# Meta-Language Support for Type-Safe Access to External Resources

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.** Meta-programming applications often require access to heterogeneous sources of information, often from different technological spaces (grammars, models, ontologies, databases), that have specialized ways of defining their respective data schemas. Without direct language support, obtaining typed access to this external, potentially changing, information is a tedious and error-prone engineering task. The Rascal meta-programming language aims to support the import and manipulation of all of these kinds of data in a type-safe manner. The goal is to lower the engineering effort to build new meta programs that combine information about software in unforeseen ways. In this paper we describe built-in language support, so called *resources*, for incorporating external sources of data and their corresponding data-types while maintaining type safety. We demonstrate the applicability of Rascal resources by example, showing resources for RSF files, CSV files, JDBC-accessible SQL databases, and SDF2 grammars. For RSF and CSV files this requires a type inference step, allowing the data in the files to be loaded in a type-safe manner without requiring the type to be declared in advance. For SQL and SDF2 a direct translation from their respective schema languages into Rascal is instead constructed, providing a faithful translation of the declared types or sorts into equivalent types in the Rascal type system. An overview of related work and a discussion conclude the paper.

## 1 Introduction

Software language engineers, such as those working in the grammarware, modelware, or ontologyware domains, write meta-programs. These programs, used for tasks such as calculating software metrics, performing static analysis, mining software repositories, and building IDEs, use information provided by a wide number of different sources. While some of these are internal to the application, some may be external, e.g., the contents of software repository commit messages, the values in a database storing bug reports, or an already-defined grammar for the language being manipulated. In general, this external information may be created using a number of different tools, each with its own data formats and methods of storing data. These data formats can be defined explicitly, such as with a table definition in a database or an SDF2 [7,16] definition for a grammar, but may also be defined implicitly, such as is found in RSF (Rigi Standard Format) files,

which provide a textual representation of binary relations (see Section 4.1), and in CSV (Comma-Separated Values) files, which provide a textual representation of tabular data.

Without direct language support for accessing external data sources, different libraries have to be developed, each targeted at a different data source. Each library provides a different mechanism for naming these data sources and determining the type of data stored in a resource, including how the external values and types map into the values and types of the host language. Actually writing out these types is a manual task, and synchronization in cases where the underlying types change (e.g., a database table that has been altered, a CSV file with added columns) also must be performed manually. The more sizable and complex data sources described by Bugzilla's entity/relationship (E/R) model are a case in point. This paper is about managing these repetitive and error-prone software language engineering tasks in the context of the Rascal meta-programming language [10].

We describe built-in Rascal language support for incorporating external sources of data (identified by URIs) and their corresponding types: *resources*. Resources allow the meta-programmer to generate type declarations statically, when a resource is imported, before type-checking takes place. This resources feature builds on essential Rascal language features, described in Section 2. This discussion of enabling features is then followed by the main contributions of this paper, given in Sections 3 and 4. Section 3 discusses the design of the Rascal resources language feature, while Section 4 demonstrates the use of resources through four case studies, including the application of resources to typical software engineering tasks. Sections 5 and 6 then close, presenting related work and a final discussion with ideas for future work, respectively.

## 2 Enabling Rascal Features

To explain Resources, we first discuss four key enabling Rascal features: *type literals* that allow types to be treated as values, *source location literals* that provide access to external resources via Uniform Resource Locators (URIs), *string templates* for code generation, and the *Rascal-to-Java bridge* to connect arbitrary Java libraries to Rascal.

### 2.1 Type Literals

**The Rascal type system** provides a uniform framework for both built-in and user-defined types, with the latter defined for both abstract datatypes and grammar non-terminals (also referred to as *concrete* datatypes). A built-in tree datatype (**node**) acts as an umbrella for both abstract and concrete datatypes. The type system is based on a type lattice with **void** at the bottom and **value** at the top (i.e., the supertype of all types). In between are the types for atomic values (**bool**, **int**, **real**, **str**, **loc**, **datetime**), types for tree values (**node** and defined abstract and concrete datatypes), and composite types with

typed elements. Examples of the latter are **list[int]**, **set[int]**, **tuple[int,str]**, **rel[int,str]**, and, for a given non-terminal type `Exp`, **map[Exp,int]**. Sub-typing is always covariant with respect to these typed elements; with functions, as is standard, return types must be covariant, while the argument types are instead contravariant. For example, for sets, **set[int]** is a subtype of **set[value]**, while for functions, **int(value)** is a subtype of **value(int)**.

**Formal type parameters** allow the definition of generic types and functions. All non-atomic types can have explicit *type parameters*, written either as `&T` or `&T <: Bound`. The former can be bound to any Rascal type, the latter only to subtypes of the type `Bound`. For example, **rel[&T,&T]** defines a generic binary relation type over elements of the same type, **list[&T <: num]** defines a list with elements that can only be one of the subtypes of the type **num**, and **list[&T] reverse(list[&T] L)** defines the type of a function with the name `reverse` that returns a list with the same element type as its argument `L`.

**Reified types** make it possible to manipulate types as ordinary values that can be passed around, queried and manipulated. Rascal's reification operator creates *self-describing* type values which contain both the reified type and all datatypes used in this type. A type can be reified using the prefix reification operator (#); we call such an expression a *type literal*. A reified type value contains a symbol to represent the type and a map of definitions for any abstract or concrete datatype dependencies. It is guaranteed to have the type **type[&T]**, where the type parameter `&T` is bound to the type that was reified. For example:

- **#int** produces a literal value `type(\int(),())` of type **type[int]**.
- **#rel[int,str,bool]** produces `type(\rel([\int(),\str(), \bool()]),())` of type **type[rel[int,str,bool]]**.

The **type** data constructor used to build type literals is built in to Rascal; the representations for type symbols and their definitions are defined as Rascal datatypes in a library module. Above, the map of definitions was empty: `()`. For abstract or concrete datatypes this map will contain the complete (possibly recursive) abstract datatype or grammar. Assume a definition for Boolean connectives:

```
data Bool = and(Bool l, Bool r) | t() | f();
```

then the reified type `#Bool` will produce the following term of type **type[Bool]** (some details have been elided):

```
type(adt("Bool"),
     (adt("Bool"):choice(...,constructor(adt("Bool"),"and",
           [label("l",adt("Bool")),label("r",adt("Bool"))]),...)))
```

Such self-describing type values are particularly useful in the context of defining resources, where we want to import, and compute types for, otherwise untyped or unknown data from outside of Rascal. Using type literals we can write library functions that bind arbitrary (external) data to specific types.

### 2.2 Source Locations

Rascal provides built-in support for location literals (values of type **loc**) that are Uniform Resource Identifiers[1] (URIs) optionally followed by text coordinates that allow the identification of specific text ranges in the information the URI points at. Location literals are quoted with bars, such as `|http://www.rascal-mpl.org|`.

In addition to the standard schemes like `file` (local file access) and `http` (remote file access), a number of Rascal-special schemes are supported such as `cwd` (current working directory), `home` (the user's home directory), `std` (the Rascal standard library), `jar` (an entry in a jar file), and `project` (an Eclipse project). The collection of schemes is openly extensible – the extension implements a contribution interface in Java.

The location datatype conveniently provides direct access to parts of the URI and gives short-hands to interact with file systems and web pages. Source locations in Rascal are very versatile and are, for instance, used for tasks such as accessing source code locations in editors and providing hyperlinking functionality in the IDE. In the context of the resources feature, we use them to identify external sources of type information.

### 2.3 String Templates and Concrete Syntax Templates

Rascal provides both string templates and concrete syntax templates for code generation, a frequently occurring operation in meta-programming. String templates are multi-line string literals with a left-margin, interpolation of arbitrary expressions, auto-indentation, and structured control flow. For example, the following code generates the definition of a Java class named `name` with a number of fields (given as `name`×`type` pairs in relation `fields`), all indented by 2 spaces:

```
str class(str name, rel[str,str] fields) =
  "class <name> {
  '  <for (<f,t> <- fields) {><t> <f>;
  '<}>
  '}";
```

Concrete syntax templates are parsed fragments of code, used for pattern matching and pattern construction. Concrete syntax fragments are supported for languages that have a grammar defined in Rascal. For example:

```
import lang::rascal::syntax::Rascal;

Module m = ‘module M imports N; ...‘;
```

The fragment within the backquotes will be parsed using the grammars defined in the current scope (here, the imported grammar of Rascal). Concrete syntax fragments allow for anti-quoting to expand variables or to match and bind parts using pattern matching. The benefit of concrete syntax fragments is that both generated code and patterns are statically guaranteed to be syntactically correct.

[1] See `http://www.ietf.org/rfc/rfc3986.txt`.

### 2.4 Rascal-to-Java Bridge

The Rascal-to-Java Bridge makes it possible to call Java functions from Rascal code and to build Rascal data values in Java code. Rascal users can extend their library reusing existing Java code or building on top of the Java standard library. This enables, for example, reuse of JDBC libraries, open Java compilers, SMT solvers, and the Apache Math library. The author of a library written in Java is responsible for producing Rascal data of the right type. Consider the `size` function for lists:

```
@javaClass{org.rascalmpl.library.Prelude}
public java int size(list[&T] lst);
```

The modifier `java` indicates that the function `size` is written in Java and the annotation `javaClass` defines in which class the method `size` can be found. The function is then implemented by the following Java code:

```
public class Prelude {
  IValueFactory vf;
  ...
  IInteger size(IList lst) {
    return vf.integer(lst.length());
  }
}
```

The Java API `IValueFactory` makes it possible to construct arbitrary Rascal values. If the returned type does not match the return type of the associated Rascal function, a run-time type exception will occur, ensuring this mechanism cannot be used to break type safety.

## 3 Rascal Resources

Given the above motivation and prerequisites, we explain the core contribution of this paper: the design and realization of user-defined resources in Rascal.

### 3.1 Design of Resources

There are four requirements for resources in Rascal. First, resources should be accessible with a uniform naming scheme. Second, access to resources should be statically typed. Third, the types of resources should be transparently observable to the Rascal programmer. Fourth, where possible, resources should be implemented directly in Rascal. We discuss the first three requirements below.

**Uniform naming scheme:** Many of the libraries for accessing external sources of data in Rascal use their own naming schemes to refer to the sources of the data. For instance, file-based resources tend to use the location of the file, encoded as a Rascal location. By contrast, JDBC resources use JDBC *connect strings*, strings encoding the information needed to connect to the database (host, database,

user id, etc), and built either directly as strings or using driver-specific functions that accept the proper parameters. Resources based on data retrieved in JSON format over HTTP may encode the query URL directly.

To obtain a uniform naming scheme, we took as inspiration work on both the Unix [14] and Plan 9[2] operating systems. Unix introduced a major innovation in the handling of I/O by providing a uniform interface for many input and output sources. This allowed special devices, such as terminals, and pseudo-devices to be treated like files, given they provided implementations of operations such as `read` and `write`. Plan 9 took this even further and allowed each resource (including processes and network resources) to have a unique path name to be accessed uniformly. This goal of providing a uniform addressing mechanism can also be realized by the URIs that we already use in Rascal locations, as described above.

It is natural to use Rascal's existing location values to identify and locate external sources of data and types. These external sources introduce new schemes, which bind to functionality for interpreting specific kinds of external data sources (like comma separated values or JDBC data sources). The authority, path and query components serve to identify, unambiguously, which particular source of data is imported and to provide all necessary parameters to do so. We use a `+` sign to split the scheme into a logical and a physical part. Two examples of locations that identify resources are:

- `|csv+file:///Users/foo/projects/data.csv|` uses a CSV resource to access a file in a folder of user foo.
- `|sdf:///languages/ansi-c/syntax/Main|` uses an SDF resource to import a grammar with main module `languages/ansi-c/syntax/Main` (using an implicit search path, see below).

The format of the information in the location is dependent on the resource—the only requirement is that it be representable as a URI.

**Access to resources is statically typed:** As already discussed, the Rascal type system provides ways to introduce type-checked identifiers, such as names of abstract datatypes and their fields (**data** `Person = person(`**str** `name,` **int** `age)`) and relations with named columns (**rel**`[AST class,` **int** `NCLOC]`). Types not only provide safety, they also provide access via meaningful identifiers to project, select and update parts of datatypes.

Without the resources feature we could already (easily) provide general access to external data. In that case, all imported data is of type **value**. To analyze and manipulate such data, pattern matching is needed extensively, as well as indexing into containers using anonymous "magic" constants. The next best thing is to use type literals (described above). If the client code of a library that imports external data provides a specification of the expected type, then at least the client code can be made type correct and use the appropriate API. For example, we can write: `readCSV(#rel[str name, int age], |home:///people.csv|)` to obtain a typed API to a person/age database stored in a CSV file. The first

[2] See `http://cm.bell-labs.com/plan9/`.

step in providing a Rascal resource is to create such a generic library with read functions that are parameterized with the expected return type.

Type literals may solve the type safety issue with external data sources, but the heavy-lifting is in the client code that provides a complete type specification for the external data. For a CSV file with twenty columns, one needs to manually infer a type literal from the CSV file that reflects their types. For an SQL database, one needs to come up with corresponding relation types for every table (imagine a relatively simple situation with 20 tables with 5 columns each). For an external grammar formalism, one needs to port each non-terminal into a Rascal syntax definition. Moreover, all these schemas are subject to evolution and maintenance, leading to cumbersome co-evolution between a data source and its reflection in the Rascal type system. Finally, the semantics of the mapping from one type system to another may require a sizable intellectual effort—recall the "impedance mismatch" between object-oriented (OO) and E/R representations of data.

Our solution to this remaining problem is to apply *code generation* at module import time. One code generator is needed for each kind of foreign data, e.g., CSV or JDBC. Such a code generator is needed in the standard library for each common kind of foreign data, and is reusable for every external data source of this kind. The generated code typically makes use of generic library code accepting type literals as arguments. A designated user-defined code generator generates all required type definitions and interface functions automatically. Since what is generated is standard Rascal code, like a a user would themselves write, the code can then be type checked by the Rascal type checker when the module is loaded, before execution. Public definitions in the generated module provide the module signature used during type checking inside the importing module, ensuring the resource is used safely. Since resources are treated as standard Rascal values, the type checker has the same limitations as with other Rascal code: most type errors are caught statically, while a few (e.g., missing fields on constructor values) are caught using dynamic checks that throw exceptions in error cases.

The reuse of code generators solves the problem of inferring complex type definitions for every new data source as well as their co-evolution. Type-safety is provided as well as an appropriate API. What remains is the specialized, one-time design of a code generator for every new kind of external data. This design influences how many static guarantees can be made and which kinds of co-evolution will go detected. For example, if a CSV resource generator does not support column names, then swapping two columns with the same inferred types in a file will go undetected by Rascal's type system.

**Resource types are transparent:** The types of external resources can be provided implicitly by way of type-inspection APIs, or explicitly by providing them as complete type definitions. We opt for the latter, since we believe that this is easier to understand for the programmer. By design, the resource generators produce the source code of full Rascal modules, containing all type definitions and access functions. The user can read the generated code like any other Rascal module, debug it if necessary, and use the same IDE support for browsing and querying Rascal modules, yet does not need to maintain it manually. Because of

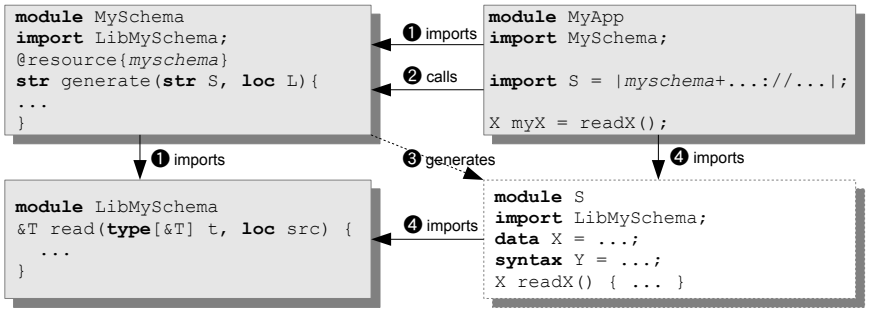


**Fig. 1.** Processing the import of an external resource using a URI scheme.

the complexity of the mapping between external and internal types and values, we believe that the ability to interact with the generated source code is an essential engineering prerequisite.

### 3.2 Syntax and Semantics of Resources

As depicted in Figure 1, to extend Rascal with a new resource type (say `myschema`) the following ingredients are needed:

- A library, e.g. `LibMySchema`, that handles the basic access to and communication with the external resource, using type literals.
- A Rascal module, e.g. `MySchema`, that contains the code generator for the `myschema` resource. The code generator function takes two arguments and is declared to work for a specific schema using a tag: `@resource{myschema}`.
- Rascal client code, e.g. `MyApp`, which imports a to-be-generated resource module using this notation: **import** `S = |myschema+...://...|;` Since it is a `myschema` resource, it starts with `myschema`. This schema name connects the use of a resource with the proper generator that has been defined for that resource.

Given these ingredients, the following steps are executed when the `MyApp` application is loaded (Figure 1):

❶ First all non-resource module dependencies are loaded. This means the `MySchema` module that implements the `myschema` resource is loaded and its declarations are executed. When the declaration of the `generate` function is evaluated, the `@resource` tag triggers the binding of `myschema` to this function. Typically the generator depends on a `LibMySchema` module to get access to the external type information, which is imported now transitively.

❷ Then the resource modules are loaded. For every specific schema used, in this case only `myschema`, its associated generator function is called providing the name of the module to be generated (`S`) and the user-provided URI (`L`)

as arguments. Resource modules are loaded after non-resource modules to ensure that the generator functions have been registered; this restriction could be relaxed, but only at the cost of a more detailed analysis to ensure that all needed generator functions are imported before the resources that use them.

❸ As a result of the call, the code generator uses the information provided in the URI to acquire and manipulate external data sources, eventually generating the source code of a statically correct module with the given name.

❹ Finally, the generated module `S` is imported, like any normal module, by the client application, which may trigger more imports if the generated module imports library modules. Note that generated modules may contain resource imports themselves, which may be useful in cases of importing resources that are themselves structured modularly.

After these steps module `MyApp` can use the freshly created API of module `S`.

## 4 Sample Resources for Software Language Engineers

When a software language engineer is studying a large software system he/she wants to tap into as many relevant information sources as possible and wants to integrate their contents into a unified result. Examples are bug reports that are stored in a relational database, metrics stored in a spreadsheet, a call graph stored in some textual format, or a complete grammar stored in an external grammar file. In the following sections we demonstrate that all these needs can be addressed by Rascal's resources concept.

### 4.1 Rigi Standard Format (RSF)

Rigi Standard Format[3] is the main file exchange format for the Rigi system [13] and is used to describe binary relations. Although RSF has been superseded by richer formats such as GXL[4] and GraphML[5] we use it here for its simplicity. An RSF file contains triples of the form `verb subject object` and can simultaneously define several binary relations. The verb part of a triple determines the relation to which the triple contributes. Here is an example:

```
call    main          printf
call    main          listcreate
data    main          FILE
data    listcreate    List
file    listcreate    list.c
lineno  listcreate    10
lineno  main          150
...
```

[3] See `http://www.rigi.cs.uvic.ca/downloads/rigi/doc/node52.html`.
[4] See `http://www.gupro.de/GXL/`.
[5] See `http://graphml.graphdrawing.org/`.

```
@resource{rsf}
public str generate(str moduleName, loc src) {
  map[str, type[value]] rels = getRSFTypes(src);

  return "module <moduleName>
         'import lang::rsf::IO;
         '<for(rname <- rels) {>
         'public <rels[rname]> <rname>() {
         '  return readRSFRelation(#<rels[rname]>, \"<rname>\", <src>);
         '}<}>"';
}
```

**Fig. 2.** Resource generator for RSF.

These triples define the relations `call`, `data`, `file` and `lineno`. The first three will correspond to Rascal relations of type **rel[str,str]**, while `lineno` will get type **rel[str,int]**. In order to create support for RSF resources, two steps are needed. First, basic I/O functionality is needed to support the resource format itself. In the case of RSF, we have extended the existing RSF library `lang::rsf::IO` to support resources. There are two essential functions:

- **map[str, type[value]]** `getRSFTypes(`**loc** `src)` returns a map of relation names and their inferred types. By default, all elements in the relations have type `str`, but consistent use of `bool`, `int`, or `real` values at fixed positions in the triples will lead to more precise typing (for instance, with `lineno` above).
- `&T readRSFRelation(`**type**`[&T] result, str name, loc src)`[6], given an expected type and a relation name, returns the typed relation with the given name for the RSF resource at location `src`.

Using this, we can create a generator that supports RSF. A simplified version is shown in Figure 2. It essentially processes the given location, extracts the relation names and types from the RSF triples at that location, and uses string templates to generate a Rascal module with declarations for typed functions to extract the various relations from the RSF triples.

Next, we illustrate RSF resources using extracted facts from JHotDraw[7]:

```
CALL AbstractConnector_2354 Figure_1715
CALL AbstractConnector_2354 Geom_3544
INHERITANCE AbstractConnector_2354 Connector_1478
CONTAINMENT AbstractConnector_2354 Figure_1715
CALL AbstractFigure_2788 ChopBoxConnector_2286
...
```

An RSF resource for this data is created by:

```
import RSF;
import JHotDraw52 = |rsf+file:///Users/.../JHotDraw52.rsf|;
```

[6] `&T` could be replaced with **rel[&T1,&T2]** to enforce that the return type is a relation; this change will be made in a future version of the code.

[7] See `http://code.google.com/p/crocopat/source/browse/tags/crocopat-2.1.4/examples/projects/JHotDraw52.rsf`.

```
module JHotDraw52
import lang::rsf::IO;

public rel[str, str] CALL() =
  readRSFRelation(#rel[str, str], "CALL", |file:///Users/.../JHotDraw52.rsf|);

public rel[str, str] INHERITANCE() =
  readRSFRelation(#rel[str, str], "INHERITANCE", |file:///Users/.../JHotDraw52.rsf|);

public rel[str, str] CONTAINMENT() =
  readRSFRelation(#rel[str, str], "CONTAINMENT", |file:///Users/.../JHotDraw52.rsf|);
```

**Fig. 3.** Generated module `JHotDraw52`.

This will generate the module `JHotDraw52` shown in Figure 3. Finally, one can use this resource, for instance, by defining a function that reads the `CALL` relation from it (accessible using function `CALL`) and computes its transitive closure:

```
rel[str, str] indirectCalls() = CALL()+;
```

The bottom-line is that Rascal can handle the untyped data in an RSF file in a fully type-safe manner.

### 4.2 Comma-Separated Values (CSV)

The CSV format was originally intended for exchanging information between spreadsheets and databases but is today used as an exchange format in many other application domains as well. A CSV file has the following structure:

- a header line consisting of field names separated by commas;
- one or more lines consisting of values separated by commas.

The CSV format differs in various respects from the RSF format:

- RSF can define several relations at once; CSV can define only one relation.
- RSF only supports binary relations; CSV supports relations of arbitrary arity.
- The RSF format is fixed; in a CSV file, the header line is optional and the default separator (comma) can be redefined.

The Rascal `lang::csv::IO` library supports the standard CSV format[8] and has been extended to support CSV resources.

The major challenge compared to RSF resources is to handle the variability mentioned above. Our solution is to use the standard query parameters in the URI that describes the location of the CSV data. In the following example we want to process the metrics collected by the Eclipse Metrics Plugin[9]. The relevant data are collected in the file `methods.csv` (a run of the metrics plugin on the source code of the Rascal system itself) in the user's home directory, and we want the function for reading the metrics resource to be named `METHOD_METRICS`:

```
import CSV;
import METHODS = |csv+home:///methods.csv?funname=METHOD_METRICS|;
```

```
module METHODS
import lang::csv::IO;

alias METHOD_METRICSType = rel[str PACKAGE, str TYPE, str METHOD, int LINE, int NOL, int NOS,
                               int FE, int NLS, int NOP, int CC, int LOCm];

public METHOD_METRICSType METHOD_METRICS() =
  return readCSV(#METHOD_METRICSType, |home:///methods.csv|, ());
```

**Fig. 4.** Generated module `METHODS`.

This will generate the `METHODS` module shown in Figure 4. For convenience, the alias `METHOD_METRICSType` is created as an abbreviation for the actual relation type. The functions in the generated module `METHOD` can, for instance, be used in the following comprehension to compute the methods with the largest cyclomatic complexities (field `CC`):

```
{ <m.PACKAGE, m.METHOD, m.CC> | m <- METHOD_METRICS(), m.CC > 50 };
```

In this particular example, four methods were found, two of which are shown:

```
rel[str, str, int]: {
  <"org.rascalmpl.interpreter","reify",53>, ...
  <"org.rascalmpl.library.vis.util","unPrintableKeyName",59>
}
```

The full power of the relational calculus that is embedded in Rascal can now be used to further explore these metrics data in a type-safe manner.

### 4.3 Java Database Connectivity (JDBC)

A number of systems, such as Bugzilla, use relational databases to store information useful in language engineering tasks. For instance, information about bug reporters is stored in Bugzilla in a table named `profiles`, which contains data that conforms to the schema shown in Figure 5.

| Column | Type | Nullable |
|---|---|---|
| userid | mediumint(9) | N |
| login_name | varchar(255) | N |
| cryptpassword | varchar(128) | Y |
| realname | varchar(255) | N |
| disabledtext | mediumtext | N |
| mybugslink | tinyint(4) | N |
| extern_id | varchar(64) | Y |
| disable_mail | tinyint(4) | N |

**Fig. 5.** Schema for table `profiles`.

One popular way to gain access to this information in Java is to use JDBC, a standard Java API for querying, updating, and exploring the metadata of databases. Using Rascal's ability to call Java functions, we have written a JDBC library that allows JDBC calls to be made from within Rascal code. For instance, to connect to a Bugzilla database, select all the records from `profiles`, and then close the connection, the following code would be run:

[8] See `http://tools.ietf.org/html/rfc4180`.
[9] See `http://eclipse-metrics.sourceforge.net/`.

```
module Profiles
import JDBC;

alias profilesType = rel[int userid, str login_name,
  Nullable[str] cryptpassword, str realname, str disabledtext,
  int mybugslink, Nullable[str] extern_id, int disable_mail];

public profilesType resourceValue() {
  registerJDBCClass("com.mysql.jdbc.Driver");
  con = createConnection("jdbc:mysql://host/bugs?user=usr&password=pass");
  profilesType res = loadTable(#profilesType, con, "profiles");
  closeConnection(con);
  return res;
}
```

**Fig. 6.** Generated JDBC resource for the `profiles` table.

```
registerJDBCClass(mysqlDriver);
con = createConnection("jdbc:mysql://host/bugs?user=usr&password=pass");
res = loadTable(con,"profiles");
closeConnection(con);
```

The first line registers the proper JDBC driver, in this case for MySQL. The second line then actually creates the connection, using a JDBC connect string formatted according to the requirements of the MySQL JDBC driver. The third line loads the data in the table into `res`; since no type information is provided, the data is loaded as a set of values, which can then be de-constructed using pattern matching. Finally, the fourth line closes the connection.

It is possible to instead load a typed representation of the data with `loadTable`, which returns the data in a relation with named fields of the proper type. However, this requires computing the type manually. As was discussed in Section 3, determining the correct type literal is a non-trivial task, here made more difficult by the need to map from native MySQL types, to JDBC types, and then to Rascal types, along with the need to properly account for `null` values (Rascal has no equivalent of `null`, so a datatype `Nullable`, parameterized by the actual column type, is used instead). For this simple table, this process would derive the following type literal representing a row in the table:

```
#tuple[int userid,str login_name,Nullable[str] cryptpassword,str realname,
str disabledtext,int mybugslink,Nullable[str] extern_id,int disable_mail]
```

Two JDBC Resources are currently defined to provide type safe access to JDBC tables. The first, `jdbctable`, provides access to a specific table, while the second, `jdbctables`, provides access to all tables in a database. The following two `import` statements import the `profiles` table and all Bugzilla tables, respectively:

```
import Profiles=|jdbctable+mysql://host/bugs/profiles?user=u&password=p|;
import AllTables=|jdbctables+mysql://host/bugs?user=u&password=p|;
```

The first of the resource imports generates a module, `Profiles`, containing the code shown in Figure 6. The second import creates similar code for each table

```
P = profiles(); B = bugs(); S = bug_status();

deltaDurations = { < p.login_name, b.bug_id,
  createDuration(ts, b.delta_ts).days > | s <- S, s.id==5, p <- P, b <- B,
  b.assigned_to==p.userid, b.bug_status==s.value,
  notnull(ts) := b.creation_ts };

perPerson = { < p,
  ( 0 | it + d | <_,d> <- deltaDurations[p] ) / size(deltaDurations[p]) > |
  p <- deltaDurations<0> };
```

---

```
rel[str, int]: { <"person1",1>, <"person2",17>, <"person3",7>, ... , <"personN",8> }
```

**Fig. 7.** Compute average days/person to resolve a bug, using the JDBC resource.

in the database, with the table name used to give a name to the function used to retrieve the resource (e.g., `resourceValue` in Figure 6). Using the imported tables, one can then perform queries over the data. For instance, one may want to find the average number of days, per person, it takes from when a bug is assigned to when it is resolved[10]. This is done using the code shown in Figure 7.

The first line in Figure 7 extracts the relations stored in tables `profiles`, `bugs`, and `bug_status` into variables `P`, `B`, and `S`, respectively. Relation `deltaDurations` is then created using a comprehension, which enumerates all bug statuses; filters these to only include tuples with `id` 5 (status "resolved" in this Bugzilla database); enumerates all profiles; enumerates all bugs; and filters the bugs to include only those bugs assigned to the user represented by the profile, with a status the same as the current status, and with a non-null creation timestamp (checked using pattern matching, with a non-null timestamp represented as timestamp `ts` wrapped in the `notnull` constructor). For each matching combination of status, profile, and bug (after accounting for all the conditions just mentioned), a tuple is added to the computed relation containing the login name of the profile, the id of the bug, and the number of days between the creation timestamp and the timestamp of the last change to the bug information, which we assume here represents the date when the bug was resolved – i.e., a relation between logins, bug ids, and days to resolve the bug. `perPerson` is built in a similar way: for each login name `p`, the days related to `p` are summed, with the result divided by the number of records to compute the average. The resulting relation is shown, in part, in Figure 7 below the code, with login names made anonymous.

### 4.4 Syntax Definition Formalism

The Syntax Definition Formalism (SDF) [7,16] is an EBNF-like grammar formalism extended with disambiguation constructs. It is generally used to define both the concrete and the abstract syntax of software languages in the same

[10] Technically, this shows the average from the creation date to the delta date, which is the date of the last change to the bug information.

```
@resource{sdf}
public str generate(str name, loc at) {
   def = loadSDF2Module(at.host, [|rascal:///|]);
   gr = fuse(sdf2grammar(name, def));
   return "module <name>
          '
          '<grammar2rascal(gr)>
          ";
}
```

**Fig. 8.** Resource generator for SDF.

definition. There are many open-source grammars available written in SDF[11]. These grammars are complex engineering artifacts [9], especially if they are written with the intention to generate a syntactically and semantically correct parser. Since it is appealing to reuse such SDF grammars, we have implemented an SDF resource that can, for instance, import an SDF syntax definition of `Java5` into a Rascal module as follows:

```
import lang::sdf2::utils::Resource;
import Java5 = |sdf://languages/java/syntax/Java5|;
```

The resource generator is implemented as shown in Figure 8. This implementation deserves some explanation:

- We use the search path for Rascal modules to search for SDF files. The URI `|rascal:///|` represents the root of the entire Rascal search path which is passed to a function that will traverse all the imports of a modular SDF specification and produce a single syntax tree listing all relevant modules. Adding an SDF grammar to any Rascal project in the Eclipse workspace will make it available for use.
- Since SDF's module system has an entirely different semantics from Rascal's module system, modules in SDF can not map to modules in Rascal. The *fuse* function flattens the internal grammar such that SDF's module semantics are implemented[12].
- The `sdf2grammar` function (800 LOC in Rascal), in particular, attempts to maintain the semantics of SDF's disambiguation features. Bouwers et al. have described some of the intricacies of the semantics of disambiguation [3]. Some disambiguation features in SDF are more powerful than their counterparts in Rascal. Their semantics have been limited in the design of Rascal to make them easier to understand and debug. At the same time, Rascal has additional disambiguation features that can replace the earlier "mis-uses" of the power of SDF. The translation is intentionally *not* complete, such that features that do not map are documented in the resulting Rascal grammar. `sdf2grammar` is not fast, it was written for brevity and clarity first, with optimization as a later goal.

[11] See `http://www.syntax-definition.org`.
[12] SDF's renaming and module parameter features are not yet implemented.

The complexity of such a translation from one EBNF-based formalism to another may be daunting, but being able to reuse it transparently via the resource feature adds all the more value. It is particularly useful to be able to read the source code of the resulting Rascal module like any normal module. We also expect that further maintenance may take place in the generated Rascal modules. This can be achieved by replacing the above import of the Java5 resource by a direct import of the generated module `Java5`.

## 5 Related Work

Rascal resources fit in the field of interfacing data sources and programming languages. A large amount of related work exists on interfaces (libraries, code transformations, language extensions) for accessing external (especially database) resources. Because of space, we only discuss the most directly related work.

### 5.1 Scripting Languages

Scripting languages, like Rascal, often serve as "glue" between systems that need to be combined. Rascal has a static type system, while most other scripting languages (Python, Ruby, Perl) have dynamic type systems. Such languages have the advantage that data conversion between two systems can be limited to the shape of the data and not much time has to be spent on bridging type systems. Dynamically typed languages serve well as glue because they pose no a priori, static, limitations on the kind of data that can be processed. Our resources concept is unnecessary for dynamically typed languages, since there is no static type system to use in the first place.

One can use Rascal as a mostly dynamically typed language when no optional type declarations are used and all data is simply of type **value**, **list[value]**, **set[value]** or **node**. XML documents can, for instance, be represented in this way. In this style, one has to use pattern matching to analyze and transform these untyped data structures, effectively encoding the type system into the program.

Compared to this dynamically typed programming model using pattern matching, Rascal resources are at the other end of the spectrum and use external datatype definitions, bringing the external data into the typed world. Note that the generated code should be type-safe unless the underlying schema has changed since the last generation of the interface. In this unusual case, Rascal produces a run-time type error if the schema change results in a type change.

### 5.2 Object/Relational Mapping

There have been many attempts, in many language paradigms, at solving the *impedance mismatch* [11] between the representation of data in a relational database and in program values. This includes PLAIN [15], a Pascal-like programming language extended with statements to query a database, and the Microsoft ADO.Net Entity Framework [12], a data access framework for Microsoft's .Net

platform. [5] provides a comprehensive overview of the problems involved in integrating programming languages and databases.

In our work on Rascal resources we have not focused specifically on access to databases, and we have not yet attempted to optimize access, instead working with a model where all data of interest is loaded into Rascal and then manipulated using Rascal expressions (see Figure 7 for an example). However, there is nothing inherent to our solution that would prevent this optimization. Since we are most interested in working with existing data, we also do not yet support writing to databases, and would need a stateful interaction library, such as that developed for working with Maude-based analysis tools [8], to do so efficiently.

### 5.3 XML Binding and AST Generators

The problem of being able to use typed interfaces on data that is serialized in an untyped or otherwise foreign notation can be found in many places. In the XML domain, this is called *binding'* and generation of typed interfaces from XSD or DTD schemas is common practice [2,4,1]. In the (compiler) front-end domain we, and many others, have generated APIs for abstract syntax trees from grammars [6]. This is also a kind of data-binding.

### 5.4 LINQ

LINQ provides language support for type-safe SQL-like query syntax on external data-sources. By implementing a Provider, library authors can relatively easily add support for foreign data representations. A LINQ provider may even examine the syntax tree of a query and decide how to implement it.

Rascal resources share a similar goal, but the design is quite different. Instead of enabling an adapter to access the remote data, Rascal resources are about transforming external data into local data representations. LINQ provides, like Rascal, guarantees for type safety of the client code.

### 5.5 F#3.0 Type Providers

Rascal's resources also resemble the feature of *Type Providers* in F#3.0[13]. We briefly discuss commonalities and some significant differences.

*Type Providers* in F#provide a hook into the type system. The user may declare, using source code annotations, certain extensions to the type system, which may manipulate the set of declared types. Implementing a Type Provider entails the implementation of an interface to produce literal representations of new types, new properties and new methods. The net effect is that after a programmer has declared the intention to use a certain kind of type provider, at every use site of the generated types the provider mechanism will query external data sources for type information and bind new type names "dynamically" at type-checking time to the relevant scope in the F#program.

[13] See `http://msdn.microsoft.com/en-us/library/hh156509(v=vs.110).aspx`.

*Granularity.* A Type Provider is a module (dll) which provides functions to add types to a relevant scope. These functions may be parameterized, for example by a location URI, such that the generated types are specific for the call site. Rascal's resources are generators of modules, which statically contain all the type information that is present via a statically known URI. They are "module providers" rather than type providers and thus have a larger granularity. They will bind the new type names for the entire module that imports the resource.

*Dynamic & Lazy versus Static & Eager* By design, Type Providers make every change in external definitions of schemas immediately visible, and by using dynamic and lazy retrieval of type information exploratory programming[13] can be supported. Still, if a schema changes the user does need to rebuild the F# project[13]. Rascal's design requires a regeneration of the type definitions, which it does every time an importing module is (re)loaded. It does so eagerly—a resource must produce a complete set of type definitions for the data that will later be loaded dynamically. For large database schemas the Rascal programmer would have to wait until an entire E/R schema is translated or individually select which tables to import, while in F# the schema can be explored on a table-by-table basis. However, the Rascal programmer always gets a complete overview of the available types as a literal Rascal program, while the F# programmer would need to exercise the generated structure to discover these types, using IDE features such as auto-completion.

*Definition.* F#'s type providers are written in F#. A new type provider consists of functions that return lists of type literals. To implement a type provider, the programmer needs access to and understanding of F#'s reflection API. The construction of the types is type safe, calls to the reflection API are type-checked and the type-correctness of the generated type definitions can be guaranteed.

For Rascal resources, we currently use string templates to generate a new Rascal module. The *generating* program thus has no static guarantees of correctness regarding the generated program, while the *generated* program is fully type-checked before execution. Note that both Rascal and F# use type parameterized type literals to link generic data to specific types.

*Example.* For comparison, Figure 9 shows Rascal code for a resource generator that produces a set of 100 datatypes, each with three kinds of constructors of which one has 100 fields and 2 functions defined on each type. This example mimics precisely the example F# type provider called `HelloWorldTypeProvider`.[14]

Rascal's and F#'s type declarations are not entirely compatible, but the two example snippets generate definitions of the same size and complexity. The size of the F# example is much larger (67 non-commented non-empty lines), compared to Rascal (14). F# also needs 5 library modules to access the reflection API and the Type Provider API, while Rascal needs no API. The benefit of F#'s added explicitness is the type-safety that Rascal does not guarantee at generation time, while Rascal's solution is simpler to understand (we claim).

[14] See `http://msdn.microsoft.com/en-us/library/hh361034(v=vs.110)`.

```
module HelloWorldResource
@resource{helloworld}
str generate(loc uri, str name)
  = "module <name>
    '<for (i <- [1..100]) {>
    '@doc{This is an example generated type definition}
    'data Type<i> = unit_<i>()
    '             | data_<i>(str x)
    '             | nested_<i>(str sp_1, <for (j <- [2..100]) {>, str sp_<j><}>)
    '             ;
    '@doc{This computes some property}
    'int property(Type<i> arg) = ...;
    '@doc{This computes some function}
    'str method(Type<i> arg, int i) = ..."
    '<}>";
```

**Fig. 9.** The Rascal Resource generator to mimic F#'s HelloWorldTypeProvider.

Rascal does, however, support the construction of statically syntax correct modules using its concrete syntax feature. For example:

```
@resource{hello}
Module generator(loc uri, str name) = ‘module <[Name] name> ...‘;
```

### 5.6 OData

Rascal resources share some of the same goals as the Open Data Protocol[15], or OData. OData also uses URIs to identify resources, and is focused on providing a standard interface to resource data, but is a protocol, not a programming language mechanism. In the future, an `OData` Rascal resource would provide a clean way for Rascal to access information shared using the OData standard.

## 6 Discussion

We have presented the design and implementation of typed, uniform access to external resources in Rascal. This brings the flexibility of managing external datatypes in dynamically typed languages to the world of statically typed languages. Although we believe that the examples in Section 4 demonstrate that we have made good progress towards satisfying the requirements listed earlier in Section 3, some comments are in order.

First, it could be beneficial to check the static safety of the generated code *at generation time*, instead of as part of the module load process. Currently, generation can (if using concrete syntax) only guarantee that the generated code is syntactically correct. This would require invoking the checker as part of the generation process. Second, we have focused so far on resources that can easily be read fully into Rascal and manipulated using Rascal code. For larger resources, this is not practical, and we instead would need a method of gradually reading in resource data. Along with this, it would be useful to leverage the optimization and

[15] See `http://www.odata.org/`.

search capabilities of external systems. For instance, the not-null check shown in the `JDBC` example in Section 4 is currently performed using pattern matching in Rascal, but could also be done directly by the database. Third, we have focused mainly on reading resources, and would like to provide better support for writing data back to external data sources after making changes within Rascal.

We intend to further explore and extend the possibilities of resources for other resource types that are relevant for the software language engineer.

**Acknowledgments:** We would like to thank the anonymous reviewers, whose helpful feedback has assisted us in improving the quality of this paper.

## References

1. The Enhydra Project: The Zeus Java-to-XML Data Binding tool. Available at `http://zeus.ow2.org/`, 2002.
2. The ExoLab Group: Castor. Available at `http://www.castor.org`, 2002.
3. E. Bouwers, M. Bravenboer, and E. Visser. Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking. In *Proceedings of LDTA'07*, volume 203 of *ENTCS*, pages 85–101. Elsevier, 2008.
4. XML/Java Data Binding and Breeze XML Binder. Technical report, The Breeze Factor, 2002. available at `http://www.breezefactor.com/whitepapers.html`.
5. W. Cook and A. Ibrahim. Integrating Programming Languages and Databases: What is the Problem? *ODBMS.ORG*, Sept 2006. Expert Article.
6. H. de Jong and P. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59:35–61, April 2004.
7. J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
8. 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.
9. P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for Grammarware. *ACM TOSEM*, 14(3):331–380, 2005.
10. 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.
11. D. Maier. Representing Database Programs as Objects. In *Advances in Database Programming Languages, Papers from DBPL-1, September 1987, Roscoff, France*, pages 377–386, 1987.
12. S. Melnik, A. Adya, and P. A. Bernstein. Compiling Mappings to Bridge Applications and Databases. In *Proceedings of SIGMOD'07*, pages 461–472. ACM, 2007.
13. H. Müller and K. Klashinsky. Rigi - A System for Programming-in-the-Large. In *Proceedings of ICSE 10 (1988)*, pages 80–86, April 1988.
14. D. Ritchie and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 57(6), July–August 1978.
15. R. P. van de Riet, A. I. Wasserman, M. L. Kersten, and W. de Jonge. High-Level Programming Features for Improving the Efficiency of a Relational Database System. *ACM Trans. Database Syst.*, 6(3):464–485, 1981.
16. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.