 and $d->v1 are
$d =& $c; // $d and $c now are aliases
$vv = "a";
$$vv = 6; // $$vv aliases $a
```

Fig. 4. Creating Aliases in PHP.

information that can be used in the other. For instance, discovering that two names are aliased can add to the set of types assigned to a name, which could then add additional elements to the Cartesian products calculated for method calls. Also, expanding the set of types of an invocation target can expand the set of methods invoked at a specific call site, which can then lead to the generation of more alias pairs. While convergence of this process can be slow, the type inference and alias analysis algorithms work over finite sets of values and are monotonic, guaranteeing that they will terminate.

4.2 Required Analysis Tasks

A number of standard tasks are required for creating any PHP analysis. We discuss four of these below: parsing, maintaining internal representations needed in the analysis, writing the rules for the analysis, and reporting the analysis results. After this we show how we implemented these four tasks in Rascal and then we sketch out how we would implement them in Maude.

Parsing PHP Scripts The purpose of executing a server-side PHP script (the standard mode of execution) is to generate an HTML page to send to a client. To make this easier for developers, PHP scripts are often a mixture of PHP code and fragments of HTML. Because of this, the parser needs to be capable of parsing intermingled PHP code and HTML markup. Whether to keep the HTML fragments is a decision based on the analysis – some analyses try to ensure that sensible HTML code is generated, while for other analyses it may be possible to discard the HTML. Since the analysis also needs to be able to handle includes properly (discussed above), parsing and later steps may also work in tandem, with new scripts parsed as new information on includes is discovered. The end result of parsing should be an internal representation of the parsed script(s) that can be used in the analysis.

Developing Internal Representations Each analysis uses a (potentially large) number of intermediate representations. Some of these can be shared between different analyses,

such as the representation of the script to analyze, while some are unique to each analysis. For the analyses here, this would include the representations of the results: sets of types assigned to expressions and access paths for the type inference analysis, and sets of alias pairs assigned to program points for the alias analysis. This would also include the intermediate representations, used during computation of the results, but often containing information that is not needed once the analysis is complete.

Writing Analysis Rules The analysis rules interact with the internal program representation and the various supporting data structures to analyze the various language constructs, computing (for instance) the inferred type of a concatenation expression, the aliases created in a reference assignment, or the set of all alias pairs at a given program point.

Reporting Analysis Results The analysis needs to provide a way to report the final analysis results. Based on the needs of the analysis clients, results could be provided using internal data structures (e.g., sets of alias pairs associated with a specific program point), external messages (visual indicators to show which types have been inferred for an expression), or some combination of the two.

4.3 Analyzing PHP in Rascal

Here we describe, in terms of the analysis tasks listed above, the current implementation of the PHP type and alias analyses we have developed in Rascal.

Parsing PHP Scripts We are currently parsing PHP using a fork of an open-source PHP parser[4]. Our parsing script pretty-prints the parse tree as a term conforming to the AST representation we have in Rascal for PHP, with location information provided as Rascal annotations. This PHP script is called directly from Rascal using a library for interacting with the command shell. Parsing the abstract syntax tree representation to Rascal's internal algebraic data-types is also a standard library function.

We are also converting an SDF parser for PHP, written as part of the PHP-front project[5], which will allow us to parse PHP code directly in Rascal. This will also create terms conforming to the Rascal PHP AST definition so we will not have to change existing code.

Developing Internal Representations The PHP AST is defined as a mutually recursive collection of Rascal datatype declarations, with base types and collection types used to represent strings, integers, lists of parameters, etc. The internal configuration of the analysis is represented as an element of a Rascal algebraic datatype, with different fields holding different pieces of analysis information. Relations are used to represent intermediate results (e.g., the relation between an access path and possible types); final results are often instead given in maps, since these provide for quicker lookup performance, and can be stored directly on the abstract syntax tree using Rascal annotations.

[4] `https://github.com/nikic/PHP-Parser/`
[5] `http://www.program-transformation.org/PHP/PhpFront`

Writing Analysis Rules Analysis rules are written as Rascal functions using a combination of switch statements and parameter-based dispatch. Each function takes at least the current configuration and a piece of abstract syntax, returning the updated configuration and (often) a partial analysis result, such as the type set computed for an expression.

The functions that use dynamic dispatch look like rewrite rules from a certain perspective, while the code that uses switch has a more procedural style. For example:

```
Infer plus(Cfg c, int(), int())   = <c, {int()}>;
Infer plus(Cfg c, int(), float()) { return <c, {float()}>; }
Infer plus(Cfg c, int(), str()) = <warn(c, msg),{int(),float()}>;
default Infer plus(Cfg c, Type l, Type r)
  = <err(c,...),{int(),float()}>
```

This code defines the `plus` function in four independent declarations. Each definition is overloaded —mutually exclusively— using pattern matching. The second definition is written using a block of statements to demonstrate the two different styles of function definition. The last definition has the **`default`** keyword, which indicates that it will be tried only after the others have failed to match the parameters passed into the call. The definitions use different kinds of data-types, namely algebraic data types (the updated configuration returned from a call to `warn(c, msg)`), tuples (`<a,b>`), and sets (using the `{...}` brackets).

The current implementation of the analysis rules has several limitations:

- The handling of "variable" constructs (variable-variables, variable functions, accesses to variable properties, etc) is currently too conservative, weakening the precision of the analysis. For instance, assigning a new type to a variable accessed through a variable-variable assigns this type to all variables currently in scope.
- The analysis can be overwhelmed as PHP scripts get larger, with both memory usage and processor usage growing to make the analysis infeasible. The imprecision in the handling of the variable constructs makes this worse, since this dramatically increases the size and number of type sets and alias pairs.
- Since the initial focus was on checking for upgrade problems between PHP4 and PHP5, some PHP5 features are not yet analyzed. This includes interfaces, traits, closures, and gotos, although some similar constructs (`break` and `continue` statements, essentially structured gotos) are currently supported.

Because of these limitations, we are currently reimplementing the analyses with a focus on performance. We are also working towards supporting the newer features of PHP that we do not yet support.

Reporting Analysis Results As mentioned above, analysis results are recorded in annotations on the AST, and are also given back as part of the final configuration. The current model of sharing the results is to use annotations, since this allows arbitrary information to be added to each node in the AST.

We do not yet show types for PHP expressions and access paths in a graphical fashion (e.g., as hovers within the Eclipse IDE), but that would be the next step if we were constructing an IDE for PHP. To display errors and warnings, the easiest

way is to register them via the Rascal standard library module that gives access to the Eclipse Problem View. A typical low-brow way of producing readable results from the annotated AST would be to use Rascal's string templates to produce a readable list of error messages and warnings.

4.4 Analyzing PHP in Maude

Here we describe what would be needed to build rewriting logic semantics versions of the type and alias analyses described above. We assume that these analyses would be run using Maude.

Parsing PHP Scripts Since the Maude parser is not capable of parsing normal PHP scripts, we instead would need to use an external parser. We may use the same external parser that Rascal used before: the parser would generate terms, in prefix form, using an algebraic signature defined to represent the abstract syntax of PHP. Some of these terms would be "located", as described in Section 3 with the RLSRunner tool, allowing source location information to be reflected in any generated error messages. Connecting the PHP front-end with Maude is done most straightforwardly by creating a shell script wrapping a call to the parser and sending the result over a pipe to Maude.

Developing Internal Representations Using Maude, all internal representations are based around terms formed over an algebraic signature. The abstract syntax for PHP would most likely be given in mixfix, allowing rules[6] to be written over an abstract syntax that looks similar to the concrete syntax. Intermediate results and the final results would also be defined as terms, representing (for instance) sets of types or maps from program points to alias pairs.

In a computation-based rewriting logic semantics [35], the current configuration – i.e., the current state of the analysis, including all intermediate results – would also be defined algebraically, as a multiset of nested *cells*. These cells can contain information such as the current computation (i.e., the remaining steps of the analysis), the current map of program names to values or storage locations, or the set of available class definitions.

Writing Analysis Rules Using a computation-based RLS, analysis rules are written as transformations of the configuration, generally involving the computation. Most language features are handled by several rules: one rule breaks apart the language construct to evaluate its pieces, while the others use the evaluation results to compute a result for the entire construct. As an example, a type inferencer may include rules such as:

```
eq k(exp(E + E') -> K) = k(exp(E,E') -> + -> K) .
eq k(val(int,int) -> + -> K) = k(val(int) -> K) .
eq k(val(int,float) -> + -> K) = k(val(float) -> K) .
eq k(val(int,str)-> + -> K) = k(warn(...)-> val(int float)-> K) .
```

The first rule indicates that we need to evaluate expressions `E` and `E'` before we can compute the result of `E + E'`; `E` and `E'` are put at the front of the computation (`k`) to

[6] We speak here of rules in the generic sense, including both equations and rewriting logic rules.

indicate they should be evaluated next, while + is put in the computation as a marker to indicate the operation being performed. The second, third, and fourth rules then give several possible behaviors, based on the results of evaluating the operands. The second rule says that, if both operands are of type `int`, so is the result. The third rule says that, if the first is `int` and the second is `float`, the overall result is `float`. Finally, the last rule shown says that, if we are trying to add an `int` to a `string` (as occurs in examples in Figure 4), we should issue a warning (the text is elided as `...`), because this may not be the operation we were intending to perform. We should also then return a set (represented by juxtaposition) containing both `int` and `float`, since, based on the contents of the string, both are possible result types.

Reporting Analysis Results There are several options for reporting the results of the analysis. The most basic is to just examine the final configuration, which will have the type assignments, alias pairs, etc. inside it. We could also use the semantics to perform a final "pretty printing" step to provide the results in string form. Other options include the use of external tools to view the results, such as was done with the RLSRunner tool in Section 3. Finally, we could return just a term with the results, not the entire configuration. This term could be used as input into another RLS-based tool which needed the computed results.

5 Related Work

In earlier work [45] we discussed the evolution of Rascal from its origins in the ASF+SDF and RSCRIPT systems. Some of the material in this paper, especially in Section 2, is based on this work, including the Rascal examples shown for illustration. Below we list other related work for Rascal, for PHP program analysis, and for analysis using rewriting logic.

Rascal: The design of Rascal is based on inspiration from many earlier languages and systems. The syntax features (grammar definition and parsing) are directly based on SDF [18], but the notation has changed and the expressivity has been increased. The features related to analysis are mostly based on relational calculus, relational algebra and logic programming systems such as Crocopat [5], Grok [26] and RSCRIPT [31], with some influence from CodeSurfer [2]. Rascal has strongly simplified backtracking and fixed point computation features reminiscent of constraint programming and logic programming systems like Moreau's Choice Point Library[36], Prolog and Datalog [8]. Rascal's program transformation and manipulation features are most directly inspired by term rewriting/functional languages such as ASF+SDF [48], Stratego [6], TOM [3], and TXL [12]. The ATerm library [47] inspired Rascal's immutable values, while the ANTLR tool-set [39], Eclipse IMP [9] and TOM [3] have been an inspiration because of their integration with mainstream programming environments.

PHP Analysis: Most research on PHP analysis has focused on detecting security vulnerabilities, including SQL injection attacks, cross-site scripting, and the use of tainted data (data that comes from outside the program, such as from a user form, and

that is not checked before being used in file writes, database queries, etc). This is the main focus of both the WebSSARI [27,28] and the Pixy [30,29] systems and of a number of individual analyses [41]. The PHP-sat[7] and PHP-tools[8] projects extend this security validation research by also adding support for additional analyses, including detecting a variety of common bug patterns (e.g., assigning the result of a function call where the body of the called function does not include a return statement), and by finding some PHP4 to PHP5 migration errors (e.g., functions with names that match new PHP5 functions). Another tool, the prototype PHP Validator [7], uses a type inferencer as part of a number of possible analyses. For instance, one example given is an analysis to detect the accidental use of + instead of . in string concatenation. However, not all of the analyses listed are actually implemented, and the PHP Validator tool does not support the object-oriented features of PHP.

Analysis in Rewriting Logic: Rewriting logic has been used extensively for program analysis. The work most similar to that discussed here is the work on policy frameworks for C [19] and SILF [24]. This work, in turn, was based on earlier work on detecting units of measurement errors in C [42] and BC [10] programs. Taking another approach, JavaFAN [15] uses Maude's state space search and LTL model checking facilities [13] to find program errors, including possible deadlocks in concurrent programs. A semantics of C [14], developed using K [43], uses standard Maude rewriting to run C programs and look for undefined behavior; state space search and model checking are used to explore the nondeterminism introduced by constructs with undefined evaluation orders.

6 Summary and discussion

We have shown a range of scenarios for building new software analysis tools in Rascal: from a purely Rascal-based solution to solutions that make use of external tools. This has been illustrated through a number of examples, including one showing the integration of Rascal with an existing rewriting logic-based analysis. The most substantial example has been the analysis of PHP code, in which we outlined the necessary steps for such an analysis, their implementation in Rascal, and a speculative implementation in Maude. We now summarize our observations and conclusions.

6.1 Observations regarding Rascal

Rascal is a programming language that provides all the features and libraries needed to create end-to-end software analysis tools. Although it has its roots in algebraic specification [4] and can be used to write programs that strictly follow the rewriting paradigm, it shows its major strengths in developing Rascal-based or hybrid solutions that cooperate with an IDE or with external tools.

[7] `http://www.program-transformation.org/PHP/PhpSat`
[8] `http://www.program-transformation.org/PHP/PhpTools`

Aspect	Maude	Rascal
Computational Model	Simpler	More complex
Semantics	Formal	Informal
Extensibility (language level)	Yes	No
Integration external tools	Only sockets, pipes	Java & Eclipse
Grammar definition & parsing	Limited	Fully integrated
Data types	Limited	Rich
Libraries	Limited	Rich
Maturity	Mature framework	Young language
Efficiency	Optimized	Not yet optimized

Table 3. Comparison of Maude and Rascal.

6.2 Comparing Maude and Rascal: General Observations

Our global experience with the two languages makes clear that both Maude and Rascal have competing benefits and shortcomings, which we summarize in Table 3.

Maude is closer to semantic foundations, with a formal semantics and a rewriting-based computational model. This makes the analysis itself amenable to formal techniques and provides a simpler conceptual core. Maude provides flexible extensibility mechanisms and (by treating computations as first class entities) powerful ways to manipulate partial computations. This can be especially useful when analyzing features that "jump", such as gotos, exceptions, and loop break and continue statements. Maude is a mature framework, with an optimized implementation, but it can still be challenging to write and maintain large specifications. Debugging tools for Maude [40] have made great progress but still have trouble scaling to large specifications, which often leads to debugging by reading through rewriting traces.

Rascal is more pragmatic, with a more complex, unformalized computational model. The Rascal language definition framework provides very strong integrated grammar definition and parsing facilities, rich data types that provide element generation, and traversal and pattern matching constructs. For integration, Rascal also provides rich libraries (e.g., supporting visualization, statistics, and data formats like HTML, XML, CSV, and SVN) and seamless extension and integration facilities with Java, Eclipse and external tools. Since Rascal is a young language, its design is not yet completely finalized and its implementation is not yet optimized, leading to potential performance problems when tackling large analysis problems.

6.3 Comparing Maude and Rascal: Observations for PHP Analysis

More specific differences can be identified on the basis of the PHP analysis.

Parsing PHP Scripts Here Rascal is clearly superior since both a completely Rascal-based parser and an externally implemented parser can be used. Also, more control is possible over the shape of the resulting trees.

Developing Internal Representations Rascal provides many more constructs, like maps, trees, and n-ary tuples and relations, "out of the box", making it easier to quickly develop the needed internal representations and to treat them as black boxes.

Writing Analysis Rules One perspective is that the same term and rule-based style can be used for writing specifications in both approaches. Specifications written in this style (this would be a functional style of programming in Rascal) are more amenable to formal analysis. This could be useful if, for instance, one wanted to prove that the type inference algorithm does not infer incorrect types. However, given the informal nature of PHP, it is not clear how possible it would be to conduct such proofs. Another perspective is that, if this level of rigor isn't needed, Rascal provides a number of programming language features that enable a richer variety of programming styles. Jumps can be modeled in Maude using K-style computations and in Rascal using either edges in graph-like program representations or using higher-order functions.

Reporting Analysis Results Maude and Rascal can both return an annotated term or string representing the analysis results. Rascal also includes facilities for error reporting and IDE integration built directly into the language.

6.4 Final Observations

We can draw several conclusions from this comparison. The bottom-line is that Maude is focused on formal specification while Rascal is focused on programming.

Maude is a better choice when the formal properties of the implemented tools are also important. Maude may be a better choice when a language has a number of jump-like constructs; these can be handled in Rascal, but require making the control context more explicit. Rascal is a better choice for end to end solutions, for instance real language environments, where parsing, integration with IDEs, and integration with external tools becomes important. With more standard control flow mechanisms, richer built-in data types, and a provided unit test definition and execution mechanism, large Rascal programs should also be easier to debug than large rewriting logic specifications for the same analysis. Finally, as shown with the RLSRunner tool, in some cases it may make sense to write analyses partly in Rascal and partly in Maude, such as when an analysis semantics already exists, or can be derived as an extension of an existing semantics, and can then be used as part of a Rascal-developed language environment.

References

1. O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of ECOOP'94*, volume 952 of *LNCS*, pages 2–26. Springer, 1995.
2. P. Anderson and M. Zarins. The CodeSurfer Software Understanding Platform. In *Proceedings of IWPC'05*, pages 147–148. IEEE, 2005.
3. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In *Proceedings of RTA'07*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
4. J. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press, 1989.
5. D. Beyer. Relational programming with CrocoPat. In *Proceedings of ICSE'06*, pages 807–810. ACM Press, 2006.
6. 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.

7. P. Camphuijsen. Soft typing and analyses of PHP programs. Master's thesis, Universiteit Utrecht, 2007.
8. S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., 1990.
9. P. Charles, R. M. Fuhrer, S. M. Sutton Jr., E. Duesterwald, and J. J. Vinju. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. In *Proceedings of OOPSLA'09*, pages 191–206. ACM Press, 2009.
10. F. Chen, G. Roşu, and R. P. Venkatesan. Rule-Based Analysis of Dimensional Safety. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 197–207. Springer, 2003.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
12. J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.
13. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In *Proceedings of WRLA'02*, volume 71 of *ENTCS*. Elsevier, 2002.
14. C. Ellison and G. Roşu. An Executable Formal Semantics of C with Applications. In *Proceedings of POPL'12*, pages 533–544. ACM Press, 2012.
15. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
16. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
17. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
18. J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
19. M. Hills, F. Chen, and G. Roşu. A Rewriting Logic Approach to Static Checking of Units of Measurement in C. In *Proceedings of RULE'08*. Elsevier. To Appear.
20. M. Hills, T. F. Şerbănuţă, and G. Roşu. A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In *Proceedings of WRLA'06*, volume 176 of *ENTCS*, pages 215–231. Elsevier, 2007.
21. M. Hills, P. Klint, T. van der Storm, and J. J. Vinju. A Case of Visitor versus Interpreter Pattern. In *Proceedings of TOOLS Europe 2011*, volume 6705 of *LNCS*, pages 228–243. Springer, 2011.
22. M. Hills, P. Klint, and J. J. Vinju. RLSRunner: Linking Rascal with K for Program Analysis. In *Proceedings of SLE'11*, volume 6940 of *LNCS*, pages 344–353. Springer, 2011.
23. M. Hills, P. Klint, and J. J. Vinju. Scripting a Refactoring with Rascal and Eclipse. In *Proceedings of WDT'12*. ACM Press, 2012. To appear.
24. M. Hills and G. Roşu. A Rewriting Logic Semantics Approach To Modular Program Analysis. In *Proceedings of RTA'10*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 151 – 160. Schloss Dagstuhl - Leibniz Center of Informatics, 2010.
25. M. Hind, M. G. Burke, P. R. Carini, and J.-D. Choi. Interprocedural Pointer Alias Analysis. *ACM TOPLAS*, 21(4):848–894, 1999.
26. R. C. Holt. Grokking Software Architecture. In *Proceedings of WCRE'08*, pages 5–14. IEEE, 2008.
27. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of WWW'04*, pages 40–52. ACM Press, 2004.
28. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Verifying Web Applications Using Bounded Model Checking. In *Proceedings of DSN '04*, pages 199–208. IEEE, 2004.

29. N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of PLAS'06*, pages 27–36. ACM Press, 2006.
30. N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, pages 258–263, 2006.
31. P. Klint. Using Rscript for Software Analysis. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008.
32. P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, pages 168–177. IEEE, 2009.
33. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.
34. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
35. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
36. 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.
37. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
38. W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.
39. T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.
40. A. Riesco, A. Verdejo, and N. Martí-Oliet. A Complete Declarative Debugger for Maude. In *Proceedings of AMAST'10*, volume 6486 of *LNCS*, pages 216–225. Springer, 2010.
41. A. Rimsa, M. d'Amorim, and F. M. Q. Pereira. Tainted Flow Analysis on e-SSA-Form Programs. In *Proceedings of CC'11*, volume 6601 of *LNCS*, pages 124–143. Springer, 2011.
42. G. Roşu and F. Chen. Certifying Measurement Unit Safety Policy. In *Proceedings of ASE'03*, pages 304 – 309. IEEE, 2003.
43. G. Roşu and T. F. Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
44. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In *Proceedings of ECOOP'03*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.
45. J. van den Bos, M. Hills, P. Klint, T. van der Storm, and J. J. Vinju. Rascal: From Algebraic Specification to Meta-Programming. In *Proceedings of AMMSE'11*, volume 56 of *EPTCS*, pages 15–32, 2011.
46. M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proceedings of CSMR'07*, pages 331–332. IEEE, 2007.
47. M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.
48. 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, *Proceedings of CC '01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.