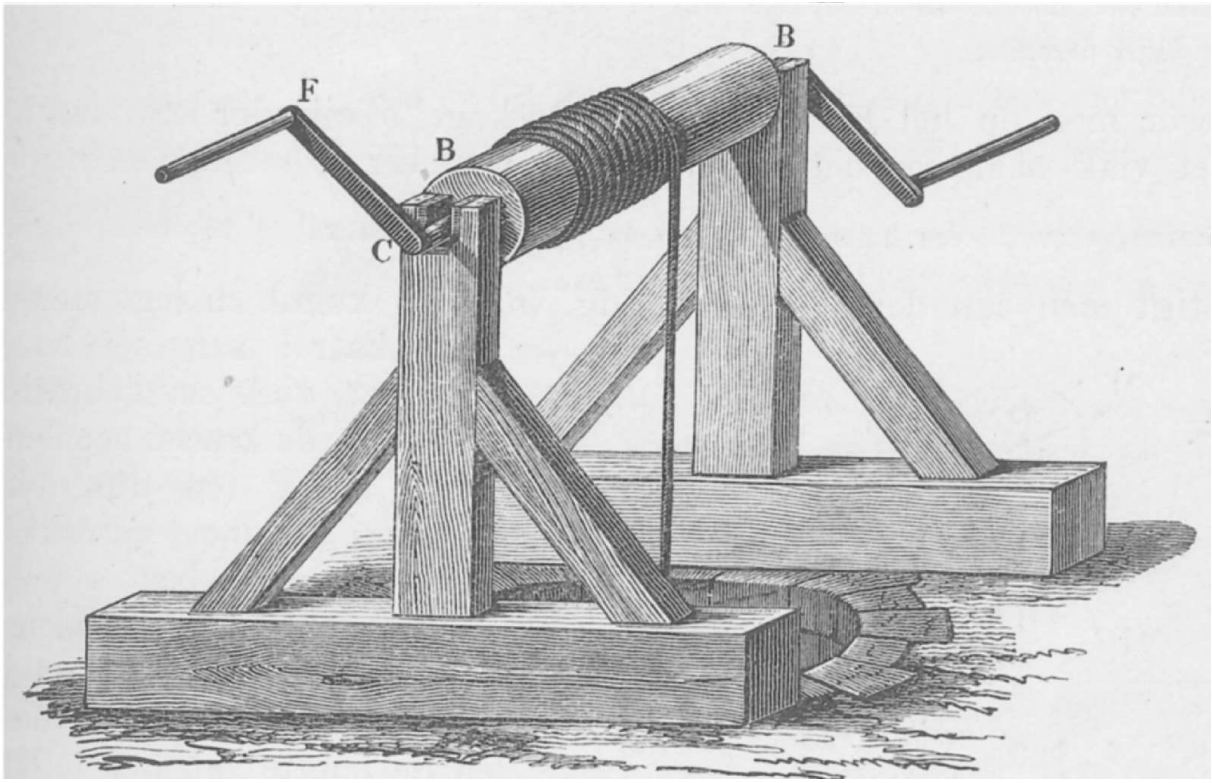
A Tutorial Introduction to RSCRIPT
—a Relational Approach to Software Analysis—



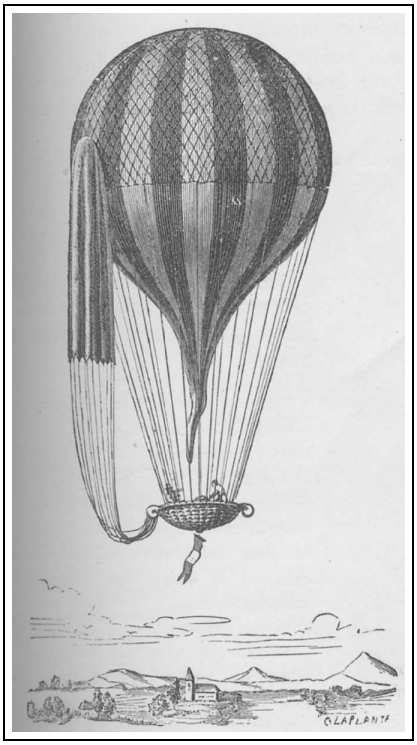
Paul Klint

DRAFT: September 4, 2007

Contents

Chapter 1

Introduction



Extract-Enrich-View paradigm: RSCRIPT is a small scripting language based on the relational calculus. It is intended for analyzing and querying the source code of software systems: from finding uninitialized variables in a single program to formulating queries about the architecture of a complete software system. RSCRIPT fits well in the extract-enrich-view paradigm shown in Figure 1.1:
Extract: Given the source text, extract relevant information from it in the form of relations. Examples are the CALLS relation that describes direct calls between procedures, the USE relation that relates statements with the variables that are used in the statements, and the PRED relation that relates a statement with its predecessors in the control flow graph. The extraction phase is outside the scope of RSCRIPT but may, for instance, be implemented using ASF+SDF [4] and we will give examples how to do this.
Enrich: Derive additional information from the relations extracted from the source text. For instance, use CALLS to compute procedures that can also call each other indirectly (using transitive closure). Here is where RSCRIPT shines.
View: The result of the enrichment phase are again bags and relations. These can be displayed with various tools like, Dot [15], Rigi [19] and others. RSCRIPT is not concerned with viewing but we will give some examples anyway.

Application of Relations to Program Analysis Many algorithms for program analysis are usually presented as *graph* algorithms and this seems to be at odds with the extensive experience of using *term* rewriting for tasks as type checking, fact extraction, analysis and transformation. The major obstacle is that graphs can and terms cannot contain cycles. Fortunately, every graph can be represented as a relation and it is therefore natural to have a look at the combination of relations and term rewriting.

Once you start considering problems from a relational perspective, elegant and concise solutions start to appear. Some examples are:

- Analysis of call graphs and the structure of software architectures.
- Detailed analysis of the control flow or dataflow of programs.
- Program slicing.
- Type checking.
- Constraint problems.

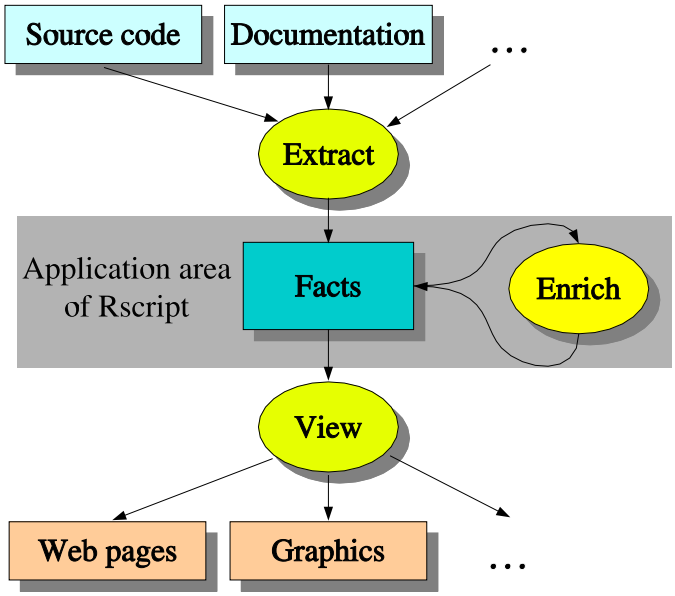


Figure 1.1: The extract-enrich-view paradigm

What's new in RSCRIPT? Given the considerable amount of related work to be discussed below, it is necessary to clearly establish what is and what is not new in our approach:

- We use sets and relations like Rigi [19] and GROK [12] do. After extensive experimentation we have decided *not* to use bags and multi-relations like in RPA [9].
- Unlike several other systems we allow *nested* sets and relations and also support n-ary relations as opposed to just binary relations but don't support the complete repertoire of n-ary relations as in SQL.
- We offer a strongly typed language with user-defined types.
- Unlike Rigi [19], GROK [12] and RPA [9] we provide a relational *calculus* as opposed to a relational algebra. Although the two have the same expressive power, a calculus increases, in our opinion, the readability of relational expressions because they allow the introduction of variables to express intermediate results.
- We integrate an equation solver in a relational language. In this way dataflow problems can be expressed.
- We introduce an *location* datatype with associated operations to easily manipulate references to source code.
- There is some innovation in syntactic notation and specific built-in functions.
- We introduce the notion of an RSTORE that generalizes the RSF tuple format of Rigi. An RSTORE consists of name/value pairs, where the values may be arbitrary nested bags or relations. An RSTORE is a language-independent exchange format and can be used to exchange complex relational data between programs written in different languages.

1.1 Background

Relation-oriented Languages There is a long tradition in Computer Science to organize languages around one a more prominent data types such as lists (Lisp), strings (SNOBOL), arrays (APL) or sets (SETL). We use sets and relations as primary datatypes and the sets and set formers in SETL [21] are the best historic reference for them. Set formers have later on be popularized in various functional languages since they were introduced in KRC [25]. An overview of languages centered around collection types such as sets and bags is given in [22]. Database languages in general and SQL in particular are described in [26]. The connection between comprehensions and relational algebra is described in [27, 24]. A further analysis of this topic is given in [6].

Systems supporting relational programming include RELVIEW [2] (intended for the interactive creation and visualization of relations and the prototyping of graph algorithms), ...

Relations and Program Analysis The idea to represent relational views of programs is already quite old. For instance, in [17] all syntactic as well as semantic aspects of a program were represented by relations and SQL was used to query them. Due to the lack of expressiveness of SQL (notably the lack of transitive closures) and the performance problems encountered, this approach has not seen wider use. In Rigi [19], a tuple format (RSF) is introduced to represent relations and a language (RCL) to manipulate them. In [20] a *source code algebra* is described that can be used to express relational queries on source text. In [5] a *query algebra* is formulated to express direct queries on the syntax tree. It also allows the querying of information that is attached to the syntax tree via annotations. Relational algebra is used in GROK [12] and Relation Partition Algebra (RPA) [9, 10, 16] to represent basic facts about software systems and to query them. In GUPRO [8] graphs are used to represent programs and to query them. In F(p)–ℓ [7] a Prolog database and a special-purpose language are used to represent and query program facts.

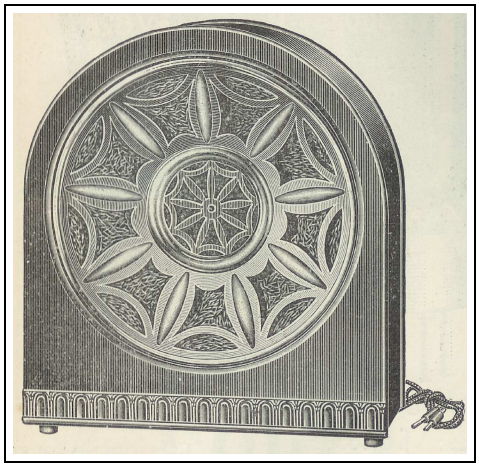
The requirements for a query language for reverse engineering are discussed in [11].

1.2 Plan for this Tutorial

In Chapter 2 we first provide a motivating example of our relational approach. In Chapter 3 follows a complete description of all the features in RSCRIPT. In the following Chapters 4 and 5 all built-in operators and functions are described. The most interesting part of this tutorial is probably Chapter 6 where we present a menagerie of larger examples ranging from computing the McCabe complexity of code, analyzing the component structure of systems, to program slicing. Chapter 8 describes how to run an RSCRIPT. Two appendices complete this tutorial: Appendix A summarizes all built-in operators and Appendix B summarizes all built-in functions.

Chapter 2

A Motivating Example



Suppose a mystery box ends up on your desk. When you open it, it contains a huge software system with several questions attached to it:

- How many procedure calls occur in this system?
- How many procedures contains it?
- What are the entry points for this system, i.e., procedures that call others but are not called themselves?
- What are the leaves of this application, i.e., procedures that are called but do not make any calls themselves?
- Which procedures call each other indirectly?
- Which procedures are called directly or indirectly from each entry point?
- Which procedures are called from all entry points?

There are now two possibilities. Either you have this superb programming environment or tool suite that can immediately answer all these questions for you or you can use RSCRIPT.

Preparations To illustrate this process consider the workflow in Figure 2.1. First we have to extract the calls from the source code. Recall that RSCRIPT does not consider fact extraction *per se* so we assume that this call graph has been extracted from the software by some other tool. Also keep in mind that a real call graph of a real application will contain thousands and thousands of calls. Drawing it in the way we do later on in Figure 2.2 makes no sense since we get a uniformly black picture due to all the call dependencies. After the extraction phase, we try to understand the extracted facts by writing queries to explore their properties. For instance, we may want to know *how many calls* there are, or *how many procedures*. We may also want to enrich these facts, for instance, by computing who calls who in more than one step. Finally, we produce a simple textual report giving answers to the questions we are interested in.

Now consider the call graph shown in Figure 2.2. This section is intended to give you a first impression what can be done with RSCRIPT. Please return to this example when you have digested the detailed description of RSCRIPT in Chapters 3, 4 and 5.

RSCRIPT supports some basic data types like integers and strings which are sufficient to formulate and answer the questions at hand. However, we can gain readability by introducing separately named types for the items we are describing. First, we introduce therefore a new type `proc` (an alias for strings) to denote procedures:

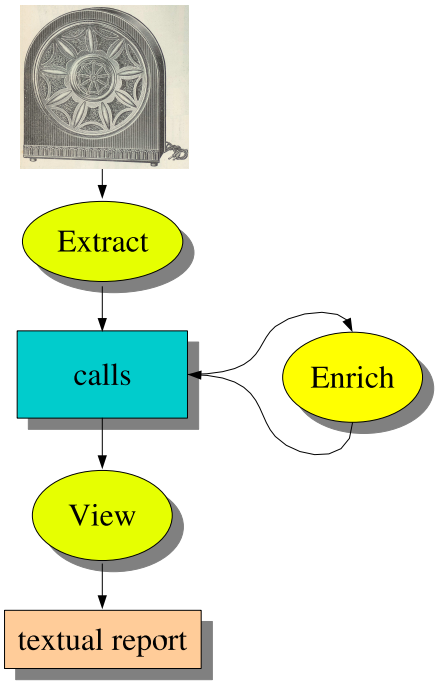


Figure 2.1: Workflow for analyzing mystery box.

```
type proc = str
```

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

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

This concludes the preparatory steps and now we move on to answer the questions.

How many procedure calls occur in this system? To determine the numbers of calls, we simply determine the number of tuples in the `Calls` relation, as follows:

```
int nCalls = # Calls
```

The operator # determines the number of elements in a bag or relation and is explained in Section 4.5.4. In this example, `nCalls` will get the value `8`.

How many procedures contains it? We get the number of procedures by determining which names occur in the tuples in the relation `Calls` and then determining the number of names:

```
set[proc] procs = carrier(Calls)
int nprocs = # procs
```

The built-in function `carrier` determines all the values that occur in the tuples of a relation. In this case, `procs` will get the value `{"a", "b", "c", "d", "e", "f", "g"}` and `nprocs` will thus get value `7`. A more concise way of expressing this would be to combine both steps:

```
int nprocs = # carrier(Calls)
```

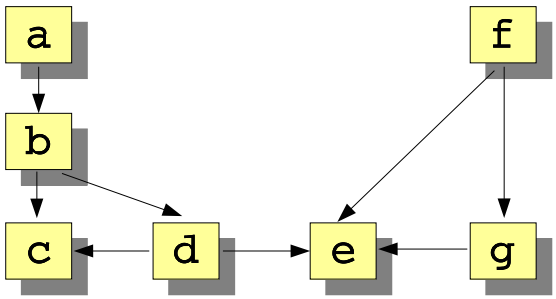


Figure 2.2: Graphical representation of the `calls` relation

What are the entry points for this system? The next step in the analysis is to determine which *entry points* this application has, i.e., procedures which call others but are not called themselves. Entry points are useful since they define the external interface of a system and may also be used as guidance to split a system in parts.

The *top* of a relation contains those left-hand sides of tuples in a relation that do not occur in any right-hand side. When a relation is viewed as a graph, its top corresponds to the root nodes of that graph. Similarly, the *bottom* of a relation corresponds to the leaf nodes of the graph. See Section 5.5.2 for more details. Using this knowledge, the entry points can be computed by determining the top of the `Calls` relation:

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

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

What are the leaves of this application? In a similar spirit, we can determine the *leaves* of this application, i.e., procedures that are being called but do not make any calls themselves:

```
set[proc] bottomCalls = bottom(Calls).
```

In this case, `bottomCalls` is equal to `{"c", "e"}`.

Which procedures call each other indirectly? We can also determine the *indirect calls* between procedures, by taking the transitive closure of the `Calls` relation:

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

In this case, `closureCalls` is equal to

```
{<"a", "b">, <"b", "c">, <"b", "d">, <"d", "c">, <"d","e">, <"f", "e">,
 <"f", "g">, <"g", "e">, <"a", "c">, <"a", "d">, <"b", "e">, <"a", "e">}
```

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

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

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

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

yields `{"e", "g"}`.

Which procedures are called from all entry points? Finally, we can determine which procedures are called from both entry points by taking the intersection of the two sets `calledFromA` and `calledFromF`

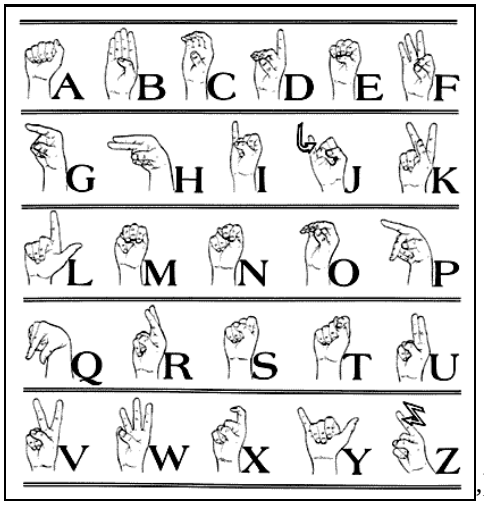
```
set[proc] commonProcs = calledFromA inter calledFromF
```

which yields `{"e"}`. In other words, the procedures called from both entry points are mostly disjoint except for the common procedure `"e"`.

Wrap-up These findings can be verified by inspecting a graph view of the calls relation as shown in Figure 2.2. Such a visual inspection does *not* scale very well to large graphs and this makes the above form of analysis particularly suited for studying large systems.

Chapter 3

The RSCRIPT Language



,]

RSCRIPT is based on *binary relations* only and has no direct support for n-ary relations with labeled columns as usual in a general database language. However, some syntactic support for n-ary relations exists. We will explain this further below.
An RSCRIPT consists of a sequence of declarations for variables and/or functions. Usually, the value of one of these variables is what the writer of the script is interested in.
The language has scalar types (Boolean, integer, string, location) and composite types (set and relation). Expressions are constructed from comprehensions, function invocations and operators. These are all described below.

3.1 Types and Values

3.1.1 Elementary Types and Values

Booleans The Booleans are represented by the type `bool` and have two values: `true` and `false`.

Integers The integer values are represented by the type `int` and are written as usual, e.g., `0`, `1`, or `123`.

Strings The string values are represented by the type `str` and consist of character sequences surrounded by double quotes. e.g., `"a"` or `"a long string"`.

Locations Location values are represented by the type `loc` and serve as text coordinates in a specific source file. They should *always* be generated automatically but for the curious here is an example how they look like: `area-in-file("/home/paulk/example.pico", area(6, 17, 6, 18, 131, 1))`.

3.2 Tuples, Sets and Relations

Tuples Tuples are represented by the type <T_1, T_2>, where T_1 and T_2 are arbitrary types. An example of a tuple type is `<int, str>`. RSCRIPT directly supports tuples consisting of two elements (also know

as *pairs*). For convenience, n-ary tuples are also allowed, but there are some restrictions on their use, see the paragraph **Relations** below. Examples are:

- `<1, 2>` is of type `<int, int>`,
- `<1, 2, 3>` is of type `<int, int, int>`,
- `<1, "a", 3>` is of type `<int, str, int>`,

Sets Sets are represented by the type `set[`T`]`, where T is an arbitrary type. Examples are `set[int]`, `set[<int,int>]` and `set[set[str]]`. Sets are denoted by a list of elements, separated by comma's and enclosed in braces as in `{`E_1`, `E_2`, ..., `E_n`}`, where the E_i ($1 \leq i \leq n$) are expressions that yield the desired element type. For example,

- `{1, 2, 3}` is of type `set[int]`,
- `{<1,10>, <2,20>, <3,30>}` is of type `set[<int,int>]`,
- `{<"a",10>, <"b",20>, <"c",30>}` is of type `set[<str,int>]`, and
- `{{"a", "b"}, {"c", "d", "e"}}` is of type `set[set[str]]`.

Relations Relations are nothing more than sets of tuples, but since they are used so often we provide some shorthand notation for them.

Relations are represented by the type `rel[`T_1`, `T_2`]`, where T_1 and T_2 are arbitrary types; it is a shorthand for `set[<`T_1`, `T_2`>]`. Examples are `rel[int,str]` and `rel[int,set[str]]`. Relations are denoted by `{<`E_{11}`, `E_{12}`>, <`E_{21}`, `E_{22}`>, ..., <`E_{n1}`, `E_{n2}`>}`, where the E_{ij} are expressions that yield the desired element type. For example, `{<1, "a">, <2, "b">, <3,"c">}` is of type `rel[int, str]`.

Not surprisingly, n-ary relations are represented by the type `rel[`T_1`, `T_2`, ..., `T_n`]` which is a shorthand for `set[<`T_1`, `T_2`, ..., `T_n`>]`. Most built-in operators and functions require binary relations as arguments. It is, however, perfectly possible to use n-ary relations as values, or as arguments or results of functions. Examples are:

- `{<1,10>, <2,20>, <3,30>}` is of type `rel[int,int]` (yes indeed, you saw this same example before and then we gave `set[<int,int>]` as its type; remember that these types are interchangeable.),
- `{<"a",10>, <"b",20>, <"c",30>}` is of type `rel[str,int]`, and
- `{{"a", 1, "b"}, {"c", 2, "d"}}` is of type `rel[str,int,str]`.

3.2.1 User-defined Types and Values

Alias types Everything can be expressed using the elementary types and values that are provided by RSCRIPT. However, for the purpose of documentation and readability it is sometimes better to use a descriptive name as type indication, rather than an elementary type. The type declaration

```
type T1 = T2
```

states that the new type name T_1 can be used everywhere instead of the already defined type name T_2. For instance,

```
type ModuleId = str
type Frequency = int
```

introduces two new type names `ModuleId` and `Frequency`, both an alias for the type `str`. The use of type aliases is a good way to hide representation details.

Composite Types and Values In ordinary programming languages record types or classes exist to introduce a new type name for a collection of related, named, values and to provide access to the elements of such a collection through their name. In RSCRIPT, tuples with named elements provide this facility. The type declaration

`type` T `= <`T_1 F_1 `, ...,` T_n F_n`>`

introduces a new composite type T, with n elements. The i-th element T_i N_i has type T_i and field name F_i. The common dot notation for field access is used to address an element of a composite type. If V is a variable of type T, then the i-th element can be accessed by V `.` F_i. For instance,[1]

```
type Triple = <int left, str middle, bool right>
Triple TR = <3, "a", true>
str S = TR.middle
```

first introduces the composite type `Triple` and defines the `Triple` variable `TR`. Next, the field selection `TR.middle` is used to define the string `S`.

Implementation Note. The current implementation severely restricts the re-use of field names in different type declarations. The only re-use that is allowed are fields with the same name and the same type that appear at the same position in different type declarations.

Type equivalence An RSCRIPT should be *well-typed*, this means above all that identifiers that are used in expressions have been declared, and that operations and functions should have operands of the required type. We use *structural equivalence* between types as criterion for type equality. The equivalence of two types T_1 and T_2 can be determined as follows:

- Replace in both T_1 and T_2 all user-defined types by their definition until all user-defined types have been eliminated. This may require repeated replacements. This gives, respectively, T_1' and T_2'.
- If T_1' and T_2' are identical, then T_1 and T_2 are equal.
- Otherwise T_1 and T_2 are not equal.

3.3 Comprehensions

We will use the familiar notation

`{`E_1`, ...,` E_m `|` G_1`, ...,` G_n`}`

to denote the construction of a set consisting of the union of successive values of the expressions $E_1, ..., E_m$. The values and the generated set are determined by $E_1, ..., E_m$ and the *generators* $G_1, ..., G_n$. E is computed for all possible combinations of values produced by the generators.

Each generator may introduce new variables that can be used in subsequent generators as well as in the expressions $E_1, ..., E_m$. A generator can use the variables introduced by preceding generators. Generators may enumerate all the values in a set or relation, they may perform a test, or they may assign a value to variables.

3.3.1 Generators

Enumerator Enumerators generate all the values in a given set or relation. They come in two flavors:

- T V `:` E: the elements of the set S (of type `set[`T`]`) that results from the evaluation of expression E are enumerated and subsequently assigned to the new variable V of type T. Examples are:

[1] The variable declarations that appear on lines 2 and 3 of this example are explained fully in Section 3.4.

  - `int N : {1, 2, 3, 4, 5}`,
  - `str K : KEYWORDS`, where `KEYWORDS` should evaluate to a value of type `set[str]`.

- `<`D_1`, ..., `D_n`> :` E: the elements of the relation R (of type `rel[<`T'_1`, ...,`T'_n`]`, where T'_i is determined by the type of each target D_i, see below) that results from the evaluation of expression E are enumerated. The i-the element ($i = 1, ..., n$) of the resulting n-tuple is subsequently combined with each target D_i as follows:
  - If D_i is a variable declaration of the form T_i V_i, then the i-th element is assigned to V_i.
  - If D_i is an arbitrary expression E_i, then the value of the i-th element should be equal to the value of E_i. If they are unequal, computation continues with enumerating the next tuple in the relation R.

  Examples are:

  - `<str K, int N> : <"a",10>, <"b",20>, <"c",30>}¡`
  - `<str K, int N> : FREQUENCIES`, where `FREQUENCIES` should evaluate to a value of type `rel[str,int]`.
  - `<str K, 10> : FREQUENCIES`, will only generate pairs with `10` as second element.

Test A test is a boolean-valued expression. If the evaluation yields `true` this indicates that the current combination of generated values up to this test is still as desired and execution continues with subsequent generators. If the evaluation yields `false` this indicates that the current combination of values is undesired, and that another combination should be tried. Examples:

- `N >= 3` tests whether `N` has a value greater than or equal `3`.
- `S == "coffee"` tests whether `S` is equal to the string `"coffee"`.

In both examples, the variable (`N`, respectively, `S`) should have been introduced by a generator that occurs earlier in the enclosing comprehension.

Assignment Assignments assign a value to one or more variables and also come in two flavors:

- T V `<-` E: assigns the value of expression E to the new variable V of type T.
- `<`R_1`, ..., `R_n`> <-` E: combines the elements of the n-tuple resulting from the evaluation of expression E with each T_i as follows:
  - If R_i is a variable declaration of the form T V_i, then the i-th element is assigned to V_i.
  - If R_i is an arbitrary expression E_i, then the value of the i-th element should be equal to the value of E_i. If they are unequal, the assignment acts as a test that fails (see above).

  Examples of assignments are:

- `rel[str,str] ALLCALLS <- CALLS+` assigns the transitive closure of the relation `CALLS` to the variable `ALLCALLS`.
- `bool Smaller <- A <= B` assigns the result of the test `A <= B` to the Boolean variable `Smaller`.
- `<int N, str S, 10> <- E` evaluates expression `E` (which should yield a tuple of type `<int, str, int>`) and performs a tuple-wise assignment to the new variables `N` and `S` *provided* that the third element of the result is equal to `10`. Otherwise the assignment acts as a test that fails.

3.3.2 Examples of Comprehensions

- `{X | int X : {1, 2, 3, 4, 5}, X >= 3}` yields the set `{3,4,5}`.
- `{<X, Y> | int X : {1, 2, 3}, int Y : {2, 3, 4}, X >= Y}` yields the relation `{<2, 2>, <3, 2>, <3, 3>}`.
- `{<Y, X> | <int X, int Y> : {<1,10>, <2,20>}}` yields the inverse of the given relation: `{<10,1>, <20,2>}`.
- `{X, X * X | X : {1, 2, 3, 4, 5}, X >= 3}` yields the set `{3,4,5,9,16,25}`.

3.4 Declarations

3.4.1 Variable Declarations

A variable declaration has the form

$T\ V$ `=` E

where T is a type, V is a variable name, and T is an expression that should have type T. The effect is that the value of expression E is assigned to V and can be used later on as V's value. Double declarations are not allowed. As a convenience, also declarations without an initialization expression are permitted and have the form

$T\ V$

and only introduce the variable V. Examples:

- `int max = 100` declares the integer variable `max` with value `100`.
- The definition

```
rel[str,int] day = {<"mon", 1>, <"tue", 2>, <"wed",3>,
                    <"thu", 4>, <"fri", 5>, <"sat",6>, <"sun",7>}
```

  declares the variable `day`, a relation that maps strings to integers.

3.4.2 Local Variable Declarations

Local variables can be introduced as follows:

E `where` $T_1\ V_1$ `=` E_1`, ...,` $T_n\ V_n$ `=` E_n `end where`

First the local variables V_i are bound to their respective values E_i, and then the value of expression E is yielded.

3.4.3 Function Declarations

A function declaration has the form

$T\ F$`(`$T_1\ V_1$`, ...,` $T_n\ V_n$`)` `=` E

Here T is the result type of the function and this should be equal to the type of the associated expression E. Each $T_i\ V_i$ represents a typed formal parameter of the function. The formal parameters may occur in E and get their value when F is invoked from another expression. Example:

- The function declaration

```
rel[int, int] invert(rel[int,int] R) = {<Y, X> | <int X, int Y> : R }
```

  yields the inverse of the argument relation `R`. For instance, `invert({<1,10>, <2,20>})` yields `{<10,1>, <20,2>}`.

Parameterized types in function declarations The types that occur in function declarations may also contain *type variables* that are written as & followed by an identifier. In this way functions can be defined for arbitrary types. Examples:

- The declaration

  ```
  rel[&T2, &T1] invert2(rel[&T1,&T2] R) = {<Y, X> | <&T1 X, &T2 Y> : R }
  ```

  yields an inversion function that is applicable to any binary relation. For instance,

  - `invert2({<1,10>, <2,20>})` yields `{<10,1>, <20,2>}`, and
  - `invert2({<"mon", 1>, <"tue", 2>})` yields `{<1, "mon">, <2, "tue">}`.

- The function

  ```
  <&T2, &T1> swap(&T1 A, &T2 B) = <B, A>
  ```

  can be used to swap the elements of pairs of arbitrary types. For instance,

  - `swap(<1, 2>)` yields `<2,1>` and
  - `swap(<"wed", 3>)` yields `<3, "wed">`.

3.5 Assertions

An assert statement may occur everywhere where a declaration is allowed. It has the form

`assert` L`:` E

where L is a string that serves as a label for this assertion, and E is a boolean-value expression. During execution, a list of true and false assertions is maintained. When the script is executed as a *test suite* (see Section 8.4) a summary of this information is shown to the user. When the script is executed in the standard fashion, the assert statement has no affect. Example:

- ```
  assert "Equality on Sets 1": {1, 2, 3, 1} == {3, 2, 1, 1}
  ```

3.6 Equations

It is also possible to define mutually dependent sets of equations:

```
equations
   initial
      T1 V1 init I1
       ...
      Tn Vn init In
   satisfy
      V1 = E1
      ...
      Vn = En
end equations
```

In the `initial` section, the variables V_i are declared and initialized. In the `satisfy` section, the actual set of equations is given. The expressions E_i may refer to any of the variables V_i (and to any variables declared earlier). This set of equations is solved by evaluating the expressions E_i, assigning their value to the corresponding variables V_i, and repeating this as long as the value of one of the variables was changed. This is typically used for solving a set of dataflow equations. Example:
```

- Although transitive closure is provided as a built-in operator, we can use equations to define the transitive closure of a relation. Recall that

$$R+ = R \cup (R \circ R) \cup (R \circ R \circ R) \cup ....$$

This can be expressed as follows.

```
rel[int,int] R =  {<1,2>, <2,3>, <3,4>}

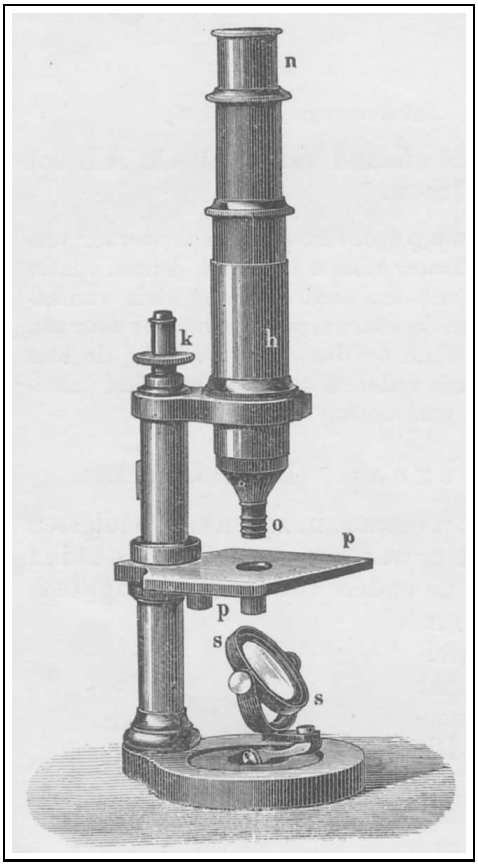
equations
  initial
     rel[int,int] T init R
  satisfy
     T = T union (T o R)
end equations
```

The resulting value of `T` is as expected:

```
{<1,2>, <2,3>, <3,4>, <1, 3>, <2, 4>, <1, 4>}
```

Chapter 4

Built-in Operators



The built-in operators can be subdivided in several broad categories:

- Operations on Booleans (Section 4.1): logical operators (`and`, `or`, `implies` and `not`).
- Operations on integers (Section 4.2): arithmetic operators (`+`, `-`, `*`, and `/`) and comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`).
- Operations on strings (Section 4.3): comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`).
- Operations on locations (Section 4.4). comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`).
- Operations on sets or relations (Section 4.5): membership tests (`in`, `notin`), comparison operators (`==`, `!=`, `<`, `<=`, `>`, and `>=`), and construction operators (`union`, `inter`, `diff`).
- Operations on relations (Section 4.6): composition (`o`), Cartesian product (`x`), left and right image operators, and transitive closures (`+`, `*`).

The following sections give detailed descriptions and examples of all built-in operators.

4.1 Operations on Booleans

$bool_1$ `and` $bool_2$	yields `true` if both arguments have the value true and `false` otherwise
$bool_1$ `or` $bool_2$	yields `true` if either argument has the value `true` and `false` otherwise
$bool_1$ `implies` $bool_2$	yields `false` if $bool_1$ has the value `true` and $bool_2$ has value `false`, and `true` otherwise
`not` *bool*	yields true if *bool* is `false` and `true` otherwise

4.2 Operations on Integers

int_1 == int_2	yields `true` if both arguments are numerically equal and `false` otherwise
int_1 != int_2	yields `true` if both arguments are numerically unequal and `false` otherwise
int_1 <= int_2	yields `true` if int_1 is numerically less than or equal to int_2 and `false` otherwise
int_1 < int_2	yields `true` if int_1 is a numerically less than int_2 and `false` otherwise
int_1 >= int_2	yields `true` if int_1 is numerically greater than or equal to int_2 and `false` otherwise
int_1 > int_2	yields `true` if int_1 is numerically greater than int_2 and `false` otherwise
int_1 + int_2	yields the arithmetic sum of int_1 and int_2
int_1 - int_2	yields the arithmetic difference of int_1 and int_2
int_1 * int_2	yields the arithmetic product of int_1 and int_2
int_1 / int_2	yields the integer division of int_1 and int_2

4.3 Operations on Strings

str_1 == str_2	yields `true` if both arguments are equal and `false` otherwise
str_1 != str_2	yields `true` if both arguments are unequal and `false` otherwise
str_1 <= str_2	yields `true` if str_1 is lexicographically less than or equal to str_2 and `false` otherwise
str_1 < str_2	yields `true` if str_1 is a lexicographically less than str_2 and `false` otherwise
str_1 >= str_2	yields `true` if str_1 is lexicographically greater than or equal to str_2 and `false` otherwise
str_1 > str_2	yields `true` if str_1 lexicographically greater than str_2 and `false` otherwise

4.4 Operations on Locations

loc_1 == loc_2	yields `true` if both arguments are identical and `false` otherwise
loc_1 != loc_2	yields `true` if both arguments are unequal and `false` otherwise
loc_1 <= loc_2	yields `true` if loc_1 is textually contained in or equal to loc_2 and `false` otherwise
loc_1 < loc_2	yields `true` if loc_1 is strictly textually contained in loc_2 and `false` otherwise
loc_1 >= loc_2	yields `true` if loc_1 textually encloses or or is equal to loc_2 and `false` otherwise
loc_1 > loc_2	yields `true` if loc_1 strictly textually encloses loc_2 and `false` otherwise

Examples In the following examples the offset and length part of a location are set to `0`; they are not used when determining the outcome of the comparison operators.

- `area-in-file("f", area(11, 1, 11, 9, 0, 0)) < area-in-file("f", area(10, 2, 12, 8, 0, 0))` yields `true`.
- `area-in-file("f", area(10, 3, 11, 7, 0,0)) < area-in-file("f", area(10, 2, 11, 8, 0, 0))` yields `true`.
- `area-in-file("f", area(10, 3, 11, 7, 0, 0)) < area-in-file("g", area(10, 3, 11, 7, 0, 0))` yields `false`.

4.5 Operations on Sets or Relations

4.5.1 Membership Tests

any `in` *set*	yields `true` if *any* occurs as element in *set* and `false` otherwise
any `notin` *set*	yields `false` if *any* occurs as element in *set* and `true` otherwise
tuple `in` *rel*	yields `true` if *tuple* occurs as element in *rel* and `false` otherwise
tuple `notin` *rel*	yields `false` if *tuple* occurs as element in *rel* and `true` otherwise

Examples

- `3 in {1, 2, 3}` yields `true`.
- `4 in {1, 2, 3}` yields `false`.
- `3 notin {1, 2, 3}` yields `false`.
- `4 notin {1, 2, 3}` yields `true`.
- `<2,20> in {<1,10>, <2,20>, <3,30>}` yields `true`.
- `<4,40> notin {<1,10>, <2,20>, <3,30>}` yields `true`.

Note If the first argument of these operators has type T, then the second argument should have type `set[`T`]`.

4.5.2 Comparisons

set_1 `==` set_2	yields `true` if both arguments are equal sets and `false` otherwise
set_1 `!=` set_2	yields `true` if both arguments are unequal sets and `false` otherwise
set_1 `<=` set_2	yields `true` if set_1 is a subset of set_2 and `false` otherwise
set_1 `<` set_2	yields `true` if set_1 is a strict subset of set_2 and `false` otherwise
set_1 `>=` set_2	yields `true` if set_1 is a superset of set_2 and `false` otherwise
set_1 `>` set_2	yields `true` if set_1 is a strict superset of set_2 and `false` otherwise

4.5.3 Construction

set_1 `union` set_2	yields the set resulting from the union of the two arguments.
set_1 `inter` set_2	yields the set resulting from the intersection of the two arguments.
set_1 `\` set_2	yields the set resulting from the difference of the two arguments.

Examples

- `{1, 2, 3} union {4, 5, 6}` yields `{1, 2, 3, 4, 5, 6}`.
- `{1, 2, 3} union {1, 2, 3}` yields `{1, 2, 3}`.
- `{1, 2, 3} union {4, 5, 6}` yields `{1, 2, 3, 4, 5, 6}`.
- `{1, 2, 3} inter {4, 5, 6}` yields `{ }`.
- `{1, 2, 3} inter {1, 2, 3}` yields `{1, 2, 3}`.
- `{1, 2, 3, 4}  \ {1, 2, 3}` yields `{4}`.
- `{1, 2, 3}  \ {4, 5, 6}` yields `{1, 2, 3}`.

4.5.4 Miscellaneous

# set	yields the number of elements in *set*.
# rel	yields the number of tuples in *rel*.

Examples

- `#{1, 2, 3}` yields `3`.
- `{<1,10>, <2,20>, <3,30>}` yield `3`.

4.6 Operations on Relations

rel_1 `o` rel_2	yields the relation resulting from the composition of the two arguments
set_1 `x` set_2	yields the relation resulting from the Cartesian product of the two arguments
rel `[-,` *set* `]`	yields the left image of the *rel*
rel `[-,` *elem* `]`	yields the left image of the *rel*
rel `[` *elem* `,-]`	yields the right image of *rel*
rel `[` *set* `,-]`	yields the right image of *rel*
set `[` *elem* `]`	yields the right image of *rel*
rel `[` *set* `]`	yields the right image of *rel*
rel `+`	yields the relation resulting from the transitive closure of *rel*
rel `*`	yields the relation resulting from the reflexive transitive closure of *rel*

Composition: `o` The composition operator combines two relations and can be defined as follows:

```
rel[&T1,&T3] compose(rel[&T1,&T2] R1, rel[&T2,&T3] R2) =
   {<V, Y> | <&T1 V, &T2 W> : R1, <&T2 X, &T3 Y> : R2, W == X }
```

Example

- `{<1,10>, <2,20>, <3,15>} o {<10,100>, <20,200>}` yields `{<1,100>, <2,200>}`.

Cartesian product: `x` The product operator combines two sets into a relation and can be defined as follows:

```
rel[&T1,&T2] product(set[&T1] S1, set[&T2] S2) =
   {<V, W> | &T1 V : S1, &T2 W : S2 }
```

Example

- `{1, 2, 3} x {9}` yields `{<1, 9>, <2, 9>, <3, 9>}`.

Left image: `[-, ]` Taking the left image of a relation amounts to selecting some elements from the domain of a relation.

The *left* image operator takes a relation and an element `E` and produces a set consisting of all elements E_i in the domain of the relation that occur in tuples of the form `<`E_i`, E>`. It can be defined as follows:

```
set[&T1] left-image(rel[&T1,&T2] R, &T2 E) =
   { V | <&T1 V, &T2 W> : R, W == E }
```

The left image operator can be extended to take a set of elements as second element instead of a single element:

```
set[&T1] left-image(rel[&T1,&T2] R, set[&T2] S) =
   { V | <&T1 V, &T2 W> : R, W in S }
```

Examples Assume that `Rel` has value `{<1,10>, <2,20>, <1,11>, <3,30>, <2,21>}` in the following examples.

- `Rel[-,10]` yields `{1}`.
- `Rel[-,{10}]` yields `{1}`.
- `Rel[-,{10, 20}]` yields `{1, 2}`.

Right image: `[ ]` and `[ ,-]` Taking the right image of a relation amounts to selecting some elements from the range of a relation.

The *right* image operator takes a relation and an element `E` and produces a set consisting of all elements E_i in the range of the relation that occur in tuples of the form `<E,` E_i`>`. It can be defined as follows:

```
set[&T2] right-image(rel[&T1,&T2] R, &T1 E) =
   { W | <&T1 V, &T2 W> : R, V == E }
```

The right image operator can be extended to take a set of elements as second element instead of a single element:

```
set[&T2] right-image(rel[&T1,&T2] R, set[&T1] S) =
   { W | <&T1 V, &T2 W> : R, V in S}
```

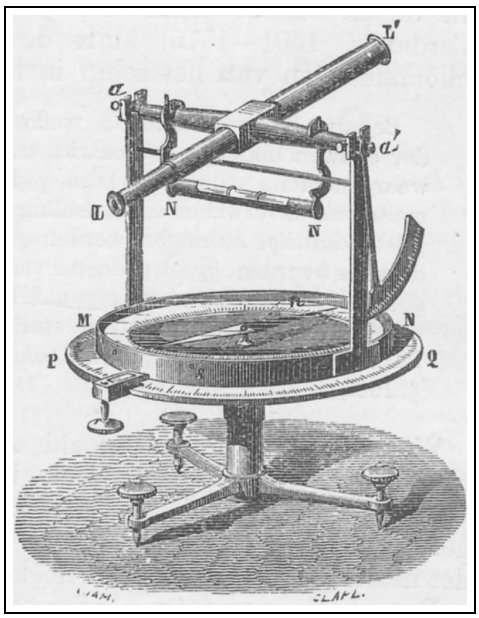
Examples Assume that `Rel` has value `{<1,10>, <2,20>, <1,11>, <3,30>, <2,21>}` in the following examples.

- `Rel[1]` yields `{10, 11}`.
- `Rel[{1}]` yields `{10, 11}`.
- `Rel[{1, 2}]` yields `{10, 11, 20, 21}`.

These expressions are abbreviations for, respectively `Rel[1,-]`, `Rel[{1},-]`. and `Rel[{1, 2},-]`.

Chapter 5

Built-in Functions



The built-in functions can be subdivided in several broad categories:

- Elementary functions on sets and relations (Section 5.1): identity (`id`), inverse (`inv`), complement (`compl`), and powerset (`power0`, `power1`).
- Extraction from relations (Section 5.2): domain (`domain`), range (`range`), and carrier (`carrier`).
- Restrictions and exclusions on relations (Section 5.3): domain restriction (`domainR`), range restriction (`rangeR`), carrier restriction (`carrierR`), domain exclusion (`domainX`), range exclusion (`rangeX`), and carrier exclusion (`carrierX`).
- Functions on tuples (Section 5.4): first element (`first`), and second element (`second`).
- Relations viewed as graphs (Section 5.5): the root elements (`top`), the leaf elements (`bottom`), reachability with restriction (`reachR`), and reachability with exclusion (`reachX`).
- Functions on locations (Section 5.6): file name (`filename`), beginning line (`beginline`), first column (`begincol`), ending line (`endline`), and ending column (`endcol`).
- Functions on sets of integers (Section 5.7): sum (`sum`), average (`average`), maximum (`max`), and minimum (`min`).

The following sections give detailed descriptions and examples of all built-in functions.

5.1 Elementary Functions on Sets and Relations

5.1.1 Identity Relation: `id`

Definition:

```
rel[&T, &T] id(set[&T] S) = { <X, X> | &T X : S}
```

Yields the relation that results from transforming each element in S into a pair with that element as first and second element. Examples:

- `id({1,2,3})` yields `{<1,1>, <2,2>, <3,3>}`.
- `id({"mon", "tue", "wed"})` yields `{<"mon","mon">, <"tue","tue">, <"wed","wed">}`.

5.1.2 Deprecated: Set with unique elements: `unique`

Definition:

```
set[&T] unique(set[&T] S) = primitive
```

Yields the set (actually the set) that results from removing all duplicate elements from S. This function stems from previous versions when we used bags instead of sets. It now acts as the identity function and is deprecated. Example:

- `unique({1,2,1,3,2})` yields `{1,2,3}`.

5.1.3 Inverse of a Relation: `inv`

Definition:

```
rel[&T2, &T1] inv (rel[&T1, &T2] R) = { <Y, X> | <&T1 X, &T2 Y> : R }
```

Yields the relation that is the inverse of the argument relation R, i.e. the relation in which the elements of all tuples in R have been interchanged. Example:

- `inv({<1,10>, <2,20>})` yields `{<10,1>,<20,2>}`.

5.1.4 Complement of a Relation: `compl`

Definition:

```
rel[&T1, &T2] compl(rel[&T1, &T2] R) = (domain(R) x range(R)) \ R}
```

Yields the relation that is the complement of the argument relation R, using the carrier set of R as universe. Example:

- `compl({<1,10>}` yields `{<1, 1>, <10, 1>, <10, 10>}`.

5.1.5 Powerset of a Set: `power0`

Definition:

```
set[set[&T]] power0(set[&T] S) = primitive
```

Yields the powerset of set S (including the empty set). Example:

- `power0({1, 2, 3, 4})` yields

```
{ {}, {1}, {2}, {3}, {4},{1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4},
  {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}, {1,2,3,4}}
```

5.1.6 Powerset of a Set: `power1`

Definition:

```
set[set[&T]] power1(set[&T] S) = primitive
```

Yields the powerset of set S (excluding the empty set). Example:

- `power1({1, 2, 3, 4})` yields

```
{ {1}, {2}, {3}, {4},{1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4},
  {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}, {1,2,3,4}}
```

5.2 Extraction from Relations

5.2.1 Domain of a Relation: `domain`

Definition:

```
set[&T1] domain (rel[&T1,&T2] R) = { X | <&T1 X, &T2 Y> : R }
```

Yields the set that results from taking the first element of each tuple in relation R. Examples:

- `domain({<1,10>, <2,20>})` yields `{1, 2}`.
- `domain({<"mon", 1>, <"tue", 2>})` yields `{"mon", "tue"}`.

5.2.2 Range of a Relation: `range`

Definition:

```
set[&T2] range (rel[&T1,&T2] R) = { Y | <&T1 X, &T2 Y> : R }
```

Yields the set that results from taking the second element of each tuple in relation R. Examples:

- `range({<1,10>, <2,20>})` yields `{10, 20}`.
- `range({<"mon", 1>, <"tue", 2>})` yields `{1, 2}`.

5.2.3 Carrier of a Relation: `carrier`

Definition:

```
set[&T]  carrier (rel[&T,&T] R) = domain(R) union range(R)
```

Yields the set that results from taking the first and second element of each tuple in the relation R. Note that the domain and range type of R should be the same. Example:

- `carrier({<1,10>, <2,20>})` yields `{1, 10, 2, 20}`.

5.3 Restrictions and Exclusions on Relations

5.3.1 Domain Restriction of a Relation: `domainR`

Definition:

```
rel[&T1,&T2] domainR (rel[&T1,&T2] R, set[&T1] S) =
   { <X, Y> | <&T1 X, &T2 Y> : R, X in S }
```

Yields a relation identical to the relation R but only containing tuples whose first element occurs in set S. Example:

- `domainR({<1,10>, <2,20>, <3,30>}, {3, 1}` yields `{<1,10>, <3,30>}`.

5.3.2 Range Restriction of a Relation: `rangeR`

Definition:

```
rel[&T1,&T2] rangeR (rel[&T1,&T2] R, set[&T2] S) =
   { <X, Y> | <&T1 X, &T2 Y> : R, Y in S }
```

Yields a relation identical to relation R but only containing tuples whose second element occurs in set S. Example:

- `rangeR({<1,10>, <2,20>, <3,30>}, {30, 10}` yields `{<1,10>, <3,30>}`.

5.3.3 Carrier Restriction of a Relation: `carrierR`

Definition:

```
rel[&T,&T] carrierR (rel[&T,&T] R, set[&T] S) =
   { <X, Y> | <&T X, &T Y> : R, X in S, Y in S }
```

Yields a relation identical to relation R but only containing tuples whose first and second element occur in set S. Example:

- `carrierR({<1,10>, <2,20>, <3,30>}, {10, 1, 20})` yields `{<1,10>}`.

5.3.4 Domain Exclusion of a Relation: `domainX`

Definition:

```
rel[&T1,&T2] domainX (rel[&T1,&T2] R, set[&T1] S) =
   { <X, Y> | <&T1 X, &T2 Y> : R, X notin S }
```

Yields a relation identical to relation R but with all tuples removed whose first element occurs in set S. Example:

- `domainX({<1,10>, <2,20>, <3,30>}, {3, 1})` yields `{<2, 20>}`.

5.3.5 Range Exclusion of a Relation: `rangeX`

Definition:

```
rel[&T1,&T2] rangeX (rel[&T1,&T2] R, set[&T2] S) =
   { <X, Y> | <&T1 X, &T2 Y> : R, Y notin S }
```

Yields a relation identical to relation R but with all tuples removed whose second element occurs in set S. Example:

- `rangeX({<1,10>, <2,20>, <3,30>}, {30, 10})` yields `{<2, 20>}`.

5.3.6 Carrier Exclusion of a Relation: `carrierX`

Definition:

```
rel[&T,&T] carrierX (rel[&T,&T] R, set[&T] S) =
   { <X, Y> | <&T1 X, &T2 Y> : R, X notin S, Y notin S }
```

Yields a relation identical to relation R but with all tuples removed whose first or second element occurs in set S. Example:

- `carrierX({<1,10>, <2,20>, <3,30>}, {10, 1, 20})` yields `{<3,30>}`.

5.4 Tuples

5.4.1 First Element of a Tuple: `first`

Definition:

```
&T1 first(<&T1, &T2> P) = primitive
```

Yields the first element of the tuple P. Examples:

- `first(<1, 10>)` yields `1`.
- `first(<"mon", 1>)` yields `"mon"`.

5.4.2 Second Element of a Tuple: `second`

Definition:

```
&T2 second(<&T1, &T2> P) = primitive
```

Yields the second element of the tuple `P`. Examples:

- `second(<1, 10>)` yields `10`.
- `second(<"mon", 1>)` yields `1`.

5.5 Relations viewed as graphs

5.5.1 Top of a Relation: `top`

Definition:

```
set[&T] top(rel[&T, &T] R) = unique(domain(R)) \ range(R)
```

Yields the set of all roots when the relation `R` is viewed as a graph. Note that the domain and range type of `R` should be the same. Example:

- `top({<1,2>, <1,3>, <2,4>, <3,4>})` yields `{1}`.

5.5.2 Bottom of a Relation: `bottom`

Definition:

```
set[&T] bottom(rel[&T,&T] R) = unique(range(R)) \ domain(R)
```

Yields the set of all leaves when the relation `R` is viewed as a graph. Note that the domain and range type of `R` should be the same. Example:

- `bottom({<1,2>, <1,3>, <2,4>, <3,4>})` yields `{4}`.

5.5.3 Reachability with Restriction: `reachR`

Definition:

```
set[&T] reachR( set[&T] Start, set[&T] Restr, rel[&T,&T] Rel) =
   range(domainR(Rel, Start) o carrierR(Rel, Restr)+)
```

Yields the elements that can be reached from set `Start` using the relation `Rel`, such that only elements in set `Restr` are used. Example:

- `reachR({1}, {1, 2, 3}, {<1,2>, <1,3>, <2,4>, <3,4>})` yields `{2, 3}`.

5.5.4 Reachability with Exclusion: `reachX`

Definition:

```
set[&T] reachX( set[&T] Start, set[&T] Excl, rel[&T,&T] Rel) =
   range(domainR(Rel, Start) o carrierX(Rel, Excl)+)
```

Yields the elements that can be reached from set `Start` using the relation `Rel`, such that no elements in set `Excl` are used. Example:

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

5.6 Functions on Locations

5.6.1 File Name of a Location: `filename`

Definition:

```
str filename(loc A) = primitive
```

Yields the file name of location A. Example:

- `filename(area-in-file("pico1.trm",area(5,2,6,8,0,0)))` yields `"pico1.trm"`.

5.6.2 Beginning Line of a Location: `beginline`

Definition:

```
int beginline(loc A) = primitive
```

Yields the first line of location A. Example:

- `beginline(area-in-file("pico1.trm",area(5,2,6,8,0,0)))` yields 5.

5.6.3 First Column of a Location: `begincol`

Definition:

```
int begincol(loc A) = primitive
```

Yields the first column of location A. Example:

- `begincol(area-in-file("pico1.trm",area(5,2,6,8,0,0)))` yields 2.

5.6.4 Ending Line of a Location: `endline`

Definition:

```
int endline(loc A) = primitive
```

Yields the last line of location A. Example:

- `endline(area-in-file("pico1.trm",area(5,2,6,8,0,0)))` yields 6.

5.6.5 Ending Column of a Location: `endcol`

Definition:

```
int endcol(loc A) = primitive
```

Yields the last column of location A. Example:

- `endcol(area-in-file("pico1.trm",area(5,2,6,8,0,0)))` yields 8.

5.7 Functions on Sets of Integers

The functions in this section operate on sets of integers. Some functions (i.e., `sum-domain`, `sum-range`, `average-domain`, `average-range`) exist to solve the problem that we can only provide sets of integers and cannot model bags that may contain repeated occurrences of the same integer. For some calculations it is important to include these repetitions in the calculation (e.g., computing the average length of class methods given a relation from methods names to number of lines in the method.)

5.7.1 Sum of a Set of Integers: `sum`

Definition:

```
int sum(set[int] S) = primitive
```

Yields the sum of the integers in set S. Example:

- `sum({1, 2, 3})` yields 6.

5.7.2 Sum of First Elements of Tuples in a Relation: `sum-domain`

Definition:

```
int sum-domain(rel[int,&T] R) = primitive
```

Yields the sum of the integers that appear in the first element of the tuples of R. Example:

- `sum-domain({<1,"a">, <2,"b">, <1,"c">})` yields 4.

Be aware that `sum(domain({<1,"a">, <2,"b"">, <1, "c">}))` would be equal to 3 because the function `domain` creates a *set* (as opposed to a bag) and its result would thus contain only one occurrence of 1.

5.7.3 Sum of Second Elements of Tuples in a Relation: `sum-range`

Definition:

```
int sum-range(set[int] S) = primitive
```

Yields the sum of the integers that appear in the second element of the tuples of R. Example:

- `sum-range({<"a",1>, <"b",2>, <"c",1>})` yields 4.

5.7.4 Average of a Set of Integers: `average`

Definition:

```
int average(set[int] S) = sum(S)/(#S)
```

Yields the average of the integers in set S. Example:

- `average({1, 2, 3})` yields 3.

5.7.5 Average of First Elements of Tuples in a Relation: `average-domain`

Definition:

```
int average-domain(rel[int,&T] R) = sum-domain(R)/(#R)
```

Yields the average of the integers that appear in the first element of the tuples of R. Example:

- `average({<1,"a">, <2,"b">, <3,"c">})` yields 2.

5.7.6 Average of Second Elements of Tuples in a Relation: `average-range`

Definition:

```
int average(rel[&T,int] R) = sum-range(R)/(#R)
```

Yields the average of the integers that appear in the second element of the tuples of R. Example:

- `average({<"a",1>, <"b",2>, <"c",3>})` yields 2.

5.7.7 Maximum of a Set of Integers: `max`

Definition:

```
int max(set[int] S) = primitive
```

Yields the largest integer in set S. Example:

- `max({1, 2, 3})` yields 3.

5.7.8 Minimum of a Set of Integers: `min`

Definition:

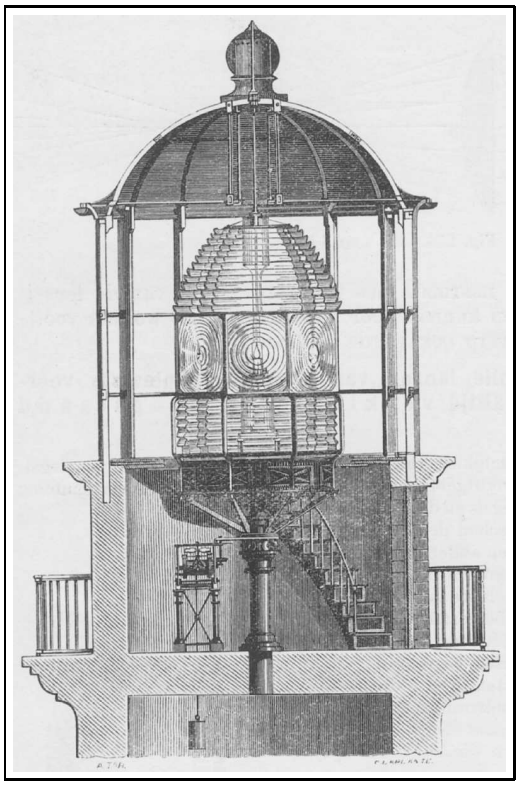
```
int min(set[int] S) = primitive
```

Yields the smallest integer in set S. Example:

- `min({1, 2, 3})` yields 1.

Chapter 6

Larger Examples



Now we will have a closer look at some larger applications of RSCRIPT. We start by analyzing the global structure of a software system. You may now want to reread the example of call graph analysis given earlier in Chapter 2 as a reminder. The component structure of an application is analyzed in Section 6.1 and Java systems are analyzed in Section 6.2. Next we move on to the detection of initialized variables in Section 6.3 and we explain how source code locations can be included in a such an analysis (Section 6.4).
As an example of computing code metrics, we describe the calculation of McCabe's cyclomatic complexity in Section 6.5. Several examples of dataflow analysis follow in Section 6.6. A description of program slicing concludes the chapter (Section 6.7).

6.1 Analyzing the Component Structure of an Application

A frequently occurring problem is that we know the call relation of a system but that we want to understand it at the component level rather than at the procedure level. If it is known to which component each procedure belongs, it is possible to *lift* the call relation to the component level as proposed in [16].

First, introduce new types to denote procedure calls as well as components of a system:

```
type proc = str
type comp = str
```

Given a calls relation `Calls2`, the next step is to define the components of the system and to define a `PartOf` relation between procedures and components.

```
rel[proc,proc] Calls = {<"main", "a">, <"main", "b">, <"a", "b">,
```

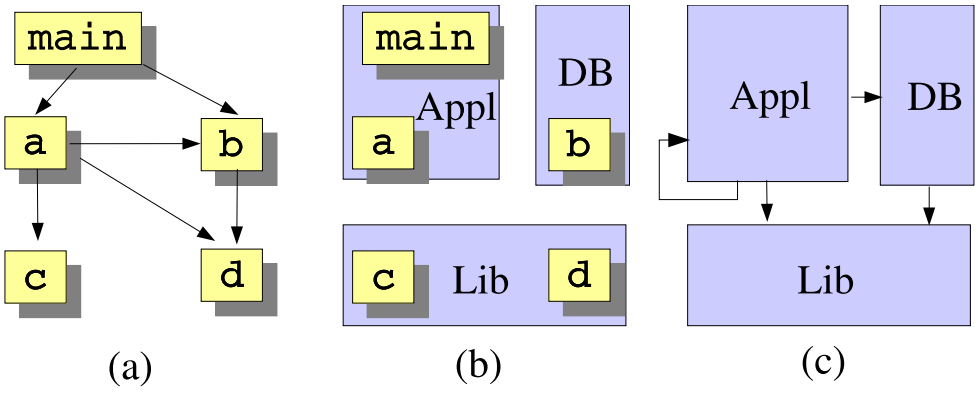


Figure 6.1: (a) `Calls2`; (b) `PartOf`; (c) `ComponentCalls`

```
                          <"a", "c">, <"a", "d">, <"b", "d">}

set[comp] Components = {"Appl", "DB", "Lib"}

rel[proc, comp] PartOf = {<"main", "Appl">, <"a", "Appl">, <"b", "DB">,
                          <"c", "Lib">, <"d", "Lib">}
```

Actual lifting, amounts to translating each call between procedures by a call between components. This is achieved by the following function `lift`:

```
rel[comp,comp] lift(rel[proc,proc] aCalls,  rel[proc,comp] aPartOf) =
   { <C1, C2> | <proc P1, proc P2> : aCalls,
                <comp C1, comp C2> : aPartOf[P1] x aPartOf[P2]
   }
```

In our example, the lifted call relation between components is obtained by

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

and has as value:

```
{<"DB", "Lib">, <"Appl", "Lib">, <"Appl", "DB">, <"Appl", "Appl">}
```

The relevant relations for this example are shown in Figure 6.1.

6.2 Analyzing the Structure of Java Systems

Now we consider the analysis of Java systems (inspired by [3]). Suppose that the type `class` is defined as follows

```
type class = str
```

and that the following relations are available about a Java application:

- `rel[class,class] CALL`: If <C_1, C_2> is an element of `CALL`, then some method of C_2 is called from C_1.
- `rel[class,class] INHERITANCE`: If <C_1, C_2> is an element of `INHERITANCE`, then class C_1 either extends class C_2 or C_1 implements interface C_2.
- `rel[class,class] CONTAINMENT`: If <C_1, C_2> is an element of `CONTAINMENT`, then one of the attributes of class C_1 is of type C_2.

```
package CH.ifa.draw.standard;

import java.awt.Point;
import CH.ifa.draw.framework.*;
/**
 * A LocatorHandle implements a Handle by delegating the location requests to
 * a Locator object.
 */
public class LocatorHandle extends AbstractHandle {
    private Locator       fLocator;
    /**
     * Initializes the LocatorHandle with the given Locator.
     */
    public LocatorHandle(Figure owner, Locator l) {
        super(owner);
        fLocator = l;
    }
    /**
     * Locates the handle on the figure by forwarding the request
     * to its figure.
     */
    public Point locate() {
        return fLocator.locate(owner());
    }
}
```

Figure 6.2: The class `LocatorHandle` from JHotDraw 5.2

To make this more explicit, consider the class `LocatorHandle` from the JHotDraw application (version 5.2) as shown in Figure 6.2. It leads to the addition to the above relations of the following tuples:

- To `CALL` the pairs `<"LocatorHandle", "AbstractHandle">` and `<"LocatorHandle", "Locator">` will be added.
- To `INHERITANCE` the pair `<"LocatorHandle", "AbstractHandle">` will be added.
- To `CONTAINMENT` the pair `<"LocatorHandle", "Locator">` will be added.

Classes in Cycles Cyclic structures in object-oriented systems makes understanding hard. Therefore it is interesting to spot classes that occur as part of a cyclic dependency. Here we determine cyclic uses of classes that include calls, inheritance and containment. This is achieved as follows:

```
rel[class,class] USE = CALL union CONTAINMENT union INHERITANCE
set[str] ClassesInCycle =
   {C1 | <class C1, class C2> : USE+, C1 == C2}
```

First, we define the `USE` relation as the union of the three available relations `CALL`, `CONTAINMENT` and `INHERITANCE`. Next, we consider all pairs `<C1, C2>` in the transitive closure of the `USE` relation such that `C1` and `C2` are equal. Those are precisely the cases of a class with a cyclic dependency on itself.

Probably, we do not only want to know which classes occur in a cyclic dependency, but we also want to know which classes are involved in such a cycle. In other words, we want to associate with each class a set of classes that are responsible for the cyclic dependency. This can be done as follows.

```
rel[class,class] USE = CALL union CONTAINMENT union INHERITANCE
set[class] CLASSES = carrier(USE)
rel[class,class] USETRANS = USE+
```

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

First, we introduce two new shorthands: `CLASSES` and `USETRANS`. Next, we consider all classes `C` with a cyclic dependency and add the pair `<C, USETRANS[C]>` to the relation `ClassCycles`. Note that `USETRANS[C]` is the right image of the relation `USETRANS` for element `C`, i.e., all classes that can be called transitively from class `C`.

6.3 Finding Uninitialized and Unused Variables in a Program

Consider the following program in the toy language Pico: [1]

```
[ 1] begin declare x : natural, y : natural,
[ 2]             z : natural, p : natural;
[ 3]  x := 3;
[ 4]  p := 4;
[ 5]  if q then
[ 6]       z := y + x
[ 7]  else
[ 8]       x := 4
[ 9]  fi;
[10]  y := z
[11] end
```

Inspection of this program learns that some of the variables are being used before they have been initialized. The variables in question are `q` (line 5), `y` (line 6), and `z` (line 10). It is also clear that variable `p` is initialized (line 4), but is never used. How can we automate these kinds of analysis?

Recall from Section 1 that we follow extract-enrich-view paradigm to approach such a problem.

The first step is to determine which elementary facts we need about the program. For this and many other kinds of program analysis, we need at least the following:

- The *control flow graph* of the program. We represent it by a relation `PRED` (for predecessor) which relates each statement with each predecessors.
- The *definitions* of each variable, i.e., the program statements where a value is assigned to the variable. It is represented by the relation `DEFS`.
- The *uses* of each variable, i.e., the program statements where the value of the variable is used. It is represented by the relation `USES`.

In this example, we will use line numbers to identify the statements in the program.[2]

Assuming that there is a tool to extract the above information from a program text, we get the following for the above example:

```
type expr = int
type varname = str
expr ROOT = 1
rel[expr,expr] PRED = { <1,3>, <3,4>, <4,5>, <5,6>, <5,8>, <6,10>, <8,10> }
rel[expr,varname] DEFS = {<3,"x">, <4,"p">, <6,"z">, <8,"x">, <10,"y">}
rel[expr,varname] USES = {<5,"q">, <6,"y">, <6,"x">, <10,"z">}
```

This concludes the extraction phase. Next, we have to enrich these basic facts to obtain the initialized variables in the program.

[1] This is an extended version of the example presented earlier in [14].

[2] In Section 6.4, we will use locations to represent statements.

So, when is a variable *V* in some statement *S* initialized? If we execute the program (starting in ROOT), there may be several possible execution path that can reach statement *S*. All is well if *all* these execution path contain a definition of *V*. However, if one or more of these path do *not* contain a definition of *V*, then *V* may be uninitialized in statement *S*. This can be formalized as follows:

```
rel[expr,varname] UNINIT =
   { <E, V> | <expr E, varname V>: USES,
              E in reachX({ROOT}, DEFS[-,V], PRED)
   }
```

We analyze this definition in detail:

- `<expr E, varname V> : USES` enumerates all tuples in the USES relation. In other words, we consider the use of each variable in turn.
- `E in reachX(ROOT, DEFS[-,V], PRED)` is a test that determines whether statement S is reachable from the ROOT without encountering a definition of variable V.
  - {ROOT} represents the initial set of nodes from which all path should start.
  - `DEFS[-,V]` yields the set of all statements in which a definition of variable V occurs. These nodes form the exclusion set for `reachX`: no path will be extended beyond an element in this set.
  - PRED is the relation for which the reachability has to be determined.
  - The result of `reachX(ROOT, DEFS[-,V], PRED)` is a set that contains all nodes that are reachable from the ROOT (as well as all intermediate nodes on each path).
  - Finally, `E in reachX(ROOT, DEFS[-,V], PRED)` tests whether expression E can be reached from the ROOT.
- The net effect is that UNINIT will only contain pairs that satisfy the test just described.

When we execute the resulting RSCRIPT (i.e., the declarations of ROOT, PRED, DEFS, USES and UNINIT), we get as value for UNINIT:

```
{<5, "q">, <6, "y">, <10, "z">}
```

and this is in concordance with the informal analysis given at the beginning of this example.

As a bonus, we can also determine the *unused* variables in a program, i.e., variables that are defined but are used nowhere. This is done as follows:

```
set[var] UNUSED = range(DEFS) \ range(USES)
```

Taking the range of the relations DEFS and USES yields the variables that are defined, respectively, used in the program. The difference of these two sets yields the unused variables, in this case `{"p"}`.

6.4 Using Locations to Represent Program Fragments

One aspect of the example we have just seen is artificial: where do these line numbers come from that we used to indicate expressions in the program? One solution is to let the extraction phase generate *locations* to precisely indicate relevant places in the program text.

Recall from Section 3.1.1, that a location consists of a file name, a begin line, a begin position, an end line, and an end position. Also recall that locations can be compared: a location A_1 is smaller than another location A_2, if A_1 is textually contained in A_2. By including locations in the final answer of a relational expression, external tools will be able to highlight places of interest in the source text.

The first step, is to define the type `expr` as aliases for `loc` (instead of `int`):

```
type expr = loc
type varname = str
```

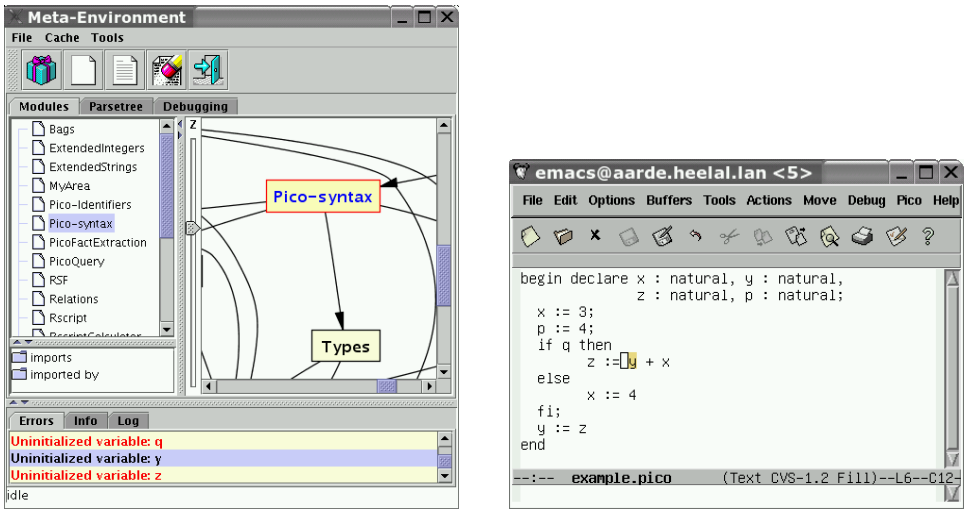


Figure 6.3: Checking undefined variables in Pico programs using the ASF+SDF Meta-Environment. On the left, main window of Meta-Environment with error messages related to Pico program shown on the right.**THIS FIGURE IS OUTDATED**

Of course, the actual relations are now represented differently. For instance, the USES relation may now look like

```
{ <area-in-file("/home/paulk/example.pico", area(5,5,5,6,106,1)), "q">,
  <area-in-file("/home/paulk/example.pico", area(6,13,6,14,127,1)), "y">,
  <area-in-file("/home/paulk/example.pico", area(6,17,6,18,131,1)), "x">,
  <area-in-file("/home/paulk/example.pico", area(10,7,10,8,168,1)), "z">
}
```

The definition of UNINIT can be nearly reused as is. The only thing that remains to be changed is to map the (expression, variable-name) tuples to (variable-name, variable-occurrence) tuples, for the benefit of the precise highlighting of the relevant variables.

We define a new type var to represent variable occurrences and an auxiliary set that VARNAMES that contains all variable names:

```
type var = loc
set[varname] VARNAMES  = range(DEFS) union range(USES)
```

Remains the new definition of UNINIT:

```
rel[var, varname] UNINIT =
   { <V, VN>| var-name VN : VARNAMES,
              var V : USES[-,VN],
              expr E : reachX({ROOT}, DEFS[-,VN], PRED),
      V <= E
   }
```

This definition can be understood as follows:

- var-name VN : VARNAMES generates all variable names.
- var V : USES[-,VN] generates all variable uses V for variables with name VN.

- As before, `expr E : reachX(ROOT, DEFS[-,VN], PRED)` generates all expressions `E` that can be reached from the start of the program without encountering a definition for variables named `VN`.
- `V <= E` tests whether variable use `V` is enclosed in that expression (using a comparison on locations). If so, we have found an uninitialized occurrence of the variable named `VN`.

In Figure 6.3 it is shown how checking of Pico programs in the ASF+SDF Meta-Environment looks like.

6.5 McCabe Cyclomatic Complexity

The *cyclomatic complexity* of a program is defined as $e - n + 2$, where e and n are the number of edges and nodes in the control flow graph, respectively. It was proposed by McCabe [18] as a measure of program complexity.

Experiments have shown that programs with a higher cyclomatic complexity are more difficult to understand and test and have more errors. It is generally accepted that a program, module or procedure with a cyclomatic complexity larger than 15 is *too complex*. Essentially, cyclomatic complexity measures the number of decision points in a program and can be computed by counting all if statement, case branches in switch statements and the number of conditional loops.

Given a control flow in the form of a predecessor relation `rel[stat,stat] PRED` between statements, the cyclomatic complexity can be computed in an RSCRIPT as follows:

```
int cyclomatic-complexity(rel[stat,stat] PRED) =
    #PRED - #carrier(PRED) + 2
```

The number of edges e is equal to the number of tuples in `PRED`. The number of nodes n is equal to the number of elements in the carrier of `PRED`, i.e., all elements that occur in a tuple in `PRED`.

6.6 Dataflow Analysis

Dataflow analysis is a program analysis technique that forms the basis for many compiler optimizations. It is described in any text book on compiler construction, e.g. [1]. The goal of dataflow analysis is to determine the effect of statements on their surroundings. Typical examples are:

- Dominators (Section 6.6.1): which nodes in the flow dominate the execution of other nodes?
- Reaching definitions (Section 6.6.2): which definitions of variables are still valid at each statement?
- Live variables (Section 6.6.3): of which variables will the values be used by successors of a statement?
- Available expressions: an expression is available if it is computed along each path from the start of the program to the current statement.
- and more.

6.6.1 Dominators

A node d of a flow graph *dominates* a node n, if every path from the initial node of the flow graph to n goes through d [1, Section 10.4]. Dominators play a role in the analysis of conditional statements and loops. In Figure 6.4, we show the function `dominators` that computes the dominators for a given flow graph `PRED` and an entry node `ROOT`. First, the auxiliary set `VERTICES` (all the statements) is computed. The relation `DOMINATES` consists of all pairs <S, S_1,...,S_n> such that

- S_i is not an initial node or equal to `S`.

```
rel[stat,stat] dominators(rel[stat,stat] PRED, int ROOT) =
   DOMINATES
where
    set[int] VERTICES = carrier(PRED)

    rel[int,set[int]] DOMINATES =
    { <V,  VERTICES \ {V, ROOT} \ reachX({ROOT}, {V}, PRED)> |  int V : VERTICES }
endwhere
```

Figure 6.4: The function `dominators`

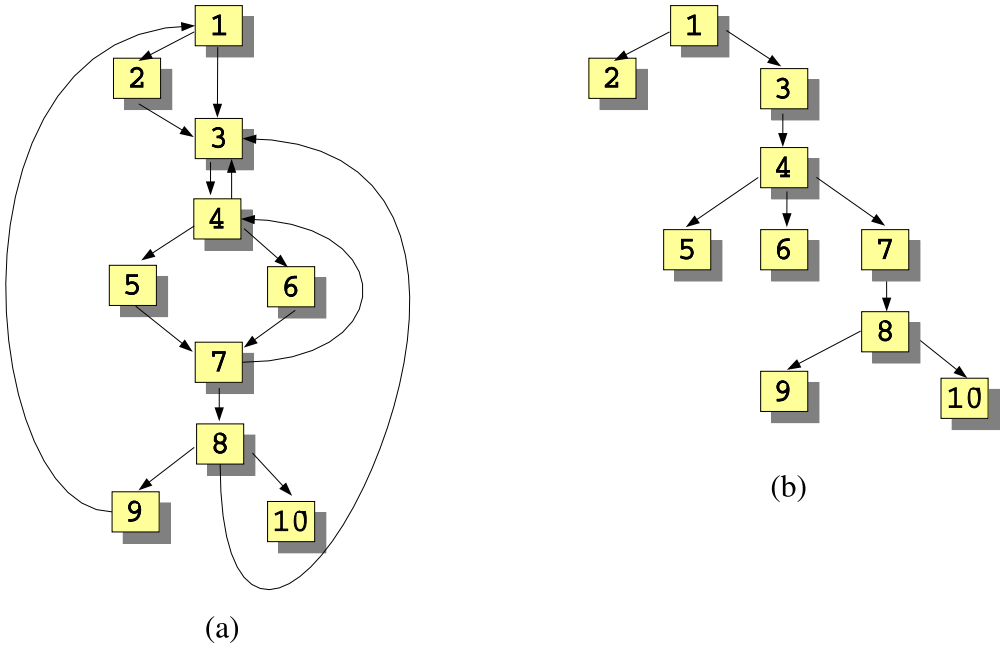


Figure 6.5: (a) Flow graph and (b) dominator tree

- S_i cannot be reached from the initial node without going through S.

Consider the flow graph

```
rel[int,int] PRED = {
<1,2>, <1,3>,
<2,3>,
<3,4>,
<4,3>,<4,5>, <4,6>,
<5,7>,
<6,7>,
<7,4>,<7,8>,
<8,9>,<8,10>,<8,3>,
<9,1>,
<10,7>
}
```

and the result of applying `dominators` to it:

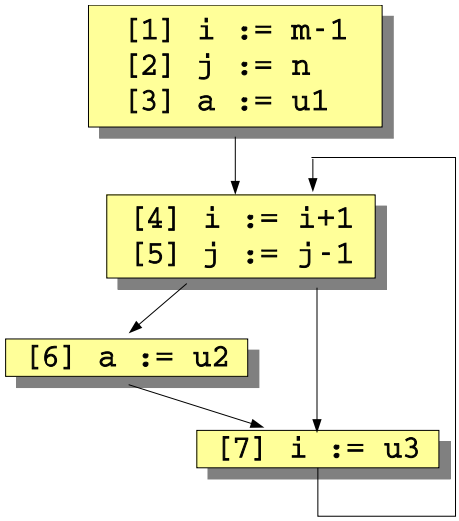


Figure 6.6: Flow graph for various dataflow problems

```
{<1, {2, 3, 4, 5, 6, 7, 8, 9, 10}>,
<2, {}>,
<3, {4, 5, 6, 7, 8, 9, 10}>,
<4, {5, 6, 7, 8, 9, 10}>,
<5, {}>,
<6, {}>,
<7, {8, 9, 10}>,
<8, {9, 10}>,
<9, {}>,
<10, {}>}
```

The original flow graph and the resulting *dominator tree* are shown in Figure 6.5. The dominator tree has the initial node as root and each node d in the tree only dominates its descendants in the tree.

6.6.2 Reaching Definitions

We illustrate the calculation of reaching definitions using the example in Figure 6.6 which was inspired by [1, Example 10.15].

As before, we assume the following basic relations PRED, DEFS and USES about the program:

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

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

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

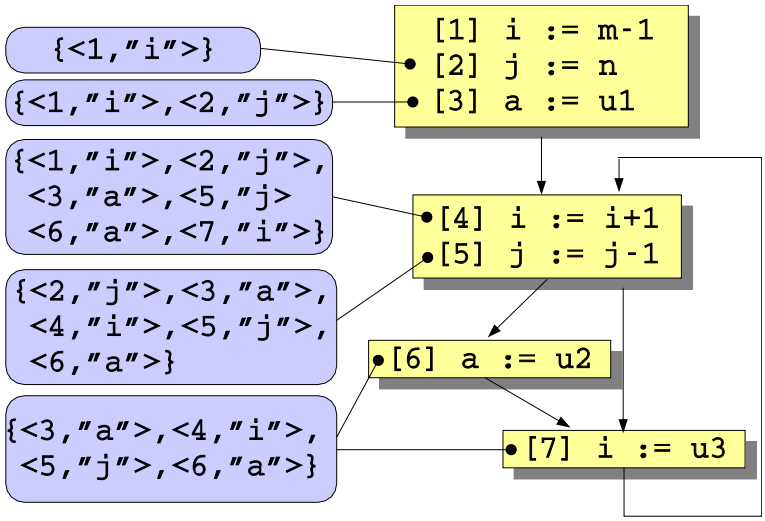


Figure 6.7: Reaching definitions for flow graph in Figure 6.6

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

The new DEF relation gets as value:

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

and USE gets as value:

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

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

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

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

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

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

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

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

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

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

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

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

This idea can be expressed in RSCRIPT quite literally:

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

First, the relations `IN` and `OUT` are declared and initialized. Next, two equations are given that very much resemble the ones given above. For our running example (Figure 6.7) the results are as follows. Relation `IN` has as value:

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

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

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

Relation `OUT` has as value:

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

Observe, again for statement `4`, that all definitions of variable `i` are missing in `OUT[4]` since they are killed by the definition of `i` in statement `4` itself. Definitions for `a` and `j` are, however, contained in `OUT[4]`. The result of reaching definitions computation is illustrated in Figure 6.7.

In Figure 6.8 the above definitions are used to formulate the function `reaching-definitions`. It assumes appropriate definitions for the types `stat` and `var`. It also assumes more general versions of `predecessor` and `successor`. We will use it later on in Section 6.7 when defining program slicing.

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

set[stat] predecessor(rel[stat,stat] P, stat S) = P[-,S]

set[stat] successor(rel[stat,stat] P, stat S) = P[S,-]

rel[stat, def] reaching-definitions(rel[stat,var] DEFS, rel[stat,stat] PRED) =
   IN
where
    set[stat] STATEMENT = carrier(PRED)

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

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

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

Figure 6.8: Reaching definitions

6.6.3 Live Variables

The live variables of a statement are those variables whose value will be used by the current statement or some successor of it. The mathematical formulation of this problem is as follows:

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

$$OUT[S] = \bigcup_{S' \in successorofS} IN[S']$$

The first equation says that a variable is live coming into a statement if either it is used before redefinition in that statement or it is live coming out of the statement and is not redefined in it. The second equation says that a variable is live coming out of a statement if and only if it is live coming into one of its successors. This can be expressed in RSCRIPT as follows:

```
 equations
   initial
         rel[stat,def] LIN init {}
         rel[stat,def] LOUT init DEF
   satisfy
         LIN  =  { < S, D> | stat S : STATEMENTS,
                             def D : USE[S] union (LOUT[S] \ (DEF[S]))}
         LOUT =  { < S, D> | stat S : STATEMENTS,
                             stat Succ : successor(S),
                             def D : LIN[Succ] }
```

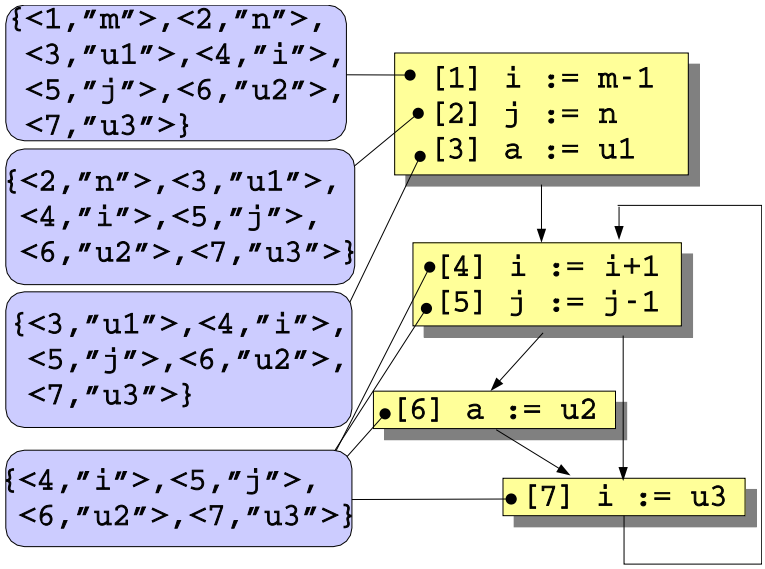


Figure 6.9: Live variables for flow graph in Figure 6.6

```
end equations
```

The results of live variable analysis for our running example are illustrated in Figure 6.9.

6.7 Program Slicing

Program slicing is a technique proposed by Weiser [28] for automatically decomposing programs in parts by analyzing their data flow and control flow. Typically, a given statement in a program is selected as the *slicing criterion* and the original program is reduced to an independent subprogram, called a *slice*, that is guaranteed to represent faithfully the behavior of the original program at the slicing criterion. An example is given in Figure 6.10. The initial program is given in Figure 6.10(a). The slice with statement `[9]` as slicing criterion is shown in Figure 6.10(b): statements `[4]` and `[7]` are irrelevant for computing statement `[9]` and do not occur in the slice. Similarly, Figure 6.10(c) shows the slice with statement `[10]` as slicing criterion. This particular form of slicing is called *backward slicing*. Slicing can be used for debugging and program understanding, optimization and more. An overview of slicing techniques and applications can be found in [23].

Here we will explore a relational formulation of slicing adapted from a proposal in [13]. The basic ingredients of the approach are as follows:

- We assume the relations `PRED`, `DEFS` and `USES` as before.
- We assume an additional set `CONTROL-STATEMENT` that defines which statements are control statements.
- To tie together dataflow and control flow, three auxiliary variables are introduced:
  - The variable `TEST` represents the outcome of a specific test of a conditional statement. The conditional statement defines `TEST` and all statements that are control dependent on this conditional statement will use `TEST`.
  - The variable `EXEC` represents the potential execution dependence of a statement on some conditional statement. The dependent statement defines `EXEC` and an explicit (control) dependence is made between `EXEC` and the corresponding `TEST`.

```
[ 1] read(n)
[ 2] i := 1
[ 3] sum := 0
[ 4] product := 1
[ 5] while i<= n do
     begin
[ 6]   sum := sum + i
[ 7]   product :=
         product * i
[ 8]   i := i + 1
     end
[ 9] write(sum)
[10] write(product)
```

(a)

```
[ 1] read(n)
[ 2] i := 1
[ 3] sum := 0
[ 5] while i<= n do
     begin
[ 6]   sum := sum + i
[ 8]  i := i + 1
     end
[ 9] write(sum)
```

(b)

```
[ 1] read(n)
[ 2] i := 1
[ 4] product := 1
[ 5] while i<= n do
     begin
[ 7]   product :=
         product * i
[ 8]   i := i + 1
     end
[10] write(product)
```

(c)

Figure 6.10: (a) Sample program, (b) slice for statement `[9]`, (c) slice for statement `[10]`

– The variable `CONST` represents an arbitrary constant.

The calculation of a (backward) slice now proceeds in six steps:

1. Compute the relation `rel[use,def] use-def` that relates all uses to their corresponding definitions. The function `reaching-definitions` as shown earlier in Figure 6.8 does most of the work.
2. Compute the relation `rel[def,use] def-use-per-stat` that relates the "internal" definitions and uses of a statement.
3. Compute the relation `rel[def,use] control-dependence` that links all `EXEC`s to the corresponding `TEST`s.
4. Compute the relation `rel[use,def] use-control-def` combines use/def dependencies with control dependencies.
5. After these preparations, compute the relation `rel[use,use] USE-USE` that contains dependencies of uses on uses.
6. The backward slice for a given slicing criterion (a use) is now simply the projection of `USE-USE` for the slicing criterion.

This informal description of backward slicing is described precisely in Figure 6.11. Let's apply this to the example in Figure 6.10 and assume the following:

```
rel[stat,stat] PRED = {<1,2>, <2,3>, <3,4>, <4,5>, <5,6>, <5,9>, <6,7>,
                       <7,8>,<8,5>, <8,9>, <9,10>}

rel[stat,var] DEFS  = {<1, "n">, <2, "i">, <3, "sum">, <4,"product">,
                       <6, "sum">, <7, "product">, <8, "i">}

rel[stat,var] USES  = {<5, "i">, <5, "n">, <6, "sum">, <6,"i">,
                       <7, "product">, <7, "i">, <8, "i">, <9, "sum">,
                       <10, "product">}

set[int] CONTROL-STATEMENT = { 5 }
```

The result of the slice

```
set[use] BackwardSlice(
    set[stat] CONTROL-STATEMENT,
    rel[stat,stat] PRED,
    rel[stat,var] USES,
    rel[stat,var] DEFS,
    use Criterion)
= USE-USE[Criterion]

where
    rel[stat, def] REACH = reaching-definitions(DEFS, PRED)

    rel[use,def] use-def =
       {<<S1,V>, <S2,V>> | <stat S1, var V> : USES, <stat S2, V> : REACH[S1]}

    rel[def,use] def-use-per-stat  =
       {<<S,V1>, <S,V2>> | <stat S, var V1> : DEFS, <S, var V2> : USES}
             union
       {<<S,V>, <S,"EXEC">> | <stat S, var V> : DEFS}
             union
       {<<S,"TEST">,<S,V>> | stat S : CONTROL-STATEMENT,
                             <S, var V> : domainR(USES, {S})}

    rel[stat, stat] CONTROL-DOMINATOR =
       domainR(dominators(PRED), CONTROL-STATEMENT)

    rel[def,use] control-dependence  =
       { <<S2, "EXEC">,<S1,"TEST">> | <stat S1, stat S2> : CONTROL-DOMINATOR}

    rel[use,def] use-control-def = use-def union control-dependence

    rel[use,use] USE-USE = (use-control-def o def-use-per-stat)*

endwhere
```

Figure 6.11: Backward slicing

```
BackwardSlice(CONTROL-STATEMENT, PRED, USES, DEFS, <9, "sum">)
```

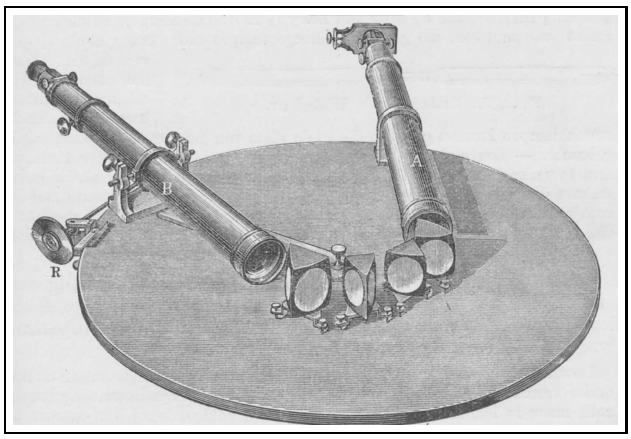
will then be

```
{ <1, "EXEC">, <2, "EXEC">,  <3, "EXEC">, <5, "i">, <5, "n">,
  <6, "sum">, <6, "i">, <6, "EXEC">, <8, "i">, <8, "EXEC">,
  <9, "sum"> }.
```

Take the domain of this result and we get exactly the statements in Figure 6.10(b).

Chapter 7

Extracting Facts from Source Code



In this tutorial we have, so far, concentrated on querying and enriching facts that have been extracted from source code. As we have seen from the examples, once these facts are available, a concise RSCRIPT suffices to do the required processing. But how is fact extraction achieved and how difficult is it? To answer these questions we first describe the workflow of the fact extraction process (Section 7.1) and then we give a more detailed account of fact extraction using ASF+SDF (Section 7.2).

7.1 Workflow for Fact Extraction

Figure 7.1 shows a typical workflow for fact extraction for a *System Under Investigation* (SUI). It assumes that the SUI uses only *one* programming language and that you need only one grammar. In realistic cases, however, several such grammars may be needed. The workflow consists of three main phases:

- Grammar: Obtain and improve the grammar for the source language of the SUI.
- Facts: Obtain and improve facts extracted from the SUI.
- Queries: Write and improve queries that give the desired answers.

Of course, it may happen that you have a lucky day and that extracted facts are readily available or that you can reuse a good quality fact extractor that you can apply to the SUI. On ordinary days you have the above workflow as fall-back.

It may come as a surprise that there is such a strong emphasis on validation in this workflow. The reason is that the SUI is usually a huge system that defeats manual inspection. Therefore we must be very careful that we validate the outcome of each phase.

Grammar In many cases there is no canned grammar available that can be used to parse the programming language dialect used in the SUI. Usually an existing grammar can be adjusted to that dialect, but then it is then mandatory to validate that the adjusted grammar can be used to parse the sources of the SUI.

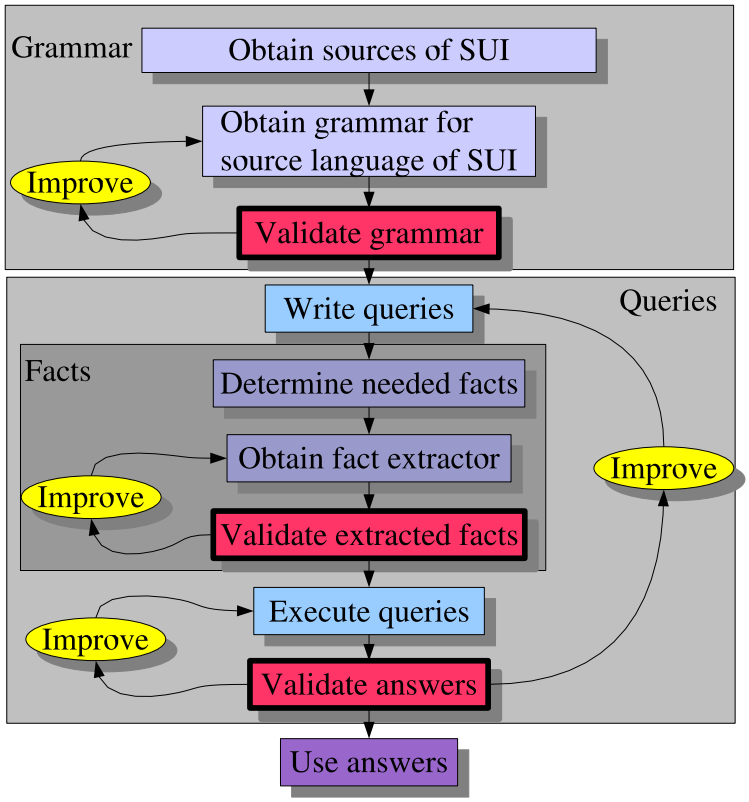


Figure 7.1: Workflow for fact extraction

Facts It may happen that the facts extracted from the source code are *wrong*. Typical error classes are:

- Extracted facts are *wrong*: the extracted facts incorrectly state that procedure `P` calls procedure `Q` but this is contradicted by a source code inspection.
- Extracted facts are *incomplete*: the inheritance between certain classes in Java code is missing.

The strategy to validate extracted facts differ per case but here are three strategies:

- Postprocess the extracted facts (using RSCRIPT, of course) to obtain trivial facts about the source code such as total lines of source code and number of procedures, classes, interfaces and the like. Next validate these trivial facts with tools like `wc` (word and line count), `grep` (regular expression matching) and others.
- Do a manual fact extraction on a small subset of the code and compare this with the automatically extracted facts.
- Use another tool on the same source and compare results whenever possible. A typical example is a comparison of a call relation extracted with different tools.

Queries For the validation of the answers to the queries essentially the same approach can be used as for validating the facts. Manual checking of answers on random samples of the SUI may be mandatory. It also happens frequently that answers inspire new queries that lead to new answers, and so on.

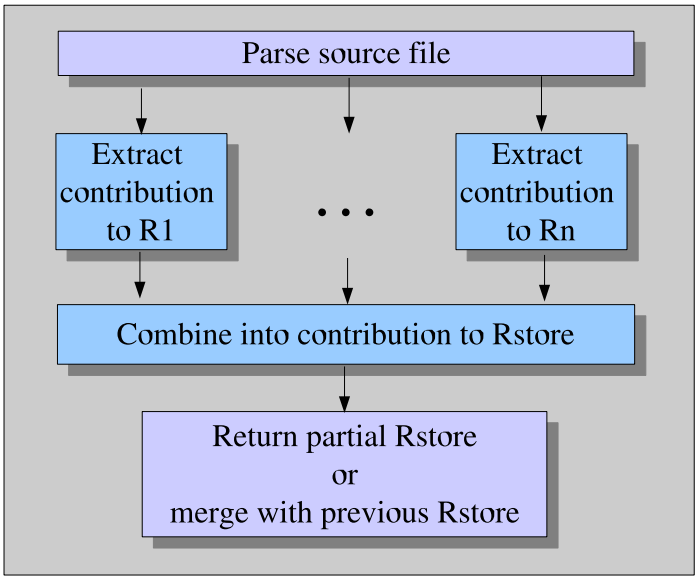


Figure 7.2: The *Separation-of-Concerns* strategy for fact extraction

7.2 Fact Extraction using ASF+SDF

7.2.1 Strategies for Fact Extraction

The following global scenario's are available when writing a fact extractor in ASF+SDF:

- *Dump-and-Merge:* Parse each source file, extract the relevant facts, and return the resulting (partial) Rstore. In a separate phase, merge all the partial Rstores into a complete Rstore for the whole SUI. The tool `merge-rstores` is available for this.

- *Extract-and-Update*: Parse each source file, extract the relevant facts, and add these directly to the partial Rstore that has been constructed for previous source files.

The experience is that the *Extract-and-Update* is more efficient.

A second consideration is the scenario used for the fact extraction per file. Here there are again two possibilities:

- *All-in-One*: Write one function that extracts all facts in one traversal of the source file. Typically, this function has an Rstore as argument and returns an Rstore as well. During the visit of specific language constructs additions are made to named sets or relations in the Rstore.

- *Separation-of-Concerns*: Write a separate function for each fact you want to extract. Typically, each function takes a set or relation as argument and returns an updated version of it. At the top level all these functions are called and their results are put into an Rstore. This strategy is illustrated in Figure 7.2

The experience here is that everybody starts with the *All-in-One* strategy but that the complexities of the interactions between the various fact extraction concerns soon start to hurt. The advice is therefore to use the *Separation-of-Concerns* strategy even if it may be seem to be less efficient since it requires a traversal of the source program for each extracted set or relation.

```
module PicoFactExtraction
imports Pico-syntax
imports basic/Integers
imports Rstore
imports utilities/PosInfo[PROGRAM] utilities/PosInfo[STATEMENT]
        utilities/PosInfo[EXP] utilities/PosInfo[PICO-ID]
exports
  context-free syntax
    cflow({ STATEMENT ";"}*)     -> <Set[[Elem]], Rel[[Elem]], Set[[Elem]]>

    uses(PROGRAM, Rel[[Elem]])   -> Rel[[Elem]] {traversal(accu,top-down,continue)}
    uses(EXP, Rel[[Elem]])       -> Rel[[Elem]] {traversal(accu,top-down,continue)}

    defs(PROGRAM, Rel[[Elem]])   -> Rel[[Elem]] {traversal(accu,top-down,break)}
    defs(STATEMENT, Rel[[Elem]]) -> Rel[[Elem]] {traversal(accu,top-down,break)}

    id2str(PICO-ID)              -> String
```

Figure 7.3: Syntax of functions for Pico fact extraction

7.2.2 Extracting Facts for Pico

After all these mental preparations, we are now ready to delve into the details of a Pico fact extractor. Figure 7.3 shows the syntax of the functions that we will need for Pico fact extraction. There are some things to observe:

- Module `Pico-syntax` is imported to make available the Pico grammar.
- Module `Rstore` is imported to get access to all functions on Rstores.
- The module `PosInfo` is imported with various sort names as parameter. For all these sorts, the function `get-location` will be defined that extracts the source text location from a given language construct.
- The function `cflow` will extract the control flow from Pico programs.
- The functions `uses` and `defs` extracts the uses and definitions of variables from the source text.
- `id2str` is an auxiliary function that converts Pico identifiers to strings that can be included in an Rstore.
- We have omitted all declarations for ASF+SDF variables to be used in the specification. The convention is that such variables all start with a dollar sign (`$`).

Extracting control flow The function `cflow` extracts the control flow from Pico programs. It takes a list of statements as input and returns a triple as output:

- A set of program elements that may enter a construct.
- A relation between the entries and exits of a construct.
- A set of program elements that form the exits from the construct.

For instance, the test in an if-then-else statement forms the entry of the statement, it is connected to the entry of the first statement of the then and the else branch. The exits of the if-then-else statement are the exits of the last statement in the then and the else branch. The purpose of `cflow` is to determine this information for individual statements and to combine this information for compound statements. Its definition is shown in Figure 7.4.

```
%% ---- control flow of statement lists
[cf1] <$Entry1, $Rel1, $Exit1> := cflow($Stat),
      <$Entry2, $Rel2, $Exit2> := cflow($Stat+)
       ============================================================
       cflow($Stat ; $Stat+) =
       < $Entry1,
         $Rel1 union $Rel2 union ($Exit1 x $Entry2),
         $Exit2
       >
[cf2] cflow() = <{}, {}, {}>

%% ---- control flow of individual statements
[cf3]  <$Entry, $Rel, $Exit> := cflow($Stat*),
       $Control := get-location($Exp)
       ============================================================
       cflow(while $Exp do $Stat* od) =
       < {$Control},
         ({$Control} x $Entry) union $Rel union ($Exit x {$Control}),
         {$Control}
       >

[cf4]  <$Entry1, $Rel1, $Exit1> := cflow($Stat*1),
       <$Entry2, $Rel2, $Exit2> := cflow($Stat*2),
       $Control := get-location($Exp)
       ============================================================
       cflow(if $Exp then $Stat*1 else $Stat*2 fi) =
       < {$Control},
         ({$Control} x $Entry1) union ({$Control} x $Entry2)
                                union  $Rel1 union $Rel2,
         $Exit1 union $Exit2
       >
[default-cf]
        $Loc := get-location($Stat)
       ============================================================
        cflow($Stat) = < {$Loc}, {}, {$Loc} >
```

Figure 7.4: Equations for `cflow`: computing control flow

Extracting uses and defs The functions `defs` and `uses` are shown in Figure 7.5. They extract the definition, respectively, the use of variables from the source code. Both functions are defined by means of an ASF+SDF *traversal function* which silently visits all constructs in a tree, and only performs an action for the constructs for which the specification contains an equation. In the case of `defs`, equation `[vd1]` operations on assignment statements and extracts a pair that relates the location of the complete statement to the name of the variable on the left-hand side. For the function `uses`, equation `[vu1]` acts on all uses of variables. For completeness sake, the figure also show the definition of utility function `id2str`.

Queries Figure 7.6 shows the syntax of the functions we will use for querying. In fact, we will demonstrate two styles of definition. In the first style, the function `extractRelation` extracts facts from a Pico program and yields an Rstore. This can be used by `pico-query` to run an arbitrary RSCRIPT on that Rstore. In the second style, fact extraction and running an RSCRIPT are done in a single function.

Figure 7.7 shows the first definition style. In equation `[er1]`, we see a step-by-step construction of an Rstore that contains all the information gathered by the extraction functions. An Rstore that contains all this information is returned as result of `extractRelations`. The function `pico-query` can then be used to run an RSCRIPT for a given Rstore.

```
%% ---- Variable definitions: <expression-location, var-name>

[vd1]  $Id := $Exp := $Stat
       ============================================================
       defs($Stat, $Rel) = $Rel union {<get-location($Stat), id2str($Id)>}

%% ---- Variable uses <var-location, var-name>

[vu1]  $Id := $Exp
       ============================================================
       uses($Exp, $Rel) = $Rel union  {<get-location($Id), id2str($Id)>}

%% ----- utilities

[i2s] id2str(pico-id($Char*)) = strcon("""" $Char* """")
```

Figure 7.5: Equations for `defs`, `uses` and `id2str`

```
module PicoQuery
imports RscriptCalculator
imports PicoFactExtraction
imports basic/Errors

exports
  start-symbols Summary
  context-free syntax
     extractRelations(PROGRAM)                    -> RSTORE
     pico-query(RSCRIPT, RSTORE, StrCon, StrCon) -> Summary
     uninit(PROGRAM)                              -> Summary
```

Figure 7.6: Syntax of function for two styles of querying

The second definition style is shown in Figure 7.8. In this case, we see that all work is done in a single (indeed large) equation. The construct `[| ... |] yield UNINIT` is particularly noteworthy since it allows the embedding of a complete RSCRIPT in an ASF+SDF equation. Also pay attention to the following:

- The RSCRIPT is first simplified as much as possible according to ordinary ASF+SDF simplification rules. This implies that variables like `$Start`, `$Rel1`, and `$Program` are replaced by their respective values. This is also the case for the functions `defs` and `uses` that occur in the RSCRIPT.
- The effect of the `[| ... |] yield UNINIT` construct is that the RSCRIPT is evaluated and that the value of `UNINIT` is returned as result.
- The definition of the function `convert2summary` is not shown: it performs a straightforward conversion of `UNINIT`'s value to the message format (`Summary`) that is used by the Meta-Environment.

7.3 Concluding Remarks

As can already be seen from the very simple Pico example, over 100 lines of ASF+SDF specification (including variable declarations and auxiliary functions we did not show) are needed to extract facts, while

```
%% ---- extractRelations

[er-1] begin $Decls $Stat+ end := $Program,
       $Start := get-location($Program),
       <$Entry, $Rel, $Exit> := cflow($Stat+),

       $Loc :=  get-location($Program),
       $Rstore1 := assign(ROOT, expr, $Loc, rstore()),
       $Rstore2 := assign(PRED, rel[expr,expr],
                          $Rel union ({$Start} x $Entry), $Rstore1),
       $Rstore3 := assign(DEFS, rel[expr,varname], defs($Program, {}), $Rstore2),
       $Rstore4 := assign(USES, rel[var,varname], uses($Program, {}), $Rstore3)
       ========================================================================
       extractRelations($Program) = $Rstore4

%% ---- pico-query

[pq1]  pico-query($Script, $Rstore, $StrCon1, $StrCon2) =
       convert2summary(
          $StrCon2,
          eval-rscript-with-rstore-and-yield($Script, $Rstore, $StrCon1)
       )
```

Figure 7.7: Build an Rstore with `extractRelations` and apply it using `pico-query`

only 10 lines of RSCRIPT are sufficient for the further processing of these facts. What can we learn from this observation?

First, that even in the simple case of Pico fact extraction is more complicated than the processing of these facts. This may be due to the following:

- The facts we are interested in may be scattered over different language constructs. This implies that the fact extractor has to cover all these cases.
- The extracted facts are completely optimized for relational processing but places a burden on the fact extractor to perform this optimization.

Second, that several research question remain unanswered:

- Is it possible to solve (parts of) the fact extraction in a language-parametric way. In other words, is it possible to define generic extraction methods that apply to multiple languages?
- Is a further integration of fact extraction with relational processing desirable? We have already shown some form of integration in Figure 7.8 where an embedded RSCRIPT occurs in an ASF+SDF specification. Is it, for instance, useful to bring some of the syntactic program domains like expressions and statements to the relational domain?

```
[ui-1] begin $Decls $Stat+ end := $Program,
       $Start := get-location($Program),
       <$Entry, $Rel, $Exit> := cflow($Stat+),
       $Rel1 := $Rel union ({$Start} x $Entry),
       $Rval := [| type expr = loc
                   type var  = loc
                   type varname = str

                   expr ROOT = $Start

                   rel[expr,expr] PRED = $Rel1
                   rel[expr,varname] DEFS  = defs($Program, {})
                   rel[var,varname] USES  = uses($Program, {})
                   set[varname] VARNAMES  = range(DEFS) union range(USES)

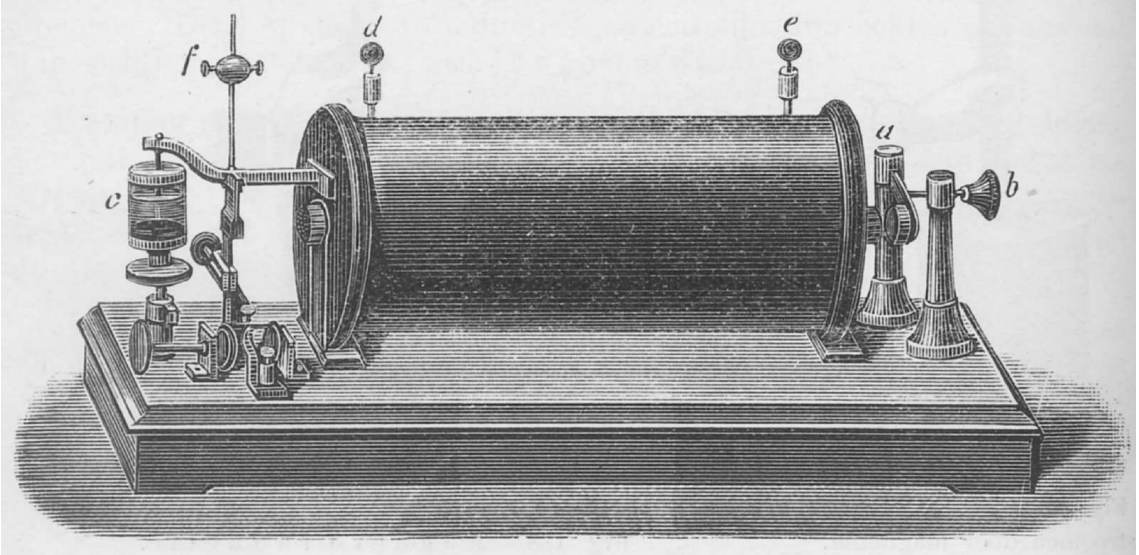
                   rel[var, varname] UNINIT =
                   { <V, VN>| var-name VN : VARNAMES,
                              var V : USES[-,VN],
                              expr E : reachX({ROOT}, DEFS[-,VN], PRED),
                              V <= E
                   }

                |] yield UNINIT
       ================================================================
       uninit($Program) = convert2summary("Uninitialized variable", $Rval)
```

Figure 7.8: Combined fact extraction and query processing

Chapter 8

Installing and Running RSCRIPT



8.1 Warning

The information in the current chapter is outdated. As of august 2007, the most convenient way to use RSCRIPT is to use `rscript-meta`, an instance of The Meta-Environment specialized for RSCRIPT. It can be downloaded (in binary and source form) from `http://www.meta-environment.org`.

8.2 Installing RSCRIPT

RSCRIPT is available[1] as `relation-calculus-0.4.tar.gz` (or a newer version). It requires a recent version (e.g. at least 1.5.3) of the ASF+SDF Meta-Environment to run. A typical installation session on a typical Unix/Linux system consists of the following steps:

- Extract all files from the distribution: `tar zxvf relation-calculus-0.4.tar.gz` This command uncompresses the distribution file, creates a subdirectory `relation-calculus-0.4` and places all directories and files from the distribution in it.

[1] `www.cwi.nl/projects/MetaEnv/...`

- Change to the new directory: `cd relation-calculus-0.4`
- Configure the sources: `./configure`
- Build and install application: `make install`

See the files `INSTALL` and `README` for more specific installation instructions.

8.3 Running RSCRIPT from the command line

8.3.1 File extensions

The following file extensions are used by the command line tools:

- `.rscript` is the required file name extension for files that contain an RSCRIPT.
- `.rstore` is the required file name extension for files that contain an rstore. As intermediate result rstores also occur in parsed form and then have the extension `.rstore.pt`. The tools transparently accept rstores in both forms.
- `.rviz` is the required file name extension for files that contain relational data that are to be visualized (see Chapter 9).

8.3.2 `rscript`: check and execute an rscript

The command `rscript` takes care of parsing, typechecking, and executing an RSCRIPT. Optionally, the script can only be checked, can use a given rstore, be executed as test suite, or yield the value of a given variable from the resulting rstore.

- `-c` or `--check-only`: Only check the rscript for syntactic errors or typechecking errors but do not execute it.
- `-h` or `--help`; print help information.
- `-i` *name* or `--input` *name*: The RSCRIPT to be processed is on file *name*. By default, the script is assumed to be on standard input.
- `-o` *name* or `--output-parse-tree` *name*: The result of the execution of the RSCRIPT is an rstore and is written to file *name*. Note that this rstore is in the form of a parse tree. By default, the resulting rstore is printed in textual form to standard output.
- `-s` *name* or `--store` *name*: The RSCRIPT is executed using an initial rstore taken from file *name*. By default, the initial rstore is empty.
- `-t` or `--testsuite`: Execute the RSCRIPT as a testsuite, i.e., execute all `assert` statements and report which ones failed.
- `-v` or `--verbose`: give verbose output.
- `-y` *name* or `--yield` *name*: Instead of producing a complete rstore as the result of executing the rscript, only return the value of the variable *name*.r

The following examples illustrate the use of the `rscript` command:

- `rscript query.rscript` executes the script `query.rscript` and prints the resulting rstore on standard output.

- `rscript -i query.rscript -s previous.rstore -o result.rstore.pt`: executes the same script, but uses `previous.rstore` as initial rstore and writes the resulting store to `result.rstore.pt¿`
- `rscript -y nCalls calls.rscript`: executes the script `calls.rscript` and prints the value of the variable `nCalls` in the resulting rstore.
- `rscript -t tests1.rscript`: executes `tests2.rscript` as test suite and reports which assert statements failed (if any).

8.3.3 `extract-relations`: extract relations from source files

The command `extract-relations` provides a common framework for ASF+SDF-based fact extraction and has the following form:

```
extract-relations [options] <input-files>
```

The following optional arguments are supported:

- `-h` or `--help`; print help information.
- `-o` *file* or `--output` *file*: the name of the resulting rstore (default: `result.rstore.pt`).
- `-e` *program* or `--executable` *program*: the executable *program* for performing extraction (default: none). This is most likely to be a compiled ASF+SDF specification.
- `-s` *sort* or `--sort` *sort*: *sort* is the sort used for parsing each input file (default: none). In other words, each input file should conform to the syntax of *sort*. Example: `CompilationUnit`.
- `-f` *name* or `--function` *name*: the extraction function to be applied to each source file (default: `extractRelations`). The definition of this function should conform to:

  ```
  extractRelations(sort) -> RSTORE
  ```

  where *sort* is the sort of each input file as defined above.
- `-p` *file* or `--parse-table` *file*: the parse table *file* used for parsing input files (default: none).

The following examples illustrate the use of the `extract-relations` command:

- `extract-relations -e JavaAnalysis -s CompilationUnit -o jhotdraw.rstore.pt -p Java.trm.tbl dir/*.java`: extract relations from Java source code in the files `dir/*.java`. Use executable `JavaAnalysis`, each input file is of sort `CompilationUnit`, use parse table `Java.trm.tbl`, and produce as output an rstore `jhotdraw.rstore.pt`.
- `extract-relations -e TBExtr -s Tscript -p Tscript.trm.tbl *.tb`: extract relations from TOOLBUS source files. Use executable `TBExtr`, each input file is of sort `Tscript`, use parse table `Tscript.trm.tbl` and extract from the source files `*.tb`.

8.3.4 `merge-rstores`: combine several rstores

The command `merge-rstores` merges several rstores into a new rstore. This command is used in a scenario where an extraction tool extracts facts from each source file in a software portfolio to be analyzed and deposits these facts in a separate rstore per source file. When the complete portfolio is to be analyzed, the separate rstores have to be merged into a single one. This merged rstore is then used as initial rstore for the execution of some rscript.

The command has as arguments, a list of names of rstores to be merged.

The following optional arguments are supported:

- `-h` or `--help`: print help information.
- `-o` *name* or `--output` *name*: the name of the merged rstore is *name*. By default, the output is `result.rstore.pt`.

Note: *tool should check for right output form*

- `-v` or `--verbose`: give verbose output.

8.4 Running RSCRIPT Interactively

You can also edit and run Rscripts interactively:

- Change directory to your checked out copy of the directory `relation-calculus/spec`.
- Start the ASF+SDF Meta-Environment with the command `meta`.
- Open the module `Rscript.sdf`.
- Open your own term, using the `Rscript` module.
- Observe that a new menu with the name `Rscript` appears in the menu bar.
- Click the parse button in the `Actions` menu of the editor: now we know whether there are syntax errors. If so, correct them.
- Click the `Check` button in the `Rscript` menu: this will perform a type check of your script. If there are type errors, correct them.
- Click the `Run` button in the `Rscript` menu to execute your script: a new editor pops up which shows all the variables at the end of the execution. `Run` also performs a type of your script so you may skip the previous step.
- Click the `Run with Rstore` button if you want to execute your RSCRIPT with an existing "Rstore": a collection of relations that are the result of previous extraction phase. Currently, a fixed name is used for this Rstore: `RSTORE.rstore`.

Running a Test suite Same as above but use `Testsuite` button instead of the `Execute` button in the `Rscript` menu. The effect is that the script is executed and that a summary is printed of `assert` statement that succeeded or failed.

8.5 Other Tools and Demos

8.5.1 Examples

The subdirectory `rscripts` contains several sample scripts. See, for instance, `tests1.rscript` and `tests2.rscript` for examples of the use of built-in operators and functions.

8.5.2 The Pico Demo

The subdirectory `demo/pico` contains in a single directory the Pico syntax, Pico fact extraction (discussed in Section 7.2) and the test on uninitialized variables presented in Section 6.3.

- Change directory to `demo/pico`: `cd demo/pico`.
- Start the ASF+SDF Meta-Environment: `meta -m Pico-syntax.sdf`.
- Open the term `exam.pico` over module `Pico-syntax.sdf`.
- Under the `Pico` menu two styles of checking are available:
  - Extract Relations: this extracts relations from the current Pico program.

- Uninitialized Vars (Style 1): uses the extracted facts and executes the RSCRIPT `uninit.rscript`, see Figure 7.7.
- Uninitialized Vars (Style 2): performs combined facts extraction and query processing, see Figure 7.8.

See Figure 6.3 for a screen dump.

8.5.3 The Java Demo

The Java demo consists of the following parts:

- `java/grammar`: Contains a SDF grammar for Java.
- `java/extraction`: defines `JavaAnalysis`, that performs basic fact extraction from Java source code. It also defines the command `extract-java`, a specialized version of `extract-relations` (see Section 8.3.3):
- `java/rscripts`: defines a script `Enrich.rscript` enrich the facts extracted by JavaAnalysis. It also defines some sample scripts that operate on the enriched rstores.
- `java/example-hotdraw`: gives data for the JhotDraw example.

The command `extract-java` provides a common framework for ASF+SDF-based fact extraction from Java programs and has the following form:

```
extract-java [options]
```

The following optional arguments are supported:

- `-h` or `--help`; print help information.
- `-i` *dir* or `--input` *dir*: the name a directory *dir* that contains the Java source code. All Java source files appearing in (subdirectories of) *dir* will be used as input. (default: current directory).
- `-o` *file* or `--output` *file*: the name of the resulting rstore (default: `result.rstore.pt`).
- `-e` *program* or `--executable` *program*: the executable *program* for performing extraction (default: `JavaAnalysis`). This is most likely to be a compiled ASF+SDF specification.
- `-f` *name* or `--function` *name*: the extraction function to be applied to each source file (default: `extractRelations`). The definition of this function should conform to:

```
extractRelations(CompilationUnit) -> RSTORE
```

The following examples illustrate the use of the `extract-relations` command:

```
extract-java -i JHotDraw5.2-sources -o jhotdraw.rstore.pt
```

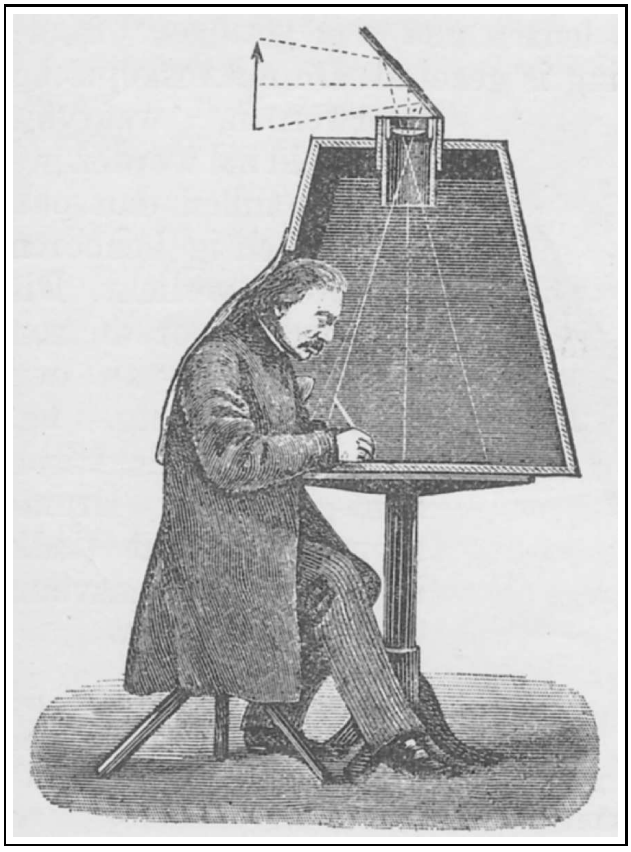
Here is a scenario to go all the way from Java source code to the visualization of the extracted and enriched facts:

```
cd example-hotdraw
extract-java -i /ufs/paulk/software/source/JHotDraw -o jhotdraw.rstore.pt
rscript -i ../rscripts/Enrich.rscript -s jhotdraw.rstore.pt -o enr.jhotdraw.rstore.pt
rscript -i validate.rscript -s enr.jhotdraw.rstore.pt
rstore2rviz -i enr.jhotdraw.rstore.pt -o jhotdraw.rviz
rviz jhotdraw.rviz
```

In the next chapter, we will further explain this scenario.

Chapter 9

Visualization of Rstores



The sets and relations constructed for all but the most trivial problems are voluminous and their textual representation is hard to grasp for the human eye. This is where information visualization techniques (**) come to our rescue. In this chapter we present some initial experiments to visualize the contents of an Rstore. This is a multi-stage process described in Section 9.2. First facts are extracted from the source (Section 9.3) and are further enriched (Section 9.4). Next the Rstore is converted to the `.rviz` format that is more amenable as input for a visualization tool. This format is described in Section 9.5 and the conversion from Rstore to this format is discussed in Section 9.6. Next, the `.rviz` file can be visualized and explored. This is the topic of Section 9.7

9.1 Warning

The ideas described here remain relevant but their specific implementation is outdated. The interactive RSCRIPT environment (`rscript-meta`) contains built-in visualization techniques for Rstores.

9.2 The visualization workflow

The process of achieving a visualization of a System under Investigation (SUI) is shown in Figure 9.1 and consists of the following steps:

- Extract facts from a source code directory `SUISrcDir` and produce an Rstore `SUI.rstore.pt`.

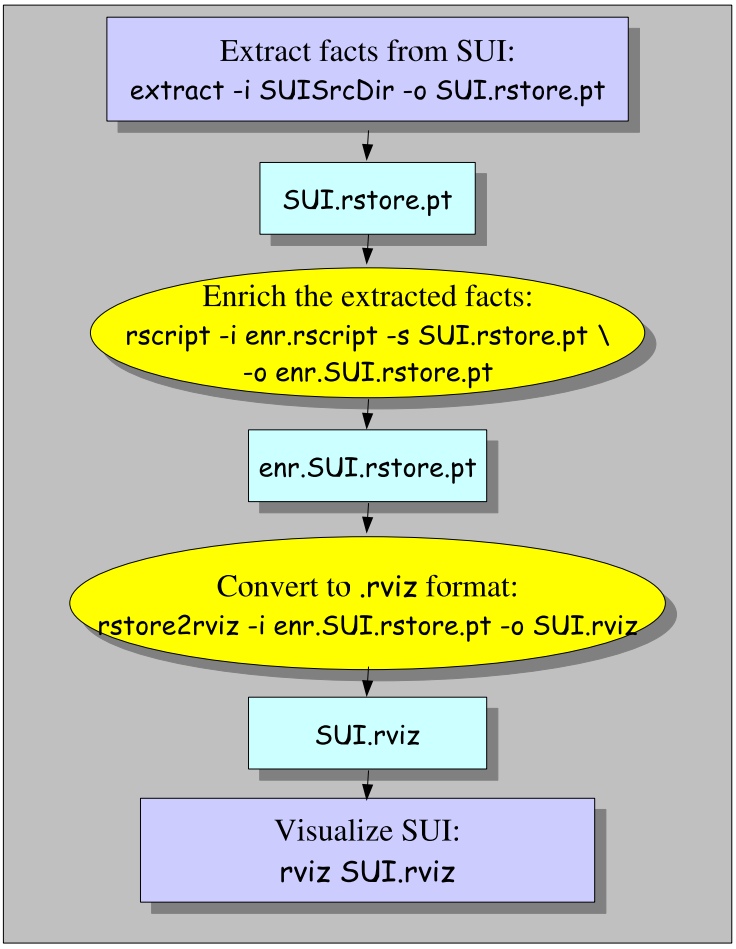


Figure 9.1: Workflow for visualization of System Under Investigation

- Enrich this Rstore by running an rscript `enr.rscript` on it, with resulting enriched Rstore `enr.rstore.pt`.
- Convert the enriched Rstore to the the visualization format `.rviz`.
- Run the visualization tool `rviz` on these data.

The details of this process are now further explained.

9.3 Extracting Facts

In Chapter 7 we have already seen how fact extraction can be organized and implemented. For the current presentation it is sufficient to assume that there exists an `extract` tool that can be used:

```
extract -i SUISrcDir -o SUI.rstore.pt
```

where `SUISrcDir` is the source directory where the source of the SUI can be found. In the distribution a tool `extract-java` is available with precisely this behavior for Java programs.

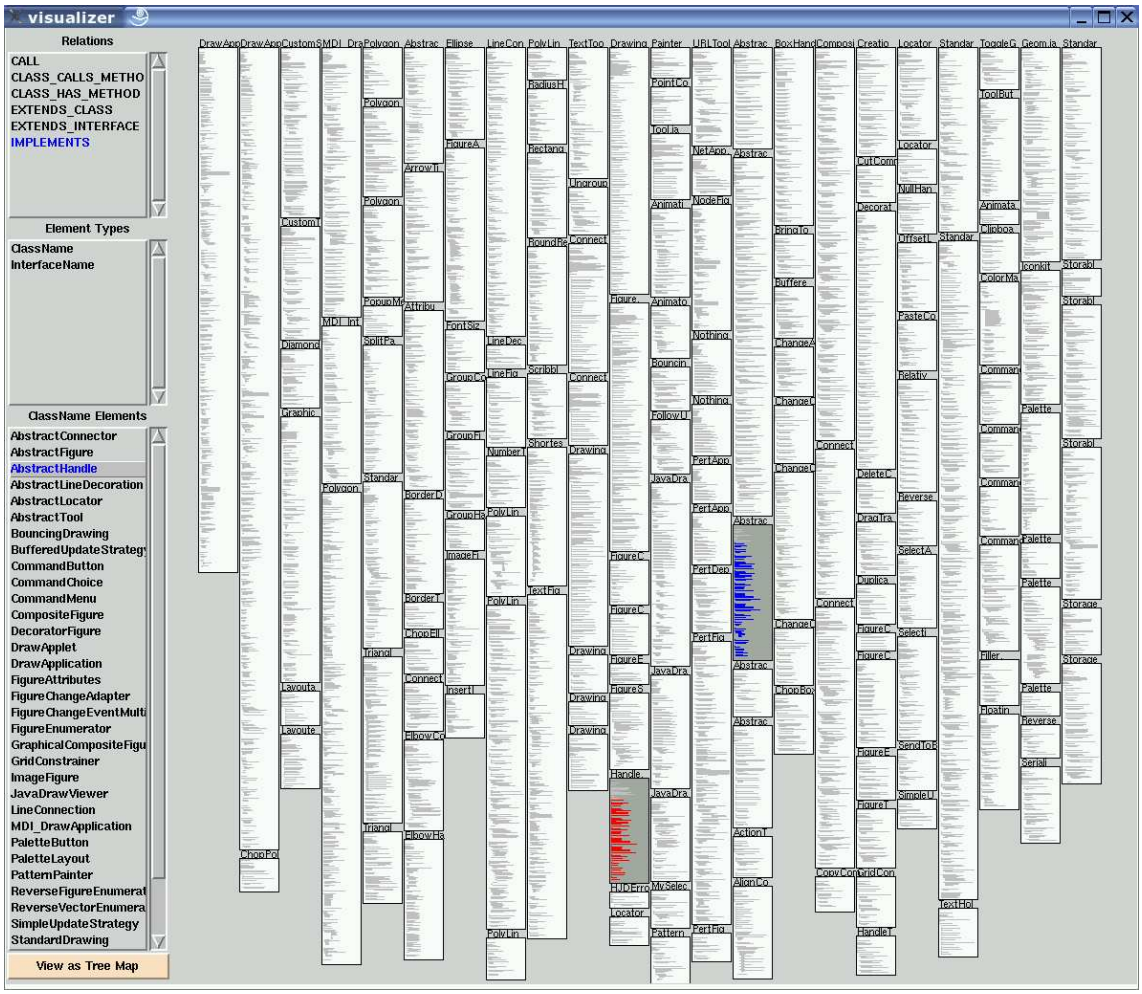


Figure 9.2: File view

9.4 Enriching Facts

Given the Rstore `SUI.rstore.pt` we enrich it by writing and executing an RSCRIPT `enr.rscript` that extends the extracted facts as required by the goal we want to achieve:

```
rscript enr.rscript -s SUI.rstore.pt -o enr.SUI.rstore.pt
```

9.5 The `.rviz` Format

Before explaining the conversion to the rviz format, it is helpful to understand this format first. Only two forms of data definitions can currently appear in an rviz file. *Elements* are values that can occur in relations and are defined by the keyword `elm`. *Tuples* are defined by the keyword `tup`. The definition of an *element* has the following form:

`elm` *Type Name File BeginLine BeginCol EndLine EndCol*

Type is the type as declared in the RSCRIPT, *Name* is the textual name of the element, and the subsequent file and location information characterize the precise coordinates of the element in the source text. An example of an element definition is:

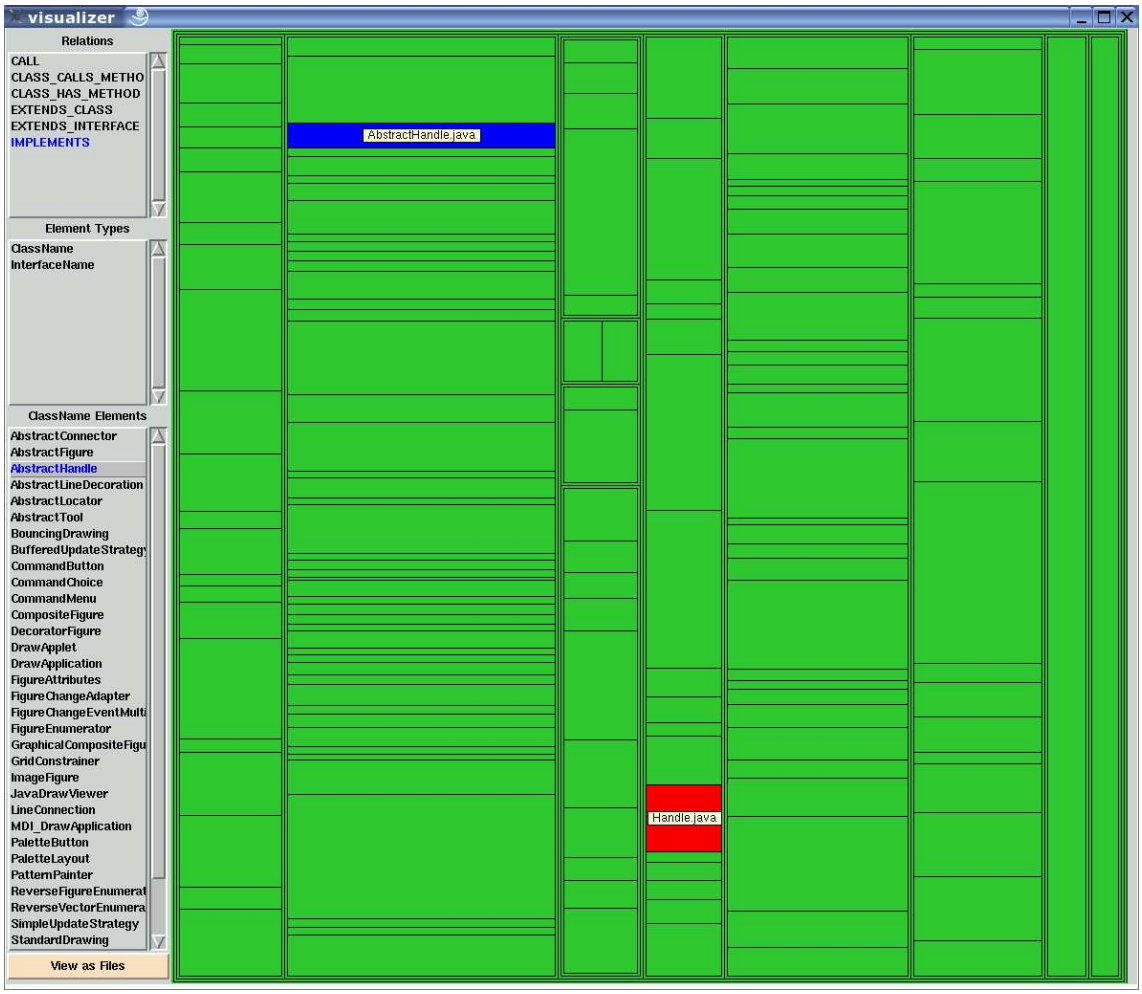


Figure 9.3: Tree map view

```
elm "ClassName" "PolyLineFigure" ".../PolyLineFigure.java" 21 0 339 1
```

The definition of a *tuple* has the following form:

`tup` *RelName Type1 Value1 Type2 Value2*

RelName is the name of the relation to which the tuple belongs. It is followed two type/value pairs that define the two items in the tuple. An example of a tuple definition is:

```
tup IMPLEMENTS "ClassName" "AbstractHandle" "InterfaceName" "Handle"
```

Discussion Observe that the current visualization format is very simple and does not allow the representation of all data that may be present in an Rstore. In particular, sets and n-ary relations ($n > 2$) are not represented. Clearly, this format will further evolve.

9.6 `rstore2rviz`: Convert Rstore to Visualization Format

The command `rstore2rviz` takes an rstore as input and converts it to the `.rviz` format that is accepted by the visualization tool `rviz`.

- `-h` or `--help`: print help information.
- `-i` *name* or `--input` *name*: rstore comes from file *name*. By default: standard input.
- `-o` *name* or `--output` *name*: the result is written to file *name*. By default, this is a file named `result.rviz`.
- `-v` or `--verbose`: give verbose output.

Example: `rstore2rviz -i enr.SUI.rstore.pt -o SUI.rviz`

9.7 `rviz`: Visualize an Rstore

Visualization is simply started by the command `rviz` with the given visualization data as input, as in:

```
rviz SUI.rviz
```

The result is a window as shown in Figure 9.2 which consists of several panes. On the left-hand side three panes occur. On top is the *Relations* pane that lists the relations that are available. One of these relations can be selected and its elements will be displayed. In the middle appears an *Element Type* pane that shows the element types that are available for the selected relation. Selecting one of these types lists in the the bottom all *Elements* that occur in the selected relation and are of the selected type. By selecting one element from the *Elements* pane, that element (and all elements it is associated with by the selected relation) will be highlighted in the large graphical pane on the right.

There are two visualization methods available that can be selected by the button at the bottom left that is alternatively labeled as `View as Files` or `View as TreeMap`. In the former case, files are shown as rectangles with a a pattern of horizontal lines inside that reflect their textual structure. In the latter case, a tree map is shown of the directory structure of all files.

Figure Figure 9.2 shows the visualization of facts extracted from the JHotDraw application. The relation `IMPLEMENTS` has been selected, and of the two possible element types `ClassName` has been chosen. From the list of possible class names, `AbstractHandle` has been selected. The result is that the element itself is shown in the file view (with all lines of its definition displayed in blue), and all its "related" elements (e.g., the interfaces it implements) shown in red.

Figure Figure 9.3 shows the same selection, but this time in the tree map view.


S
B
B
A
A
H
O

Appendix A

Tables of Built-in Operators

Operator	Description	Section
`and`	Boolean and	4.1
`implies`	Boolean implication	4.1
`in`	Membership test on sets/relations	4.5
`inter`	Intersection of sets/relations	4.5
`not`	Boolean negation	4.1
`notin`	Non-membership test on sets/relations	4.5
`or`	Boolean or	4.1
`union`	Union of sets/relations	4.5
`==`	Equality of integers	4.2
`==`	Equality of strings	4.3
`==`	Equality of locations	4.4
`==`	Equality of sets/relations	4.5
`!=`	Inequality of integers	4.2
`!=`	Inequality of strings	4.3
`!=`	Inequality of locations	4.4
`!=`	Inequality of sets/relations	4.5
`<=`	Less than or equal of integers	4.2
`<=`	Less than or equal of strings	4.3
`<=`	Textual inclusion of locations	4.4
`<=`	Subset of sets/relations	4.5
`<`	Less than of integers	4.2
`<`	Less than of strings	4.3
`<`	Strict textual inclusion of locations	4.4
`<`	Strict subset of sets/relations	4.5
`>=`	Greater than or equal of integers	4.2
`>=`	Greater than or equal of strings	4.3
`>=`	Textual containment of locations	4.4
`>=`	Superset of sets/relations	4.5
`>`	Greater than of integers	4.2
`>`	Greater than of strings	4.3
`>`	Strict textual containment of locations	4.4
`>`	Strict superset of sets/relations	4.5
`+`	Addition of integers	4.2

-	Subtraction of integers	4.2
*	Multiplication of integers	4.2
/	Division of integers	4.2
\	Difference of sets/relations	4.5
o	Composition of relations	4.6
x	Carthesian product of sets	4.6
#	Number of elements of set	4.5.4
#	Number of tuples of relation	4.5.4
[-,]	Left image of relation	4.6
[,-]	Right image of relation	4.6
[ ]	Right image of relation	4.6
+	Transitive closure of a relation	4.6
*	Reflexive transitive closure of a relation	4.6

Appendix B

Tables of Built-in Functions

Function	Description	Section
`average`	Average of a set of integers	5.7.4
`average-domain`	Average of first elements of tuples in relation	5.7.5
`average-range`	Average of second elements of tuples in relation	5.7.6
`begincol`	First column of a location	5.6.3
`beginline`	Beginning line of a location	5.6.2
`bottom`	Bottom of a relation	5.5.2
`carrier`	Carrier of a relation	5.2.3
`carrierR`	Carrier restriction of a relation	5.3.3
`carrierX`	Carrier exclusion of a relation	5.3.6
`compl`	Complement of a relation	5.1.4
`endcol`	Last column of a location	5.6.5
`endline`	Ending line of a location	5.6.4
`filename`	File name of a location	5.6.1
`first`	First element of a tuple	5.4.1
`id`	Identity relation	5.1.1
`inv`	Inverse of a relation	5.1.3
`domain`	Domain of a relation	5.2.1
`domainR`	Domain restriction of a relation	5.3.1
`domainX`	Domain exclusion of a relation	5.3.4
`min`	Minimum of a set of integers	5.7.7
`max`	Maximum of a set of integers	5.7.7
`power0`	Powerset of a set	5.1.5
`power1`	Powerset of a set	5.1.6
`range`	Range of a relation	5.2.2
`rangeR`	Range restriction of a relation	5.3.2
`rangeX`	Range exclusion of a relation	5.3.5
`reachR`	Reachability with restriction	5.5.3
`reachX`	Reachability with exclusion	5.5.4
`second`	Second element of a tuple	5.4.2
`sum`	Sum of a set of integers	5.7.1
`sum-domain`	Sum of first elements of tuples in relation	5.7.2
`sum-range`	Sum of a first elements of tuples in relation	5.7.3
`top`	Top of a relation	5.5.1
`unique`	Deprecated: Set with unique elements	5.1.2

Acknowledgments

Thanks to Tijs van der Storm for many useful discussions and experiments. Thanks to Murat Ahat, Jan van Eijck, Taeke Kooiker, Tijs van der Storm, Ivan Vankov, and Jurgen Vinju for comments on this tutorial.

Illustrations

Most illustrations used in this tutorial concern physical instruments for measurement or observation and are taken from H. van de Stadt, *Beknopt Leerboek der Natuurkunde* (Concise Text-book of Physics) Tjeenk Willink, Zwolle, 1902. On the front page appears a windlass that amplifies manual power and is used in water wells, drilling devices, and wind mills. Page 7 shows a hot air balloon combined with a parachute (circa 1900). On page 23 appears a composite microscope as proposed by Drebbel (1621). On page 29 appears a *declinatorium* used to measure the difference between the magnetic and geographic north pole. On page 37 the cross section is shown of a lighthouse as used along the Dutch cost. The spectroscope on page 53 is a design using four prisms by Steinheil and is used for the improved dispersion and analysis of the light emitted by sodium vapor. On page 61 appears Ruhmkorff's induction-coil (1851) used to create high-Voltage electric currents. Page 67 shows a variation of the *camera obscura* as used for producing realistic drawings of a landscape. Lassell's telescope (1863) appears on page 72.

The photograph on page 11 is the "Caruso" loudspeaker and appeared in an advertisement in J. Corver, *Het Draadloos Amateurstation* (The Wireless Amateur (Radio) Station), Veenstra, 's Gravenhage, 1928. The sign alphabet on page 15 has been taken from `www.inquiry.net/outdoor/skills/b-p/signaling.htm`

Bibliography

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

[2] R. Behnke, R. Berghammer, E. Meyer, and P. Schneider. RELVIEW – a system for calculating with relations and relational programming. In E. Astesiano, editor, *Proceedings of Fundamental Approaches to Software Engineering (FASE)*, volume 1382 of *Lecture Notes in Computer Science*, pages 318–321. Springer-Verlag, 1998.

[3] D. Beyer, A Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE 2003)*, 2003. To appear.

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

[5] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[6] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[7] G. Canfora, A. de Lucia, and G.A. Lucca. A system for generating reverse engineering tools: a case study of software modularisation. *Automated Software Engineering*, 6:233–263, 1999.

[8] J. Ebert, R. Gimnich, H.H. Stasch, and A. Winter. *GUPRO: Generische Umgebung zum Programverstehen*. Koblenzer Schriften zur Informatik. Fölbach, 1998.

[9] L.M.G. Feijs, R. Krikhaar, and R.C. Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, april 1998.

[10] L.M.G. Feijs and R.C. Ommering. Relation partition algebra—mathematical aspects of uses and part-of relations. *Science of Computer Programming*, pages 163–212, 1999.

[11] R. C. Holt, A. Winter, and J. Wu. Towards a common query language for reverse engineering. Technical report, Institute for Computer Science, Universität Koblenz-Landau, August 2002.

[12] R.C. Holt. Binary relational algebra applied to software architecture. CSRI 345, University of Toronto, march 1996.

[13] D.J. Jackson and E.J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, volume 19 of *ACM SIGSOFT Software Engineering Notes*, pages 2–10, 1994.

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

[15] E. Koutsofios and S.C. North. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1996. See also `www.graphviz.org`.

[16] R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[17] M. A. Linton. Implementing relational views of programs. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 132–140, 1984.

[18] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[19] H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE 10),*, pages 80–86,, April 1988.

[20] S. Paul and A. Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[21] J.T. Schwartz, R.B.K. Dewar, and E. Dubinsky anmd E. Schonberg. *Programming with Sets; An Introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[22] V. Tannen. Tutorial: languages for collection types. In *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 150–154, 1994.

[23] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[24] P. Trinder and P. L. Wadler. List comprehensions and the relational calculus. In C. Hall, J. Hughes, and J. T. O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 187–202, Glasgow, UK, 1989. Department of Computer Science, University of Glasgow.

[25] D.A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 1–28. Cambridge University Press, 1982.

[26] J.D. Ullman. *Principles of Database Systems, Second Edition*. Computer Science Press, 1982.

[27] P. L. Wadler. List comprehensions. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 15. Prentice Hall, 1987.

[28] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.