Chapter 1. Rscript --- a Relational Approach to Software Analysis

Paul Klint

2007-10-21 09:59:56 +0200 (Sun, 21 Oct 2007)

Table of Contents

Introduction .................................................................................................................. 2
A Motivating Example ............................................................................................... 3
Preparations .............................................................................................................. 4
Questions .................................................................................................................. 5
The Rscript Language ............................................................................................... 6
Types and Values ...................................................................................................... 7
Tuples, Sets and relations .......................................................................................... 7
Comprehensions ........................................................................................................ 8
Declarations ................................................................................................................ 10
Built-in Operators ..................................................................................................... 12
Operations on Booleans .......................................................................................... 13
Operations on Integers ............................................................................................ 13
Operations on Strings .............................................................................................. 13
Operations on Locations ........................................................................................ 14
Operations on Sets or Relations ............................................................................. 14
Operations on Relations ......................................................................................... 15
Built-in Functions ...................................................................................................... 17
Elementary Functions on Sets and Relations ......................................................... 18
Extraction from Relations ....................................................................................... 19
Restrictions and Exclusions on Relations ............................................................. 20
Tuples ....................................................................................................................... 21
Relations viewed as graphs ..................................................................................... 21
Functions on Locations ........................................................................................... 22
Functions on Sets of Integers ................................................................................. 23
Larger Examples ........................................................................................................ 25
Analyzing the Component Structure of an Application ........................................ 25
Analyzing the Structure of Java Systems .............................................................. 26
Finding Uninitialized and Unused Variables in a Program ..................................... 27
Using Locations to Represent Program Fragments .............................................. 29
McCabe Cyclomatic Complexity ......................................................................... 30
Dataflow Analysis ..................................................................................................... 31
Program Slicing ......................................................................................................... 37
Extracting Facts from Source Code ....................................................................... 40
Workflow for Fact Extraction ................................................................................. 40
Strategies for Fact Extraction ............................................................................... 41
Fact Extraction using ASF+SDF ............................................................................ 42
Concluding remarks .................................................................................................. 42
Table of Built-in Operators ..................................................................................... 43
Table of Built-in Functions ..................................................................................... 44
Bibliography .............................................................................................................. 45
Warning

This document is in the process of being converted to DocBook.

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, “The extract-enrich-view paradigm” [2].

**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 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 [KN96], Rigi [MK88] and others. Rscript is not concerned with viewing but we will give some examples anyway.

**Figure 1.1. The extract-enrich-view paradigm**

---

**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.

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 [MK88] and GROK [Hol96] do. After extensive experimentation we have decided not to use bags and multi-relations like in RPA [FKO98].
• 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 [MK88], GROK [Hol96] and RPA [FKO98] 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.

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 1.2, “Workflow for analyzing mystery box” [4]. 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 1.3, “Graphical representation of the calls relation” [4] 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.

Figure 1.2. Workflow for analyzing mystery box

Now consider the call graph shown in Figure 1.3, “Graphical representation of the calls relation” [4]. 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 the section called “The Rscript Language” [6] the section called “Built-in Operator” [12] and the section called “Built-in Functions” [17].

Figure 1.3. Graphical representation of the calls relation

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:
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.

**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 set or relation and is explained in the section called "Miscellaneous" [15]. 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)
```

**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 the section called "Bottom of a Relation: bottom" [22] 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">, <"a", "d">, <"a", "f">,
<"b", "e">, <"e", "g">, <"g", "e">, <"a", "e">, <"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 1.3, “Graphical representation of the calls relation”[4]. 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.

**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
Types and Values

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)).

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 \(<\text{int}, \text{str}>\). Rscript directly supports tuples consisting of two elements (also known 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 \(<\text{int}, \text{int}>\),
- \(<1, 2, 3>\) is of type \(<\text{int}, \text{int}, \text{int}>\),
- \(<1, \text{"a"}, 3>\) is of type \(<\text{int}, \text{str}, \text{int}>\).

Sets. Sets are represented by the type \(\text{set}[T]\), where \(T\) is an arbitrary type. Examples are \(\text{set}[\text{int}]\), \(\text{set}[<\text{int},\text{int}>]\) and \(\text{set}[\text{set}[\text{str}]]\). Sets are denoted by a list of elements, separated by commas 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 \(\text{set}[\text{int}]\),
- \(\{<1,10>, <2,20>, <3,30>\}\) is of type \(\text{set}[<\text{int},\text{int}>]\),
- \(\{<\text{"a"},10>, <\"b\"",20>, <\"c\",30>\}\) is of type \(\text{set}[<\text{str},\text{int}>]\), and
- \(\{<\text{"a"}, \"b\">, \{\text{"c"}, \"d\", \"e\"\}\}\) is of type \(\text{set}[\text{set}[\text{str}]]\).

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 \(\text{rel}[T_1, T_2]\), where \(T_1\) and \(T_2\) are arbitrary types; it is a shorthand for \(\text{set}[<T_1, T_2>]\). Examples are \(\text{rel}[\text{int}, \text{str}]\) and \(\text{rel}[\text{int}, \text{set}[\text{str}]]\). Relations are denoted by \(\{<E_1, E_2>, <E_3, E_3>, ..., <E_n, E_n>\}\), where the \(E_i\)\((1 \leq i \leq n)\) are expressions that yield the desired element type. For example, \(\{<1, \text{"a"}>, <2, \text{"b"}>, <3, \text{"e"}>\}\) is of type \(\text{rel}[\text{int}, \text{str}]\). Not surprisingly, \(n\)-ary relations are represented by the type \(\text{rel}[T_1, T_2, ..., T_n]\) which is a shorthand for \(\text{set}[<T_1, T_2, ..., T_n>]\). 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 \(\text{rel}[\text{int}, \text{int}]\) (yes indeed, you saw this same example before and then we gave \(\text{set}[<\text{int}, \text{int}]\) as its type; remember that these types are interchangeable.).
• \{"a",10>, <"b",20>, <"c",30>\} is of type rel[\text{str,int}], and
• \{("a", 1, "b"), ("c", 2, "d")\} is of type rel[\text{str,int,str}].

### 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

\[
type \ T_1 = T_2
\]

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

\[
type \ ModuleId = \text{str} \\
\text{type} \ Frequency = \text{int}
\]

introduces two new type names \( \text{ModuleId} \) and \( \text{Frequency} \), both an alias for the type \( \text{str} \). 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

\[
type \ T = <T_1 \ F_1, \ldots, T_n \ F_n>
\]

introduces a new composite type \( T \), with \( n \) elements. The \( i \)-th element \( T_i \ F_i \) 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,\footnote{The variable declarations that appear on lines 2 and 3 of this example are explained fully in the section called "Declarations" [10].}

\[
type \ Triple = <\text{int} \ \text{left}, \ \text{str} \ \text{middle}, \ \text{bool} \ \text{right}> \\
\text{Triple} \ TR = <3, \ "a", \ \text{true}> \\
\text{str} \ S = \text{TR}.\text{middle}
\]

first introduces the composite type \( \text{Triple} \) and defines the \( \text{Triple} \) variable \( \text{TR} \). Next, the field selection \( \text{TR}.\text{middle} \) 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.

### Comprehensions

We will use the familiar notation
\( \{E_1, \ldots, E_m \mid G_1, \ldots, G_n\} \)

to denote the construction of a set consisting of the union of successive values of the expressions \( E_1, \ldots, E_m \). The values and the generated set are determined by \( E_1, \ldots, E_m \) and the generators \( G_1, \ldots, G_n \). \( E \) 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 \( E_1, \ldots, E_m \). 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.

**Generators**

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

- \( TV : E \): the elements of the set \( S \) (of type \( \text{set}[T] \)) that results from the evaluation of expression \( E \) are enumerated and subsequently assigned to the new variable \( V \) of type \( T \). Examples are:
  
  - \( \text{int} \ N : \{1, 2, 3, 4, 5\} \).
  - \( \text{str} \ K : \text{KEYWORDS} \), where \( \text{KEYWORDS} \) should evaluate to a value of \( \text{set}[\text{str}] \).
  - \( \langle D_1, \ldots, D_n \rangle : E \): the elements of the relation \( R \) (of type \( \text{rel}[T'_1, \ldots, T'_n] \), 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 (\( i=1, \ldots, n \)) 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:
  
  - \( \langle \text{str} \ K, \text{int} \ N \rangle : \{\langle "a",10\rangle, \langle "b",20\rangle, \langle "c",30\rangle\} \).
  - \( \langle \text{str} \ K, \text{int} \ N \rangle : \text{FREQUENCIES} \), where \( \text{FREQUENCIES} \) should evaluate to a value of \( \text{rel}[\text{str},\text{int}] \).
  - \( \langle \text{str} \ K, 10 \rangle : \text{FREQUENCIES} \), will only generate pairs with 10 as second element.

**Test.** A test is a boolean-valued expression. If the evaluation yields \text{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 \text{false} this indicates that the current combination of values is undesired, and that another combination should be tried. Examples:

- \( N \geq 3 \) tests whether \( N \) has a value greater than or equal 3.
  
- \( S == "\text{coffee}" \) 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:

- \( TV <- E \): assigns the value of expression \( E \) to the new variable \( V \) of type \( T \).
  
- \( \langle R_1, \ldots, R_n \rangle <- E \): combines the elements of the \( n \)-tuple resulting from the evaluation of expression \( E \) with each \( T_j \) as follows:
    - If \( R_i \) is a variable declaration of the form \( T_j \ V_i \), 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:

• \( \text{rel}[\text{str},\text{str}] \text{ ALLCALLS} \leftarrow \text{CALLS}+ \) assigns the transitive closure of the relation \( \text{CALLS} \) to the variable \( \text{ALLCALLS} \).

• \( \text{bool Smaller} \leftarrow A \leq B \) assigns the result of the test \( A \leq B \) to the Boolean variable \( \text{Smaller} \).

• \( \langle \text{int N, str S, 10} \rangle \leftarrow E \) evaluates expression \( E \) (which should yield a tuple of type \( \langle \text{int, str, int} \rangle \)) 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.

Examples of Comprehensions

• \( \{X \mid \text{int } X : \{1, 2, 3, 4, 5\}, X \geq 3\} \) yields the set \( \{3,4,5\} \).

• \( \{<X, Y> \mid \text{int } X : \{1, 2, 3\}, \text{int } Y : \{2, 3, 4\}, X \geq Y\} \) yields the relation \( \{<2, 2>, <3, 2>, <3, 3>\} \).

• \( \{<Y, X> \mid \text{int } X, \text{int } Y : \{<1,10>, <2,20>\}\} \) yields the inverse of the given relation: \( \{<10,1>, <20,2>\} \).

• \( \{X, X \times X \mid X : \{1, 2, 3, 4, 5\}, X \geq 3\} \) yields the set \( \{3,4,5,9,16,25\} \).

Declarations

Variable Declarations

A variable declaration has the form

\[
T \ V = E
\]

where \( T \) is a type, \( V \) is a variable name, and \( E \) 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:

• \( \text{int } \max = 100 \) declares the integer variable \( \max \) with value 100.

• The definition

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

declares the variable \( \text{day} \), a relation that maps strings to integers.

Local Variable Declarations

Local variables can be introduced as follows:

\[
E \text{ where } T_1 \ V_1 = E_1, \ldots, T_n \ V_n = E_n \text{ end where}
\]

First the local variables \( V_i \) are bound to their respective values \( E_i \), and then the value of expression \( E \) is yielded.
**Function Declarations**

A function declaration has the form

\[ T \, F(T_1 \, V_1, \ldots, \, T_n \, V_n) = E \]

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

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

yields the inverse of the argument relation \( R \). For instance, \( \text{invert}({\langle 1,10 \rangle, \ <2,20 \rangle}) \) 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

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

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

- \( \text{invert2}({\langle 1,10 \rangle, \ <2,20 \rangle}) \) yields \( \{<10,1>, <20,2>\} \).
- \( \text{invert2}({\langle "mon", 1 \rangle, \ <"tue", 2 \rangle}) \) yields \( \{1, "mon">, <2, "tue">\} \).

- The function

\[
\langle\&T2, \&T1\rangle \ \text{swap}(\&T1 \ A, \&T2 \ B) = \langle B, A \rangle
\]

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

- \( \text{swap}(<1, 2>) \) yields \( <2,1> \) and
- \( \text{swap}(<"wed", 3>) \) yields \( <3, "wed"> \).

**Assertions**

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

\[
\text{assert } L; \ E
\]

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 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:

- \( \text{assert } "Equality on Sets 1"; \ {1, 2, 3} == \ {3, 2, 1, 1} \)

**Equations**

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

\[
equations
\]

\[
\]

\[
\]

\[
\]

\[
\]
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 $E_j$ 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 $E_j$, 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.

  ```plaintext
  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:

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

### Built-in Operators

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

- Operations on Booleans (the section called "Operations on Booleans[13]"): logical operators (and, or, implies and not).

- Operations on integers (the section called "Operations on Integer[13]"): arithmetic operators (+, -, *, and /) and comparison operators (==, !=, <, <=, >, and >=).

- Operations on strings (the section called "Operations on String[13]"): comparison operators (==, !=, <, <=, >, and >=).

- Operations on locations (the section called "Operations on Location[14]"). comparison operators (==, !=, <, <=, >, and >=).

- Operations on sets or relations (the section called "Operations on Sets or Relation[14]": membership tests (in, not in), comparison operators (==, !=, <, <=, >, and >=), and construction operators (union, inter, diff).

- Operations on relations (the section called "Operations on Relation[15]"): 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.

### Operations on Booleans

**Table 1.1. Operations on Booleans**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool₁ and bool₂</td>
<td>yields true if both arguments have the value true and false otherwise</td>
</tr>
<tr>
<td>bool₁ and bool₂</td>
<td>yields true if either argument has the value true and false otherwise</td>
</tr>
<tr>
<td>bool₁ implies bool₂</td>
<td>yields false if bool₁ has the value true and bool₂ has value false, and true otherwise</td>
</tr>
<tr>
<td>not bool</td>
<td>yields true if bool is false and true otherwise</td>
</tr>
</tbody>
</table>

### Operations on Integers

**Table 1.2. Operations on Integers**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int₁ == int₂</td>
<td>yields true if both arguments are numerically equal and false otherwise</td>
</tr>
<tr>
<td>int₁ != int₂</td>
<td>yields true if both arguments are numerically unequal and false otherwise</td>
</tr>
<tr>
<td>int₁ &lt;= int₂</td>
<td>yields true if int₁ is numerically less than or equal to int₂ and false otherwise</td>
</tr>
<tr>
<td>int₁ &lt; int₂</td>
<td>yields true if int₁ is numerically less than int₂ and false otherwise</td>
</tr>
<tr>
<td>int₁ &gt;= int₂</td>
<td>yields true if int₁ is numerically greater than or equal than int₂ and false otherwise</td>
</tr>
<tr>
<td>int₁ &gt; int₂</td>
<td>yields true if int₁ is numerically greater than int₂ and false otherwise</td>
</tr>
<tr>
<td>int₁ + int₂</td>
<td>yields the arithmetic sum of int₁ and int₂</td>
</tr>
<tr>
<td>int₁ - int₂</td>
<td>yields the arithmetic difference of int₁ and int₂</td>
</tr>
<tr>
<td>int₁ * int₂</td>
<td>yields the arithmetic product of int₁ and int₂</td>
</tr>
<tr>
<td>int₁ / int₂</td>
<td>yields the arithmetic division of int₁ and int₂</td>
</tr>
</tbody>
</table>

### Operations on Strings

**Table 1.3. Operations on Strings**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str₁ == str₂</td>
<td>yields true if both arguments are equal and false otherwise</td>
</tr>
<tr>
<td>str₁ != str₂</td>
<td>yields true if both arguments are unequal and false otherwise</td>
</tr>
<tr>
<td>str₁ &lt;= str₂</td>
<td>yields true if str₁ is lexicographically less than or equal to str₂ and false otherwise</td>
</tr>
<tr>
<td>str₁ &lt; str₂</td>
<td>yields true if str₁ is lexicographically less than str₂ and false otherwise</td>
</tr>
<tr>
<td>str₁ &gt;= str₂</td>
<td>yields true if str₁ is lexicographically greater than or equal to str₂ and false otherwise</td>
</tr>
<tr>
<td>str₁ &gt; str₂</td>
<td>yields true if str₁ is lexicographically greater than str₂ and false otherwise</td>
</tr>
</tbody>
</table>
Operations on Locations

Table 1.4. Operations on Locations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>loc1 == loc2</code></td>
<td>yields true if both arguments are identical and false otherwise</td>
</tr>
<tr>
<td><code>loc1 != loc2</code></td>
<td>yields true if both arguments are not identical and false otherwise</td>
</tr>
<tr>
<td><code>loc1 &lt;= loc2</code></td>
<td>yields true if <code>loc1</code> is textually contained in or equal to <code>loc2</code> and false otherwise</td>
</tr>
<tr>
<td><code>loc1 &lt; loc2</code></td>
<td>yields true if <code>loc1</code> is strictly textually contained in <code>loc2</code> and false otherwise</td>
</tr>
<tr>
<td><code>loc1 &gt;= loc2</code></td>
<td>yields true if <code>loc1</code> is textually encloses or is equal to <code>loc2</code> and false otherwise</td>
</tr>
<tr>
<td><code>loc1 &gt;= loc2</code></td>
<td>yields true if <code>loc1</code> is textually encloses <code>loc2</code> and false otherwise</td>
</tr>
</tbody>
</table>

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.

Operations on Sets or Relations

Membership Tests

Table 1.5. Membership Tests

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any in set</td>
<td>yields true if <code>any</code> occurs as element in <code>set</code> and false otherwise</td>
</tr>
<tr>
<td>any notin set</td>
<td>yields false if <code>any</code> occurs as element in <code>set</code> and false otherwise</td>
</tr>
<tr>
<td>tuple in rel</td>
<td>yields true if <code>tuple</code> occurs as element in <code>rel</code> and false otherwise</td>
</tr>
<tr>
<td>tuple notin rel</td>
<td>yields false if <code>tuple</code> occurs as element in <code>rel</code> and false otherwise</td>
</tr>
</tbody>
</table>

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]`. 
Comparisons

Table 1.6. Comparisons

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{set}_1 == \text{set}_2)</td>
<td>yields true if both arguments are equal sets and false otherwise</td>
</tr>
<tr>
<td>(\text{set}_1 != \text{set}_2)</td>
<td>yields true if both arguments are unequal sets and false otherwise</td>
</tr>
<tr>
<td>(\text{set}_1 \leq \text{set}_2)</td>
<td>yields true if (\text{set}_1) is a subset of (\text{set}_2) and false otherwise</td>
</tr>
<tr>
<td>(\text{set}_1 &lt; \text{set}_2)</td>
<td>yields true if (\text{set}_1) is a strict subset of (\text{set}_2) and false otherwise</td>
</tr>
<tr>
<td>(\text{set}_1 \geq \text{set}_2)</td>
<td>yields true if (\text{set}_1) is a superset of (\text{set}_2) and false otherwise</td>
</tr>
<tr>
<td>(\text{set}_1 &gt; \text{set}_2)</td>
<td>Yields true if (\text{set}_1) is a strict superset of (\text{set}_2) and false otherwise</td>
</tr>
</tbody>
</table>

Construction

Table 1.7. Construction

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{set}_1 \cup \text{set}_2)</td>
<td>yields the set resulting from the union of the two arguments</td>
</tr>
<tr>
<td>(\text{set}_1 \cap \text{set}_2)</td>
<td>yields the set resulting from the intersection of the two arguments</td>
</tr>
<tr>
<td>(\text{set}_1 \setminus \text{set}_2)</td>
<td>yields the set resulting from the difference of the two arguments</td>
</tr>
</tbody>
</table>

Examples.

- \(\{1, 2, 3\} \cup \{4, 5, 6\}\) yields \(\{1, 2, 3, 4, 5, 6\}\).
- \(\{1, 2, 3\} \cup \{1, 2, 3\}\) yields \(\{1, 2, 3\}\).
- \(\{1, 2, 3\} \cup \{4, 5, 6\}\) yields \(\{1, 2, 3, 4, 5, 6\}\).
- \(\{1, 2, 3\} \cap \{4, 5, 6\}\) yields \(\{\}\).
- \(\{1, 2, 3\} \cap \{1, 2, 3\}\) yields \(\{1, 2, 3\}\).
- \(\{1, 2, 3, 4\} \setminus \{1, 2, 3\}\) yields \(\{4\}\).
- \(\{1, 2, 3\} \setminus \{4, 5, 6\}\) yields \(\{1, 2, 3\}\).

Miscellaneous

Table 1.8. Miscellaneous

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># \text{set}</td>
<td>yields the number of elements in \text{set}</td>
</tr>
<tr>
<td># \text{rel}</td>
<td>yields the number of elements in \text{rel}</td>
</tr>
</tbody>
</table>

Examples.

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

Operations on Relations
### Table 1.9. Operations on Relations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rel_1 \circ rel_2$</td>
<td>yields the relation resulting from the composition of the two arguments</td>
</tr>
<tr>
<td>$set_1 \times set_2$</td>
<td>yields the relation resulting from the Cartesian product of the two arguments</td>
</tr>
<tr>
<td>$rel[-, set]$</td>
<td>yields the left image of $rel$</td>
</tr>
<tr>
<td>$rel[-, elem]$</td>
<td>yields the left image of $rel$</td>
</tr>
<tr>
<td>$rel[ set, -]$</td>
<td>yields the right image of $rel$</td>
</tr>
<tr>
<td>$rel[ elem, -]$</td>
<td>yields the right image of $rel$</td>
</tr>
<tr>
<td>$rel[ elem]$</td>
<td>yields the right image of $rel$</td>
</tr>
<tr>
<td>$rel[ set]$</td>
<td>yields the right image of $rel$</td>
</tr>
<tr>
<td>$rel +$</td>
<td>yields the relation resulting from the transitive closure of $rel$</td>
</tr>
<tr>
<td>$rel ^*$</td>
<td>yields the relation resulting from the reflexive transitive closure of $rel$</td>
</tr>
</tbody>
</table>

#### Composition: $\circ$.

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

$$
rel[&T1,&T3] \; \text{compose}(rel[&T1,&T2] \; R_1, \; rel[&T2,&T3] \; R_2) = \\
\{ \langle V, Y \rangle \mid \langle \&T1 \; V, \; \&T2 \; W \rangle : R_1, \; \langle \&T2 \; X, \; \&T3 \; Y \rangle : R_2, \; W == X \} 
$$

**Example.**

- $\{<1,10>, <2,20>, <3,15>\} \circ \{<10,100>, <20,200>\}$ yields $\{<1,100>, <2,200>\}$.

#### Cartesian product: $\times$.

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

$$
rel[&T1,&T2] \; \text{product}(set[&T1] \; S_1, \; set[&T2] \; S_2) = \\
\{ \langle V, W \rangle \mid \&T1 \; V : S_1, \; \&T2 \; W : S_2 \}
$$

**Example.**

- $\{1, 2, 3\} \times \{9\}$ 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:

$$
set[&T1] \; \text{left-image}(rel[&T1,&T2] \; R, \; \&T2 \; E) = \\
\{ \; V \mid \langle \&T1 \; V, \; \&T2 \; W \rangle : R, \; W == E \}
$$

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

$$
set[&T1] \; \text{left-image}(rel[&T1,&T2] \; R, \; set[&T2] \; S) = \\
\{ \; V \mid \langle \&T1 \; V, \; \&T2 \; W \rangle : R, \; W \; \text{in} \; S \}
$$

**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 $\{1\}$. 

---
• \text{Rel}[-,\{10, 20\}] \text{ 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:

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

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

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

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

• \text{Rel}[1] \text{ yields } \{10, 11\}.

• \text{Rel}[\{1\}] \text{ yields } \{10, 11\}.

• \text{Rel}[\{1, 2\}] \text{ yields } \{10, 11, 20, 21\}.

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

**Built-in Functions**

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

• Elementary functions on sets and relations (the section called “Elementary Functions on Sets and Relations” [18]): identity (id), inverse (inv), complement (compl), and powerset (power0, power1).

• Extraction from relations (the section called “Extraction from Relations” [19]): domain (domain), range (range), and carrier (carrier).

• Restrictions and exclusions on relations (the section called “Restrictions and Exclusions on Relations” [20]: domain restriction (domainR), range restriction (rangeR), carrier restriction (carrierR), domain exclusion (domainX), range exclusion (rangeX), and carrier exclusion (carrierX).

• Functions on tuples (the section called “Tuples” [21]: first element (first), and second element (second).

• Relations viewed as graphs (the section called “Relations viewed as graphs” [21]: the root elements (top), the leaf elements (bottom), reachability with restriction (reachR), and reachability with exclusion (reachX).

• Functions on locations (the section called “Functions on Locations” [22]: file name (filename), beginning line (beginline), first column (begincol), ending line (endline), and ending column (endcol).

• Functions on sets of integers (the section called “Functions on Sets of Integers” [23]: sum (sum), average (average), maximum (max), and minimum (min).

The following sections give detailed descriptions and examples of all built-in functions.
Elementary Functions on Sets and Relations

Identity Relation: \(\text{id}\)

Definition:

\[
\begin{align*}
\text{id}(\text{set}[T] S) &= \{ <X, X> \mid T X : S \}
\end{align*}
\]

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

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

Deprecated: Set with unique elements: \(\text{unique}\)

Definition:

\[
\begin{align*}
\text{unique}(\text{set}[T] S) &= \text{primitive}
\end{align*}
\]

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:

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

Inverse of Relation: \(\text{inv}\)

Definition:

\[
\begin{align*}
\text{inv}(\text{rel}[T1, T2] R) &= \{ <Y, X> \mid <T1 X, T2 Y> : R \}
\end{align*}
\]

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:

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

Complement of a Relation: \(\text{compl}\)

Definition:

\[
\begin{align*}
\text{compl}(\text{rel}[T1, T2] R) &= (\text{domain}(R) \times \text{range}(R)) \setminus R
\end{align*}
\]

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

- \(\text{compl}({<1,10>})\) yields \{<1, 1>, <10, 1>, <10, 10>\}.

Powerset of a Set: \(\text{power0}\)

Definition:

\[
\begin{align*}
\text{power0}(\text{set}[T] S) &= \text{set[set[T]]}
\end{align*}
\]
primitive

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

- \( \text{power0}([1, 2, 3, 4]) \) yields

  \[
  \{\{\}, \{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}\}
  \]

**Powerset of a Set: power1**

Definition:

\[
\text{set}[\text{set}[\&T]] \ \text{power1}(\text{set}[\&T] \ S) =
\]

primitive

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

- \( \text{power1}([1, 2, 3, 4]) \) yields

  \[
  \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}\}
  \]

**Extraction from Relations**

**Domain of a Relation: domain**

Definition:

\[
\text{set}[\&T] \ \text{domain}(\text{rel}[\&T1,\&T2] \ R) =
\]

\[
\{ \ X \ | \ <\&T1 \ X, \ &T2 \ Y> : \ R \}
\]

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

- \( \text{domain}([[1,10], [2,20]]) \) yields \( 1, 2 \).

- \( \text{domain}([[\text{"mon"}, 1], [\text{"tue"}, 2]]) \) yields \{\text{"mon"}, \text{"tue"}\}.

**Range of a Relation: range**

Definition:

\[
\text{set}[\&T2] \ \text{range}(\text{rel}[\&T1,\&T2] \ R) =
\]

\[
\{ \ Y \ | \ <\&T1 \ X, \ &T2 \ Y> : \ R \}
\]

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

- \( \text{range}([[1,10], [2,20]]) \) yields \( 10, 20 \).

- \( \text{range}([[\text{"mon"}, 1], [\text{"tue"}, 2]]) \) yields \( 1, 2 \).

**Carrier of a Relation: carrier**

Definition:

\[
\text{set}[\&T] \ \text{carrier}(\text{rel}[\&T,\&T] \ R) =
\]

\[
\text{domain}(R) \ \text{union} \ \text{range}(R)
\]
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:

- \( \text{carrier}([<1,10>, <2,20>]) \) yields \( \{1, 10, 2, 20\} \).

## Restrictions and Exclusions on Relations

### Domain Restriction of a Relation: \text{domainR}

Definition:

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

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

- \( \text{domainR}([<1,10>, <2,20>, <3,30>], \{3, 1\}) \) yields \( [1,10>, <3,30>] \).

### Range Restriction of a Relation: \text{rangeR}

Definition:

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

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

- \( \text{rangeR}([<1,10>, <2,20>, <3,30>], \{30, 10\}) \) yields \( [1,10>, <3,30>] \).

### Carrier Restriction of a Relation: \text{carrierR}

Definition:

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

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

- \( \text{carrierR}([<1,10>, <2,20>, <3,30>], \{10, 1, 20\}) \) yields \( [1,10>] \).

### Domain Exclusion of a Relation: \text{domainX}

Definition:

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

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

- \( \text{domainX}([<1,10>, <2,20>, <3,30>], \{3, 1\}) \) yields \( [2,20>] \).

### Range Exclusion of a Relation: \text{rangeX}

Definition:

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

- \( \text{rangeX}([<1,10>, <2,20>, <3,30>], \{30, 10\}) \) yields \( \{2, 20\} \).

**Carrier Exclusion of a Relation: carrierX**

Definition:

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

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

- \( \text{carrierX}([<1,10>, <2,20>, <3,30>], \{10, 1, 20\}) \) yields \( \{3,30\} \).

**Tuples**

**First Element of a Tuple: \( \text{first} \)**

Definition:

\[
^{\&T1} \text{first}(<^{\&T1},^{\&T2}> \ P) = \\
\text{primitive}
\]

Yields the first element of the tuple \( P \). Examples:

- \( \text{first}(<1, 10>) \) yields 1.
- \( \text{first}(<"mon", 1>) \) yields "mon".

**Second Element of a Tuple: \( \text{second} \)**

Definition:

\[
^{\&T2} \text{second}(<^{\&T1},^{\&T2}> \ P) = \\
\text{primitive}
\]

Yields the second element of the tuple \( P \). Examples:

- \( \text{second}(<1, 10>) \) yields 10.
- \( \text{second}(<"mon", 1>) \) yields 1.

**Relations viewed as graphs**

**Top of a Relation: \( \text{top} \)**

Definition:

\[
\text{set}[^{\&T}] \text{ top}(\text{rel}[^{\&T},^{\&T}] \ R) = \\
\text{unique}(%domain(R)) \ \text{\backslash} \ \text{range}(R)
\]

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:

- \( \text{top}([<1,2>, <1,3>, <2,4>, <3,4>]) \) yields \( \{1\} \).
Bottom of a Relation: bottom

Definition:

\[
\text{set}[^T\text{]} \ \text{bottom}(\text{rel}[^T,^T\text{]} \ R) = \\
\quad \text{unique}(\text{range}(R)) \ \setminus \ \text{domain}(R)
\]

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:

- \( \text{bottom}([<1,2>, <1,3>, <2,4>, <3,4>]) \) yields \( \{4\} \).

Reachability with Restriction: reachR

Definition:

\[
\text{set}[^T\text{]} \ \text{reachR}( \text{set}[^T\text{]} \ \text{Start}, \text{set}[^T\text{]} \ \text{Restr}, \ \text{rel}[^T,^T\text{]} \ \text{Rel}) = \\
\quad \text{range}(\text{domainR}(\text{Rel}, \ \text{Start}) \ o \ \text{carrierR}(\text{Rel}, \ \text{Restr}))+
\]

Yields the elements that can be reached from set \( \text{Start} \) using the relation \( \text{Rel} \), such that only elements in set \( \text{Restr} \) are used. Example:

- \( \text{reachR}(\{1\}, \{1, 2, 3\}, [<1,2>, <1,3>, <2,4>, <3,4>]) \) yields \( \{2, 3\} \).

Reachability with Exclusion: reachX

Definition:

\[
\text{set}[^T\text{]} \ \text{reachX}( \text{set}[^T\text{]} \ \text{Start}, \text{set}[^T\text{]} \ \text{Excl}, \ \text{rel}[^T,^T\text{]} \ \text{Rel}) = \\
\quad \text{range}(\text{domainR}(\text{Rel}, \ \text{Start}) \ o \ \text{carrierX}(\text{Rel}, \ \text{Excl}))+
\]

Yields the elements that can be reached from set \( \text{Start} \) using the relation \( \text{Rel} \), such that no elements in set \( \text{Excl} \) are used. Example:

- \( \text{reachX}(\{1\}, \{2\}, [<1,2>, <1,3>, <2,4>, <3,4>]) \) yields \( \{3, 4\} \).

Functions on Locations

File Name of a Location: filename

Definition:

\[
\text{str} \ \text{filename}(\text{loc} \ A) = \\
\quad \text{primitive}
\]

Yields the file name of location \( A \). Example:

- \( \text{filename(area-in-file("picol.trm",area(5,2,6,8,0,0)))} \) yields "picol.trm".

Beginning Line of a Location: beginline

Definition:

\[
\text{int} \ \text{beginline}(\text{loc} \ A) = \\
\quad \text{primitive}
\]

Yields the first line of location \( A \). Example:

- \( \text{beginline(area-in-file("picol.trm",area(5,2,6,8,0,0)))} \) yields 5.
First Column of a Location: \texttt{begincol}

Definition:

\begin{verbatim}
int begincol(loc A) =
  primitive
\end{verbatim}

Yields the first column of location \texttt{A}. Example:

\begin{itemize}
  \item \texttt{begincol(area-in-file("pico1.trm", area(5,2,6,8,0,0)))} yields 2.
\end{itemize}

Ending Line of a Location: \texttt{endline}

Definition:

\begin{verbatim}
int endline(loc A) =
  primitive
\end{verbatim}

Yields the last line of location \texttt{A}. Example:

\begin{itemize}
  \item \texttt{endline(area-in-file("pico1.trm", area(5,2,6,8,0,0)))} yields 6.
\end{itemize}

Ending Column of a Location: \texttt{endcol}

Definition:

\begin{verbatim}
int endcol(loc A) =
  primitive
\end{verbatim}

Yields the last column of location \texttt{A}. Example:

\begin{itemize}
  \item \texttt{endcol(area-in-file("pico1.trm", area(5,2,6,8,0,0)))} yields 8.
\end{itemize}

Functions on Sets of Integers

The functions in this section operate on sets of integers. Some functions (i.e., \texttt{sum-domain}, \texttt{sum-range}, \texttt{average-domain}, \texttt{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.)

Sum of a Set of Integers: \texttt{sum}

Definition:

\begin{verbatim}
int sum(set[int] S) =
  primitive
\end{verbatim}

Yields the sum of the integers in set \texttt{S}. Example:

\begin{itemize}
  \item \texttt{sum({1, 2, 3})} yields 6.
\end{itemize}

Sum of First Elements of Tuples in a Relation: \texttt{sum-domain}

Definition:

\begin{verbatim}
int sum-domain(rel[int,&T] R) =
  primitive
\end{verbatim}

Yields the sum of the integers that appear in the first element of the tuples of \texttt{R}. Example:
• \text{sum-domain}([<1,"a">, <2,"b">, <1,"c">]) \text{ yields 4.}

Be aware that \text{sum(domain}([<1,"a">, <2,"b">, <1,"c">])) \text{ 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.}

\textbf{Sum of Second Elements of Tuples in a Relation: sum-range}

\textbf{Definition:}

\begin{verbatim}
int sum-range(set[int] S) =
  primitive
\end{verbatim}

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

• \text{sum-range}([<"a",1>, <"b",2>, <"c",1>]) \text{ yields 4.}

\textbf{Average of a Set of Integers: average}

\textbf{Definition:}

\begin{verbatim}
int average(set[int] S) =
  sum(S)/(\#S)
\end{verbatim}

Yields the average of the integers in set \( S \). Example:

• \text{average}([1, 2, 3]) \text{ yields 3.}

\textbf{Average of First Elements of Tuples in a Relation: average-domain}

\textbf{Definition:}

\begin{verbatim}
int average-domain(rel[int,&T] R) =
  sum-domain(R)/(\#R)
\end{verbatim}

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

• \text{average-domain}([<1,"a">, <2,"b">, <3,"c">]) \text{ yields 2.}

\textbf{Average of Second Elements of Tuples in a Relation: average-range}

\textbf{Definition:}

\begin{verbatim}
int average-range(rel[int,&T] R) =
  sum-range(R)/(\#R)
\end{verbatim}

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

• \text{average-range}([<"a",1>, <"b",2>, <"c",3>]) \text{ yields 2.}

\textbf{Maximum of a Set of Integers: max}

\textbf{Definition:}

\begin{verbatim}
int max(set[int] S) =
  primitive
\end{verbatim}

Yields the largest integer in set \( S \). Example:
**Minimum of a Set of Integers:** \( \min \)

**Definition:**

\[
\text{int } \min(\text{set}[\text{int}] \ S) = \text{primitive}
\]

Yields the largest integer in set \( S \). Example:

\[
\min(\{1, 2, 3\}) \text{ yields 3}.
\]

---

**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 the section called “A Motivating Example”[3] as a reminder. The component structure of an application is analyzed in the section called “Analyzing the Component Structure of an Application” [25] and Java systems are analyzed in the section called “Analyzing the Structure of Java Systems” [26]. Next we move on to the detection of initialized variables in the section called “Finding Uninitialized and Unused Variables in a Program”[27] and we explain how source code locations can be included in a such an analysis (the section called “Using Locations to Represent Program Fragments” [29]). As an example of computing code metrics, we describe the calculation of McCabe's cyclomatic complexity in the section called “McCabe Cyclomatic Complexity”[30]. Several examples of dataflow analysis follow in the section called “Dataflow Analysis”[31]. A description of program slicing concludes the chapter (the section called “Program Slicing” [37]).

---

**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 [Kri99]. 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 \( \text{Calls2} \), the next step is to define the components of the system and to define a \( \text{PartOf} \) relation between procedures and components.

```
rel[proc,proc] \text{Calls} = \{ <\text{main}, \text{a}>, <\text{main}, \text{b}>, <\text{a}, \text{b}>,
                      <\text{a}, \text{c}>, <\text{a}, \text{d}>, <\text{b}, \text{d}>
\}

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

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

Actual lifting, amounts to translating each call between procedures by a call between components. This is achieved by the following function \( \text{lift} \):

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

---
In our example, the lifted call relation between components is obtained by
\[
\text{rel[comp,comp]} \text{ ComponentCalls} = \text{lift(Calls2, PartOf)}
\]
and has as value:
\[
\{\langle "DB", "Lib" \rangle, \langle "Appl", "Lib" \rangle, \langle "Appl", "DB" \rangle, \langle "Appl", "Appl" \rangle\}
\]
The relevant relations for this example are shown in Figure 1.4. “(a) Calls2; (b) PartOf; (c) ComponentCalls.” [26].

**Figure 1.4. (a) Calls2; (b) PartOf; (c) ComponentCalls.**

---

**Analyzing the Structure of Java Systems**

Now we consider the analysis of Java systems (inspired by [BNL03]. Suppose that the type class is defined as follows
\[
\text{type class} = \text{str}
\]
and that the following relations are available about a Java application:

- \[
\text{rel[class,class]} \text{ CALL}: \text{If } \langle C_1, C_2 \rangle \text{ is an element of CALL, then some method of } C_2 \text{ is called from } C_1.
\]

- \[
\text{rel[class,class]} \text{ INHERITANCE}: \text{If } \langle C_1, C_2 \rangle \text{ is an element of INHERITANCE, then class } C_1 \text{ either extends class } C_2 \text{ or } C_1 \text{ implements interface } C_2.
\]

- \[
\text{rel[class,class]} \text{ CONTAINMENT}: \text{If } \langle C_1, C_2 \rangle \text{ is an element of CONTAINMENT, then one of the attributes of class } C_1 \text{ is of type } C_2.
\]

To make this more explicit, consider the class LocatorHandle from the JHotDraw application (version 5.2) as shown here:

```java
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.
     */
```
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.

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:

\[
\text{rel}[\text{class, class}] \quad \text{USE} = \text{CALL} \cup \text{CONTAINMENT} \cup \text{INHERITANCE}
\]

\[
\text{set}[\text{str}] \quad \text{ClassesInCycle} = \{ \text{C1} \mid \langle \text{class C1, class C2} \rangle : \text{USE}+, \text{C1} == \text{C2} \}
\]

First, we define the USE relation as the union of the three available relations CALL, CONTAINMENT and INHERITANCE. Next, we consider all pairs \(\langle \text{C1, C2} \rangle\) in the transitive closure of the USE relation such that \(\text{C1}\) and \(\text{C2}\) 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.

\[
\text{rel}[\text{class, class}] \quad \text{USE} = \text{CALL} \cup \text{CONTAINMENT} \cup \text{INHERITANCE}
\]

\[
\text{set}[\text{class}] \quad \text{CLASSES} = \text{carrier}(\text{USE})
\]

\[
\text{rel}[\text{class, set[class]}] \quad \text{ClassCycles} = \{ \langle \text{C, USETRANS[C]} \rangle \mid \text{class C} : \text{CLASSES}, \langle \text{C, C} \rangle \text{ in USETRANS} \}
\]

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

### Finding Uninitialized and Unused Variables in a Program

Consider the following program in the toy language Pico: (This is an extended version of the example presented earlier in [Kli03].)

```
[ 1] begin declare x : natural, y : natural,
[ 3]  x := 3;
[ 5]  if q then
```
Rscript --- a Relational Approach to Software Analysis

\[ \begin{align*}
[6] & \quad z := y + x \\
[7] & \quad \text{else} \\
[8] & \quad x := 4 \\
[9] & \quad \text{fi;} \\
[10] & \quad y := z \\
[11] & \quad \text{end}
\end{align*} \]

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 ?? 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 \( \text{PRED} \) (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 \( \text{DEFS} \).

- The uses of each variable, i.e., the program statements where the value of the variable is used. It is represented by the relation \( \text{USES} \).

In this example, we will use line numbers to identify the statements in the program. (In the section called “Using Locations to Represent Program Fragments"[29], we will use locations to represent statements.) Assuming that there is a tool to extract the above information from a program text, we get the following for the above example:

\[
\begin{align*}
type\ expr &= \text{int} \\
type\ varname &= \text{str} \\
expr\ \text{ROOT} &= 1 \\
rel[expr,expr] \text{PRED} &= \{ <1,3>, <3,4>, <4,5>, <5,6>, <5,8>, <6,10>, <8,10> \} \\
rel[expr,varname] \text{DEFS} &= \{ <3,\"x\">, <4,\"p\">, <6,\"z\">, <8,\"x\">, <10,\"y\"> \} \\
rel[expr,varname] \text{USES} &= \{ <5,\"q\">, <6,\"y\">, <6,\"x\">, <10,\"z\"> \}
\end{align*}
\]

This concludes the extraction phase. Next, we have to enrich these basic facts to obtain the initialized variables in the program. So, when is a variable \( V \) in some statement \( S \) initialized? If we execute the program (starting in \( \text{ROOT} \)), 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:

\[
\begin{align*}
\text{rel[expr,varname] UNINIT} &= \{ <E,V> | <\text{expr E}, \text{varname V}>: \text{USES}, E\ \text{in}\ \text{reachX}((\text{ROOT}), \text{DEFS}[-,V], \text{PRED}) \}
\end{align*}
\]

We analyze this definition in detail:

- \(<\text{expr E}, \text{varname V}> : \text{USES} \) enumerates all tuples in the \( \text{USES} \) relation. In other words, we consider the use of each variable in turn.

- \( E\ \text{in}\ \text{reachX}((\text{ROOT}), \text{DEFS}[-,V], \text{PRED}) \) is a test that determines whether statement \( S \) is reachable from the \( \text{ROOT} \) without encountering a definition of variable \( V \).

- \( \{\text{ROOT}\} \) represents the initial set of nodes from which all path should start.
• $\text{DEFS}[-,V]$ yields the set of all statements in which a definition of variable $V$ occurs. These nodes form the exclusion set for $\text{reachX}$: no path will be extended beyond an element in this set.

• $\text{PRED}$ is the relation for which the reachability has to be determined.

• The result of $\text{reachX}({\text{ROOT}}, \text{DEFS}[-,V], \text{PRED})$ is a set that contains all nodes that are reachable from the $\text{ROOT}$ (as well as all intermediate nodes on each path).

• Finally, $E$ in $\text{reachX}({\text{ROOT}}, \text{DEFS}[-,V], \text{PRED})$ tests whether expression $E$ can be reached from the $\text{ROOT}$.

• The net effect is that $\text{UNINIT}$ will only contain pairs that satisfy the test just described.

When we execute the resulting Rscript (i.e., the declarations of $\text{ROOT}$, $\text{PRED}$, $\text{DEFS}$, $\text{USES}$ and $\text{UNINIT}$), we get as value for $\text{UNINIT}$:

$\{<\text{5}, \ "q">, \ <\text{6}, \ "y">, \ <\text{10}, \ "z">\}$

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

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

\[
\text{set[\text{var}]} = \text{range(\text{DEFS})} \ \setminus \ \text{range(\text{USES})}
\]

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

### Using Locations to Represent Program Fragments

**Warning**

Fix the following

\begin{verbatim}
\begin{figure}[tb] \begin{center} \epsfig{figure=figs/meta-pico.eps,width=6cm} \hspace*{0.5cm} \epsfig{figure=figs/pico-example.eps,width=6cm} \end{center} \hrulefill \caption{\label{FIG:meta-pico}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. [\textbf{THIS FIGURE IS OUTDATED}] } \end{figure}
\end{verbatim}

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 the section called “Elementary Types and Values” [7], 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 $\text{expr}$ as aliases for $\text{loc}$ (instead of $\text{int}$):

\begin{verbatim}
type expr = loc
type varname = str
\end{verbatim}

Of course, the actual relations are now represented differently. For instance, the $\text{USES}$ relation may now look like

\begin{verbatim}
{ <area-in-file("/home/paulk/example.pico", area(5,5,5,6,106,1)), "q">, <area-in-file("/home/paulk/example.pico",
\end{verbatim}
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:

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

Remains the new definition of UNINIT:

```r
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.

**Warning**

Fix reference

In Figure~\ref{FIG:meta-pico} it is shown how checking of Pico programs in the ASF+SDF Meta-Environment looks like.

### 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 [McC76] 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 statements, 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:

```r
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.
Dataflow Analysis

Dataflow analysis is a program analysis technique that forms the basis for many compiler optimizations. It is described in any textbook on compiler construction, e.g. [ASU86]. The goal of dataflow analysis is to determine the effect of statements on their surroundings. Typical examples are:

• Dominators (the section called “Dominators”[31]): which nodes in the flow dominate the execution of other nodes?

• Reaching definitions (the section called “Reaching Definitions”[33]): which definitions of variables are still valid at each statement?

• Live variables (the section called “Live Variables”[36]): 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.

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 $d$ [ASU86] (Section 10.4). Dominators play a role in the analysis of conditional statements and loops. The function dominators that computes the dominators for a given flow graph $PRED$ and an entry node $ROOT$ is defined as follows:

```plaintext
rel[stat,stat] dominators(rel[stat,stat] PRED, int ROOT) =

where
    set[int] VERTICES = carrier(PRED)

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

endwhere
```

First, the auxiliary set $VERTICES$ (all the statements) is computed. The relation $DOMINATES$ consists of all pairs $<S_i, (S_1, ..., S_n)>$ such that

• $S_i$ is not an initial node or equal to $S$.

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

Consider the flow graph

```plaintext
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>
}
```

It is illustrated in Figure 1.5, “Flow graph” [32]
The result of applying dominators to it is as follows:

```plaintext
{<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 resulting dominator tree is shown in Figure 1.6. “Dominator tree”[32]. The dominator tree has the initial node as root and each node \( d \) in the tree only dominates its descendants in the tree.
Reaching Definitions

We illustrate the calculation of reaching definitions using the example in Figure 1.7, “Flow graph for various dataflow problems” [33] which was inspired by [ASU86] (Example 10.15).

Figure 1.7. Flow graph for various dataflow problems

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

```plaintext
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:

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

The new DEF relation gets as value:

```plaintext
{ <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:

```plaintext
{ <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 =
\{<S_1, <S_2, V>> | <stat S_1, var V> : DEFS,
  <stat S_2, V> : DEFS, 
  S_1 \neq S_2 \}

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:

**Warning**

Fix expression

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

\\[ OUT[S] = \text{DEF}[S] \cup (IN[S] \setminus 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 : \text{predecessor}(S),
             def D : OUT[P]\}
  OUT = \{(S, D) | stat S : STATEMENTS,
             def D : \text{DEF}[S] \cup (IN[S] \setminus 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 1.8, “Reaching definitions for flow graph in Figure 1.7, “Flow graph for various dataflow problems” [35]) the results are as follows (see Figure 1.8, “Reaching definitions for flow graph in Figure 1.7, “Flow graph for various dataflow problems”[35]).

Relation IN has as value:

\[
\begin{align*}
\{ &<1, "i">, <2, "j">, <3, "a">, <5, "j">, <6, "a">, <7, "i"> \} \\
&\{ <1, "i">, <3, "a">, <5, "j"> \} \\
&\{ <1, "i">, <2, "j">, <3, "a">, <5, "j">, <6, "a"> \} \\
&\{ <3, "a">, <4, "i">, <5, "j">, <6, "a"> \}
\end{align*}
\]

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

Relation OUT has as value:

\[
\begin{align*}
\{ &<1, "i">, <3, <1, "i">, <4, <3, "a">, <4, <4, "i">, <5, <3, "a">, <5, <5, "j">, <6, <6, "a">, <7, <7, "i">, <7, <3, "a">, <7, <6, "a">\}
\end{align*}
\]
Figure 1.8, “Reaching definitions for flow graph in Figure 1.7, “Flow graph for various dataflow problems”” [35] 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 the section called “Program Slicing” [37] when defining program slicing. Here is the definition of reaching-definitions:

```plaintext

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


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
```

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:

**Warning**

Fix expression

\[ \{ \text{IN}[S] = \text{USE}[S] \cup \text{OUT}[S] - \text{DEF}[S] \} \]
\[ \{ \text{OUT}[S] = \text{bigcup}_{S \text{ \in \ successor of S}} \text{ IN}[S'] \} \]

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:

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

The results of live variable analysis for our running example are illustrated in Figure 1.9, “Live variables for flow graph in Figure 1.7, “Flow graph for various dataflow problems”” [37].

Figure 1.9. Live variables for flow graph in Figure 1.7, “Flow graph for various dataflow problems” [33]

Program Slicing

Program slicing is a technique proposed by Weiser [Wei84] 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 will illustrate this:

\[
\begin{align*}
\text{[1]} & \quad \text{read(n)} \\
\text{[2]} & \quad i := 1 \\
\text{[3]} & \quad \text{sum} := 0 \\
\text{[4]} & \quad \text{product} := 1 \\
\text{[5]} & \quad \text{while } i \leq n \text{ do} \\
& \quad \text{begin} \\
& \quad \text{[6]} \quad \text{sum} := \text{sum} + i \\
& \quad \text{end} \\
\text{[7]} & \quad \text{product} := \text{sum} \\
\end{align*}
\]
The initial program is given as (a). The slice with statement [9] as slicing criterion is shown in (b): statements [4] and [7] are irrelevant for computing statement [9] and do not occur in the slice. Similarly, (c) shows the slice with statement [10] as slicing criterion. This particular form of slicing is called \textit{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 [Tip95]. Here we will explore a relational formulation of slicing adapted from a proposal in [JR94]. The basic ingredients of the approach are as follows:

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

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

- Compute the relation \textit{rel[use,def] use-def} that relates all uses to their corresponding definitions. The function \textit{reaching-definitions} as shown earlier in the section called \textit{“Reaching Definitions”} [33] does most of the work.
- Compute the relation \textit{rel[def,use] def-use-per-stat} that relates the internal definitions and uses of a statement.
- Compute the relation \textit{rel[def,use] control-dependence} that links all EXECs to the corresponding TESTs.
- Compute the relation \textit{rel[use,def] use-control-def} combines use/def dependencies with control dependencies.
- After these preparations, compute the relation \textit{rel[use,use] USE-USE} that contains dependencies of uses on uses.
- The backward slice for a given slicing criterion (a use) is now simply the projection of \textit{USE-USE} for the slicing criterion.

This informal description of backward slicing can now be expressed in Rscript:
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

Let's apply this to the example from the start of this section 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">, <5, "i">, <7, "product">, <7, "i">, <8, "i">, <9, "sum">, <10, "product">
  }
set[int] CONTROL-STATEMENT = { 5 }
The result of the slice
BackwardSlice(CONTROL-STATEMENT, PRED, USES, DEFS, <9, "sum">)
will then be

\[
\{ <1, \text{"EXEC"}>, <2, \text{"EXEC"}>, <3, \text{"EXEC"}>, <5, \text{"i"}>, <5, \text{"n"}>, <6, \text{"sum"}>, <6, \text{"i"}>, <6, \text{"EXEC"}>, <8, \text{"i"}>, <8, \text{"EXEC"}>, <9, \text{"sum"}> \}
\]

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

## 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 (the section called “Workflow for Fact Extraction” [40]) and give strategies for fact extraction (the section called “Strategies for Fact Extraction” [41]).

**Figure 1.10. Workflow for fact extraction**

![Workflow Diagram](image)

**Workflow for Fact Extraction**

Figure 1.10, “Workflow for fact extraction”[40] 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.

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 \( \text{wc} \) (word and line count), \( \text{grep} \) (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.

Strategies for Fact Extraction

The following global scenario's are available when writing a fact extractor:

- 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 1.11, “Separation-of-Concerns strategy for fact extraction” [42].
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.

**Figure 1.11. Separation-of-Concerns strategy for fact extraction**

---

**Fact Extraction using ASF+SDF**

Although facts can be extracted in many ways, ASF+SDF is the tool of preference to do this. Examples are given In XXX.

**Warning**

Fix reference

**Concluding remarks**

It is not unusual that the effort that is needed to write a fact extractor is much larger than the few lines of Rscript that are sufficient for the further processing of these facts. What can we learn from this observation? First, that even in simple cases 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 questions 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? Is it, for instance, useful to bring some of the syntactic program domains like expressions and statements to the relational domain?
## Table of Built-in Operators

### Table 1.10. Table of Built-in Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Boolean and</td>
<td>[13]</td>
</tr>
<tr>
<td>implies</td>
<td>Boolean implication</td>
<td>[13]</td>
</tr>
<tr>
<td>in</td>
<td>Membership test on sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>inter</td>
<td>Intersection of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>not</td>
<td>Boolean negation</td>
<td>[13]</td>
</tr>
<tr>
<td>notin</td>
<td>Non-membership test on sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>or</td>
<td>Boolean or</td>
<td>[13]</td>
</tr>
<tr>
<td>union</td>
<td>Union of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>==</td>
<td>Equality of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>==</td>
<td>Equality of strings</td>
<td>[13]</td>
</tr>
<tr>
<td>==</td>
<td>Equality of locations</td>
<td>[14]</td>
</tr>
<tr>
<td>==</td>
<td>Equality of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality of strings</td>
<td>[13]</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality of locations</td>
<td>[14]</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal of strings</td>
<td>[13]</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Textual inclusion of locations</td>
<td>[14]</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Subset of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than of strings</td>
<td>[13]</td>
</tr>
<tr>
<td>&lt;</td>
<td>Strict textual inclusion of locations</td>
<td>[14]</td>
</tr>
<tr>
<td>&lt;</td>
<td>Strict subset of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal of integer</td>
<td>[13]</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal of strings</td>
<td>[13]</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Textual containment of locations</td>
<td>[14]</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Superset of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than of strings</td>
<td>[13]</td>
</tr>
<tr>
<td>&gt;</td>
<td>Strict textual containment of locations</td>
<td>[14]</td>
</tr>
<tr>
<td>&gt;</td>
<td>Strict superset of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>+</td>
<td>Addition of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>/</td>
<td>Division of integers</td>
<td>[13]</td>
</tr>
<tr>
<td>\</td>
<td>Difference of sets/relations</td>
<td>[14]</td>
</tr>
<tr>
<td>o</td>
<td>Composition of relations</td>
<td>[15]</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
<td>Section</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>(\times)</td>
<td>Cartesian product of sets</td>
<td>[15]</td>
</tr>
<tr>
<td>#</td>
<td>Number of elements of set/relation</td>
<td>[15]</td>
</tr>
<tr>
<td>([-, ])</td>
<td>Left image of relation</td>
<td>[15]</td>
</tr>
<tr>
<td>([, -])</td>
<td>Right image of relation</td>
<td>[15]</td>
</tr>
<tr>
<td>[ ]</td>
<td>Right image of relation</td>
<td>[15]</td>
</tr>
<tr>
<td>+</td>
<td>Transitive closure of a relation</td>
<td>[15]</td>
</tr>
<tr>
<td>*</td>
<td>Reflexive transitive closure of a relation</td>
<td>[15]</td>
</tr>
</tbody>
</table>

---

### Table of Built-in Functions

**Table 1.11. Table of Built-in Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>average</td>
<td>Average of a set of integers</td>
<td>[24]</td>
</tr>
<tr>
<td>average-domain</td>
<td>Average of first elements of tuples in relation</td>
<td>[24]</td>
</tr>
<tr>
<td>average-range</td>
<td>Average of second elements of tuples in relation</td>
<td>[24]</td>
</tr>
<tr>
<td>begincol</td>
<td>First column of a location</td>
<td>[23]</td>
</tr>
<tr>
<td>beginline</td>
<td>Beginning line of a location</td>
<td>[22]</td>
</tr>
<tr>
<td>bottom</td>
<td>Bottom of a relation</td>
<td>[22]</td>
</tr>
<tr>
<td>carrier</td>
<td>Carrier of a relation</td>
<td>[19]</td>
</tr>
<tr>
<td>carrierR</td>
<td>Carrier restriction of a relation</td>
<td>[20]</td>
</tr>
<tr>
<td>carrierX</td>
<td>Carrier exclusion of a relation</td>
<td>[21]</td>
</tr>
<tr>
<td>compl</td>
<td>Complement of a relation</td>
<td>[18]</td>
</tr>
<tr>
<td>endcol</td>
<td>Last column of a location</td>
<td>[23]</td>
</tr>
<tr>
<td>endline</td>
<td>Ending line of a location</td>
<td>[23]</td>
</tr>
<tr>
<td>filename</td>
<td>File name of a location</td>
<td>[22]</td>
</tr>
<tr>
<td>first</td>
<td>First element of a tuple</td>
<td>[21]</td>
</tr>
<tr>
<td>id</td>
<td>Identity relation</td>
<td>[18]</td>
</tr>
<tr>
<td>inv</td>
<td>Inverse of a relation</td>
<td>[18]</td>
</tr>
<tr>
<td>domain</td>
<td>Domain of a relation</td>
<td>[19]</td>
</tr>
<tr>
<td>domainR</td>
<td>Domain restriction of a relation</td>
<td>[20]</td>
</tr>
<tr>
<td>domainX</td>
<td>Domain exclusion of a relation</td>
<td>[20]</td>
</tr>
<tr>
<td>min</td>
<td>Minimum of a set of integers</td>
<td>[25]</td>
</tr>
<tr>
<td>max</td>
<td>Maximum of a set of integers</td>
<td>[24]</td>
</tr>
<tr>
<td>power0</td>
<td>Powerset of a set</td>
<td>[18]</td>
</tr>
<tr>
<td>power1</td>
<td>Powerset of a set</td>
<td>[19]</td>
</tr>
<tr>
<td>range</td>
<td>Range of a relation</td>
<td>[19]</td>
</tr>
<tr>
<td>rangeR</td>
<td>Range restriction of a relation</td>
<td>[20]</td>
</tr>
<tr>
<td>rangeX</td>
<td>Range exclusion of a relation</td>
<td>[20]</td>
</tr>
<tr>
<td>reachR</td>
<td>Reachability with restriction</td>
<td>[22]</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Section</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>reachX</td>
<td>Reachability with exclusion</td>
<td>[22]</td>
</tr>
<tr>
<td>second</td>
<td>Second element of a tuple</td>
<td>[21]</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of a set of integers</td>
<td>[23]</td>
</tr>
<tr>
<td>sum-domain</td>
<td>Sum of first elements of tuples in relation</td>
<td>[23]</td>
</tr>
<tr>
<td>sum-range</td>
<td>Sum of first elements of tuples in relation</td>
<td>[24]</td>
</tr>
<tr>
<td>top</td>
<td>Top of a relation</td>
<td>[21]</td>
</tr>
<tr>
<td>unique</td>
<td>Deprecated: Set with unique elements</td>
<td>[18]</td>
</tr>
</tbody>
</table>

---

## Bibliography


