Projektarbeit

Towards an Abstract Machine for Xcerpt's Simulation Unification

Michael Brade

12. Dezember 2004

Betreut durch
Prof. Dr. François Bry, Tim Furche und Sebastian Schaffert

Abstract

This thesis presents an initial abstract machine (called SUAM) for Xcerpt's non-standard unification algorithm, the simulation unification. The high-level simulation unification algorithm used in Xcerpt's reference implementation was adapted to allow for an easy to understand implementation at machine level. The result is a purely sequential algorithm that differs quite considerably from the standard prolog unification. The machine language of the SUAM is tailored to the algorithm and comprises some very simple and some higher-level instructions. Most of the important properties of Xcerpt's simulation unification have been implemented and are shown using an extensive example at the end of this paper.

Zusammenfassung

In dieser Projektarbeit wird eine erste abstrakte Maschine (die SUAM) für Xcerpt's Unifikationsalgorithmus, die Simulationsunifikation, vorgestellt. Der high-level Algorithmus für die Simulationsunifikation aus der Referenzimplementierung von Xcerpt wurde angepaßt, um eine einfach zu verstehende Implementierung auf Maschinenebene zu erreichen. Das Resultat ist ein ausschließlich sequentieller Algorithmus, der sich signifikant von der Standard-Prolog-Implementation unterscheidet. Die Maschinensprache der SUAM ist speziell entwickelt für den neuen Algorithmus und besteht aus einigen sehr einfachen und einigen höhersprachigen Befehlen. Die meisten wichtigen Eigenschaften der Simulationsunifikation sind implementiert und werden mit Hilfe von einem ausführlichen Beispiel am Ende dieser Arbeit erläutert.

Acknowledgements

I want to thank Tim Furche for proof-reading this thesis and for all the important and helpful suggestions he has made. Furthermore, Sebastian Schaffert always provided helpful answers to all difficulties I had with Xcerpt.

Contents

Listings

1

Introduction

Nowadays XML query languages play an increasingly important role, especially with regards to the Semantic Web. The Semantic Web is an endeavor to enrich the existing conventional Web with metadata and metadata-processing to allow computer systems to *draw inferences* from the data instead of merely *rendering* them. But to actually be able to use and process data on the Semantic Web in this way it is necessary to provide languages for Web data and metadata retrieval, evaluation, and updating. Existing Semantic Web query languages, such as DQL and TRIPLE, are special purpose languages. They are designed for querying and drawing inferences from special representations such as OWL or RDF, but these languages are not capable of processing generic Web data. The Web query language *Xcerpt*, on the other hand, is a general purpose language that can query, evaluate, update and maintain any kind of XML data, which includes data on the conventional Web as well as the Semantic Web. Likewise, Xcerpt is able to make logical deductions from any kind of XML data.

In this thesis an abstract machine for a core aspect of Xcerpt's evaluation, the simulation unification, is introduced. This abstract machine, called the SUAM, allows to match queries expressed as Xcerpt query patterns to some data represented as Xcerpt data term. Before discussing the details of this approach, the following section motivates the need for an abstract machine for Xcerpt in general and for the simulation unification in particular.

1.1 Motivation

Because abstract machines are intermediate and low-level architectures they can support implementations of many programming languages. An abstract machine (AM) for any language is always a benefit for the following reasons:

- An AM is a way to accurately define and describe the operational semantics of a computer language. Its instruction set is similar to the ingredients of a cooking recipe, the definition of an instruction corresponds to an ingredient, the execution of a program and thus the operational semantics match the description of the recipe, and the result of the program corresponds to what the recipe will yield after the cooking phase.

- Since the AM only provides the abstract structure and not a realization on a concrete physical machine, and since the AM discussed in this thesis is carefully tuned to only use and require the very basic functionalities, implementations on a wide variety of devices and platforms should be possible. Facilitation to port an interpreter to different platforms is naturally provided because the AM well defines the platform-independent part of the interpreter using the pseudo-code and leaves the platform-dependent issues up to the actual implementations.

- Portable code is another advantage: A program can be compiled on one architecture and executed on another one since all interpreters implement the same instruction set.

1.2 Contributions

This thesis's aim is to present the first steps towards a complete abstract machine for Xcerpt, the XcAM. Specifically, it develops an abstract machine for Xcerpt's simulation unification. This thesis is among the first research projects regarding abstract machines for XML query languages in general.

The following abstract machine, the SUAM, represents a formal definition and the operational semantics of Xcerpt's simulation unification. The declarative semantics for Xcerpt and the simulation unification have already been defined, they are to be found in [2].

The SUAM has very simple and standard data structures; some random access memory with insertions and deletions, if at all, only done in a stack-like manner. The instruction set is also quite simple, there are only a few but meaningful instructions with almost no overlap in functionality. The instructions include but are not limited to jumps, comparisons, and memory allocations and deallocations. The coupling between instructions is rather loose to allow for reordering and optimization.

There is one special benefit that is peculiar to the SUAM. It not only allows for several data terms to be queried with one query term translation but also the other way round: It is possible to read and store one data term and then execute several queries on it without having to reparse the data term.

1.3 Scope

Of course, an abstract machine as complex as one that completely covers Xcerpt cannot be developed at once, small steps have to be done properly one after another. Therefore, this thesis is only a part of the path towards the complete XcAM and the following will detail what it covers and what is still open for research.

The current SUAM is designed to unify a query term and a data term, it is not possible to unify two query terms yet. A data term must not contain variables or other special Xcerpt keywords; a data term is always total, and its children can be ordered or unordered and are data terms themselves.

Queries can be partial or total; ordered and unordered query terms are supported, and they can contain variables. The descendant construct is implemented and variable restrictions are possible. The remainder of Xcerpt's keywords and features is unsupported.

Also missing is the support for cyclic data terms, which can be created using references.

It is important to note that the current abstract machine does not do any constraint solving and thus only computes and creates the compact representation of the constraint store, the matrix.

Last, there is no error checking for the correctness of the code of a translated query term yet. The effect of an incorrectly used instruction is implementation dependent.

1.4 Overview

The following is a short overview of the thesis and where to find what.

The following chapter, Chapter 2, is an introduction to Xcerpt and the simulation unification. It also explains a little about a possible implementation of the simulation unification using a matrix.

Chapter 3 sets forth the general theory and the design decisions and issues. The principles of the SUAM are summarized and the important data structures are shown. Finally, the chapter informally introduces the core of the unification algorithm of the SUAM.

Following up the algorithm Chapter 4 defines the translation of a query term into SUAM code in great detail. This chapter is useful for those who wish to implement a parser and translator for query terms.

To understand the individual instructions the SUAM knows about, Chapter 5 shows each of them with an informal explanation, an example, and an (abstract) implementation. If an instance of the SUAM has to be implemented this chapter is the definite reference guide.

Chapter 6 finally comes with an extensive example of how to translate a query term and then shows the changes happening in the SUAM when executing every single instruction on its own. This chapter may give a quick overview of what the SUAM really does.

There is still a lot to be researched and improved. Chapter 7 gives a short outline of what has to and will be done in the future regarding the SUAM.

Chapter 8 summarizes a few of the most important points of this thesis.

2

Preliminaries

2.1 Xcerpt

Xcerpt is a declarative rule-based query language for Web data, i.e. XML documents or semistructured databases. Xcerpt is based on logic programming: An Xcerpt program comprises one or more *goals* and zero or more *rules*. Goals and rules consist of query and construction patterns called *terms* similar to other logic programming languages. Terms are tree- or rooted graph[1]-like structures. The children of a node can either be *ordered* as in standard XML or *unordered* as is common in relational databases. Terms are subdivided into *data terms*, *query terms* and *construct terms*.

Data terms represent XML documents or the data items of a semistructured database in general. They are similar to ground functional programming expressions and logical atoms. A *database* is a (multi)set of data terms, such as the Web, for example.

Query terms are patterns that are matched to Web resources represented by data terms. Query terms are data terms augmented by

- *variables* for selecting data items,
- variables with *variable restrictions* to limit possible bindings to particular subterms,
- *partial term specifications* to omit subterms irrelevant to the query and
- special query constructs for *subterm negation*, *optional subterm specification*, and *descendant specification*.

Construct terms reassemble the variables' bindings that were specified in query terms to create new data terms. Construct terms are data terms augmented by *variables* acting as place holders for data selected in a query, and the *grouping construct* `all` for collecting all instances resulting from different variable bindings.

[1] A "rooted graph" is a graph with a distinguished node called "root".

Construct-query rules, rules for short, relate a construct term to a query consisting of **AND**-and/or **OR**-connected query terms. Rules can be seen as *views* specifying how documents in the form of the construct term can be obtained by evaluating the query on Web resources. Xcerpt rules can be *chained* like active or deductive database rules to form complex query programs.

Examples

The first one is a real-world example that can be easily understood and that shows a fair few of Xcerpt's features: It is an XML document containing an address book. The corresponding *data term* will be:

```
<addressbook>
  <entry>
    <name>Robert Bart</name>
    <phone>
      <home>03529/3170</home>
      <mobile>0162/4576214</mobile>
      <work>03528/723712</work>
    </phone>
    <address>
      23 George Str, Village 3120
    </address>
  </entry>
  <entry>
    <name>Julia Flower</name>
    <mobile>0034-1252-6829</mobile>
    <email>flower@work.com</email>
  </entry>
  <entry>
    <name>Neil Fisher</name>
    <phone>
      <mobile>0174/3421390</mobile>
      <home>01282/709</home>
    </phone>
    <address>
      <street>45 Shorthorn Street</street>
      <postalcode>40307</postalcode>
    </address>
  </entry>
</addressbook>
```

In Xcerpt syntax the first part of the example would look as follows.

```
addressbook [
  entry [
    name [
      Robert Bart
    ]
    phone [
      home [
        03529/3170
      ]
      mobile [
        0162/4576214
      ]
      work [
        03528/723712
      ]
    ]
    address [
      "23␣George␣Str,␣Village␣3120"
    ]
  ]
  ...
]
```

Suppose the intention is to extract the mobile phone numbers and email addresses, if available. Then the query term could be this one:

```
addressbook {{
  entry {{
    desc mobile [ var Mobile ]
    optional email [ var Email ]
  }}
}}
```

The query is partial and unordered (`{{ }}`) because only two elements out of an unspecified number are known and desired and the order of properties in an address book entry usually does not matter. The mobile phone number is a `desc`endant of the <`entry`> element, that is, it needs not to be a direct child of <`entry`>. And finally, the email address is `optional`, for there are address entries that simply do not contain email addresses at all. However, for these entries the mobile phone number still has to be extracted. The `optional` keyword allows that entries without an <`email`> element are matched while still extracting the email for entries that contain one.

To create a new document with all available mobile phone numbers and email addresses, the below construct term can be used, which specifies how to construct the new document:

```
result [
  mobiles [ all var Mobile ]
  email-addresses [ all var Email ]
]
```

If this is put into a complete construct-query rule it constitutes the first working Xcerpt program:

```
GOAL
  result [
    mobiles [
      all var Mobile
    ]
    email-addresses [
      all var Email
    ]
  ]
FROM
  addressbook {{
    entry {{
      desc mobile [ var Mobile ]
      optional email [ var Email ]
    }}
  }}
END
```

The result of this query is the anticipated document:

```
result [
  mobiles [
    "0162/4576214",
    "0034-1252-6829",
    "0174/3421390"
  ]
  email-addresses [
    "flower@work.com"
  ]
]
```

The next example using artificial labels for brevity is used throughout the rest of this thesis. Note that the data term has two children with identical structure and labels:

```
f [
  g [ a, b ],
  g [ a, b ],
  h [ c, d ]
]
```

The query term shall use the unordered specification, the descendant construct, and a variable. Put into a construct-query rule using a very simple construct term, this yields:

```
GOAL
  result [
    third_term [
      var X
    ]
  ]
FROM
  f {
    g { a, b },
    desc b,
    var X
  }
END
```

The result of this program would be

```
result [
  third_term [
    h [ c, d ]
  ]
]
```

Xcerpt heavily relies on term unification to get the desired result. And it is this very unification, the simulation unification, that is this thesis's subject; the next section will examine it in a little more detail.

2.2 Simulation Unification

Simulation Unification is the non-standard unification used by the Xcerpt language and is one of its core concepts.

Xcerpt cannot make use of the standard unification algorithm used by Prolog, for example, because it must be possible to unify two terms with different numbers of children in case of a partial query, two terms with the same children in a different order in case of an unordered

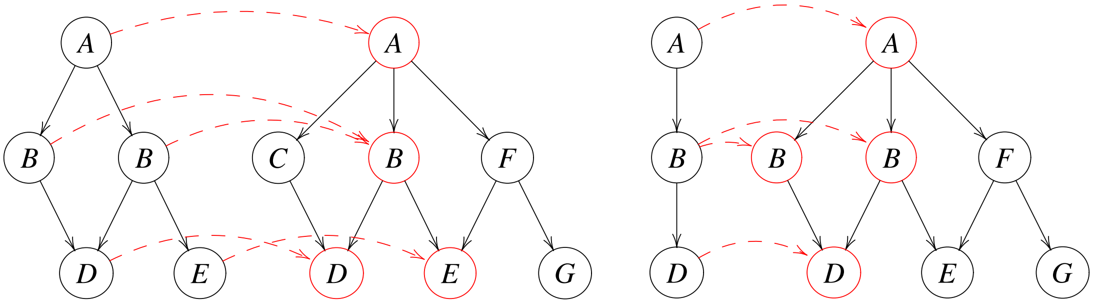


Figure 2.1: Two graph simulations with respect to label equality

query, and two terms where a subterm is nested deeper in one than the other term in case of a descendant query. Moreover, special Xcerpt constructs such as `optional`, `as`, or `not` have to be dealt with.

The Simulation Unification can handle all of these and a few more. It is based on a rooted graph simulation. A directed rooted graph G consists of a set of vertices V, a set of edges E, and a distinguished vertex r called the root of the graph so that in G there is a path from r to every other vertex.

A graph G_1 intuitively simulates in a graph G_2 if the node and edge structure of G_1 exists as a subgraph in G_2. It is important to note that by this definition a node in G_1 can simulate in two or more nodes in G_2 or the other way round if the nodes are equal with respect to label equality; this is not a one-to-one node mapping. Figure 2.1 is an example of two simulations with respect to label equality.

There can be defined restrictions as to which node and edge structure should simulate in a given graph. Xcerpt offers the so-called *subterm specification* that allows to specify whether a whole subgraph has to match exactly, whether only part of a subgraph (either horizontally or vertically) has to match, or whether the order of terms is significant.

This subterm specification is done using different kinds of brackets in the term definition. Table 2.1 on the next page shows the possible subterm specifications and their compatibilities. Specifications in each of the four different groups can possibly simulate successfully if the condition in the third column holds.

The following definition is taken from [2]. A function $f : N \to M$ is called total if f is defined for every element of N, partial otherwise. Given two term sequences $M = \langle s_1, \ldots, s_m \rangle$ and $N = \langle t_1, \ldots, t_n \rangle$. A partial or total mapping $\pi : M \to N$ is called

- *index injective* if $\forall s_i, s_j \in M : index(s_i) \neq index(s_j) \Rightarrow index(\pi(s_i)) \neq index(\pi(s_j))$
- *index monotonic* if $\forall s_i, s_j \in M : index(s_i) < index(s_j) \Rightarrow index(\pi(s_i)) < index(\pi(s_j))$
- *index bijective* if it is index injective and $\forall t_k \in N \, \exists s_i \in M : \pi(s_i) = t_k$

where $index(s_i)$ is the position of the element s_i in the term sequence M or N.

Query Term	Data Term	Condition
$l_1[s_1,\dots,s_m]$	$l_2[t_1,\dots,t_n]$	π is *index bijective* and *index monotonic*
$l_1\{s_1,\dots,s_m\}$	$l_2[t_1,\dots,t_n]$	π is *index bijective*
	$l_2\{t_1,\dots,t_n\}$	π is *index bijective*
$l_1[[s_1,\dots,s_m]]$	$l_2[t_1,\dots,t_n]$	π is *index monotonic*
	$l_2[[t_1,\dots,t_n]]$	π is *index monotonic*
$l_1\{\{s_1,\dots,s_m\}\}$	$l_2[t_1,\dots,t_n]$	
	$l_2\{t_1,\dots,t_n\}$	
	$l_2[[t_1,\dots,t_n]]$	
	$l_2\{\{t_1,\dots,t_n\}\}$	

Table 2.1: Conditions for successful simulation

Example: $f[a,b]$ does not simulate in $f\{a,b\}$, but $f\{a,b\}$ simulates in $f[a,b]$. $f[a,b]$ and $f[b,a]$ do not simulate because there exists no index monotonic mapping π. However, $f\{a,b\}$ and $f[b,a]$ do simulate because no further conditions are imposed on π in this case.

Although graph simulation allows to map two nodes of one graph to one single node of another graph, it is important to note that the simulation unification in this thesis is injective in every case. Injective means that each query subterm is mapped onto a distinct data subterm.

Further information and deeper technical detail on the simulation unification can be found in [1] and in [2]. In the next section a possible way to efficiently implement the simulation unification is discussed.

2.3 Matrix Theory

A very simple and straight-forward way to compute the simulation unification would be to generate a disjunction of all possible combinations of unifications of the subterms. These combinations would then be solved separately. However, this approach is seriously inefficient, there are $m!/(m-n)!$ different total injective mappings from $\{t_1,\dots,t_n\}$ to $\{s_1,\dots,s_m\}$. Thus, $n \cdot m!/(m-n)!$ unification steps would be required and many recursive unifications would be computed on the same pairs of subterms.

A possible solution to this problem is the use of a matrix to store unification results, thus saving the redundant steps.

The idea of the matrix as it was developed by Sebastian Schaffert for the reference implementation[2] in [2] is to not compute all possible sets of variable substitutions at once, thereby possibly evaluating certain variable substitutions several times, but to first evaluate every unique variable substitution once and store it in a matrix. Then the combination of them to the legal

[2]The reference implementation can be found at `http://www.xcerpt.org`.

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	unify($var\ X$, a)	unify($var\ X$, b)	unify($var\ X$, c)	unify($var\ X$, d)
c	unify(c, a)	unify(c, b)	unify(c, c)	unify(c, d)

This matrix will immediately be evaluated to:

$t_1 \backslash t_2$	a	b	c	d
$var\ X$	$var\ X \leq a$	$var\ X \leq b$	$var\ X \leq c$	$var\ X \leq d$
c	$false$	$false$	$true$	$false$

Figure 2.2: Example of a Unification Matrix

sets of substitutions can be done at a later point in time when all intermediate results are known. The number of single variable substitutions is growing linearly with the number of terms and variables, the number of combinations of them is obviously growing exponentially and this way the number of evaluations is kept linear.

A matrix is associated with a pair of terms whose unification is supported by the matrix. It represents a set of one or more constraints (variable substitutions), which are the possible combinations (according to the term specification) to unify two of the direct children of each of the terms to be unified. And each of these possible combinations of substitutions is represented by a different submatrix. If there are no children, the matrix represents the result of the unification of the two terms themselves.

The rows in a matrix correspond to the query terms, the columns correspond to the data terms. The cells contain the substitution(s) that the unification of the term of the matching row and the term of the matching column yields. In case of more than one constraint the cell is a submatrix itself. Therefore, the whole matrix is a matrix of nested matrices.

Since a row represents the different possible solutions for the query term of that row and all rows together form the complete query term, the columns are connected with *OR* and the rows are connected with *AND*.

This optimized approach requires at most $n \cdot m$ unification steps on each level, except for the descendant construct (see the example in Figure 2.3 on the facing page). The exponential result can then be collected later by following so-called paths in the matrix, dependent on the specification of the query term.

Figure 2.2 is an example for the matrix that would be created when unifying the two terms $t_1 := f\{\{var\ X, c\}\}$ and $t_2 := f\{a, b, c, d\}$.

A more complex example with submatrices can be found in Figure 2.3 on the facing page. The matrix would be the result of the unification of $t_1 := f\{\{desc\ a\{\{desc\ b\{\{desc\ c\}\}\}\}\}\}$ and

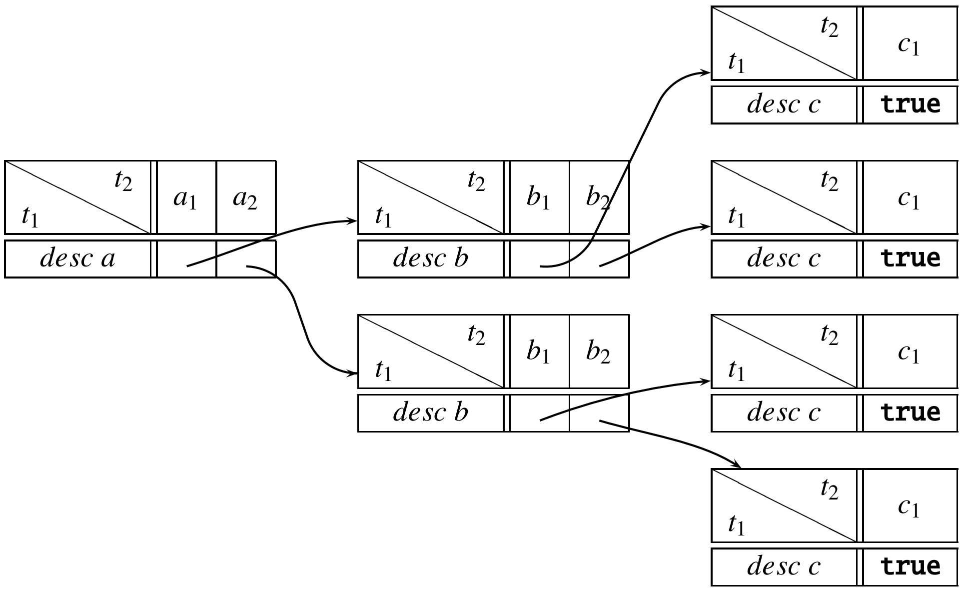


Figure 2.3: Example of a complex Unification Matrix

$t_2 := f\{a_1\{a_2\{b_1\{b_2\{c\}\}\}\}\}$. Here can be seen the exponential time requirement for the descendant construct: e.g. at level 3 in the query term (*desc c*) not only $n \cdot m = 1$ steps, but rather

$$nd_{level\ 1} \cdot nd_{level\ 2} \cdot nd_{level\ 3} \approx O(|DT|^{|QT|})$$

steps are needed, with $nd_{level\ x}$ being the number of matching descendants on level x.

3

Architecture of the Abstract Machine

In this chapter the theoretical aspects of the abstract machine for Xcerpt's simulation unification, the SUAM, will be explained. The SUAM is a part of and the first step towards the complete abstract machine for Xcerpt, the XcAM.

Following an overview of the general principles of simulation unification at machine level, the architecture and data structures of the SUAM are presented. The chapter finally demonstrates the detailed process of unification in the abstract machine.

3.1 Principles of the Abstract Machine

The following will examine the "big picture" of the SUAM and list its design principles.

First, *the data term must not be changed during a query*. It is in other words read-only and none of the SUAM instructions writes, updates or deletes anything in the data term memory area. This is the requirement to be able to execute several queries on one data term without having to reparse it.

Second, wherever possible, *an instruction should only access the next element of the data term* in the order of a left-most, pre-order, depth-first search. No instruction must require random access to the data term. The goal of this principle is to ease the needed work to allow for stream processing of XML documents in the future.

The reason stream processing does not work in the current SUAM is that a single element of the data term can have more than one unification possibility, except for ordered and total query terms. The unification algorithm in this thesis completely determines one matrix cell including all its submatrices before advancing to the next cell, which results in the corresponding data terms to be read more than once. Moreover, the arity of the elements in the data term is needed to construct the matrix, which also requires to read all of an element's children.

Third, the goal of this abstract machine is to *create the constraint store in form of a matrix*. The matrix is created on the heap, therefore the only memory the SUAM will allocate is memory on the heap. Since the matrix only grows, no garbage collection algorithm is needed at this point.

It is probable that garbage collection will be indispensable in the future, should it be possible to implement matrix compacting; see Chapter 7 for some further information.

And last, once the unification process is completed *the machine has to be left in a clean state*. The stack is left with only one cell pointing to the created matrix and the heap is left with just the matrix. In principle the abstract machine is ready to process yet another query.

3.2 Data Structures and Memory Allocation

An abstract machine is similar to a real computing machine: It has an instruction set, a memory, and manipulates memory areas at runtime. The SUAM consists of three registers, a stack, a heap, and a code segment.

3.2.1 Stack

The stack is necessary to keep track of the position in the matrix hierarchy to which the current unification result has to be written and also of the path that leads to this point to be able to backtrack one level; this is very similar to the way a stack-based depth-first search in a tree would be implemented. The stack hence pulsates with the depth of the data term that is currently processed.

Moreover, the stack is used to store the corresponding data term of each matrix, which is stored in the cell immediately following the cell referencing the matrix. It can be seen as the data term that is responsible for the creation of the submatrix.

Read- and write access to the stack is granted at any position and at any time. Additional memory is allocated at the top of the stack only; similarly, only the last cell on the top of the stack, referenced by SP, can be removed.

The stack cells contain only pointers to cells on the heap with the exception being the descendant processing which adds pointers to the stack itself.

3.2.2 Heap

The heap is exclusively used to store the constraint store in form of the matrix. Only the matrix itself, including submatrices and pointers to submatrices, will ever end up on the heap.

Since the matrix will only grow and never shrink, no garbage collection or memory management algorithms are needed for the heap. Memory is allocated only at "the end" of the heap and is never freed again. If optimization techniques such as matrix compacting are introduced this will have to be changed (cf. Chapter 7).

The heap is obviously required because that is the memory where the result, the matrix, will be stored. The matrix cannot be stored on the stack because in case of the descendant evaluation parts of the matrix are built on the stack *and* on the heap *at the same time*. In other words, two separately growing memory areas are needed. Also, the way the memory is and will be managed for the matrix is not possible using a stack. For instance, on a stack only the last cell can be removed without significant overhead.

3.2.3 Registers SP (Stack Pointer), DP (Descendant Pointer), and PC (Program Counter)

The stack pointer SP contains the address of the top allocated cell of the stack, the next available empty cell is at SP + 1. Please note that the illustrations in the following chapters will show the top of the stack at the bottom and the stack therefore grows down.

The descendant pointer DP points to the most recent descendant frame on the stack or contains the value 0 if there is no such frame.

The address of the instruction that is to be executed next is stored in the program counter PC.

3.3 Main Cycle

Of course the SUAM needs a main routine that controls its operation. The idea is to execute the instruction at PC in the code memory and then increment the program counter to point to the next instruction. Every instruction in the SUAM takes up exactly one memory cell so that the PC register can be incremented in steps of one.

Because branch instructions change the program counter, however, it is necessary to first increment the program counter and then execute the instruction at the old value. This prevents the main loop from incrementing the newly set value of the program counter too early which would consequently skip the first instruction at the jump destination. Thus, the main cycle of the abstract machine looks as follows:

```
main()
{
    do
    {
        PC = PC + 1;
        execute instruction at code[PC - 1];
    }
}
```

At the beginning the PC register is initialized with 0 and the program starts at code[0].

3.4 Representation of and Access to the Data Term

The abstract machine tries to store the data term with as little additional information as possible to not deviate too much from a stream representation. The only real extension is the attached number and order of children for each term.

label
number of children n
order of children
child 1
child 2
$\vdots$
child n
`null`

Figure 3.1: Illustration of the Data Term Format

The data term is stored at the very beginning of the heap, it is read-only and it will never be modified.

The format of the data term is illustrated in Figure 3.1: The first three cells comprise the header where the first cell contains the term's label, the second one the number of children, and the third one the order of the children. Following this header information there are the children, each of them also in the data term format, and as a delimiter for the end of a term there is a cell containing **`null`**.

The data term can be accessed at any position using pointers to the heap.

3.5 Representation of the Matrix

The matrix is built on the heap. Since a matrix is a 2-dimensional structure, it has to be linearized. In case of the SUAM the matrix is stored column by column, i.e. first all cells of the first column, then all cells of the second column, and so on, to create the consecutive cells on the heap representing the matrix. This way of linearizing the matrix stems from the fact that columns represent the data terms: Only once the evaluation for one data term is completed the next one is touched, which means the data term has been unified with all possible candidates in the query term—the rows.

A matrix has a header of three cells to store the number of rows and columns, and the order specification of the query term that created the matrix.

Matrix cells can contain the values **`true`** and **`false`** or a pointer to other cells on the heap. These pointers are either temporary pointers to a data term or pointers to another submatrix, the latter case creating a parent-child relation between two matrices.

See Figure 3.2 on the following page for an illustration of the format of a matrix.

number of rows m
number of columns n
order of query term
row 1, column 1
row 2, column 1
⋮
row m, column 1
row 1, column 2
⋮
row m, column 2
⋮
row 1, column n
⋮
row m, column n

Figure 3.2: Illustration of the Matrix Format

variable name
variable binding
indicator if variable binding (in)valid

Figure 3.3: Illustration of the Variable Structure

3.6 Representation of a Variable

The whole purpose of the SUAM and the simulation unification is to create variable bindings so that the query term and the data term simulate.

Each variable binding is represented by a variable structure with three cells on the heap. The first cell contains the variable name, the second cell contains a reference to the data term this variable is bound to, and the third cell holds the result of the unification of the variable's restriction with the data term.

See Figure 3.3 for an illustration of the format of a variable structure.

3.7 Overview of the Unification Algorithm in the SUAM

The SUAM will traverse the data term tree similar to a left-most, pre-order, depth-first search. However, if the children of a query subterm are unordered each of them can potentially be unified with each of the children on the corresponding level in the data term. Because of this some branches in the data term tree may have to be read more than once.

For example, for each element in the data term all of the siblings of the corresponding element in the query term on the corresponding level, including the element itself, will be checked for possible unification. The result of this unification is written to the matrix. With each unification step the matrix will be further extended and completed.

3.7.1 Unification of Nodes

To unify a query term node and a data term node a reference to the data term node is put on the stack first. To remember where to store the result the SUAM creates an indirection from the stack to the data term via the cell in the matrix where the result is to be stored. The result of a unification can be a new submatrix, the value **false**, or the value **true**.

Recall that two nodes match if their labels are the same, their order is compatible, the number of children of the query term is less or equal to the number of children in the data term, and the children of the nodes match.

So the SUAM first checks if the label of the node of the data term matches the label of the node of the query term by following the pointer from the top of the stack to the matrix to the data term. Then the same is done for the number of children and the order specification of the children. If either of the tests fails the SUAM writes **false** to the referenced matrix cell.

If all tests succeed it depends on the number of children what is to be done next. If the query term has no children the unification succeeded and **true** is written to the matrix cell referenced by the top of the stack. If the query term has one or more children, a new submatrix is created and the possible combinations of the children of each of the two terms are unified.

3.7.2 Finding Descendants

The descendant construct's semantics is to find *all* descendants of the current data term that match the descendant argument term and connect them with OR. The latter is because, for example, the query *var X as desc author* has to yield several variable bindings, each of them containing one of the *author* descendants of the current data term. The variables are therefore bound to a complete subtree of the data term. So the query has several separate and alternative solutions, hence the solutions are ORed together.

The idea to implement the above is to create a one-dimensional matrix with just one row and several columns. Recall that the columns in the matrix used by the SUAM are connected with *OR* and the rows are connected with *AND*. Thus, a one-dimensional matrix serves as a list of possible alternative solutions.

In general if a query term specifies a descendant all of the descendants of the corresponding data term have to be checked for unification. The SUAM has an optimization to only check descendants that at least have the same label. This optimization can of course only be performed if the argument of the descendant construct is not just a variable declaration.

The evaluation of the descendant construct is a little more complicated than the rest of the unification. The problem is that the number of descendants is not known from the beginning on and hence it is not possible to allocate an appropriate number of heap cells for the new matrix that will contain the (pointers to the) results of the descendant unification until the complete data term has been traversed.

However, while traversing the data term each encountered subterm that passed the label test has to be evaluated further. But this evaluation will then create new matrices, just as any other unification, and those matrices will end up on the heap. Therefore it is not even possible to allocate the descendant matrix cell by cell on the heap, for these cells would not be guaranteed to be consecutive.

The solution is to collect the growing contents of the descendant matrix on the stack while traversing the data term. All further matrices are created on the heap, as usual, but the pointers to them are stored on the stack. Since after completing a matrix there will be left on the stack no cells but the reference to the matrix itself the stored pointers on the stack are consecutive.

Two things are left to think about: The SUAM has to remember where the stack cells with the contents of the descendant matrix begin. And it must be possible to have nested descendant constructs.

Introducing a stack frame, called the descendant frame, solves both of these issues. The descendant pointer `DP` points to the start of the descendant frame with the contents of the descendant matrix. The descendant frame furthermore holds a backup of the old value of the descendant pointer and a reference to the data term from where to continue descending. A third cell stores the depth relative to the original data term to stop once the whole data term has been traversed.

This frame makes it possible to have nested descendant constructs since the whole frame is removed from the stack once the data term has been fully traversed: First, a new matrix is created on the heap with the size being equal to the number of cells in the descendant frame. Then the cells of the frame are copied into the matrix, the descendant pointer `DP` is set to the value stored in the frame and the frame is removed from the stack. Only one cell with a pointer to the newly created matrix remains on the stack. Therefore, a descendant frame can be used inside another descendant frame.

An example of how a descendant frame works can be found in Chapter 6.

3.7.3 Unification of Variables

Unification of variables and variable restrictions is done with one single structure comprising three cells on the heap. The first cell contains the variable name, the second cell contains a pointer to the data term that is bound to this variable, and the third cell is used to check that the variable binding fulfills the variable restriction, so it contains a pointer to just the same data term to which the variable is bound.

The verification of the variable restriction is just another unification—if the unification succeeds, the variable is bound to the data term in the second cell of the variable structure; if the unification fails, the variable is not bound to anything and the whole variable structure stands for the value **`false`**.

In case no variable restriction was given the third cell in the variable structure is replaced with `true` immediately, indicating a valid variable binding.

For each possible variable binding a new variable structure is created in the matrix on the heap. This means that there can be more than one instance of one and the same variable where each instance is bound to a different data term. Conflict resolution of variable bindings is not done in the SUAM yet.

4

Query Translation

This chapter will explain the translation of a query term into SUAM code. It defines a (recursive) translation function T that takes a query term and generates the corresponding SUAM code fragment:

$$T : \langle \textit{query-term} \rangle \longrightarrow \text{SUAM code fragment}$$

The formal definition of each of the instructions can be found in Chapter 5. The query terms that the translation function T can translate adhere to the following grammar:

⟨*query-term*⟩	::=	⟨*named-node*⟩
	|	⟨*desc-node*⟩
	|	⟨*var-node*⟩
⟨*named-node*⟩	::=	⟨*label*⟩ ⟨*child-spec*⟩
⟨*label*⟩	::=	⟨*string*⟩
⟨*desc-node*⟩	::=	'`desc`' (⟨*named-node*⟩ | ⟨*var-node*⟩)
⟨*var-node*⟩	::=	'`var`' ⟨*var-name*⟩ ['`as`' ⟨*var-restriction*⟩]
⟨*var-name*⟩	::=	⟨*string*⟩
⟨*var-restriction*⟩	::=	⟨*query-term*⟩
⟨*child-spec*⟩	::=	⟨*ordered-total-list*⟩
	|	⟨*unordered-total-list*⟩
	|	⟨*ordered-partial-list*⟩
	|	⟨*unordered-partial-list*⟩
⟨*ordered-total-list*⟩	::=	'`[`' [⟨*children*⟩] '`]`'

$\langle$*unordered-total-list*$\rangle$ ::= '`{`' [$\langle$*children*$\rangle$] '`}`'

$\langle$*ordered-partial-list*$\rangle$ ::= '`[[`' [$\langle$*children*$\rangle$] '`]]`'

$\langle$*unordered-partial-list*$\rangle$ ::= '`{{`' [$\langle$*children*$\rangle$] '`}}`'

$\langle$*children*$\rangle$ ::= $\langle$*query-term*$\rangle$ ('`,`' $\langle$*query-term*$\rangle$)*

Although the translation function may suggest that a linear, one-pass translation is possible a two-pass translation or a similar fix-up operation is required because the branch instructions take code position numbers that are not available at the time when they are generated.

4.1 Program Skeleton

Every SUAM program begins with the `init` instruction and ends with the `halt` instruction. The translation of the (root of) the query term becomes the main part of the program:

```
init

T[[⟨query-term⟩]]

halt
```

Listing 4.1: Program Skeleton

The `init` instruction sets the SUAM up for unification by reading, parsing, and storing the data term and by making the first stack cell indirectly point to it. Indirectly, because the stack cell has to point to a heap cell that contains the address of the data term.

`halt` is currently only a placeholder, the SUAM is not in the need of any post-unification clean-up. And because it should be possible to do several queries on one data term, `halt` does not initiate the shutdown of the abstract machine instance.

The remainder of this chapter defines the translation function T.

4.2 Named Nodes

The translation of a named node is twofold and consists of the translation of the node's label and the child specification[1], together referred to as the node's properties, and the translation of its children:

[1]The "child specification" is a specification of how to interpret the children of a node, e.g. whether all children or only a partial set of children is specified and whether the order of the children is significant.

$$T[\![\langle \textit{named-node} \rangle]\!] = T[\![\langle \textit{label} \rangle]\!]$$
$$T[\![\langle \textit{child-spec} \rangle]\!]$$

Listing 4.2: Named Node Translation

The translation of the children is part of the translation of the child specification. The following two sections explain each of the two named node translation phases.

4.2.1 Property Translation

The property translation makes sure that the data term node with which to unify the current query term node has compatible properties. Please see Section 2.2 on page 15 for a detailed explanation of the unification requirements.

The label is translated as follows:

$$T[\![\langle \textit{label} \rangle]\!] = \texttt{check\_label}\ \langle \textit{string} \rangle$$

Listing 4.3: Property Translation: The Label

The following code fragments are created by the translation function depending on the node type that is to be translated. The first of these is the unordered and total specification of the children:

$$T[\![\text{‘}\texttt{\{}\text{’}\ \langle \textit{children} \rangle\ \text{‘}\texttt{\}}\text{’}]\!] = \texttt{check\_total}\ \ \textit{length}(\langle \textit{children} \rangle)$$
$$T[\![\langle \textit{children} \rangle]\!](\texttt{unordered})$$

Listing 4.4: Property Translation: Unordered Specification

This means that the number of children have to be the same, otherwise the unification will fail.

If the node has only a partial list of children, i.e. the node is $label\{\{c_1, c_2, \ldots, c_n\}\}$, then the only difference to the above case is that the number of children in the data term node has to be at least the number of children in the query term node, reflected by the `check_partial` instruction:

$$T[\![\text{‘}\texttt{\{\{}\text{’}\ \langle \textit{children} \rangle\ \text{‘}\texttt{\}\}}\text{’}]\!] = \texttt{check\_partial}\ \ \textit{length}(\langle \textit{children} \rangle)$$
$$T[\![\langle \textit{children} \rangle]\!](\texttt{unordered})$$

Listing 4.5: Property Translation: Unordered Partial Specification

In both cases so far it was irrelevant if the data term node is ordered or unordered. However, if the query term node requires a specific order of its children the data term node has to be ordered as well and an additional check has to be added.

The first of the two possible ordered specifications is the total query, i.e. the node that generates the code below is $label[c_1, c_2, \ldots, c_n]$.

$T[\![$‘`[`’ $\langle children\rangle$ ‘`]`’$]\!]$ = `check_ordered`
`check_total` $length(\langle children\rangle)$
$T[\![\langle children\rangle]\!]$(`ordered`)

Listing 4.6: Property Translation: Ordered Specification

And the same goes for the partial child specification where the query term node is $label[[c_1, c_2, \ldots, c_n]]$.

$T[\![$‘`[[`’ $\langle children\rangle$ ‘`]]`’$]\!]$ = `check_ordered`
`check_partial` $length(\langle children\rangle)$
$T[\![\langle children\rangle]\!]$(`ordered`)

Listing 4.7: Property Translation: Ordered Partial Specification

As can be seen, in both cases the only difference to the unordered specification is the added `check_ordered` instruction.

4.2.2 Child Translation

The translation of a node's children ensures the creation of the submatrix and initiates the unification of the children themselves. It also ends a unification branch if the node has no children at all.

Leaves—Nodes without Children

This is the simple case. There are no children, hence the only thing to be done is to check if the previous property checking succeeded. This is done by the `check_successful` instruction:

$T[\![$‘`[]`’ | ‘`{}`’ | ‘`[[]]`’ | ‘`{{}}`’$]\!]$ = `check_successful`

Listing 4.8: Child Translation for Leaves

Inner Nodes—Nodes with Children

When a node has at least one child a new submatrix needs to be created and each of the children in the query term node has to be unified with each of the children in the data term node. The results of these unifications are stored in the corresponding matrix cells.

To get all possible unification combinations of the query term node's and data term node's children a loop is created which iterates over all children of the data term node. In every iteration the current data term child is unified with all children of the query term node. So the translation of all the children of the query term node has to be inserted in this loop.

```
   T⟦⟨children⟩⟧(order) = branch_if_null FAIL
2                         create_matrix length(⟨children⟩) order

4         CHILDREN:       push_child
                          T⟦⟨children⟩[1]⟧
6                         pop_entries 1
                          ⋮
8                         push_child
                          T⟦⟨children⟩[length(⟨children⟩)]⟧
10                        pop_entries 1

12                        next_child
                          branch_if_not_null CHILDREN
14
                          pop_entries 2
16        FAIL:
```

Listing 4.9: Child Translation for Inner Nodes

In this code the notation "⟨*children*⟩[*n*]" refers to the *n*th element of the ⟨*children*⟩ non-terminal.

Since the property checking has just been done when this code is executed the first thing to verify is if it succeeded. This is done in line 1, and when the nodes turn out not to have matched the whole child unification is skipped by jumping directly to the label `FAIL`.

Before the children are going to be evaluated the matrix has to be created in line 2. The number of children in the query term is passed as first argument, the order of the children as second argument.

All that the `push_child` instruction right before the code for each of the children does is take the reference to the data term from the top of the stack and copy it into the next empty matrix cell. A new cell with a pointer to the afore mentioned matrix cell is added to the top of the stack. It serves as an indirect pointer to the data term that is to be evaluated by the following code, carrying with it the destination of where to store the unification result: the matrix cell.

The `pop_entries` instructions after the translation of a child remove the reference to the result of the unification from the stack. Once these lines are reached the unification of the subterm is completed and the corresponding matrix is completely filled or the corresponding matrix cell contains either **`true`** or **`false`**. This result is thus not needed anymore and the direct reference to it from the stack can be discarded. Of course, not all reference to it is deleted, the parent matrix will still point to it and the first cell on the stack still points to the root matrix.

With this something fairly important should become clear: After the SUAM code for a node in the query term including `push_child` and `pop_entries 1` has been executed, the stack looks just the same as before its execution.

Now having said that, line 15 is the logical consequence. Since the translation of a node's children creates a new matrix, which adds two stack cells, the two additional cells need to be removed by the child unification code again so that the stack is left with the same amount of cells that it had before the execution of the child unification code above. Moreover, after all children have been evaluated and the matrix is completed the additional information in those two stack cells is not needed anymore and can indeed be discarded.

The `next_child` instruction in line 12 changes the pointer on top of the stack to point to the next child of the data term that corresponds to the matrix, which means that the pointer will then reference the next sibling of the data term that had been referenced by it before the call of `next_child`. If there are no further siblings `next_child` puts **`null`** on the top of the stack and the branch in line 13 will not be triggered. Otherwise the execution continues after the branch at the label `CHILDREN` and pushes the new child onto the stack.

It has to be noted that the labels in line 4 and line 16 are code position numbers in the real translation. The labels are there to support better understanding.

There is a possible optimization: In case the query term node is ordered and total no matrix but a simple list is enough and the loop in the code is not needed. This optimization has not been implemented yet, see Chapter 7 on page 77 for some more information.

4.3 Descendants: desc

First of all, note that the argument term of the descendant construct '`desc`' ⟨*desc-arg*⟩ is really a normal query term. This especially means that it can contain further descendant constructs or variable declarations.

The translation of the descendant construct is almost independent of the term it refers to, the generated code is always the same except for the `next_desc` command. The argument term of the descendant construct is translated like any other term and inserted at the indicated position in the code below.

```
T[[⟨desc-node⟩]] = create_desc_frame

           DESC: push_desc
                 T[[⟨desc-arg⟩]]
                 pop_entries 1

                 next_desc [⟨label⟩]
                 branch_if_not_null DESC

                 create_desc_matrix
```

Listing 4.10: Descendant Translation

The argument of `next_desc` is optional, it is an optimization to skip data terms that are known not to match. It is only inserted if the non-terminal ⟨*desc-arg*⟩ is of the following form:

⟨*desc-arg*⟩ ::= ⟨*label*⟩ ⟨*child-spec*⟩ | '`var`' ⟨*var-name*⟩ '`as`' ⟨*label*⟩ ⟨*child-spec*⟩

In all other cases `next_desc` has no argument.

An already mentioned problem is the fact that the SUAM does not know *how many* descendants there will be and consequently does not know how big the matrix is going to be. The heap of the SUAM has no support for dynamically growing matrices, therefore the contents of the final matrix have to be put temporarily on the stack.

To remember which entries on the stack are the results of a descendant unification a frame on the stack is used to collect the results. It is created by `create_desc_frame`, which saves the register with the old descendant pointer (`DP`) and changes it to point to the new frame, stores the start and current position of the descendant search and remembers the level of the current position. The actual results of the unifications of the argument term with the potential descendants found by `next_desc` are appended to the frame each time a unification is completed, and with each result the frame grows by one cell.

Here becomes important the earlier mentioned property of the SUAM that every completed unification, particularly the unification done by the code inserted in line 4, leaves the stack with no additional cells: It is indispensable to have the end of the descendant frame on top of the stack at the time of the call of `push_desc` so that no gaps emerge and the results in the frame stay contiguous.

It is possible to write the unification result, or a pointer to a newly created matrix with the results to the stack due to the nature of `push_desc`: It still creates an indirection to the data term, the descendant found by `next_desc`, but in contrast to `push_child` the middle cell is not on the heap but on the stack. `push_desc` therefore adds two stack cells, the first one being the matrix cell in the frame, the second one being the starting point for all other instructions.

Note that `create_desc_frame` does not check for the first term to really have the label *label*. Thus, the first call to `push_desc` in line 3 possibly takes a data term that does not match the `desc` argument at all. This is nevertheless no problem, for the translation of the argument term will include the label check. It is in fact an advantage because for data terms that do not contain a matching descendant term no special treatment is needed—line 3 is executed in any case and makes sure that the descendant frame contains *at least* one entry, saving an additional check for an empty descendant frame in `create_desc_matrix`.

However, the `next_desc` command looks for the next child that matches the label *label* (unless none is given) and therefore the label check that is done during the property checking of the argument term's translation is redundant from the second time onwards. This may be optimized in the future.

4.4 Variables and Variable Restrictions: var and as

The translation of variable declarations such as `var` *name* in the query term creates the following short code fragment:

$T[\![$'`var`' $\langle$*var-name*$\rangle$ $]\!]$ = `create_variable` $\langle$*var-name*$\rangle$
`check_successful`

Listing 4.11: Variable Translation

Variable declarations with a restriction are translated to:

$T[\![$'`var`' $\langle$*var-name*$\rangle$ '`as`' $\langle$*var-restriction*$\rangle]\!]$ = `create_variable` $\langle$*var-name*$\rangle$
$T[\![\langle$*var-restriction*$\rangle]\!]$

Listing 4.12: Translation of Variables with a Restriction

The data term that should be bound to the variable is referenced indirectly by the top of the stack at the time of the `create_variable` call.

The variable translation is based upon the idea to put into the SUAM variable structure two pointers, each of them pointing to the data term that will be the content of the variable. The first pointer represents the variable binding itself and this is the one that is dereferenced when reading the variable. The second pointer is used to verify the variable restriction by unifying the variable restriction with the variable's contents.

The separate second pointer is needed for the verification because the unification process modifies the pointer itself and changes it to point to the resulting matrix. If there was only one pointer in the variable structure, the variable binding itself would be lost after the verification of the variable restriction.

When reading a variable, the second pointer has to point to a matrix that evaluates to **`true`**, otherwise the variable binding is invalid and considered non-existent.

The `create_variable` instruction takes a reference to the second pointer of the variable structure and pushes it onto the stack, thereby creating a stack configuration identical to the one created by `push_child`. Then the variable restriction can be unified just as any data term is unified in the usual case.

When comparing the unification of a variable restriction to the unification of a normal query term it should be noted that by adding the '`var`' $\langle$*var-name*$\rangle$ '`as`' part in front of a data term the only addition to the generated code is the `create_variable` instruction. There is absolutely no change in the unification process itself.

5

Instruction Set of the Abstract Machine

In this chapter all instructions the SUAM understands will be presented and explained. For each of the instructions there is a section describing syntax, semantics, and implementation, respectively. An example is provided when appropriate.

Since this chapter is a full specification of all instructions and therefore of rather technical nature. See Chapter 6 on page 61 for a step-by-step example with detailed references to the instructions.

The code used to present the implementations of the instructions is C-style pseudo code. It uses == for the equality test, !== for the inequality test, and = as an assignment operator.

Important to note is that the **and** and **or** tests are short circuiting and therefore not commutative. This makes it possible to test a pointer to be non-**null** and then use it in the same **if**-clause for further tests. Here is an example:

```
if ( pointer !== null and heap[pointer] )
{
    // do stuff
}
```

If `pointer` is **null** the whole test can never evaluate to **true**, therefore the second statement is not executed anymore and can never result in an illegal memory access.

true, **false**, and **null** are architecture dependent constants to be defined in the abstract machine by the actual implementations.

The functions in the code below are without type definitions, this is left up to the implementations. If a function does not return a value or does not take an argument these are simply left out. If a function takes an optional argument, then this argument is marked with a "?".

5.1 SUAM-Internal Functions

The instruction implementations in this chapter use the following special functions to shorten the code.

access_dt is used to obtain the heap address of the data term indirectly referenced by the stack at position stack_pos.

```
heap_address access_dt( stack_pos )
{
    if ( HEAP_ADDRESS(stack[stack_pos]) )
        return heap[stack[stack_pos]]
    else
        return stack[stack[stack_pos]]
}
```

The function HEAP_ADDRESS() is architecture dependent and returns **true** if the given address is on the heap, **false** if it is a stack address.

The next instruction is used to replace the direct reference to the data term in the matrix with a certain value given as argument res. The heap cell in question is referenced by the stack at position stack_pos.

```
result_to_matrix( stack_pos, res )
{
    if ( HEAP_ADDRESS(stack[stack_pos]) )
        heap[stack[stack_pos]] = res;
    else
        stack[stack[stack_pos]] = res;
}
```

And last, SUAM implementations must provide an architecture dependent heap_alloc call. It should allocate the given amount of contiguous memory on the heap and return a pointer to the first cell of it.

5.2 Initialization and Clean-Up

5.2.1 init

Syntax

`init`

Semantics

Precondition: None.

Effect: The purpose of this instruction is to read and store the data term, and to set up the stack and the heap for the first unification instruction.

This means the data term has to be parsed and stored on the heap, an additional heap cell has to be allocated that points to the root of the data term, and the first stack cell has to point to the mentioned heap cell. This indirection using an additional heap cell, which will be discarded later anyway, is needed for the following instructions to work without a condition if used for the first time.

Note that if the abstract machine is supposed to evaluate two different queries on the same data term the second program must not start with the `init` instruction, otherwise the data term needs to be reread.

Postcondition: All registers are reset. The stack pointer `SP` points to the one and only cell on the stack, which contains the address of a heap cell that points to the root of the data term, also sitting on the heap.

Example

This is an example of what the state needs to be after the execution of the `init` instruction when reading the data term `f[ g[a,b], g[a,b], h[c,d] ]`.

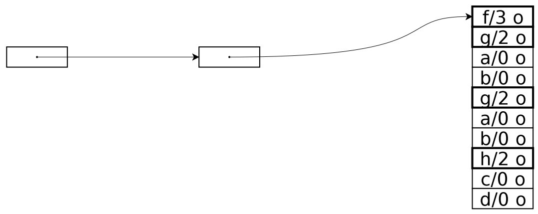


Figure 5.1: Example of the SUAM state after execution of the `init` instruction

Implementation

Since part of the tasks of this instruction are architecture and environment dependent, most notably the parsing and loading of the data term and the instructions, only a partial implementation will be given.

```
init()
{
    SP = 0;   // reset registers
    DP = 0;
    PC = 0;

    // allocate dummy heap cell to create the indirection to the data term
    stack[SP] = heap_alloc( 1 );

    read_parse_and_store_data_term();
    read_parse_and_store_instructions();
    result_to_matrix( SP, address_of_data_term_root() );
}
```

Listing 5.1: `init`

Different means for reading the data term and writing the resulting matrix are likely to be provided, e.g. files, URLs, or streams. However, specification of these is not within the scope of this thesis.

5.2.2 halt

Syntax

`halt`

Semantics

`halt` it is supposed to do the cleanup in the abstract machine after a unification has been completed. However, currently the instruction is a no-op. No cleanup is needed in the SUAM and it is not desired to formally stop the machine at this point, for its output (the unification result) will most probably be processed further. Also, it should still be possible to run several query programs without having to restart the machine.

Implementation

```
halt()
{ }
```

Listing 5.2: `halt`

5.3 Branch Instructions

The SUAM uses standard branch instructions that about any other abstract machine implements as well. They are used for conditional jumps and loops by modifying the program counter PC.

5.3.1 branch_if_null

Syntax

`branch_if_null` *`address`*

Semantics

If the top of the stack contains **`null`** `branch_if_null` sets the program counter PC to *`address`*.

Implementation

```
branch_if_null( address )
{
    if ( stack[SP] == null )
    {
        PC = address;
    }
}
```

Listing 5.3: `branch_if_null`

5.3.2 branch_if_not_null

Syntax

`branch_if_not_null` *`address`*

Semantics

If the top of the stack contains anything but **`null`** `branch_if_not_null` sets the program counter PC to *`address`*.

Implementation

```
branch_if_null( address )
{
    if ( stack[SP] !== null )
    {
        PC = address;
    }
}
```

Listing 5.4: `branch_if_not_null`

5.4 Stack Manipulation

Here are explained the instructions that change solely the stack. Currently, the SUAM has only one in this vein, other stack operations are specific to certain functionalities and part of other instructions.

5.4.1 pop_entries

Syntax
`pop_entries` *`number`*

Semantics
This instruction removes and deallocates *`number`* cells from the top of the stack.

Implementation

```
pop_entries( number )
{
    SP = SP - number;
}
```

Listing 5.5: `pop_entries`

5.5 Basic Node Unification Tests

The instructions in this section are used to do the property checking (cf. Section 2.2 on page 15 and Section 4.2.1 on page 30) of nodes in the data term.

Recall that before a matrix cell can actually contain a unification result it first needs to hold a pointer to the data term with which the current query term will be unified to remember the position of where to store the result of the unification. Therefore, all of the following instructions assume that the top of the stack points to a matrix cell that points to a data term and check this very data term for certain properties.

If a check succeeds and the data term meets its properties nothing is changed. If a check fails the instructions put **`false`** in the referenced matrix cell and replace the pointer on the top of the stack with **`null`** to indicate the failure to the following instructions. That way they do not need to check for their properties, for the unification failed already. This is why all of the instructions in this section do something if and only if the top of the stack does not contain **`null`**.

Having said that, there is one exception: `check_successful`. So that several checks are possible the other instructions only put **`false`** in the referenced matrix cell and never **`true`**. But this is the very reason `check_successful` is needed to finish the checking sequence: When all previous instructions succeeded and there is hence no **`null`**-cell on top of the stack, this instruction writes **`true`** to the matrix cell. This way the "finalization" of a property test is separated from the actual tests, avoiding either duplicating their functionalities or parameterizing the instructions.

5.5.1 check_label

Syntax

`check_label` *`label`*

Semantics

Precondition: The top of the stack is non-**`null`**, otherwise the instruction does nothing.

`check_label` further assumes that, if the above applies, `stack[SP]` points to a matrix cell that in turn points to a data term. No error behavior is specified in case this assumption does not hold. An instruction like `push_child` creates this state.

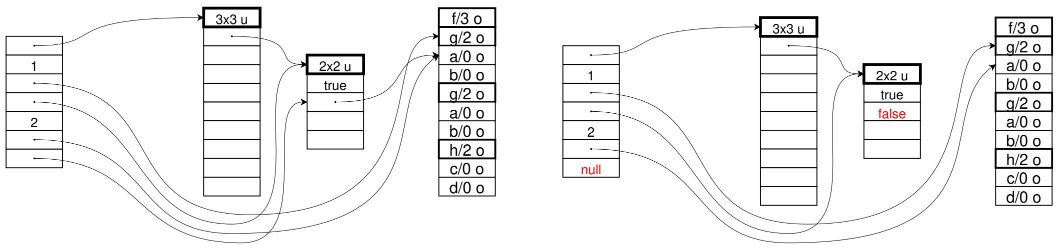
Effect: `check_label` tests if the data term's label matches *`label`*. If so, the instruction does nothing else. If this is not the case `check_label` writes **`false`** to the matrix cell referenced by the top of stack and replaces the top of the stack with **`null`**.

Postcondition: Either `stack[SP]` still points to the matrix cell, which means the test was successful, or `stack[SP]` is **`null`**, which means that the test was unsuccessful.

The label check is an equality test of strings at the moment. Xcerpt offers regular expressions in the query term but it is out of the scope of this research to develop an abstract machine for regular expression matching.

Example

Only the case of non-matching labels is shown here, nothing is changed in the other cases.



(a) SUAM state before the execution of `check_label b`

(b) SUAM state after the execution of `check_label b`

Figure 5.2: Example of a failed label check

Implementation

```
check_label( label )
{
    if ( stack[SP] !== null and heap[access_dt( SP )] !== label )
    {
        result_to_matrix( SP, false );
        stack[SP] = null;
    }
}
```

Listing 5.6: `check_label`

5.5.2 check_ordered

Syntax

`check_ordered`

Semantics

Precondition: The top of the stack is non-**`null`**, otherwise the instruction does nothing.

`check_ordered` also assumes that, if the above applies, `stack[SP]` points to a matrix cell that in turn points to a data term. No error behavior is specified in case this assumption does not hold. An instruction like `push_child` creates this state.

Effect: `check_ordered` tests if the order of the children of the data term is significant for its meaning. That is the case if changing the succession of the children would yield a semantically different data term.

If the test is successful the instruction does nothing else. If this is not the case `check_label` writes **`false`** to the matrix cell referenced by the top of stack and replaces the top of the stack with **`null`**.

This shows that an ordered query term will never match an unordered query term.

Postcondition: Either `stack[SP]` still points to the matrix cell, which means the test was successful, or `stack[SP]` is **`null`**, which means that the test was unsuccessful.

Example

Please see the example for the `check_label` instruction, the changes to the SUAM state in case of a failed test are exactly the same, only the test itself differs.

Implementation

```
check_ordered()
{
    if ( stack[SP] !== null and heap[access_dt( SP ) + 2] !== ordered )
    {
        result_to_matrix( SP, false );
        stack[SP] = null;
    }
}
```

Listing 5.7: `check_ordered`

5.5.3 check_total

Syntax

`check_total` *`arity`*

Semantics

Precondition: The top of the stack is non-**`null`**, otherwise the instruction does nothing.

`check_total` also assumes that, if the above applies, `stack[SP]` points to a matrix cell that in turn points to a data term. No error behavior is specified in case this assumption does not hold. An instruction like `push_child` creates this state.

Effect: This instruction tests if the data term's arity equals *`arity`*. The test is needed if the query term is total, for then the number of children of the query term and the data term has to be the same, otherwise they would not match (cf. Section 2.2 on page 15).

Recall that if the query term is total the data term has to be total as well for a successful unification. But a data term is always total by definition, therefore the arity check is sufficient and an additional test for the data term to be total is not needed.

If the arity test is successful the instruction does nothing else. If this is not the case `check_total` writes **`false`** to the matrix cell referenced by the top of stack and replaces the top of the stack with **`null`**.

Postcondition: Either `stack[SP]` still points to the matrix cell, which means the test was successful, or `stack[SP]` is **`null`**, which means that the test was unsuccessful.

Example

Please see the example for the `check_label` instruction, the changes to the SUAM state in case of a failed test are exactly the same, only the test itself differs.

Implementation

```
check_total( arity )
{
    if ( stack[SP] !== null and heap[access_dt( SP ) + 1] !== arity )
    {
        result_to_matrix( SP, false );
        stack[SP] = null;
    }
}
```

Listing 5.8: `check_total`

5.5.4 check_partial

Syntax

`check_partial` *`arity`*

Semantics

Precondition: The top of the stack is non-**`null`**, otherwise the instruction does nothing.

`check_partial` further assumes that, if the above applies, `stack[SP]` points to a matrix cell that in turn points to a data term. No error behavior is specified in case this assumption does not hold. An instruction like `push_child` creates this state.

Effect: If the query term is partially specified this instruction is used to check if the corresponding data term has *`arity`* or more children. Data terms are always totally and never partially specified, therefore this arity check is sufficient and no additional check for the data term's specification is needed.

If the arity test is successful the instruction does nothing else. If this is not the case `check_partial` writes **`false`** to the matrix cell referenced by the top of stack and replaces the top of the stack with **`null`**.

Postcondition: Either `stack[SP]` still points to the matrix cell, which means the test was successful, or `stack[SP]` is **`null`**, which means that the test was unsuccessful.

Example

Please see the example for the `check_label` instruction, the changes to the SUAM state in case of a failed test are exactly the same, only the test itself differs.

Implementation

```
check_partial( arity )
{
    if ( stack[SP] !== null and heap[access_dt( SP ) + 1] < arity )
    {
        result_to_matrix( SP, false );
        stack[SP] = null;
    }
}
```

Listing 5.9: `check_partial`

5.5.5 check_successful

Syntax

`check_successful`

Semantics

Precondition: The top of the stack is non-**`null`**, otherwise the instruction does nothing.

It further assumes that, if the above applies, `stack[SP]` points to a matrix cell. No error behavior is specified in case this assumption does not hold. An instruction like `push_child` creates this state.

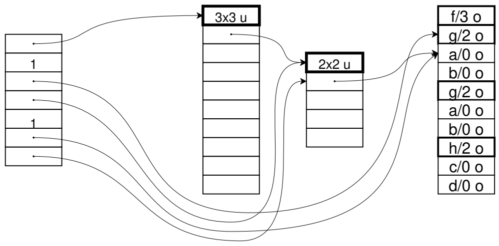
Effect: `check_successful` replaces the contents of the matrix cell with the constant **`true`**.

This instruction is used to check if all previous `check_`-instructions (if any) have been successful. It is needed for cases where the unification process should stop successfully and write **`true`** to the matrix, e.g. when a term is unified that has no children.

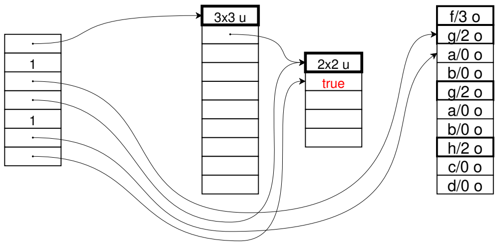
Postcondition: `stack[SP]` is not changed, thus still points to a matrix cell or is **`null`**. In case the former holds, the referenced matrix cell now contains **`true`**.

Example

This instruction only changes the SUAM state in case the top of the stack is non-**`null`**, so only this case is shown. In Figure 5.3 on the facing page the previous link to the data term is replaced by **`true`** because the current query term has been matched successfully to that data term.



(a) SUAM state before the execution of `check_successful`



(b) SUAM state after the execution of `check_successful`

Figure 5.3: Example of a successful leaf unification

Implementation

```
check_successful()
{
    if ( stack[SP] !== null )
    {
        result_to_matrix( SP, true );
    }
}
```

Listing 5.10: `check_successful`

5.6 Handling Variables

The instructions that deal with variables are presented in this section. Both types, unrestricted and restricted variables, are handled by the following instruction.

5.6.1 create_variable

Syntax

`create_variable` *name*

Semantics

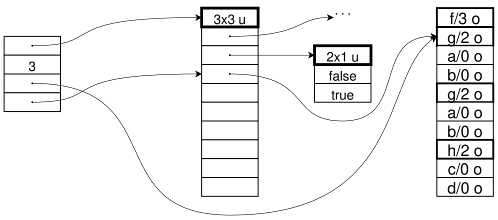
Precondition: The top of the stack points to a matrix cell that itself points to a data term, i.e. an instruction like `push_child` has been executed.

Effect: `create_variable` allocates a new variable structure on the heap. The structure has 3 entries; the name of the variable and two pointers to one and the same data term. The first pointer's destination is the content of the variable or the variable binding. The second one is used to verify the variable restriction by unifying the restriction and the data term the pointer is referencing.

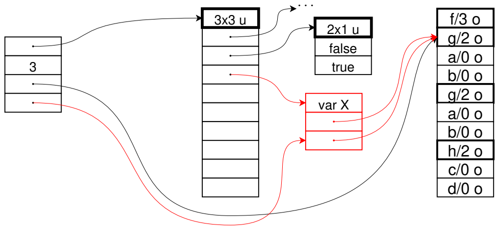
Postcondition: `create_variable` leaves the SUAM in a state similar to the one of `push_child`: the top of the stack indirectly points to a data term. It is the same data term to which the variable was bound and is thus ready to be unified with a variable restriction.

Example

The example below creates a new variable structure, the address of which is stored in the matrix cell referenced by the top of the stack. The variable's binding is the data term $g[a,b]$.



(a) SUAM state before the execution of `create_variable X`



(b) SUAM state after the execution of `create_variable X`

Figure 5.4: Example of the creation of a variable structure

Implementation

```
create_variable( name )
{
    // tmp storage holding reference to the new variable structure
    SP = SP + 1;

    // allocate 3 heap cells: variable name, contents, and restriction
    stack[SP] = heap_alloc( 3 );
    heap[stack[SP] + 0] = "var␣" + name;
    heap[stack[SP] + 1] = access_dt( SP - 1 );
    heap[stack[SP] + 2] = access_dt( SP - 1 );

    result_to_matrix( SP - 1, stack[SP] );    // put variable in matrix

    // point to last cell of variable structure for variable restriction,
    // the top of the stack holds the reference to the first cell
    stack[SP - 1] = stack[SP] + 2;
```

```
    SP = SP - 1;  // tmp storage not needed anymore
}
```

Listing 5.11: `create_variable`

5.7 Finding and Storing Children

The following instructions are used to traverse the direct children of a data term and to collect them in a matrix. The creation of the matrices is handled as well.

5.7.1 create_matrix

Syntax

`create_matrix` *`arity order`*

Semantics

Precondition: The top of the stack points to a matrix cell that itself points to a data term, i.e. an instruction like `push_child` has been executed.

Effect: This instruction creates a new matrix on the heap. The matrix will have the size of the arity of the data term, which can be obtained by double dereferencing the pointer from the top of the stack, multiplied with the arity argument passed to `create_matrix`.

The heap cell (usually a matrix cell) that is referenced by the top of the stack and the top of the stack itself are made to point to the new matrix. Two new stack cells are allocated, the first one containing an offset excluding the matrix header size to get to the next empty matrix cell, and the second one containing a pointer to the next child of the data term that is to be unified using this matrix.

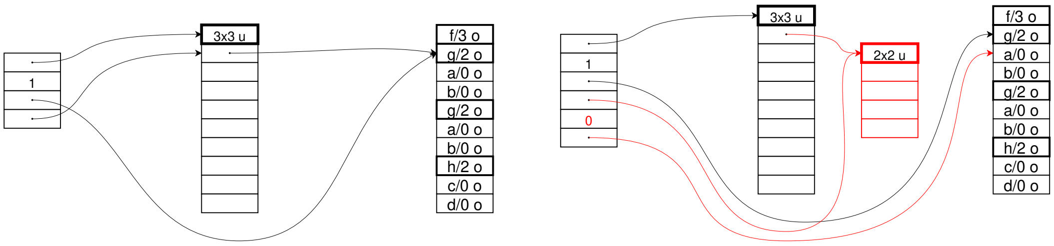
The arity of the query term and the data term as well as the order are stored in the matrix header.

Example

Notice how the new matrix in Figure 5.5 becomes a child-matrix of the already existing one. The new matrix is empty and referenced from the stack. The top of the stack points to the first child of data term that corresponds to the parent matrix.

Implementation

There is one noteworthiness about this implementation: The arity of the original data term that is needed for the size and the header of the new matrix is retrieved by going back two cells in the data term. Strictly speaking, this violates one of the design principles of the SUAM but is done here for code brevity. The “correct” way of doing this would be to allocate an additional stack cell with a backup of the arity before advancing to the first child of the data term.



(a) SUAM state before the execution of `create_matrix 2 u`

(b) SUAM state after the execution of `create_matrix 2 u`

Figure 5.5: Example of the creation of a matrix

```
create_matrix( arity, order )
{
    SP = SP + 2;                             // allocate 2 new stack cells
    stack[SP - 1] = 0;                       // no matrix cells used yet
    stack[SP] = access_dt( SP - 2 ) + 3;     // 1. child, DT hdr is 3 cells

    // now create the new matrix

    // allocate heap cells: arity QT * arity DT + size of matrix header
    result_to_matrix( SP - 2, heap_alloc(arity * heap[stack[SP] - 2] + 3) );

    // push matrix by replacing reference to parent matrix on the stack
    stack[SP - 2] = access_dt( SP - 2 );     // this time address of matrix!

    // put data in matrix header
    heap[stack[SP - 2]] = heap[stack[SP] - 2];   // arity QT: rows
    heap[stack[SP - 2] + 1] = arity;             // arity DT: columns
    heap[stack[SP - 2] + 2] = order;             // order of the QT
}
```

Listing 5.12: `create_matrix`

5.7.2 push_child

Syntax

`push_child`

Semantics

Precondition: The first three stack cells from the top of the stack are part of a matrix: the third cell is a pointer to the matrix on the heap, the second one is the number of matrix cells already occupied and the top of the stack directly points to a data term.

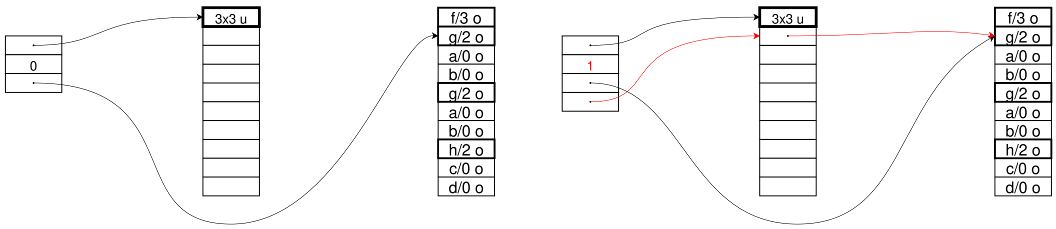
Effect: This instruction takes the reference to the data term and puts it in the next empty cell of the most recent matrix assumed to be found two cells below the top of the stack. A new stack cell is allocated and made to point to the modified matrix cell.

The matrix cell thus filled will later contain the result of the unification of the next query term and the data term just taken. To remember where to store the result `push_child` creates this indirect path from the stack to the data term using the matrix cell as intermediate cell.

Postcondition: On top of the stack is a cell with a pointer to a matrix cell of the matrix mentioned in the precondition that contains a reference to the data term that itself was on top of the stack before the execution of this instruction.

Example

As can be seen below, the `push_child` instruction copies the address of the data term that is referenced by the top of the stack to the first empty cell of the most recent matrix. The new stack cell is made to point to the modified matrix cell and the counter of used matrix cells is increased.



(a) SUAM state before the execution of `push_child`

(b) SUAM state after the execution of `push_child`

Figure 5.6: Example of pushing a data term onto the stack

Implementation

No check for the matrix being full is added, for when the translation of a query term is done properly it will never occur that `push_child` is called although the matrix is complete.

```
push_child()
{
    SP = SP + 1;                                   // new stack cell
    // point to next cell in matrix, matrix header is 3 cells
    stack[SP] = stack[SP - 3] + stack[SP - 2] + 3;
    stack[SP - 2] = stack[SP - 2] + 1;             // 1 more matrix cell used
    result_to_matrix( SP, access_dt( SP - 1 ) ); // matrix cell points to DT
}
```

Listing 5.13: `push_child`

5.7.3 next_child

Syntax

`next_child`

Semantics

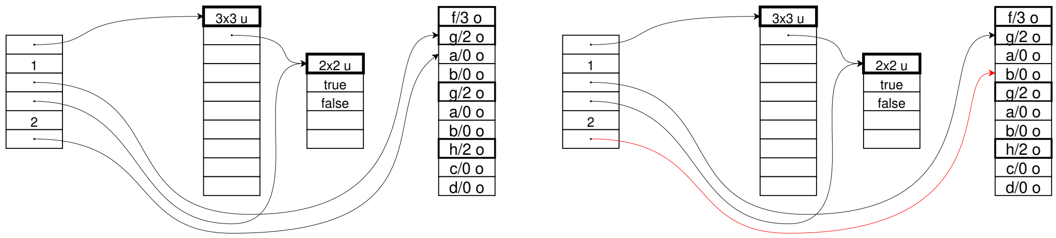
Precondition: The top of the stack contains a direct reference to a data term.

Effect: `next_child` skips all children of this data term and changes the top of the stack to point to the data term's next sibling. If there is no following sibling **`null`** is written to the top of the stack.

Postcondition: The top of the stack contains a direct reference to a data term or **`null`**.

Example

Nothing but the top of the stack is changed by this instruction in Figure 5.7. Its pointer is moved to the next sibling of the previously referenced data term.



(a) SUAM state before the execution of `next_child`

(b) SUAM state after the execution of `next_child`

Figure 5.7: Example of searching a following sibling

Implementation

The implementation stores the level in the data term tree in a new cell on the stack. It is initialized with 0 and updated as `next_child` traverses the tree. Each encountered term (a non-**`null`** cell) increases the level, each end tag (a **`null`** cell) decreases the level. As soon as 0 is reached again it is clear that the term at which `next_child` started has been completely traversed. If the following cell is yet another end tag the term has no following siblings anymore.

```
next_child()
{
    SP = SP + 1;                            // cell for level counter
    stack[SP] = 0;                          // start at level 0

    do
    {
        if ( heap[stack[SP - 1]] !== null )
        {
            stack[SP] = stack[SP] + 1;          // one level down
            stack[SP - 1] = stack[SP - 1] + 3;  // next term
        }
        else
        {
            stack[SP] = stack[SP] - 1;          // one level up
            stack[SP - 1] = stack[SP - 1] + 1;  // next term or null
        }
    }
    until ( stack[SP] == 0 );

    SP = SP - 1;                            // free the level counter

    if ( heap[stack[SP]] == null )          // no siblings?
        stack[SP] = null;
}
```

Listing 5.14: `next_child`

5.8 Finding and Storing Descendants

Similar to the previous section, the following instructions fetch the children of a data term. This time though, *all* children, including the children's children, are fetched. Since the number of them is unknown beforehand, a special frame on the stack and a special, one-dimensional matrix on the heap have to be created.

5.8.1 create_desc_frame

Syntax

`create_desc_frame`

Semantics

Precondition: The top of the stack points to a matrix cell that itself references a data term. An instruction like `push_child` creates this state.

Effect: `create_desc_frame` prepares the stack for the execution of the following descendant instructions. It creates a frame on the stack that contains

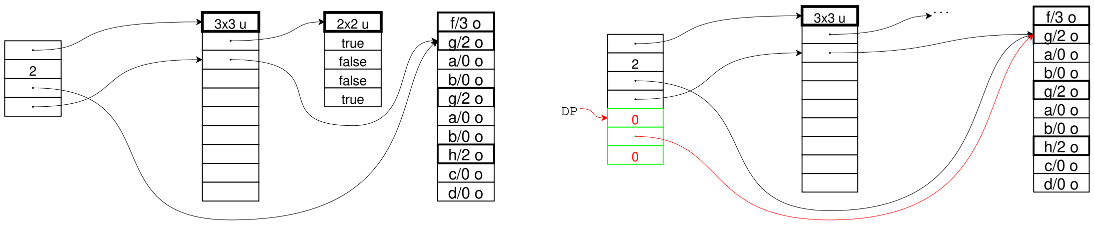
- a pointer to a possible parent descendant frame, initialized using `DP`
- a "next descendant" (ND) pointer: a pointer to the data term from where to continue descending, initialized by taking the top of the stack at the time right before `create_desc_frame` was executed and dereferencing it twice; this pointer can then be advanced by `next_desc` to traverse the data term's children
- the level of the pointer in the data term, relative to the initial data term; it is initialized with 0

`create_desc_frame` sets `DP` to point to the first of these frame cells.

Postcondition: Three new stack cells comprising the descendant frame, and the register `DP` containing the stack address of the first of these cells.

Example

This example shows the creation of the special frame on the stack that will hold the possibly matching descendants.



(a) SUAM state before the execution of `create_desc_frame`

(b) SUAM state after the execution of `create_desc_frame`

Figure 5.8: Example of creating a new descendant frame on the stack

Implementation

```
create_desc_frame( label )
{
    SP = SP + 3;                          // allocate the frame
    stack[SP - 2] = DP;                   // old DP
    stack[SP - 1] = access_dt( SP - 3 );  // next desc pointer
    stack[SP] = 0;                        // relative level of next desc
    DP = SP - 2;                          // update DP
}
```

Listing 5.15: `create_desc_frame`

5.8.2 push_desc

Syntax

`push_desc`

Semantics

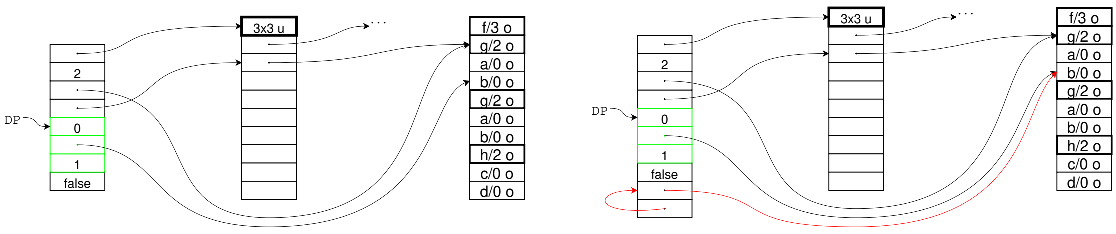
Precondition: A proper descendant frame exists on the stack, `DP` holds the address of its first cell and the second cell is a pointer to a data term.

Effect: `push_desc` takes the descendant found by the last call of `next_desc` and puts it on top of the stack. Because it is unknown how many descendants there will be altogether, `push_desc` sets up the indirect path from the top of the stack to the data term (the descendant) via a stack cell instead of a heap cell. `push_desc` therefore allocates two new stack cells, the first one being a copy of the "next descendant" cell in the descendant frame, pointing to the data term to be unified, and the second cell simply with a pointer to the previous cell.

Postcondition: Top of the stack pointing to the second stack cell from the top which itself references a data term.

Example

Now the difference to `push_child` should have become obvious: The descendants are collected on the stack and not in a matrix because the number of descendants is unknown until the last one is found. Therefore two new stack cells are needed, whereas `push_child` only allocates one new stack cell and occupies one new matrix cell.



(a) SUAM state before the execution of `push_desc`

(b) SUAM state after the execution of `push_desc`

Figure 5.9: Example of pushing a descendant onto the stack

Implementation

```
push_desc()
{
    SP = SP + 2;
    stack[SP - 1] = stack[DP + 1];  // pointer to next desc, a data term
    stack[SP] = SP - 1;             // set up indirection
}
```

Listing 5.16: `push_desc`

5.8.3 next_desc

Syntax

`next_desc` [*label*]

Semantics

Precondition: A proper descendant frame exists on the stack and `DP` holds the address of its first cell.

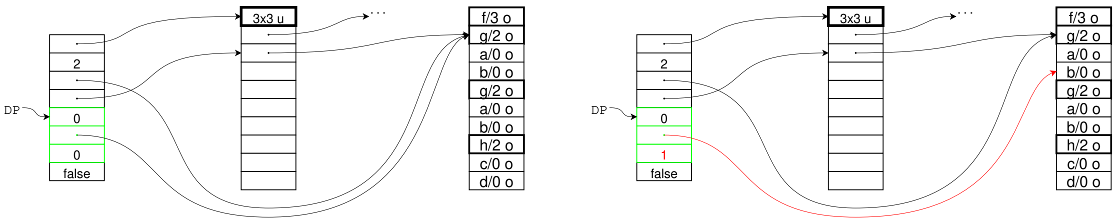
Effect: `next_desc` checks the “next descendant” (ND) pointer in the most recently created descendant frame referenced by the `DP` register. If the pointer contains the address of a data term, i.e. a non-**`null`** cell, `next_desc` continues traversing the data term and its children until a label is found that matches *`label`*. If no label was given, the next encountered child is taken without checking its label.

If there is no matching term `next_desc` writes **`null`** to the ND-pointer in the descendant frame and allocates a new cell on the stack containing **`null`** just as well so that the branch-instructions can check for this case.

While traversing the data term `next_desc` updates the level entry in the descendant frame and writes the current relative depth to it, which is increased every time a data term is encountered and decreased every time a **`null`**-cell is encountered. All of the data term's descendants have been traversed when the depth becomes 0 after decreasing it.

Postcondition: A new stack cell containing **`null`** if there is no further descendant or the ND pointer of the current descendant frame advanced to the next matching descendant.

Example



(a) SUAM state before the execution of `next_desc`

(b) SUAM state after the execution of `next_desc`

Figure 5.10: Example of searching a descendant

Implementation

`next_desc` works much like `next_child` and knows when it has scanned the whole data term by looking at the level entry in the descendant frame. If it contains 0, all children of the initial data term where checked and the end is reached.

In contrast to `next_child`, where only the level at the time of the call of the instruction is important, here the level needs to be stored across several `next_desc`-calls. This is because `next_child` traverses the subtree at once to reach the next sibling, whereas `next_desc` stops every time a matching term is found and continues the traversal only the next time it is called.

```
next_desc( label? )
{
    do
    {
        if ( heap[stack[DP + 1]] !== null )     // a data term
        {
            stack[DP + 2] = stack[DP + 2] + 1;  // one level down
            stack[DP + 1] = stack[SP + 1] + 3;  // next term
        }
        else                                    // end of a data term
```

```
        {
            stack[DP + 2] = stack[DP + 2] - 1;  // one level up
            stack[DP + 1] = stack[DP + 1] + 1;  // next term or null
        }
    }
    until ( stack[DP + 2] == 0 or !label or heap[stack[DP + 1]] == label );

    // the loop stops when the data term is completely traversed OR
    // no label was given OR it has found a potential descendant

    if ( stack[DP + 2] == 0 )                   // no more children
    {
        SP = SP + 1;
        stack[SP] = null;
    }
}
```

Listing 5.17: `next_desc`

5.8.4 create_desc_matrix

Syntax

`create_desc_matrix`

Semantics

Precondition: There is a descendant frame on the stack, referenced by `DP`. The descendant frame contains at least one entry, which follows the descendant frame header.

Effect: When all descendants have been collected and unified with their respective query terms the results will be on the stack in the descendant frame. `create_desc_matrix` is then used to create a new matrix on the heap containing all these results. This matrix has only one row and as many columns as there are entries in the descendant frame.

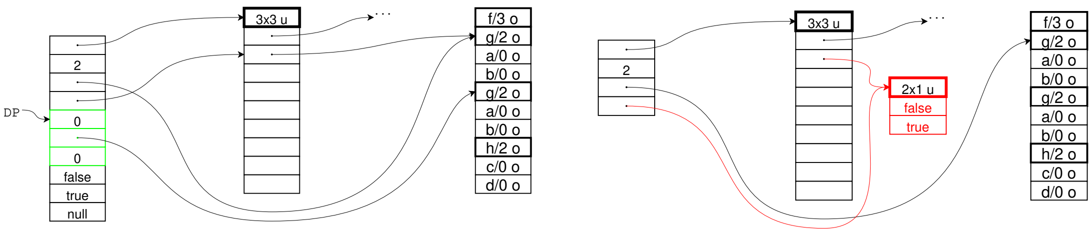
The new matrix is linked to the parent matrix cell that the stack cell preceding the descendant frame points to.

If no descendants have been found the descendant frame only holds one entry containing **`false`** and a matrix with just the value **`false`** is created.

`create_desc_matrix` also deletes the descendant frame from the stack and replaces the indirection from the now top of the stack to the new matrix with a direct pointer to the new matrix.

Postcondition: The top of the stack points to the newly created matrix, the most recent descendant frame has been removed.

Example



(a) SUAM state before the execution of `create_desc_matrix`

(b) SUAM state after the execution of `create_desc_matrix`

Figure 5.11: Example of creating a matrix from a descendant frame

Implementation

```
create_desc_matrix()
{
    // pop the null cell of the last unsuccessful next_desc instruction
    SP = SP - 1;

    // allocate matrix with as many cells as the descendant frame had
    // (descendant frame header size == matrix header size)
    result_to_matrix( DP - 1, heap_alloc( SP - DP ) );

    // push new matrix by resolving and discarding the indirection
    stack[DP - 1] = access_dt( DP - 1 );  // this time address of matrix!

    // fill matrix header
    heap[stack[DP - 1] + 0] = 1;            // only one row
    heap[stack[DP - 1] + 1] = SP - DP - 3;  // as many columns as results
    heap[stack[DP - 1] + 2] = unordered;    // descendants are unordered

    // copy the content from the descendant frame to the matrix
    while ( SP > DP + 3 )         // not the descendant frame header yet?
    {
        // elements in the desc frame have the same "height" as in the
        // matrix, since size of desc frame header == size of matrix header
        heap[stack[DP - 1] + SP - DP] = stack[SP];
        SP = SP - 1;
```

```
    }

    SP = DP;            // pop descendant frame except for the old DP
    DP = stack[DP];     // restore old DP
    SP = SP - 1;        // pop old DP as well
}
```

Listing 5.18: `create_desc_matrix`

6

Example of a Unification in the SUAM

This chapter is dedicated to an example. The first part is the translation of the example query term, the second part will explain the beginning of the unification process to show most of the important issues of unification in the SUAM.

The following is the query term used in the example:

```
f{ g{ a[], b[] }, desc b[], var X }
```

The term represents all elements *f* that have exactly three children in no specified order:

- an element *g* with two child elements *a* and *b*
- an element that contains a descendant element *b*
- an unspecified element that will be bound to the variable *X*

The query term is evaluated using the data term shown below.

```
f[ g[ a[], b[] ], g[ a[], b[] ], h[ c[], d[] ] ]
```

6.1 Translation of the Query Term

The translation of the query term yields the following machine code. The code is nicely indented and the branch instructions have labels instead of instruction pointers to make for easy reading.

```
    init

    check_label f
    check_total 3
    branch_if_null ENDF

```

```
      create_matrix 3 u
F:        push_child

          check_label g
          check_total 2
          branch_if_null ENDG

          create_matrix 2 u
G:            push_child

              check_label a
              check_ordered
              check_total 0
              check_successful

              pop_entries 1
              push_child

              check_label b
              check_ordered
              check_total 0
              check_successful

              pop_entries 1
              next_child

              branch_if_not_null G
              pop_entries 2

ENDG:     pop_entries 1
          push_child

          create_desc_frame
DESC:         push_desc

              check_label b
              check_ordered
              check_total 0
              check_successful

              pop_entries 1
```

```

            next_desc b
            branch_if_not_null DESC
        create_desc_matrix

        pop_entries 1
        push_child

        create_variable X
        check_successful

        pop_entries 1

        next_child
        branch_if_not_null F
        pop_entries 2

ENDF: halt
```

For more details on the translation please see Chapter 4 on page 28.

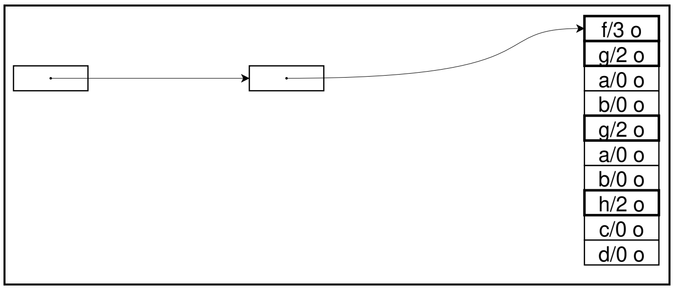
6.2 Unification in the Abstract Machine

In this section the effect of executing the above SUAM program will be illustrated step by step. Each of the instructions that changes the SUAM state is shown using a picture. The red parts are the changes done by the instruction compared to the previous state. The line numbers are the ones from the code above for reference.

On the left there's stack, in the middle of a picture is the growing matrix, and on the right there's the data term. The data term is shortened for clarity by putting the label, the number of children, and the order in one cell. The same holds for the matrix, here the rows, the columns, and the order are written in the first bold cell.

The unification in the SUAM is started with the `init` instruction. It initializes the registers and the stack, and it reads and stores the data term.

```
init
```



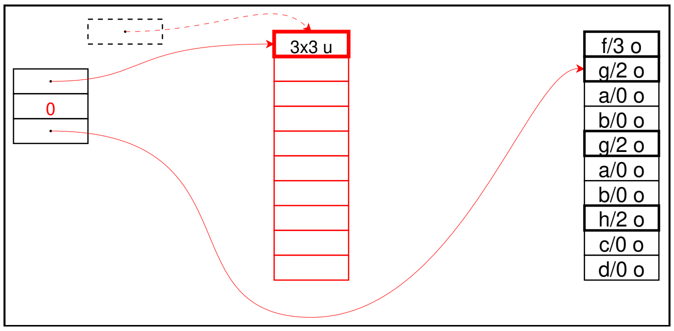
The next two instructions, which check the properties of the root node of the data term, are successful and do not change the SUAM state:

```
check_label f
check_total 3
branch_if_null ENDF
```

Thus, at the time the branch instruction is executed there is a non-**null**-pointer on top of the stack and the branch is not triggered.

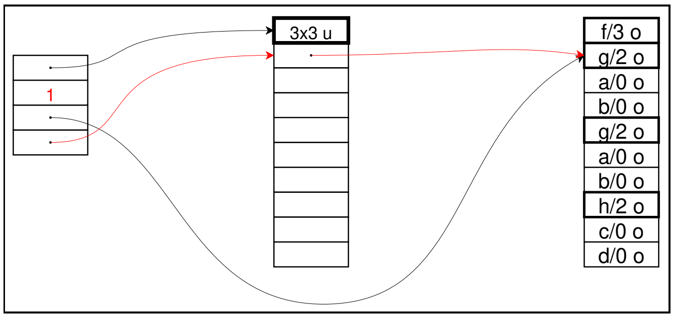
Then the first matrix has to be created, the following image illustrates the state. The dashed stack cell shows the cell that was only needed to save an additional conditional check in most of the instructions. It will not be referenced or shown anymore from now on.

```
create_matrix 3 u
```



Note that the creation of the matrix also sets up a pointer on top of the stack to reference the first child of the corresponding data subterm. The following instruction then copies this reference into the matrix where the result of its unification should be stored and saves that matrix cell on the stack.

```
push_child
```

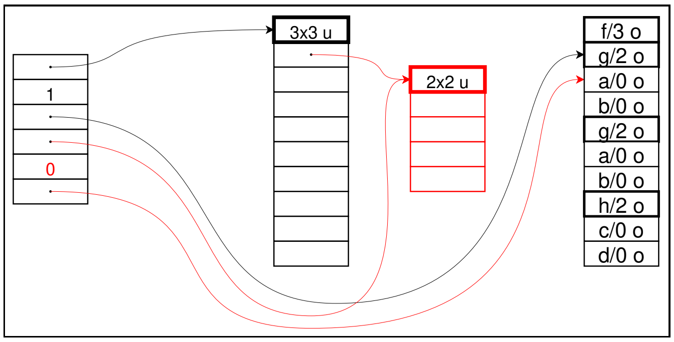


Again, the next instructions are successful and nothing is changed. No branch is executed.

```
check_label g
check_total 2
branch_if_null ENDG
```

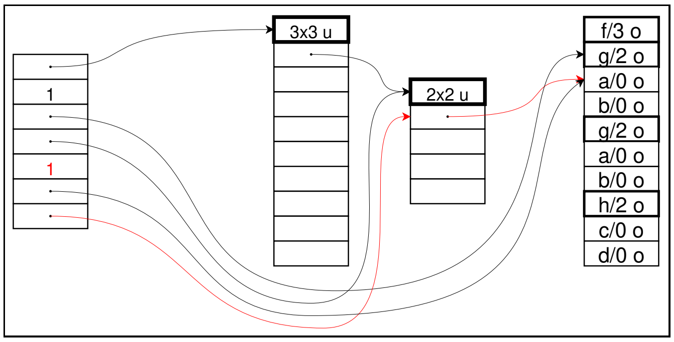
Descending further in the imaginary query term tree, another matrix has to be created.

```
create_matrix 2 u
```



The first child of g is taken for further evaluation

```
push_child
```

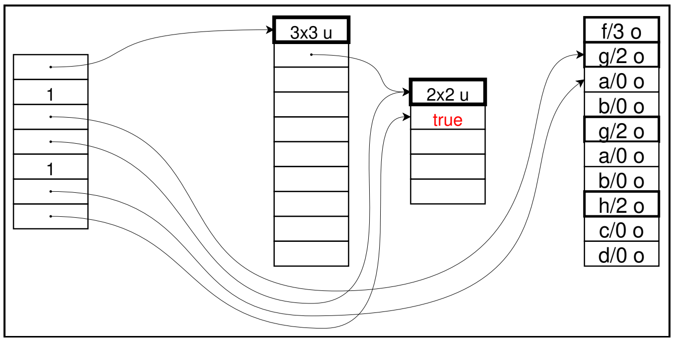


and successfully unified—the three check instructions below do not change anything.

```
check_label a
check_ordered
check_total 0
```

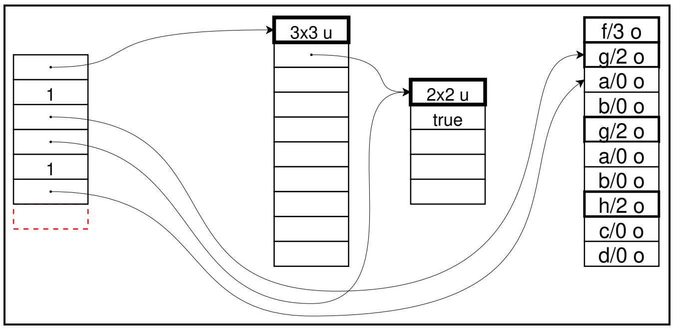
Therefore the subsequent check if that (part of the whole) unification succeeds and writes **true** to the matrix cell to indicate the success of the unification.

```
check_successful
```



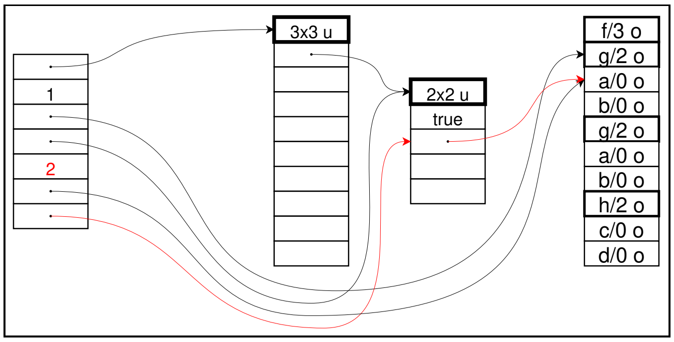
Since this part of the matrix is completed the reference to it is not needed anymore and can be removed from the stack.

```
pop_entries 1
```



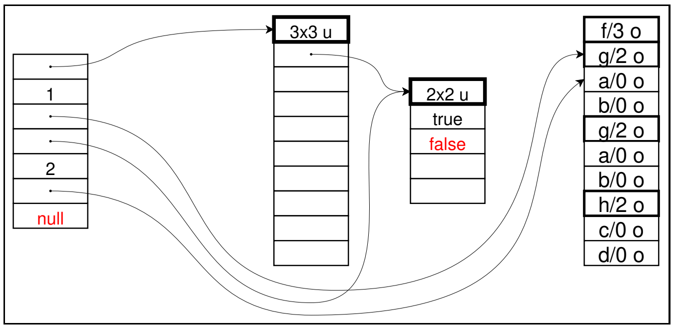
The child of the data term is taken again for possible unification:

```
push_child
```



In this case the label check with the next sibling in the query term failed, however. The matrix cell is set to **false** and the reference on the stack is replaced with **null** to indicate the failure to the following check instructions.

```
check_label b
```

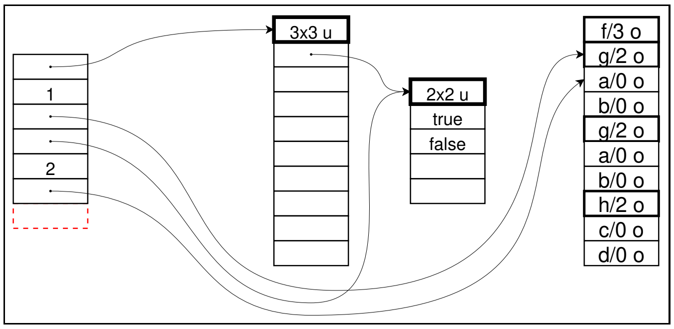


These will therefore not change anything.

```
check_ordered
check_total 0
check_successful
```

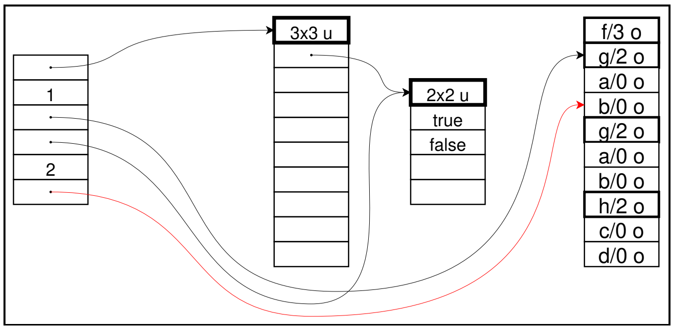
The reference to the finished matrix cell is removed from the stack.

```
pop_entries 1
```



And then the next sibling of the currently referenced data term on top of the stack is searched. If such exists, as in this case, the top of the stack is changed to point to it.

```
31 next_child
```

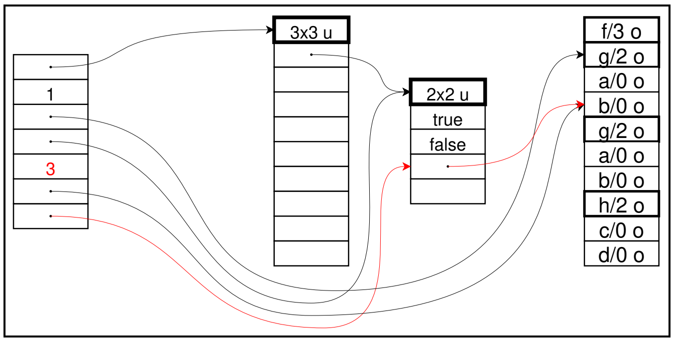


So the top of the stack is not **`null`** and the branch jumps back to the given label.

```
32 branch_if_not_null G
```

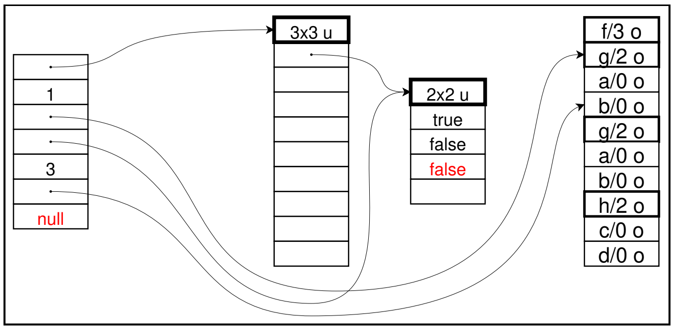
The same query subterms will be checked again against the now new data subterm on top of the stack. The next instruction takes the data subterm into the matrix and onto the stack.

```
15 push_child
```



The label check fails, obviously, and the stack and the matrix are changed accordingly.

```
17 check_label a
```



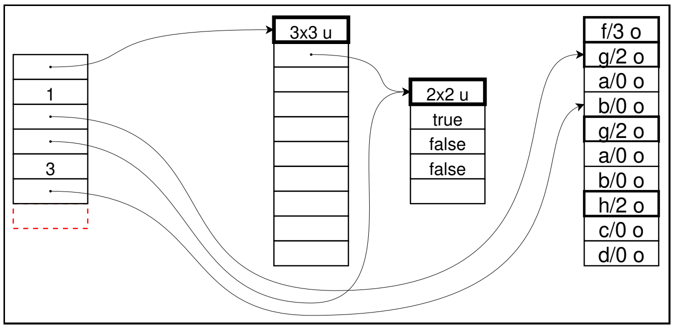
The current SUAM state is retained by the following check instructions.

```
18 check_ordered
```

```
check_total 0
check_successful
```

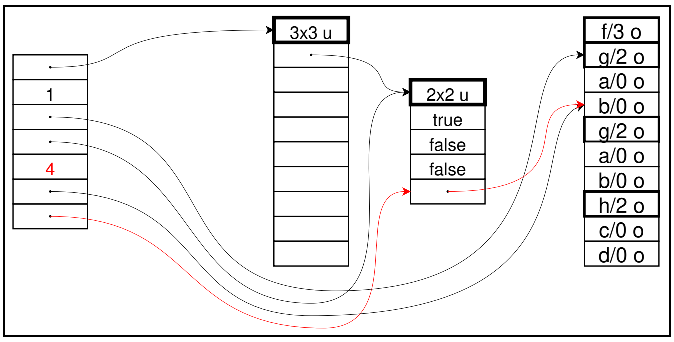
The completed matrix does not need to be directly referenced from the stack anymore.

```
pop_entries 1
```



Then the data term has to be put on the stack for the next part of the unification.

```
push_child
```

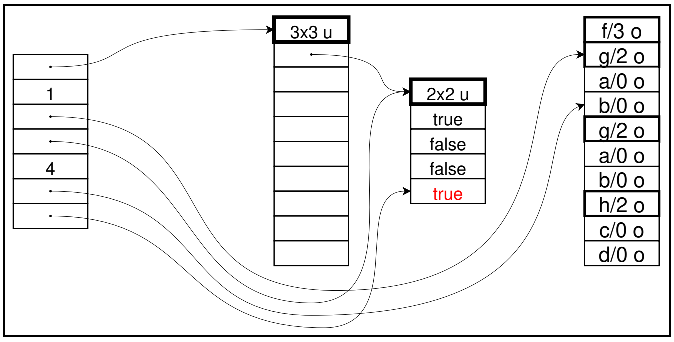


Which is successful and again does not change anything of the SUAM state.

```
check_label b
check_ordered
check_total 0
```

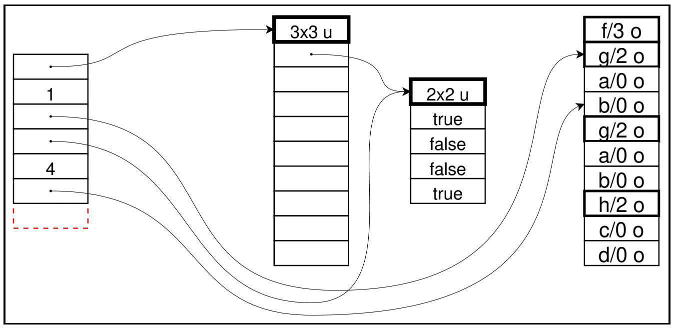
So next the success has to be stored in the matrix,

```
check_successful
```



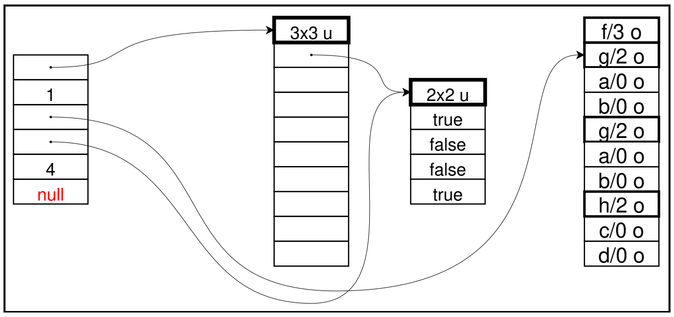
and the reference removed from the stack.

```
pop_entries 1
```



Since all the children of the data term `g` have been evaluated, `next_child` does not find any more of them and indicates this with a **`null`** on top of the stack.

```
next_child
```

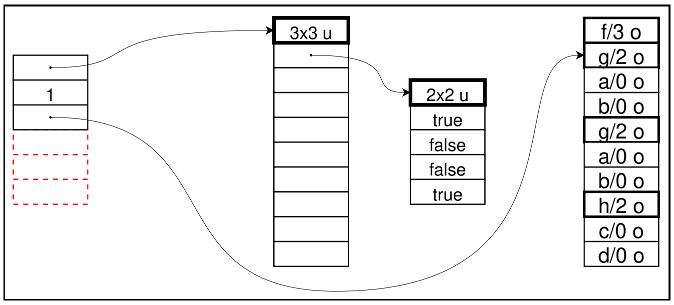


Hence, no branch is executed.

```
branch_if_not_null G
```

Now the first submatrix is complete and since it is referenced from a cell in its parent matrix no reference to it is needed anymore on the stack. Also, the helper data cells on the stack can be removed.

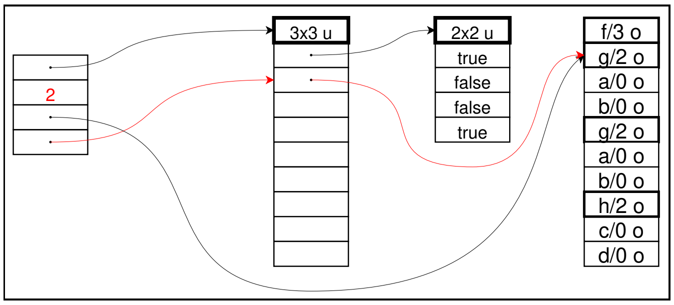
```
pop_entries 2

pop_entries 1
```



Remaining on top of the stack is now a pointer to the first child of the data term `f`, namely `g`. It is the child that has now been completely unified once with the first child of the query term `f` and is about to be unified with the second child, the descendant construct.

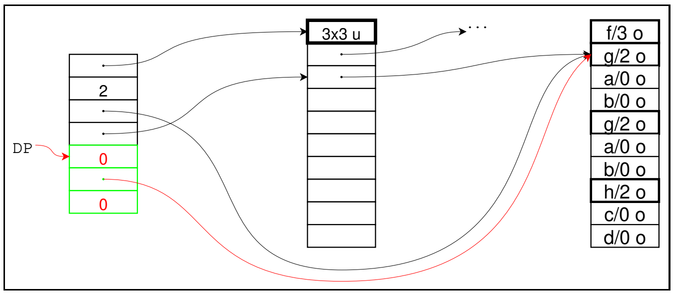
First of all, the data term has to be put on the stack and into the matrix again:

```
push_child
```



Then a descendant frame is created. Its header stores the a pointer to the data term that first constitutes the start of the search and that will later be used to remember the current search position in the data term. Furthermore, the header stores the level of the pointer relative to the data term at which the descendant search started. The descendant frame header's cells are in green color.

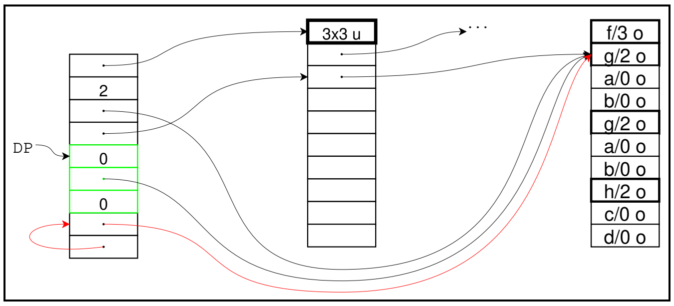
```
create_desc_frame
```



To avoid a possible corner case when there cannot be found any descendant candidate, the data term that is supposed to contain the descendants is taken as descendant candidate itself *without* first checking its label, i.e. without calling `next_desc`. This guarantees that the descendant frame always contains at least one entry.

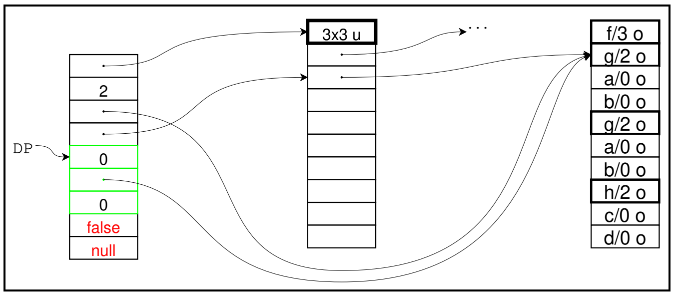
Recall that the final number of descendants is unknown at this point, hence a matrix cannot be created yet. Therefore the results of the descendant unifications are gathered on the stack first.

```
push_desc
```



Once the descendant frame has been set up the usual procedure of unifying terms can continue. The following label check fails:

```
check_label b
```

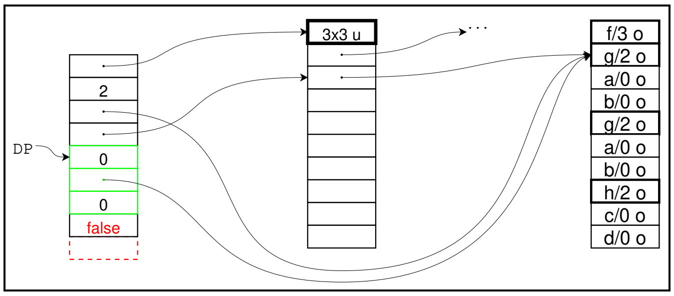


So the rest of the check instructions do not do anything.

```
check_ordered
check_total 0
check_successful
```

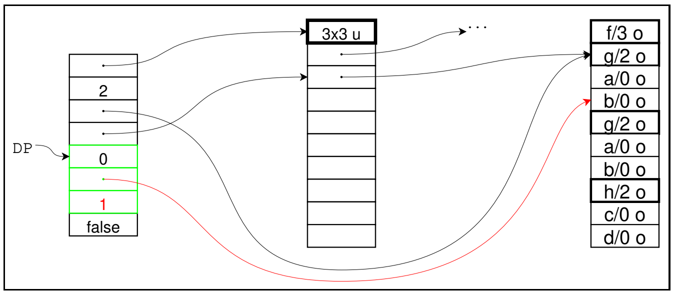
The reference to the result is removed from the top of the stack.

```
pop_entries 1
```



And then the SUAM searches for the next descendant by traversing the data term, starting at the subterm that is referenced by the pointer in the second cell of the descendant frame header.

```
next_desc b
```

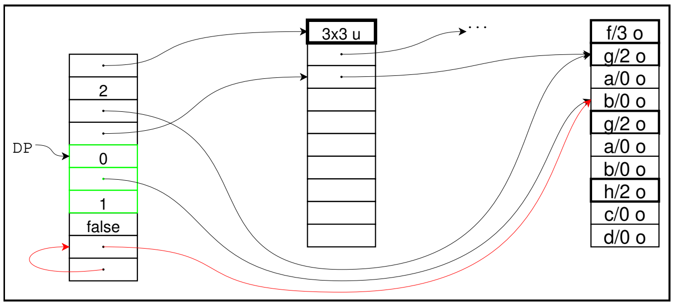


A possibly matching data term was found, so the SUAM jumps back to the unification of a descendant.

```
branch_if_not_null DESC
```

As before, the term is taken onto the heap

```
push_desc
```

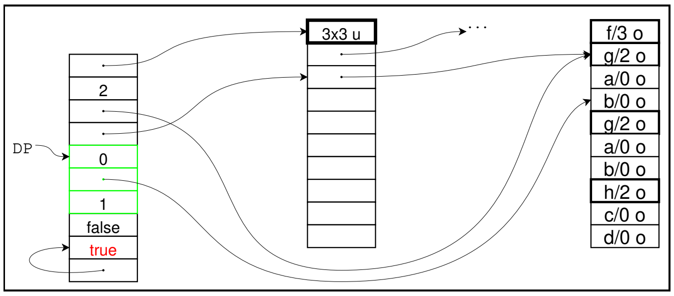


and then unified with the query term—this time successfully.

```
check_label b
check_ordered
check_total 0
```

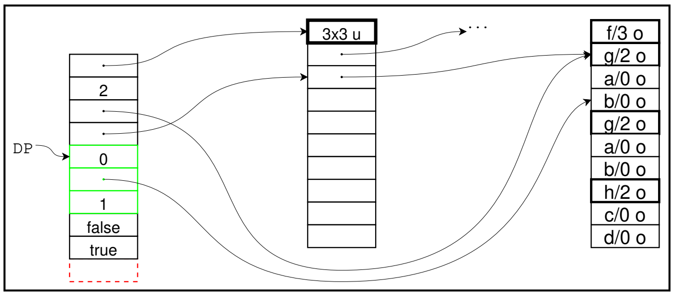
The following instruction replaces the pointer to the data term with **`true`**.

```
check_successful
```



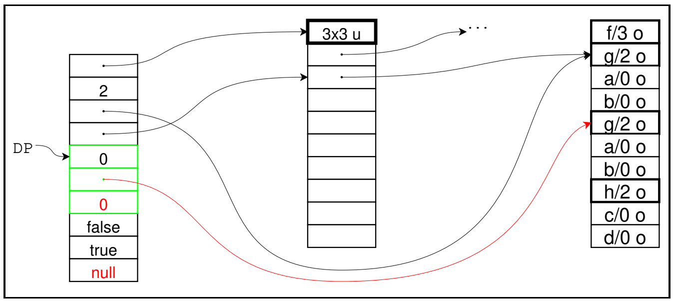
Again, the pointer to the result is not needed anymore.

47 `pop_entries 1`



`next_desc` now hits a data term that is a sibling of the data term it started at, indicated by a 0 in the fourth cell of the descendant frame header. This means there is no further matching descendant and **`null`** is put on top of the stack.

49 `next_desc b`

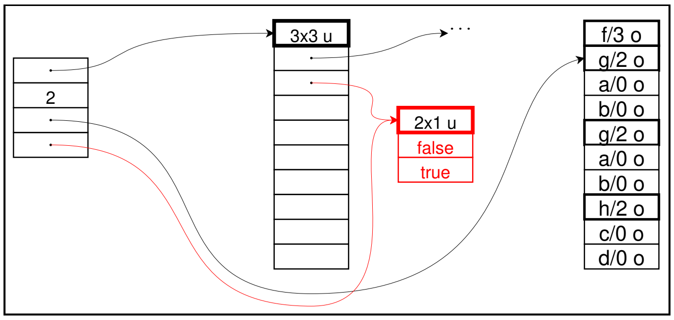


This value also prevents the branch from being executed.

50 `branch_if_not_null DESC`

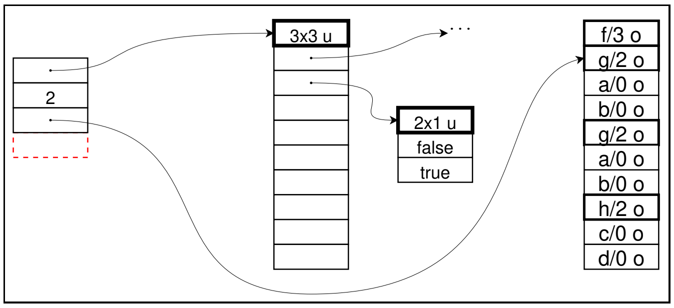
Because all descendants have been collected and their unification results are on the stack in the most recent descendant frame it is possible to finally create the matrix. The following instruction does exactly this and copies the results into the new matrix. It also removes the descendant frame and places a pointer to the newly created matrix on the top of the stack. The descendant pointer `DP` is set to 0 since there is no parent descendant frame.

51 `create_desc_matrix`



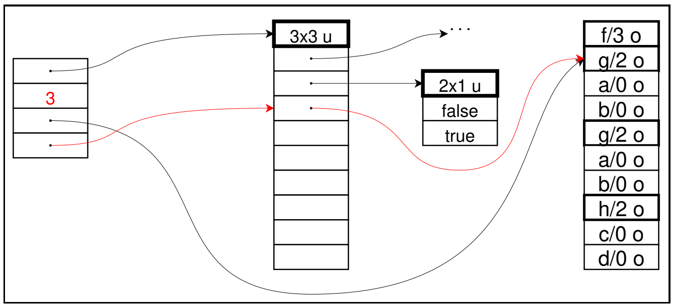
In this case the pointer to this matrix is not needed anymore on the stack, thus it can safely be removed.

```
pop_entries 1
```



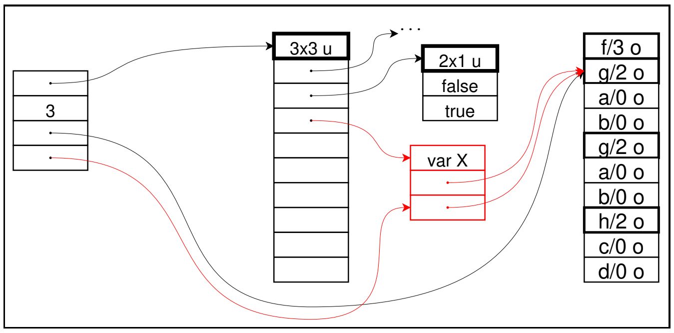
The first child of the root of the data term is now taken once more for unification with the third child of the root of the query term, the variable declaration.

```
push_child
```



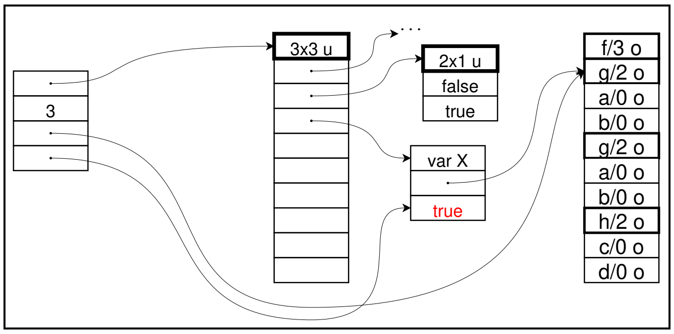
Then a new structure to store a variable is created on the heap. It has two cells, the first one being a pointer to the data term that is supposed to be its content and the second one being a pointer to the very same data term to be able to unify it with the variable's restriction.

```
create_variable X
```



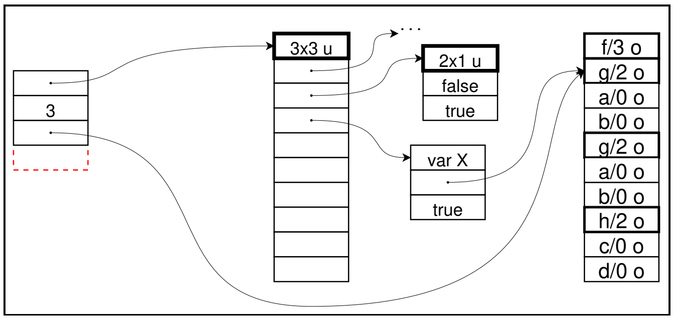
Since there is no variable restriction `check_successful` is called. Of course, it finds a pointer and not **`null`** on top of the stack, so it writes **`true`** to the matrix cell referenced by it—which is exactly the intention: to indicate that the unification of the variable restriction was successful and the variable binding is valid.

```
57 check_successful
```



The variable is complete and its reference can now be removed from the top of the stack.

```
59 pop_entries 1
```



What remains on the stack is the root of the matrix tree, waiting to be completed by the unification results of the other two data term children that were not yet evaluated. Since the process will be exactly the same as the above with only varying results, the example will stop here.

This chapter has shown the exact workings of the SUAM and its memory states. To read and understand the illustrations in this chapter is probably the easiest way to understand most of the SUAM.

7

Future Work and Extensions

First of all, the next step for the SUAM to be extended is the implementation of cycle detection in the data term. The `next_child` and `next_desc` instructions must not fail on such data terms. Also, there needs to be developed an extension of the data term representation to allow for cycles.

To optimize the memory needs of the SUAM the following observation is important: Most matrices for regular query and data terms contain *a lot* of cells with **`false`**. The reference implementation solves this by compacting the matrix. It has to be evaluated if this is also feasible for the SUAM. It is quite possible that at machine level there would be no benefit for the following reasons:

- It may be inefficient to remove single matrix cells because an additional cell to store the number of its previous row and column is needed somewhere
- Runtime performance may suffer because now the algorithm needs to check for the additional information of the cells and cannot rely on memory positions anymore.

Stream processing might be desirable for the SUAM in the future. It is certainly no easy task since the whole algorithm would have to be adapted. Also a different architecture of the heap and thus the creation of the matrix is needed for streaming to work—it has to be possible to incrementally create and fill the matrix cell by cell instead of allocating space right from the beginning. The reason is that the arity of the terms will not be known anymore in this case.

Last but not least the (incremental) consistency checking of variable bindings could be integrated in the whole process by adding some special instructions always after finishing a (sub)matrix. Similarly, instructions could be added to check if a matrix that only contains cells with boolean values collapses to **`true`** or **`false`**.

8

Conclusion and Summary

In this thesis a first version of an abstract machine for Xcerpt's simulation unification is presented. A simple, straight-forward and easy to understand unification algorithm at machine level is developed. Most of the important properties of the simulation unification are implemented and the SUAM thus differs considerably from the standard prolog unification.

The abstract machine is quite close to current hardware architectures and utilizes a stack, a heap, and three registers. This thesis's architecture does not need any garbage collection algorithms, for the data on the heap is only increased during evaluation.

The SUAM architecture allows for several queries to be executed on a single data term without the need for rereading and reparsing the data term. That is especially important when the source of data term is slow, such as the Web, for example.

The abstract machine is not only a definition of the operational semantics of the simulation unification but also an evaluation model for the cost estimation of Xcerpt queries. One of its implementation goals was to better understand the intricacies of simulation unification. Therefore it was desirable to have the unification algorithm in the abstract machine close to the reference implementation in [2] so that improvements made here can be transferred directly to the prototype.

Efficiency problems due to the algorithm being purely sequential and due to the "nested-for-loops" approach in connection with the descendant search have been identified and will be further investigated in the future.

Bibliography

[1] François Bry and Sebastian Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of International Conference on Logic Programming, Copenhagen, Denmark (29th July–1st August 2002)*, volume 2401 of *LNCS*, 2002.

[2] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Institute for Informatics, University of Munich, December 2004.