Proceedings des Hauptseminars *Methoden zur Spezifikation der Semantik von Programmiersprachen* WS 2006/2007, TU Dresden

February 26, 2007

Semantics in Philosophy and Computer Science

Enrico Leonhardt
Computer Science Dept.
Technical University
Dresden, Germany
Enrico.Leonhardt@inf.tu-dresden.de

Dipl. Ing. Simone Röttger[*]
Computer Science Dept.
Technical University
Dresden, Germany
Simone.Röttger@inf.tu-dresden.de

ABSTRACT

The term 'formal semantics' is quite important in computer science since it is used to define programming languages. The techniques behind this term are very powerful but hard to understand. To get a better understanding of the whole area it might be beneficial to have a wider impression of semantics in general. This paper tries to present why and how semantics is used in philosophy and what the correlations are to semantics in computer science.

Keywords

Semantics, Philosophy, Computer Science

1. INTRODUCTION

Semantics (Greek semanticos, giving sings, significant), in general, refers to the aspects of meaning that are expressed in either a natural or an artificial language.

Nowadays, semantics are important in many fields such as linguistics, philosophy, psychology, information theory, logic, and computer science. It is used in different ways whereby in linguistics and philosophy, the most established lines of meaning investigations are published. Many of them discuss the meaning of complex terms that are deviated from simple terms in consideration of syntax, and try to answer the question whether a phrase is true or not, which is known as semantic theory of truth.

To investigate a natural language, philosophy abstracts the content from natural phrases to build a formal language, and uses logical concepts.

The remainder of the paper is organized as follows. In Section 2, the problem of truth in philosophy is presented with some methods of resolutionis. Followed by a short overview of the diffrent areas of semiotics in Section 3. Section 4 talks about formal semantics in Computer Science and finally Section 5 gives the conclusion.

[*]Supervisor

2. THEORY OF TRUTH

One goal in philosophy is to find a formal definition for the 'true' predicate according to the term of truth. By doing this for linguistic entities the difference between "sentence" and "statement" must be clarified as Stegmüller pointed out [2]. Sentences can contain so-called indicators such as 'I', 'you', 'now' etc. that create different statements with different meanings depending on speakers and situations. So, it is not possible to decide whether the sentence

"`He is not here, today.`"

is true or false.

In order to give a consistent definition of "truth" it is necessary to formalize languages or use formal languages without such indicators.

2.1 TARSKI Scheme

Only the theoretical meaning of the 'true' predicate is of importance here. As the logician and philosopher Alfred Tarski [3] pointed out, this is comparable with the intuitive association of 'truth' of a statement, which argues that something is so and so, and at the end it is really so and so (in the real world). Although this might be satisfying in terms of simpleness, it is a kind of fungous definition and not really clear and correct. In order to achieve a better 'correctness' Tarski came up with a scheme that is:

"`X is true if and only if p.`"

Where p can be replaced by any statement, and X refers to the name of it. One example where this scheme can be used is an intuitive verbalization such as:

"`The statement 'the sun is shining' is true`
`if and only if the sun shines.`"

Here, the use of the Tarski scheme, where p is 'the sun is shining' inclusive the quotation marks and X is some identifier for p, gives a partial definition of the 'true' predicate because the theoretical meaning of true is defined for this particular statement.

However, a partial definition is not a definition of the 'true' predicate within a colloquial language (natural language), which is in demand and has the requirements:

- adequate in respect of content
- formal correct

Adequate in respect of content means that "every statement" such as the example above with a 'true' predicate in it used with the Tarski scheme is logical determinable. Even though this is possible for some examples it does not work for "every statement" because of paradox, or antinomies respectively that is presented in the next section.

2.2 Antinomies of Truth

A paradox or antinomy is something where a conflict is generated in spite of faultless using logical and mathematical deductive methods of reasoning.

Mostly the terms paradox and antinomy are set equally. However, there is a difference between them, which should be clarified. An antinomy for instance is a logical paradox whereas there also paradox definitions and paradox act commandments exist.

2.2.1 Paradox definition

One example for a paradox definition is:

```
“ A suicide murderer kills all
that do not kill themselves."
```

Since the question "Does he kills himself or not?" can be answered in both ways, there is a conflict. However, this situation only results from the assumption that such a person really exists. On the other hand, the inference says that such a person can not exists because the question above would generate a conflict.

For this reason, it is not an antinomy since there is no proof that such a situation exists. Respectively there is no logical problem.

2.2.2 Paradox act commandment

It is also possible to use the term paradox for act commandments that never ends. An example for such an endless act commandment is

```
          ‘‘Give someone the commandment
    ‘follow the instructions on a sheet of paper’
          where and on both sites is written
                ‘please turn around’."
```

Even though this really exists, it does not generate a logical problem as well.

2.2.3 Logical paradox - Antinomy

In contrast to paradox definition, a logical paradox or antinomy arises once there is a proof for the statement S, whereas S contains two parts. The part of a statement that claims the opposite of another part S_i is called the negation $\neg S_i$. So, a proof for S exists if there are proofs for both sites because S is the 'and'- catenation of its parts $(S_i.\neg S_i)$.

From section 2.1 it is clear that a formal definition of the 'true' predicate must satisfy the requirement 'adequate in respect of content'. This fails once a partial definition with the Tarski scheme is not logical determinable.

Now, if the 'true' predicate is used in such a statement antinomies can be created. One "popular" example is

```
“All Greeks are liars, said a Greek.”
```

Another more accurate version is from Lukasiewicz:

```
“The statement on page 8 is not true.”
```

As Stegmüller pointed out on page 26 the use of the Tarski scheme and the faultless use of logical deductive methods of reasoning ends in a conflict for these examples.

Therefore, it is not possible to give a formal definition of the 'true' predicate in this way, which is also proven by some other different antinomies. However, all of them have two conditions in common as Tarski pointed out:

- the languages that are used to construct antinomies contain 'true' predicates
- the validity of logical basic laws

To find a solution for the problem of antinomies one of these conditions must be eliminated. Since it is not possible to eliminate or give up the validity of logical basic laws, the elimination of the first condition is inescapable.

2.3 Division of Object and Meta language

For this reason, Tarski divided the natural language in two languages. The first one can be used to describe anything in the objective world. This is called the object language. It does not contain any 'true' predicates and cannot say anything about other statements. An example is:

```
“The table is white.”
```

The second one can be used to say everything. This is called Meta language and contains 'true', 'false' predicates etc. that might be used in order to say something about other statements (that could also formulated in object language).

```
“The previous example is not true.”
```

In addition, a statement in object language is called statement of order one whereas a statement in Meta language is at least of order two. If a statement refers to another, which is already of a higher order the actual statement is one order above (table 1).

Table 1: Order hierarchy of statements

Statement	Order
"`The sun is shining today.`"	one
"`The statement above is true.`"	two
"`The second statement here is true.`"	three

Through this division of the natural language, the construction of antinomies is not possible anymore as Stegmüller presented on page 40 for the antinomy of a statement such as

"`The statement on page 8 is not true.`"

By using object and Meta language, the order must be included in the statement:

"`The statement of order one on page 8 is not true.`"

Assuming there is a page 8 and only this statement is written there an empirical verification would show that there is no sentence of order one on page 8. The decision whether that statement is true or not depends now on the method of analyzes.

Since the method by B Russel is well accepted, the statement is false by using it because it contains the partial statement:

"`There is a statement of order one on page 8 that is false.`"

In this way, it is possible to eliminate antinomies. However, this is not a proof that a definition of the 'true' predicate is working in that way. There are still sentences possible that are not determinable such as:

"`The color is too late.`"

In order to investigate a language and solve the truth problem in philosophy it is necessary to formalize languages and prevent those situations. This is done by the following three different techniques that are part wise mutually, also called "semiotics":

3. SEMIOTICS

Semiotics, in general, is the study of signs and symbols. It can be used for every scientific investigation of language systems whereas there are two different approaches for different fields. Semiotics can be empirical or "pure". The empirical semiotics is used in order to investigate historic traditional language systems. On the other hand, the "pure" semiotics helps to create new artificial language systems and investigate them as well. In both areas semiotics have three branches, namely (1) syntax, (2) semantics, and (3) pragmatics.

3.1 Empirical semiotics

Empirical semiotics is used in Linguistics and Philosophy.

3.1.1 Syntax

Syntax concentrates only on the formal structure of a statement. It is the study of rules, or "pattern relations". For instance the sentence

``` 
''The color is late."
```

is a correct English sentence construction in terms of "pattern relations" (subject + verb + adjective).

To find general laws that govern the syntax of all natural languages, modern research attempts to systematize a descriptive grammar.

3.1.2 Semantics

Semantics is the study of aspects of meaning. It analyzes the meaning of a statement only by its colloquialism and its content whereas two different sorts of meaning a significant expression may have:

- the relation that a sign has to other signs (sense)
- the relation that a sign has to objects and objective situations, actual or possible (reference)

For the example "The color is late." the sense considers the relation between the subject "color" and adjective "late". Does it make any sense? The reference investigates the meaning that this statement has in the objective world.

In addition, there are different syntactic levels of semantics:

- the meaning of each individual word is analyzed by lexical semantics
- relationships between objects within a sentences is referred by structural semantics
- combination of sentences as real or hypothetical facts
- texts of different persons that interacts somehow (discussion, dialog,)

The connection between these levels is realized by the Frege principle, which says that the meaning of a complex sign is a function of meanings of their sub meanings.

```
MEANING(the color is late) = f(MEANING (the),
MEANING (color), MEANING (is), MEANING (late))
```

3.1.3 Pragmatics

Pragmatics is the most extensive technique. It considers all factors of the environment such as (1) the speaker, (2) the colloquialism (statement structure), and (3) the content that is focused by the speaker.

3.2 Pure semiotics

To define new artificial language systems or formal languages in logic, mathematics, information theory and computer science "pure" semiotics also called formal semiotics is used.

3.2.1 Syntax

Syntax defines the formal grammar, or simple grammar. It provides sets of rules for how strings in a language can be generated, and rules for how a string can be analyzed to determine whether it is a member of the language or not.

3.2.2 Semantics

Semantics defines a mathematical model, which describes the possible computations of a formal language especially programming languages in computer science. The different approaches of semantics are called:

- Denotational semantics
- Operational semantics
- Axiomatic semantics

Whereas in logic other modern approaches of semantics are important e.g.:

- Model-theoretic semantics
- Proof-theoretic semantics
- Truth-value semantics
- Game-theoretical semantics
- Probabilistic semantics

3.2.3 Pragmatics

As pragmatics of empirical semiotics, pragmatics for formal languages considers the environment. Such environments are e.g. different compilers, operating systems or machines.

4. FORMAL SEMANTICS

In computer science 'pure' semiotics are used to define artificial languages respectively programming languages.

The syntax defines formal grammars that are often context-free, to describe the set of reserved words and possible token-combinations of a programming language.

Where semantics defines a mathematical model of computation by the following techniques:

- **Denotational semantics** is used to translate each phrase in the language into a denotation, i.e. a phrase in some other language. Denotational semantics loosely corresponds to compilation, although the "target language" is usually a mathematical formalism rather than another computer language. For example, denotational semantics of functional languages often translates the language into domain theory;
- **Operational semantics** is used to describe the execution of the language (rather than by translation). Operational semantics loosely corresponds to interpretation, although again the "implementation language" of the interpreter is generally a mathematical formalism. Operational semantics may define an abstract machine (such as the SECD machine), and give meaning to phrases by describing the transitions they induce on states of the machine. Alternatively, as with the pure lambda calculus, operational semantics can be defined via syntactic transformations on phrases of the language itself;
- **Axiomatic semantics** is used one gives meaning to phrases by describing the logical axioms that apply to them. Axiomatic semantics makes no distinction between a phrase's meaning and the logical formulas that describe it; its meaning is exactly what can be proven about it in some logic. The canonical example of axiomatic semantics is Hoare logic.

The distinctions between the three broad classes of approaches can sometimes be blurry, but all known approaches to formal semantics use the techniques above, or some combination thereof.

However, it would be wrong to view at these styles separately. In fact, all of them are highly dependant on each other as Winskel pointed out [4]. For example, showing that the proof rules of an axiomatic semantics are correct relies on an underlying denotational or operational semantics.

4.1 Dynamic and Static Sematics

Apart from the choice between denotational, operational, or axiomatic approaches, there are two more formal semantics introduced by Consel and Danvy [1]:

- **Static semantics** considers all properties that do not change during the execution.
- **Dynamic semantics** considers all properties that might change during the execution.

5. CONCLUSION

In this paper a rough overview is given about how important semantics are in philosophy and how this is related to computer science.

6. REFERENCES

[1] C. Consel and O. Danvy. Static and dynamic semantics processing. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–24, New York, NY, USA, 1991. ACM Press.

[2] W. Stegmueller. *Das Wahrheitsproblem und die Idee der Semantik.* Springer Verlag, Vienna, 1957.

[3] A. Tarski. *THE SEMANTIC CONCEPTION OF TRUTH AND THE FOUNDATIONS OF SEMANTICS.* Philosophy and Phenomenological Research 4, University of California, Berkeley, 1944.

[4] G. Winskel. *The Formal Semantics of Programming Languages.* Foundation of Computer Series, 1994.

An Introduction To Attribute Grammars

Sven Karol
Department of Computer Science
Technische Universität Dresden
Germany
Sven.Karol@mailbox.tu-dresden.de

ABSTRACT

Beside the syntax, semantic is a very important part of programming languages. Without a semantic a program would no longer be a program but a lifeless sequence of characters which is part of the language.
The dynamic semantic properties of a program are determined during execution at runtime. Typically they directly depend on the input values of the program. By contrast the static semantic properties of a program are computed at compile-time when it is translated from source code representation into some kind of goal representation. The static semantic of a programming language is used to determine those properties.

The concept of attributed context-free grammars (Attribute Grammars) addresses this aspect. It enables a user to create complete specifications of static semantics. This article gives a short introduction on the topic of attributed context-free grammars. It shows how to define such a construct and introduces the concept of synthesised and inherited attributes which can be associated to grammar symbols. Furthermore it will be shown how semantic rules on attributes can be defined. The final part of the paper gives an introduction into different types of attribute grammars and attribute evaluators.

1. INTRODUCTION

Attribute Grammars(AGs) are a widely known approach to express the static semantics of programming languages. They were first introduced as a formal mechanism by Donald E. Knuth in 1968 [1]. Previously they already have been used informally by many compiler developers to define the static semantics of a programming language: In the 1960s there has been a big discussion about how to specify the semantics of context-free languages. Many experts tried out to find a declarative concept but did not succeed while computers went better, and programming languages went more complex. Hence the compiler parts which dealt with the semantics began to become very complex and nearly unmaintainable.
Finally, in 1967 [4], Knuth realised that many people used the same concept of attributes in compilers and that there are especially attributes which only flow from top-down or bottom-up through a parse tree - the idea idea of attribute grammars was born.
Today most compiler-generators use AGs to generate the components for the semantics analysis phase out of a user's specification automatically. In the process of compilation the semantics analysis is the third phase following lexical and syntactical analysis which only deal with context free (syntactic) properties of a language. A lexical analyser(lexer) converts an input stream of characters into a stream of tokens or rather, a stream of terminal symbols. Tokens are the smallest unit a parser can handle. Hence a syntactical analyser(parser) converts an input stream of tokens into a(n) (attributed) syntax tree. The third phase addresses context dependent properties, especially those which carry static semantic. So, in the case of attribute grammars, a semantic analyser(attribute evaluator) takes an unevaluated attributed syntax tree as input and has the evaluated attributed syntax tree as output. Note that typically an attribute evaluator is not the only part of the semantic analysis phase.

A context dependent property of a programming language has static semantics if it can be computed at compile-time of a program. Such a property might be the type information of variables or the result type of arithmetic expressions. In difference to that, a property with dynamic semantic must be computable during execution at runtime. Hence such properties often depend on the input values of a program directly and might change during multiple program executions.

The first of the next three sections deals with the notation and the definition of Attribute Grammars and shows how a context free grammar can be enriched by semantic constructs. Afterwards a simple example is introduced which is used in the whole article to enhance the readers understanding of the different topics. The second section deals with the circularity of Attribute Grammars. Different ways for detecting cycles in *AG*s are explained. The standard approaches for attribute evaluators presented in the third section have no abilities to evaluate *AG*s containing cycles. Hence it might be useful to find them before an evaluation takes place. The last section gives an overview on dynamic and static attribute evaluators. Additionally it introduces L-

attributed grammars as a more restrictive class of Attribute Grammars.
I would recommend readers to be at least familiar with the concept of context free grammars and top-down LL parsing approaches.

2. DEFINING ATTRIBUTE GRAMMARS

The concept of AGs extends the notion of context free grammars through two different kinds of attributes. Inherited attributes are used to specify the flow of information from a node in the abstract syntax tree top-down to lower nodes. Contrary synthesised attributes characterise an information flow in bottom-up direction. Relations between attribute values of different nodes are expressed with semantic rules. Let $G = (N, T, P, S)$ be a context-free grammar with

N - a set of non-terminal symbols,
T - a set of terminal symbols,
P - a set of grammar production rules,
S - the starting symbol,
and $N \cap T = \emptyset$, $p \in P$: $X_0 \rightarrow X_1 \dots X_i \dots X_{np}$ ($X_0 \in N, X_i \in (N \cup T)$), $S \in N$.

The value np represents the count of production elements on a grammar-rule's right side. It might be equal to 0 so that the right side is empty. Such a production is called an ϵ-production deducing the empty word. In the following we use $V = N \cup T$ to represent the grammar vocabulary.

For defining an Attribute Grammar we have to extend the context-free notation with

INH - a set of inherited attributes,
SYN - a set of synthesised attributes,
INH_X - a set of inherited attributes for $X \in V$,
SYN_X - a set of synthesised attributes for $X \in V$,
f - semantic rules bound to syntactic rules and attributes, and $INH \cap SYN = \emptyset$, $INH_X \subseteq INH$, $SYN_X \subseteq SYN$.
Additionally every attribute $a \in INH \cup SYN$ has to be associated with a range of values T_a which one could imagine as type T of a.

Defining f formally is slightly more difficult. If $p = (X_0 \rightarrow X_1 \dots X_i \dots X_{np})$ is a production of the context-free grammar then f has to be defined for every $a_0 \in SYN_{X_0}$ and it has to be defined for every $a_i \in INH_{X_i} (1 \leq i \leq np)$. The arguments of f might consist of any value of any attribute of any grammar symbol in p. The notation $f_{(p,i,a)}$ denotes that a definition of f evaluates the attribute a of the grammar symbol on the ith position in the production p. What does that mean? If $p2$ is a production like $A \rightarrow XA$ in any AG G with

$$INH_A = \{i1\}$$
$$INH_X = \{i2\}$$
$$SYN_A = \{s1\}$$
$$SYN_X = \{s2\}$$

then exactly the semantic rules $f_{(p2,0,s1)}$, $f_{(p2,1,i2)}$ and $f_{(p2,2,i1)}$ have to be defined for $p2$ and none else.

The evaluation of attributes takes place on the abstract syntax tree (AST) which can be regarded as a composition of elemental trees. For every production p in G an elemental tree t_p can be created whose root is X_0 and whose leaves are $\{X_i | 1 \leq i \leq np\}$ (fig. 1).

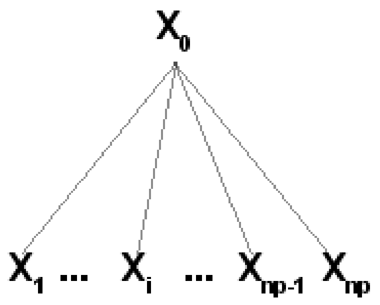

Figure 1: an elemental tree

So any possible AST can be created by repeatedly merging leaves of a subtree with the root of an elemental tree which is labelled by the same non-terminal symbol. In most compilers the parser does this work.
Every production p of a grammar occurs only once but t_p can occur multiple times in an AST. Hence all occurrences embody the same attributes but different attribute exemplars. This means that similar nodes carry different attribute values.

Now we can use the definition from above to construct a concrete instance of an AG. The example will contain a context free grammar which describes a potential representation of hexadecimal numbers enriched by semantic information for computing the decimal value of those numbers. It is a variation of the example binary notation used by [1]. The form of notation leans against the syntax for AGs used in [2].

attribute grammar AG_{hex}
nonterminals

$$N = \{NUMBER, NUM1, NUM2, HEX\}$$
$$S = NUMBER$$

terminals

$$T = \{., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

attributes

$$INH = \{position : int\}$$
$$SYN = \{dvalue : real\}$$
$$SYN_{NUMBER} = \{dvalue\}, INH_{NUMBER} = \emptyset$$
$$SYN_{NUM1} = SYN_{NUM2} = \{dvalue\}, INH_{NUM1} = INH_{NUM2} = \{position\}$$
$$SYN_{HEX} = \{dvalue\}, INH_{HEX} = \emptyset$$

rules

r1: $NUMBER \rightarrow NUM1$
$NUMBER.dvalue = NUM1.dvalue$

$NUM1.position = 0$

r2: $NUMBER \rightarrow NUM1\ .\ NUM2$
$NUMBER.dvalue = NUM1.dvalue + NUM2.dvalue$
$NUM1.position = 0$
$NUM2.position = 1$

r3: $NUM1 \rightarrow NUM1\ HEX$
$NUM1_0.dvalue = HEX.dvalue * 16^{NUM1_0.position} + NUM1_1.dvalue$
$NUM1_1.position = NUM1_0.position + 1$

r4: $NUM1 \rightarrow \epsilon$
$NUM1.dvalue = 0$

r5: $NUM2 \rightarrow HEX\ NUM2$
$NUM2_0.dvalue = HEX.dvalue * 16^{-NUM2_0.position} + NUM2_1.dvalue$
$NUM2_1.position = NUM2_0.position + 1$

r6: $NUM2 \rightarrow \epsilon$
$NUM2.dvalue = 0$

r7: $HEX \rightarrow \{0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F\}$
$HEX.dvalue = getValueForChar()$

Note that in this example the semantic rules are directly annotated below the syntactic rule they correspond to. In practice semantic rules can become very large thus they normally would be stored in programming-libraries.
The grammar defines a number to consist of one part if it is a whole number or two parts separated by a point if it is a real number. The left-recursive non-terminal $NUM1$ represents the integer part while the right-recursive $NUM2$ is used to represent the real part. It would be possible to use only one of both non-terminals but then one would have to introduce an other inherited attribute counting the length of a side at first.
Hence AG_{hex} uses two attributes only. The synthesised attribute *dvalue* is used to accumulate the computed decimal values of the hexadecimal cyphers. Those values are published by every $NUM1$ or $NUM2$ node in the AST to its predecessor. The attribute *position* is an inherited attribute which is used to count the actual position in a hexadecimal number. It is incremented and published top down to the corresponding $NUM1$ or $NUM2$ node. There it is used in the exponent of the computation of the decimal values. Finally after all attributes have been evaluated the result can be found in the root of the AST.
It is very likely that any meaningful context-free grammar contains productions with multiple occurrences of the same grammar symbol - at least in recursions. So there must be a possibility to distinguish between them. Otherwise it would not be possible to define semantic rules for such productions. Typically this problem is solved by introducing an implicit numeration as it was done in the example. For instance an argument A_0 of a semantic rule would correspond to the first occurence of the grammar-symbol A in a syntactic rule $r = (A \rightarrow XAY)$ while A_1 addresses the second exemplar.
Another interesting property of AG_{hex} can be found. If an attribute occurs on a left side of a grammar production's semantic rules it never occurs on the right side and vice versa. This property is called the **normal form** of an AG. More formally:

Let AG be an attribute grammar and $p \in P$ a production then AG is in **normal form** if all arguments arg of semantic rules can be found in $(INH_{X0} \cup SYN_{Xi})(1 \leq i \leq np)$.

Let us take a look on the concrete evaluated derivation tree T_{ex} for the number string $h = ' 08.15'$(fig.2). Obviously 08.15_{hex} seems to be equal to 8.082_{dec} as it is stored in $NUMBER.dvalue$. In this notation synthesised attributes are always noted to the right of a node while inherited attributes are noted to the left.

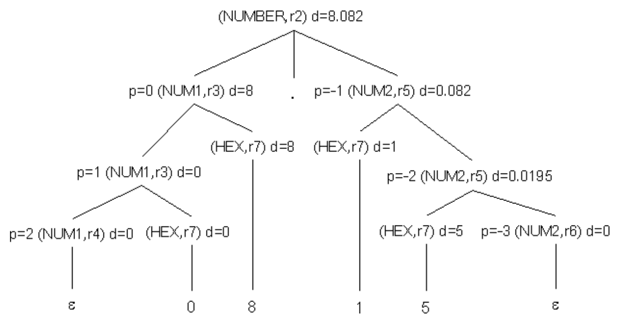

Figure 2: example: attributed abstract syntax tree for h='08.15'

Inherited attributes are not necessarily needed. So for every attribute grammar $A(INH \neq \emptyset)$ it is possible to create at least one equivalent attribute grammar $B(INH = \emptyset)$ by introducing a synthesised attribute c to transfer all information about the nodes in the tree at the root [1]. However a usage of synthesised attributes only often leads to very complicated semantic rules and an additional overhead for the evaluation of the attribute c at the root. Hence typically inherited attributes lead to a better readability and less computation effort.
Nevertheless there are exceptions to that rule. For instance in our example it would be possible to use synthesised attributes without introducing much more effort. Therefore we could invert the recursion in the rules r_3 and r_5. Additionally we would have to convert the position attribute into a synthetic attribute. Furthermore the semantic rules of $r1$ and $r2$, which are responsible for initialization of *position*, have to be replaced by similar methods in r_4 and r_6.

3. DEPENDENCIES AND CYCLES

Testing for circularity is a very important topic for AGs because it is impossible to evaluate attributes which depend on themselves. Fortunately there are methods to pre-calculate dependencies of attribute exemplars at generation time, e.g. when a compiler is generated or written by hand. Hence the information can be used to solve two problems.
First of all a grammar can automatically be checked on errors before anything will be created. So it can be used to support the users of compiler-compilers to solve conflicts in grammars. Secondly a cycle free dependency information may be used to pre-compute an order of evaluation steps for a static attribute evaluator.

For any deducible derivation tree T of a grammar G an evaluator has to take into account all relations between attribute exemplars. Those relations can be expressed by a *dependency graph* $D(T)$. The nodes of $D(T)$ correspond to attribute exemplars in T and can be written as $N.a$, where N is a node in T while a is an attribute of the grammar symbol being the label of N in T. Let $X1 = label(N1)$ and $X2 = label(N2)$ be the grammar symbols which are label of $N1$ and $N2$, then $D(T)$ contains directed edges from $N1.a1$ to $N2.a2$ if there is a applicable semantic rule $X2.a2 = f(.., X1.a1, ..)$ for a production p with $X1 \in p$ and $X2 \in p$ in this context. According to our example, $D(T_{hex})$ is the graph in figure 3.

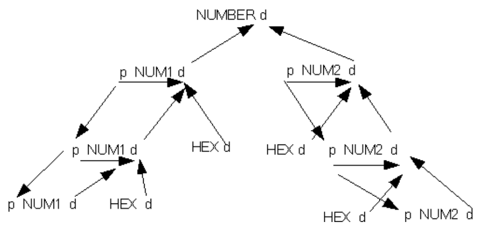

Figure 3: dependency graph $D(T_{hex})$

The transitive closure $D^+(T)$ contains an edge from a node $N1.a1$ to a node $N2.a2$ if, and only if, there is a path between both nodes. If $D^+(T)$ does not contain any edge of the type $(N.a, N.a)$, $D(T)$ contains no cycles and can be evaluated.

An attribute grammar G is **well-formed** if $D(T)$ contains no cycle for every possible derivation tree T.

It is impossible to pre-compute all $D(T)$ because typically it is possible to create an infinite number of derivation trees for G. Hence the dependency information has to be computed directly out of the grammar rules. So it is useful to regard the local dependency graph $D(p)$ for a production p. Let p be a production then $D(p)$ contains nodes $N.a$ for every grammar-symbol $X \in p$ with $X = label(N)$ and all attributes a with $a \in INH_X \cup SYN_X$. There is an arc from $N1.a1$ to $N2.a2$ in $D(p)$ only if there is a semantic rule $X2.a2 = f(.., X1.a1, ..)$ defined for p. $D(p)$ can be constructed for every production in a grammar and any possible $D(T)$ can be created through pasting various $D(p)$ together. Figure 4 shows the $D(p)$s according to our example. None of them contains any cycle. So AG_{hex} is **locally free of cycles** which is the weakest condition for a cycle free grammar: an AG contains no local cycles if every graph in $\{D(p)|p \in P\}$ is free of cycles. As every normalized AG contains no local cycles the test for this property is a co-product of the normalization process [3].

In order to test globally if an attribute grammar contains any cycle some additional constructs are needed [3]. The operation root-projection $rp(D)$ applied on a dependency graph $D(T)$ results in graph which only contains nodes which correspond to the attributes of the underlying tree's root. Additionally $rp(D)$ contains all edges which lead from an attribute of the root node to an attribute of the root node. Another important operation is the overlay operation. Let p be a production, and for $1 \leq i \leq np$ let G_i be any directed graph with nodes $\{a|a \in SYN_{X_i} \cup INH_{X_i}\}$ then

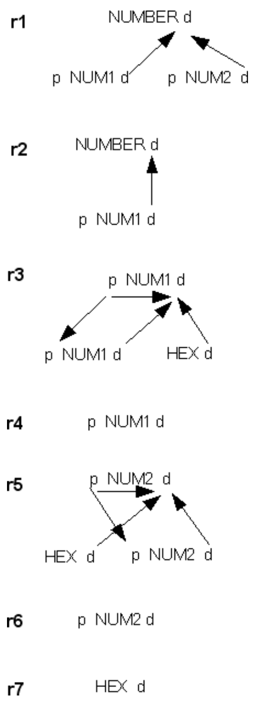

Figure 4: local dependency graphs of AG_{hex}

$D(p)[G_1, \ldots, G_{np}]$ is the local dependency graph for p overlayed with the edges of G_i.
Figure 5 shows both operations applied on some local dependency graphs of our example. The grey edges in that figure do not belong to any graph. They only shall clarify the importance of the order of $D(p)[\ldots]$'s arguments.

With these operations we can calculate sets $S(X)$ for every

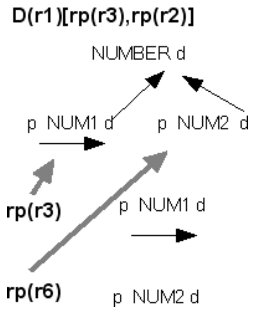

Figure 5: $rp()$ **and** $D(p)[\ldots]$

grammar-symbol X of an attributed context free grammar. Such a set contains every possible dependency graph between the attributes of X. This information will be enough to determine if any dependency graph $D(T)$ can contain any cycle. We use the following algorithm to compute the $S(X)$:

1. **foreach** $n \in N$: set $S(n) = \emptyset$
2. **foreach** $t \in T$: set $S(t) = \{(SYN_t \cup INH_t, \emptyset)\}$
In other words: Initialize the $S(X)$ for all grammar-symbols. $S(X)$ for non-terminals contains no graph while for terminals $S(X)$ is initialized with a graph having nodes only. Typically terminals are not attributed, hence in most cases $S(X)$

will be empty too.

3. **repeat**

```
- choose a production $p \in P$
- for $1 \leq i \leq np$ choose $G_i \in S(X_i)$
- compute the overlay $D(p)[G_1, \ldots, G_{np}]$
- compute the root-projection $r$ for the transitive
    closure
$rp(D(p)^+[G_1, \ldots, G_{np}])$
- add $r$ to $S(X_0)$ if, and only if $!(r \in S(X_0))$
```

until there are no more possibilities to create a new graph which could be added to any $S(X)$.
In other words: Try to construct new graphs by overlaying any rule dependency graph with root-projected graphs which already have been created and add them to $S(X)$. If no new graphs can be added the algorithm terminates.

Now our circularity test is nearly complete. We only have to check if any overlayed graph $D(p)[G_1, \ldots, G_i, \ldots, G_{np}]$ with $G_i \in S(X_i)$ contains any cycle.
If not, the grammar is **free of cycles**. Unfortunately the algorithm seems to have a high time complexity because there are many combination possibilities to create a new graph in $S(X)$. Fortunately there is an other criterion which is even more strict and causes less computation effort. Instead of the set $S(X)$ only one graph $s(X)$ per X can be created. It could be regarded as a merge of all graphs in $S(X)$. For computing the $s(X)$s the above algorithm has to be slightly modified: instead of repeatedly adding graphs to $S(X)$ these will be stepwise merged in $s(X)$ until none of the $s(X)$ can be changed any more. Afterwards we have to check if any overlayed graph $D(p)[G_1, \ldots, G_i, \ldots, G_{np}]$ with $G_i = s(X_i)$ contains any cycle.
If not, the grammar is **absolutely free of cycles**.

It should be said that if an AG is *absolutely free of cycles* the grammar also is *free of cycles* but not vice versa! The test for *absolute cycle freedom* might fail while the standard test for *cycle freedom* does not fail. Hence a test procedure for cycles in a compiler generator might look like in the following:

1. Try to transform the AG into normal form, if that fails return 'grammar contains cycles' otherwise continue with step 2.
2. Apply the test for *absolute cycle freedom*, if that fails continue with step 3 otherwise return 'grammar is free of cycles'.
3. Apply the test for *cycle freedom*, if that fails return 'grammar contains cycles'.

The results of that test do not have to be thrown away. Instead the $s(X)$ could be used by an attribute evaluator at compile-time. Therefore some more theoretical constructs are needed which will not be discussed here. For deepening I would recommend to read [2,chapter 9.4].

4. ATTRIBUTE EVALUATORS

At a glance there are two groups of attribute evaluators. The group of dynamic evaluators computes attribute dependency information on runtime only. In contrast static evaluators use dependency information which has been computed during generation time of a compiler. The basic ideas for that were shown in the last section.
Some of the approaches lead to restrictions on the attribute definition. Hence they also lead to some more restricted classes of AGs.

Data-driven evaluator
This is a very intuitive approach for a dynamical evaluation. It can be applied to all possible attributed context free grammars regardless of if they contain cycles or not - data-driven evaluation can detect cycles dynamically.
An evaluation takes place on the AST where, in every iteration, the algorithm searches for evaluable attributes. Initially those could be found as synthesised attributes of the leafs or on any other node where constant values have been assigned to an attribute. In case of our example this meets attributes of nodes which are evaluated by one of the semantic rules $f_{(r1,1,position)}$, $f_{(r2,1,position)}$, $f_{(r2,2,position)}$, $f_{(r4,0,dvalue)}$, $f_{(r6,0,dvalue)}$ and $f_{(r7,0,dvalue)}$.
The algorithm works as follows:

1. **foreach** node $n \in T$: assign all directly computable values to the corresponding attributes
2. **if** there are more evaluable attributes continue with step 1, **else** goto step 3.
3. **if** there are still unevaluated attributes return 'grammar contains cycle', **else** return 'all attributes evaluated'

Obviously the algorithm has to iterate over all attribute exemplars in the AST during a single pass. In the worst case it finds only one attribute it can evaluate per step. So the time complexity would be c^2 (if c is the number of attributes).

Demand-driven evaluator
This dynamical approach is slightly different from the data-driven approach. It tries to evaluate attributes on the AST regardless if they can be computed directly or depend on other ones. In the case that an attribute depends on others which not have been evaluated too, a demand-driven evaluator tries to evaluate those recursively. Typically the algorithm starts with the synthesised attributes on the root of the AST.
In the abstract syntax tree of our example (fig. 2) the computation could start on the attribute *dvalue* of the $NUMBER$ node.

Due to
$f_{(r1,0,dvalue)} = NUM1.dvalue + NUM2.dvalue$

the algorithm continues with
$f_{(r3,0,dvalue)} = HEX.dvalue * 16^{NUM1_0.position} + NUM1_1.dvalue$

and computes it's first value with
$f_{(r7,0,dvalue)} = getValueForChar()$.

After that the computation continues with the evaluation of $f_{(r3,0,dvalue)}$'s second part. Typically the implementation is a simple recursive function like (for more detail see [3]):

```
public AttrValue Node::eval(Attribute attr){
    if(values.get(attr)!=null)
        return values.get(attr);
    if(working.get(attr)==true)
        throw new Exception("contains cycle");
    working.set(true);
    //find out which semantic rule must be applied
    {....
    //recursion
    someOtherNode.eval(someAttribute);
    ....}
    values.set(attr,resultValue);
    working.set(attr,false);
    return resultValue;
}
```

The algorithm terminates if all attributes which are reachable from the root could have been evaluated - unreachable attributes are ignored. It also detects cycles in the moment when the method body is entered a second time before having the corresponding attribute evaluated.

L-attributed grammars
L-attributed grammars belong to a class of *AG*s which can be evaluated during syntax analysis. They do not require an explicit construction of the syntax tree. Instead attribute values are annotated to elements on the parser stack. Hence compilers which use this technique typically become very efficient. Unfortunately some restrictions on how attributes are allowed to be defined must be introduced. The well-established deterministic parse algorithms for *LL* and *LR* grammars 'traverse' on the parse-tree from top-down in a depth-first manner. Hence attributes have to be evaluated in the same order. This means that a node n in the parse-tree (or grammar-symbol which is currently derivated) should only carry attributes which directly depend on synthesised attributes of nodes which are left siblings of n. Of course it may depend on inherited attributes of its father and on the synthesised attributes of its direct children too. Figure 6 shows such a situation for a production $p = (X_0 \rightarrow X_1 \dots X_i \dots X_{np})$.

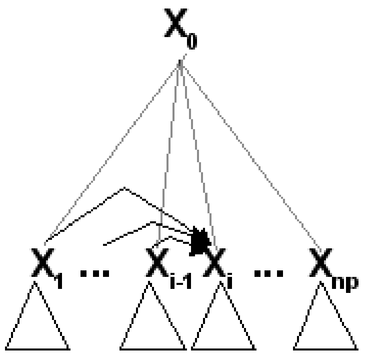

Figure 6: allowable attibute flows from siblings to X_i

More formally a normalized attribute grammar G is called $L\!-\!attributed$ if for each $p \in P$ with $p = (X_0 \rightarrow X_1 \dots X_{np})X$ and for every $a \in INH_{X_j}(1 \leq j \leq np)$ the semantic rule $f_{(p,j,a)}$ has arguments $b \in (\bigcup SYN_{X_k})(1 \leq k \leq j-1)$ or $b \in INH_{X_0}$ only.

LL-attributed grammars
L-attributed grammars having an underlying context-free LL(1) grammar are called LL-attributed grammars.
Hence it is possible to extend the standard parsing techniques for LL(1). In the table driven approach a parser holds only minimal information per step. This information has to be enriched by a kind of label which marks when all symbols in a production have been completely derived and the synthesised attributes of the production's root X_0 can be evaluated. This might be done by a special symbol placed behind a production on the parser stack. Additionally the attributes themselves have to be stored somewhere because after every shift step a grammar-symbol is removed or replaced from the parser stack. Hence the parse tree has to be built-up somehow in parallel, at least temporary. For instance symbols on the stack could be associated with production elements in the derivation queue which such a parser normally builds up. There attribute values could be stored and later be read. An other option would be an explicit partial build-up of the parse tree with nodes currently needed only.
A recursive descent parser for LL(1) would have all these things included automatically. Such a parser can easily be constructed from recursive functions which directly correspond to the grammar productions. Due to the recursive calls a partial parse tree is implicitly constructed on the runtime-stack. A production has been completely derived when a function returns.
Unfortunately our *AG* AG_{hex} is not a LL(1) grammar. So we use only the right recursive part of our grammar to demonstrate how to create an attribute evaluator:

attribute grammar AG_{hexLL1}
...
rules

r1: $NUMBER \rightarrow NUM$
$NUMBER.dvalue = NUM.dvalue$
$NUM.position = 1$

r2: $NUM \rightarrow HEX\ NUM$
$NUM_0.dvalue = HEX.dvalue * 16^{-NUM_0.position} +$
$NUM_1.dvalue$
$NUM_1.position = NUM_0.position + 1$

r3: $NUM \rightarrow \epsilon$
$NUM.dvalue = 0$

r4: $HEX \rightarrow \{0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F\}$
$HEX.dvalue = getValueForChar()$

With this we can construct a very simple recursive descent LL(1) parser.

```
public real number(){
    int position=1;
    return num(position);
}
public real num(int position){
    if(getLookAhead().isHexValue()){
        return hex()*pow(16,-position)+num(position+1);
    }
    else if(getLookAhead().isEndOfStream()){
        return 0;
    }
    throw new Exception("Unexpected character");
}
public real hex(){
    return getValueForChar();
}
```

As one can see it seems to be very intuitive to create such a parser. It might be a suitable solution for small problems.
Creating a L-attributed LR(1) is much more complicated, because such parsers do not explicitly 'expand' grammar productions as LL(1) parsers do. Additionally there might be unresolvable conflicts between different items in a parser state on how to compute inherited attributes of another item. So we do not discuss LR attributed grammars here. It just should be said that a SLR(1),LR(1) or LALR(1) grammar possibly might not be L-attributable!

S-attributed grammars
S-attributed grammars are a subclass of L-attributed grammars containing synthesised attributes only. Hence there are no more restrictions concerning inherited attributes. So a S-attributed LL(1) grammar can always be called LL-attributed as well as a S-attributed LR(1) can always be called LR-attributed.

Ordered attribute grammars
This type of attribute grammars can be evaluated using a statically precomputed total order for the attributes of every grammar symbol of an AG. An evaluator visits every node in an abstract syntax tree multiple times. During a visit, at least one attribute of a node has to be evaluated. A *visit oriented evaluator* enters a node in the tree after it has computed one or more dependent values in the upper tree. Then it computes some values of attributes in that node which might be important for lower nodes and descents to visit some of the lower nodes. This is called a sequence of visits. Such a sequence can be determined by regarding the attribute dependency graphs of grammar symbols. The set $s(X)$ from the above cycle test can be used. Unfortunately $s(X)$ only contains attribute flow from inherited attributes of a grammar symbol X to synthesised attributes of X. So it only has information about the attribute flow through subtrees in which X is label of the root. For computing a complete attribute dependency information $R(X)$ one has to consider the upper attribute flow $t(x)$ too. The computation of $t(X)$ is described in [2] and will not be regarded here. $R(X)$ is defined as $R(x) = s(X) \cup t(X)$. Figure 7 shows $R(NT)$ for a hypothetical non-terminal NT with attributes $INH_{NT} = \{x\}$ and $SYN_{NT} = \{y, z\}$.
Each i-th visit Vi_X of nodes N with $X = label(N)$ consists of a set of inherited attributes and a set of depending synthesised attributes. These attributes will be evaluated during a visit by computing the inherited ones first. For instance

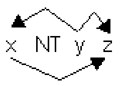

Figure 7: $R(X)$ for a non-terminal NT

$V1_{NT}$ of our short example would contain $V1_{NT}.INH = \emptyset$ because y in $V1_{NT}.SYN = \{y\}$ does not depend on any inherited attribute of NT. After the second visit $V2$ with $V2_{NT}.INH = \{x\}$ and $V2_{NT}.SYN = \{z\}$ the attributes of NT are completely evaluated.
Typically visit oriented evaluators are implemented as a recursive procedure on the nodes of the AST. Note that it might not be possible to compute a sequence of visits for a grammar out of the $R(X)$. Unfortunately this can not easily be checked,e.g. it can not be checked by introducing some syntactical restrictions as L-attributed grammars do.

5. CONCLUSION

Attribute grammars seem to be a good choice for defining the static semantics of a context free language. The concept of inherited attributes and synthesised attributes is quite simple and easily applicable.
Attribute evaluators can be generated automatically, but not all methods can be used for that. For instance ordered attribute grammars and LR-attributed grammars introduce restrictions which are not only of syntactical kind. Hence those methods might be difficult to use for developers who do not have in-depth knowledge in compiler construction. So in most cases and for small maybe domain oriented languages it would be better to use more simpler methods like the demand-driven approach or LL-attributed grammars.

6. REFERENCES

[1] Donald E. Knuth: *Semantics of Context-Free Languages,* Mathematical Systems Theory Vol. 2, No. 2 (1968)

[2] R. Wilhelm, D. Maurer: *Uebersetzerbau,* ISBN 3-540-61692-6 (Springer, 1997)

[3] Lothar Schmitz: *Syntaxbasierte Programmierwerkzeuge,* out of print (Teubner, 1995)

[4] Donald E. Knuth: *The genesis of attribute grammars,* Proceedings of the international conference on Attribute grammars and their applications, 112 (1990)

The JastAdd Semantic Specification Tool

Christoff Bürger
Fakultät Informatik, Technische Universität Dresden
Dresden, Germany
Christoff.Buerger@gmx.net

ABSTRACT

Attribute grammars are a common way to specify the semantics of formal languages. They are very powerful but their evaluation and especial specification tend to become confusing as the semantic specifications have to meet syntactical necessities and different semantical parts are blended. If they are only used to specify the semantic of a language on paper and the implementation is done imperative its even harder to recognize, track and understand semantic evaluations in the final implementation covering syntactical and semantical analyses mixed together. Attribute grammar tools try to solve this problem as they assist the user to specify semantics in an easy way and distinct semantic modules. One of this tools is JastAdd, which will be introduced in this text.

JastAdd is a modern and very powerful tool to specify semantics with attribute grammars. It extends the basics introduced by Knuth in 1967 in several ways. It supports references to attributes far away in the parsing tree so its a referenced attribute grammar tool (RAG). Circular attribute evaluation dependencies are possible and can be evaluated if they are terminating confirming fix point semantics. This supports recursive evaluation specifications. Parsing trees can be rewritten based on declarative graph rewriting rules before and during attribute evaluation. Parameterized attributes are possible. Last but not least semantic specifications can be done both ways imperative as well as declarative. All features are embedded in an object oriented design. This paper examines all the functionalities above, their specification and usage, advantages and disadvantages and introduce into the JastAdd system in a formal way.

Categories and Subject Descriptors

F. Theory of Computation [**F.3 Logics And Meanings Of Programs**]: F.3.2 Semantics of Programming Languages; F. Theory of Computation [**F.4 Mathematical Logic And Formal Languages**]: F.4.2 Grammars and Other Rewriting Systems; D. Software [**D.2. Software Engineering**]: D.2.13 Reusable Software

General Terms

semantic generator

Keywords

semantic specification, semantic evaluation, attribute grammar, circular attribute grammar, referenced attribute grammar, compiler generator, aspect oriented programming, object oriented programming

1. INTRODUCTION

The task to transform a most times text based input, which confirms against a specification into some well defined output is a common task in computer science. If the output confirms the input specification of another translator or is an executable program or program parts such translators are well known as compilers. But compilers are only a part of the translator hierarchy, like pretty printer, interpreter, filter etc.

Always a syntax specification for the input exists, most times its a context free grammar. A lot of work and research has been done to develop techniques to read the input, check it against its syntax specification and represent it in some structured way enriched with input informations, so that it can be further analysed without to worry about its syntactical correctness. This is done by lexers which recognise the inputs atomic elements, also called tokens and parsers which take a token stream and construct a parsing tree, representing the order of specification rules applied to create this input as well as the token informations associated with each applied rule. There are well known theories about language specifications and how to check in an acceptable time if an input word is part of a language, like LL(k), LR(k), SLR(k) and LALR(k). It is possible to implement generators which are taking a context free language specification and automatically create lexers and parsers for it. The output of the generated parser is most times a parsing tree, where nodes represent non terminals and leaves terminals of the grammar. The children of each node are the right sides of a grammar rule for this non terminal. The terminal leaves contain the associated input token informations. So the complete syntactical process can be easy specified and syntactical analyse tools generated.

But every syntax is nothing without some meaning, its associated semantic. To specify, analyse and apply the semantic of a language is a much harder task. A common way to do so are attribute grammars. They allow to overcome the gap between a context free syntax and its context sensitive semantic, as well as a formal specified way how to describe the grammars semantic. The formal theory behind leads to techniques for checking if an input word confirms the languages semantic rules and to evaluate its semantic if possible. The following text will introduce JastAdd [1] , an attribute grammar based semantic tool, which can be used to specify the semantic of a context free grammar as well as to compute an input words semantic. The basic input for its semantic evaluator are parsing trees, so a parser generator can be used to create a parser for a given context free grammar and the generated parser computes the parsing trees of input words.

In the following we will show JastAdds features, their usage and specification. To do so we will start with a short overview about JastAdd's design goals, then showing its features and extensions for attribute grammars concluding into a summary how JastAdd reaches its design goals. After doing so we can examine in detail how JastAdd specifications look like. At least we will give a short summary and subjective conclusion about JastAdd.

2. ATTRIBUTE GRAMMAR TOOLS GOALS AND JASTADDS FEATURES

2.1 Attribute grammar tools design goals

To specify and implement the semantic for a given context free syntax is a time consuming and not easy task. To support language developers and guide them through the development process of attribute grammars main guidelines have been developed. It is common to do the following steps :

1. Identification of semantic categories. This are all values or properties of interest gathered from tokens or computed based on other such values. Each necessary value is associated with an attribute and can be a synthetic or inherit ones.
2. Association of semantic categories with syntactic categories, so which syntactical symbol (most times non terminals) has to have which attributes. As information has to flow through the parsing tree there are often temporary attributes with copy rules.
3. Checking of each context and constructing its local evaluation graph. Every grammar rule is a context and its left sided synthetic attributes have to be evaluated in a given context as well as its right sided inherit ones. If a grammar rule is represented as sub tree, an evaluation graph has to be constructed, showing on which other attributes a given attribute depends or if its value is an atomic one (taken from a token or statically constant).
4. Specifying concrete evaluation functions.

[1]The JastAdd attribute grammar tool had been developed at the Computer Department of Lund University, Sweden. Its predecessor tool was the referenced attribute grammar (RAG) tool APPLAB (APPlication language LABoratory) developed at Lund University as well. JastAdds first final version was ready for public in 2003. This text is based on the version released at 15. Septemper 2006.

It has been discovered that there are a lot of common semantic tasks specified for most languages like checking if an identifier is declared, computing declaration levels, checking type safety, identifying parameter lists, evaluating mathematical expressions and many others. It would be a good idea to save development time and resources if those tasks are specified in an universal way, so their solutions can be reused. This covers common specifications as well as evaluating functions.

Another overhead in the development of attribute grammars are above mentioned copy rules. They have to be specified but do nothing useful at all instead passing attributes from one node to another. Additionally they waste time and memory while attribute evaluation. The memory wastage can be solved if pointers are used. But RAGs, referenced attribute grammars, are a much better way. They allow to link attributes of far parts in the parsing tree together, so a node can get needed attributes for its attribute evaluation directly. The needed attributes don't have to be passed along the parsing tree.

"Never change a running system" may be a nice statement but we all know that systems have to be expanded and developed further. There may be a need to expand an existing language with additional functionalities or just some syntactical sugar. It would be great if those additions can be added separated from already existing language specifications and implementations and if its easy to extend a given language with additional syntax and the associated semantic.

We already mentioned reuse a lot but we can even ask for more, the possibility to mix existing semantic components to build or extend a new language processing systems like compilers. It is a different if some specification or code fragment can be used in several projects or if some kind of component model is supported to mix and assemble software systems in a well defined way.

Another problem are the evaluation functions for attributes. Often recursive like specifications are needed or are much easier to do than iterative ones. But recursive attribute evaluations means, that the attribute grammar is circular. Another annoying point is, that even the easiest evaluation functionalities have to be implemented. It would be good, if a declarative way to specify attribute dependencies, as know from specifications on paper, which automatically leads to the implementation of the specified evaluations is possible. Also the specification languages and implementation languages for evaluation functionalities should look identical, so users don't have to learn several syntaxes. If specification and implementation could be easy exchanged and all the work is done in one language, some kind of meta specification for attribute grammars can be supported as known from composition systems, where the composition recipe language is the same as the language in which the components are written and recipes can be composed from other recipes (meta composition). The languages for specification and implementation of the attribute grammars should at least look familiar with well known programming or specification languages.

Often a languages context free grammar has to be changed in some way, so that it can be syntactical analysed. For example it needs to be factorized for LL analyses. As attribute evaluations are associated with grammar rules, the semantic analyse is influenced by syntactical necessities. If the grammar can be specified more easy for semantic analyse it would be good if a translation of parsing trees from the parser into easier parsing trees for the semantic analyse, so called abstract syntax treess (AST), can be done.

2.2 JastAdds design goals

2.2.1 *Main design goals and their realisation*

JastAdd has the following main design goals, which all focus around some kind of reuse, so that it is possible just to add additional semantics and their evaluation by expanding existing specifications or by using already existing ones :

1. Separation of different semantic tasks / aspects and reuse of already specified semantic aspects.
2. Straight forward specifications of attribute evaluation rules focusing on the problems semantic and not their exactly solving implementation. This means, that attribute evaluation specifications have to be easy to write and straight forward to implement without the need to translate from intuitive specifications on paper into real code implementations.
3. Abstracting from syntactical necessities of parsing trees and focusing only on semantics of a grammar for a language.
4. The user shouldn't have to worry how attribute evaluation is done and if attribute dependencies are evaluated in the right order.

In the following we will discuss how JastAdd realises these goals.

1) JastAdd supports the reuse of semantic specifications and evaluation functions by a module based system. The modules specifying the evaluation of attributes can contain declarative as well as imperative specifications [2]. Every module specifies some semantic aspect, so the modules are called aspects. Systems can be easy expanded by just adding already implemented semantic modules. But these modules are not linked together like modules known from existing programming languages like C, C++ or Java classes / packages. The different modules are weaved together by an aspect weaver. The aspect weaving of different semantic modules allows to group the same semantic aspect alone into one module and to distinguish different semantic tasks in a given language, helping to keep overview of the specifications.

2) JastAdd module specifications can be done in a declarative way as well as attribute evaluations, allowing much smaller specifications, their automatically implementation and easier evaluation changes. To make the declarative, but also possible imperative specifications smaller and to focus on problem solving, referenced attributes are supported. There is no need for copy rules. Circular declarative specifications are possible, resulting in smaller, easier to understand and implement attribute evaluations. To make attribute grammar specifications as familiar as possible JastAdd uses a Java similar syntax and an object oriented design. But its necessary to mention that even the used specification language is similar to Java, its still a proprietary one. Also the meta specifications mentioned in 2.1 is not real supported and possible, as there is no kind of component model. JastAdd is still only aspect based. Imperative specifications are done in the Java programming language, so they support some kind of meta specification, but are limited to Java's component model. They also loose the advantages of declarative specifications, that's the easy way to develop them. Of course their real code implementation is done manually, as these specifications are real Java code. Imperative and declarative specifications can be mixed, so the JastAdd developer call their evaluation specifications intertype declarations.

3) Abstracting from syntactical necessities can be done by declarative graph (parsing tree) rewriting rules, which make parsing trees easier for semantic analyse. To focus on semantics only, the context free grammar used for syntactical analyse can be changed for an easier one. To do so, an abstract grammar specification is supported, which has to be specified for every attribute grammar realised with JastAdd. Parsing trees from the parser have to confirm this one. An automatic mechanism to create confirming parsing trees is not supported, but it is most times easy to create those trees by introducing small translation actions into the parser generator specifications. A method called syntaxCheck, which checks if a given parsing tree confirms the specified abstract semantic is supported. There is no problem to use any parser generator to generate a parser for the syntactical analyse. Its only necessary, that the parser generates abstract parsing trees, so it has to generate a Java class object composition. That's why Java parser generators are easier to glue with JastAdd.

4) Attribute evaluation specifications, even in different aspect modules, can be specified without the need to worry about dependencies between them and the weaving of them, as well as the evaluation of given parsing trees is done automatically. Rewriting rules and attribute evaluation rules are computed in the right order by JastAdd. Rewriting rules are applied as soon as they can. No evaluation is done, if the graph can be rewritten, until the rewriting has been done. The JastAdd attribute evaluator guarantees that always an attribute is accessed its correct value is computed before. Additional the attribute evaluator is demand driven and attribute values are only computed once and cached afterwards. If an attribute has already been accessed its cached result is returned instead computing it another time. But there are also a few drawbacks. Firstly demand driven evaluation also means that not all attributes have to be computed. Attributes not needed for the main results (the synthetic attributes at the root) and not accessed by their access method will not be computed at all. On the other hand attributes may be accessed several times, even they are only computed once. Most times this is a good thing, but if side effects are expected while attribute evaluation, because the user implemented them in evaluation functions, its possible

[2]The JastAdd designer call this intertype declarations.

```
A ->B C          A ::= B C;
                 abstract B;
B ->token1       B1 :  B ::= <token1>;
B ->C token2     B2 :  B ::= C <token2>;
C ->token3       C ::= <token3>;
```

Figure 1: Grammar rules and their corresponding AST specification

```
class A {
     B getB() {...};
     C getC() {...};
}

abstract class B {
}

class B1 extends B {
     String getToken1() {...};
}

class B2 extends B {
     C getC() {...};
     String getToken2() {...};
}

class C {
     String getToken3() {...};
}
```

Figure 2: The generated AST-classes

that they do not occur, as the evaluation function was never called or they may occur several times, if the side effects are triggered while accessing the attribute. Also its not possible to tell the evaluation order statically. JastAdd guarantees that the evaluation will be done in a right order, but there may exist several orders to compute the semantics of a parsing tree. At all this are no big draw backs, as the semantic of a programming language, specified by an attribute grammar should never relay on side effects. Additionally this are no restrictions for possible attribute grammars. Its still possible to specify and evaluate any kind of correct attribute grammar.

2.2.2 *Implementation of basic attribute grammar concepts*

Above we examined, which JastAdd feature is essential to reach which design goal, but we didn't check until now how JastAdd realises the implementation of the basic attribute grammar concepts as introduced by Knuth in 1967! Of course JastAdd implements the basic attribute grammar concepts, that's synthetic and inherit attributes and attribute evaluation functions realising information flow along the parsing tree and computation of attributes by equivalences. The basic concepts are implemented in an object oriented way.

Attributes are declared in abstract syntax tree classes (AST-classes) realizing the association of semantic categories to syntactical ones. Their values are computed, as known, by equations. Those equations are using AST-class methods if they are complex. An AST-class represents a non terminal or an abstraction, if a non terminal has several rules. For each non terminal rule exists a class implementation of the class abstraction for this non terminal. Every AST-class object is a node in the abstract parsing tree, so abstract parsing trees are represented by object oriented class hierarchies in a composition. The access to attributes is done by a method named like the attribute, with the guarantee that this method will always return the correct attribute value. This object oriented design has all the advantages of object oriented modelling, the abstraction and concretion of relationships by a "is a" relationship, resulting in the possibility to :

- Swap concrete implementations of syntactical categories.
- Outsourcing of general and default semantics in super classes. Subclasses can reuse this semantics (semantic evaluation methods) as well as overwrite them.
- Subclasses automatically inherit semantic categories of super classes, that means, they inherit the attributes associated with super classes.

2.2.3 *Extensions of the basic attribute grammar concepts and summary of JastAdds features*

JastAdd extends the basic attribute grammar concepts by :

- reference valued attributes (RAG)
- parameterized attributes
- circular defined attributes
- non terminal attributes and rewriting rules for AST's

Most of these concepts have been discussed in the part "2.2.1. Main design goals and their realisation". We will only mention additional informations here.

RAG's are not more powerful than basic attribute grammars. There specifications are just smaller, so the specifications are easier to implement and there is no need for copy rules. That saves memory and time while evaluation. The object oriented design together with the ability to use referenced attributes also makes the collecting of informations into data structures much easier. Its no hard task to imagine that the Java class libraries well designed *Collections* can be used to create collections of references of attributes scattered across the parsing tree. Referenced attributes allow to break the rule, that the attribute flow has to follow the parsing trees structure.

Parameterized attributes are attributes with some functionality, so attributes behave like methods. This together with referenced attributes is a nice feature making specifications much more easy to implement and understand. It also allows to model more object oriented. But at the end its just syntactical sugar and doesn't introduce additional power.

Table 1: AG tool design goals and their realisation in JastAdd

Tool feature	Feature realisation in JastAdd
familiar specification languages	Java like specifications; object oriented modelling
easy implementation of attribute evaluation functions	declarative specifications; object oriented design; parameterized attributes; referenced attributes
easy attribute handling and collection	referenced attributes; correct evaluation order
preparation of the syntactical parsing tree for semantic analyse focusing on semantic aspects only	abstract grammar and AST specification; graph rewriting
reusability of implementations for semantic tasks	declarative / imperative semantic modules (aspects); independency of semantic modules; aspect oriented weaving to implement different semantic modules at the same time; no need to worry about attribute declaration, their usage in different modules and evaluation
manageable evaluation	demand driven evaluator; caching evaluated attributes

Circular defined attributes together with the ability to specify specifications in a declarative way make life much more easy. They realise iterative evaluations without the need to know how termination is done. Just let the fix point theory solve this problem. Of course if there's no fix point the evaluation will not terminate, but that's exactly the semantic of a not terminating recursion : Its undefined. Every circular defined attribute evaluation can be translated into an iterative evaluation function, so no additional power is introduced.

It is possible to extend the AST during semantic analyse based on declarative rules with additional AST nodes. The JastAdd developer call this non terminal attributes and distinguish it from the JastAdd rewriting feature even it is some special kind of it. Declarative rewriting rules allow to change the AST in any way. The rewriting is done automatically as soon as possible and rewriting rules can be based on each other, so one rewriting step may lead to another. As mentioned in "2.2.1. Main design goals and their realisation" rewriting allows to prepare the parsing tree from the parser for semantic analyse.

Let us summary how JastAdd reaches it design goals of reuse, extensibility, easy implementation and familiarity. There are three main ideas :

1. Object oriented design : AST's are implemented by object oriented Java class hierarchies. AST-class objects represent AST nodes.

2. Static aspects : Extension of AST's with additional features without the need to change their code manually. Extensions are done automatically by weaving semantic aspects into the AST-classes (the aspects are already implemented in modules).

3. Declarative computations : JastAdd supports declarative specifications to describe the semantic of a language. Attributes, rewriting rules as well as attribute evaluation functions can be specified declarative. The user doesn't need to worry in which order attributes or rewriting rules of even different aspect modules are evaluated for concrete AST's. This allows to split the program in independent aspect modules. Declarative specifications are also much smaller and easier to implement than imperative ones. Of course JastAdd still allows imperative specifications done in Java.

3. IMPLEMENTING JASTADD SPECIFICATIONS AND USING JASTADD

Every JastAdd specification relays on two main parts : [3]

1. AST specification (*.ast)

2. Declarative / imperative semantic specifications (*.jadd / *.jrag). Declarative and imperative specifications can be mixed in the same module and are always associated with an AST-class specified in the AST specification, thats why the JastAdd developers call their attribute evaluation specifications intertype declarations. The JastAdd developer introduced *.jadd files for imperative apsects and *.jrag for declarative ones. Both can contain the same kind of specifications. But it is good design to distinguishe between declarative and imperative modules, where declarative modules add attributes, equations, and rewrites to the AST classes and imperative modules add ordinary fields and methods to the AST classes.

The AST specification is always needed. Declarative or imperative semantic specifications don't have to occur [4]. The result of a generated semantic evaluator for a given language are the AST-classes.

[3]The semantic modules are parts of the semantic specifications. So we talk sometimes about semantic modules and sometimes about specifications, but both is the same. Only specifications on paper may be different from the modules. Also the modules are called semantic aspects, as its the idea, that every module specifies one semantic aspect.

[4]Its often useful to specify base attribute structures and evaluations declarative and let imperative operations work on them.

The interface for the results of the syntactical parser is the AST specification. All parsing trees the parser delivers have to confirm the AST specification and are a composition of objects of this specification. They are a composition of AST-class objects, the parser generated. So AST-classes and this way parsing trees are type save. At all the AST specification represents an abstract grammar and allows independendt from the underlaying parser the semantic specification and implementation.

The AST-classes code is the semantic of the language implemented. The final AST-classes are generated by weaving the modules (aspects) together into the skeletons of the AST-classes generated out of the AST specification. Attributes are specified in the semantic modules. [5]

We will now examine each of the specifications in detail.

3.1 AST specifications

AST specifications are written in *.ast files. They represent an abstract grammar. To explain them we will introduce some automatic translation process, which translates context free grammars and their rules into AST specifications. This process can be used to translate the context free grammar used for a parser generator into an AST specification, so that the integration of a parser with JastAdd can be done, if the parser just generates every time it reduces with a grammar rule the corresponding AST-class object and sets its references [6]. Of course, in reality an AST specification abstracts form the context free grammar as syntactical necessities are not needed for semantic analyse.

For every non terminal grammar symbol, which has several rules

$$r_1 = A- > \alpha, \ldots, r_n = A- > \beta$$

an abstract non terminal class A is defined and n concrete non terminal classes [7] X are defined, which implement the abstract A. Every of the n X implements another of the rules [8]

$$r_1, \ldots, r_n.$$

The productions are translated into a sequence of abstract non terminal classes, non terminal classes and token. The translated sequence of a production is specified together with the corresponding non terminal class specification.

[5]To be exactly the semantic aspects are translated into ordinary Java code, which is than weaved into the AST-class skeletons, which had been generated themself before, by translation the AST specification into ordinary Java code.
[6]So the parser generates the abstract syntax tree, which is in JastAdd a Java object composition of AST-class objects
[7]We will call concrete non terminal classes, which are implementations of abstract non terminal classes in the following just non terminal class. The abstract non terminal classes will be called by full name. It is necessary to take naming seriously. Firstly we have the context free grammar, just called grammar and its grammar rules, just called rules or productions. Secondly we have the corresponding specification for this grammar, consisting of abstract non terminal classes, non terminal classes and token.
[8]We call the rules also productions. If we do so, we mean the right side of a rule.

Lets examine how the productions are translated into sequences :

- For every token in the production a part *<TokenName>* is generated in the sequence at the position, which corresponds the one of the token in the production. *TokenName* is the name of the token in the production.
- For every non terminal in the production a part in the sequence called like the grammars non terminal corresponding abstract non terminal class is generated at the position, which corresponds the one of the non terminal in the production.

If the non terminal in the production has only one rule, no abstract non terminal class is needed, instead an non terminal class with the grammar's non terminal name is specified and used every where, where usual the abstract non terminal class would be used.

The following formalism are used in the specification, to explain the operations mentioned above.

1. *::=* ... The equal symbol, which the production translation follows. On the Left side is a non terminal class name or a construct like 3.
2. *abstract NonTerminalClassName;* ... Specifies an abstract non terminal class.
3. *NewNonTerminalClassName : AbstractNonTerminalClassName* ... The *NewNonTerminalClassName* is a non terminal class implementation of the abstract non terminal class *AbstractNonTerminalClassName* and represents one of the rules for the non terminal *AbstractNonTerminalClassName.*

The syntax for a non terminal class specification is :

$$NonTerminalClassName ::= ProductionTranslation;$$

where *NonTerminalClassName* is a construct like 3. or a new non terminal class name.

We recommend to check "Figure 1 : Grammar rules and their corresponding AST specification" for a short example.

Of course the specification scheme above is expanded with syntactical sugar. There are four possible main subcomponents [9] for non terminal classes :

1. list ... *class : superclass ::= superclass*;*
2. optional ... *class ::= [superclass];*
3. token ... *class ::= <token>;*
4. aggregate ... *class ::= class2 class3 class class4;* An aggregate may have any production sequence containing other non terminal classes / abstract non terminal classes or token.

[9]productions

This syntactical sugar helps, to make AST specifications as short and easy as possible and to become rid of syntactical parsing tree components not needed.

The additional operations in AST specifications do the following :

- *NonTerminalClassName** ... Specifies, that the abstract non terminal class *NonTerminalClassName* can occur several times. *A : B ::= B*;* and *C : B ::= <t>* equals *B ->t B* and *B ->ø* with *ø* is the empty word.
- *[NonTerminalClassName]* ... Specifies, that the (abstract) non terminal class *NonTerminalClassName* can occur but doesn't have to do so. *A ::= [N]* equals *A ->N* and *A ->ø* with *ø* is the empty word.

The AST specification is automatically translated into the AST classes skeletons. We are not going to describe these translation process here in detail. Instead we just name a few properties of the generated AST classes. Keep in mind, that the AST specification only generates the AST class skeletons, without semantic module weaving into the AST classes.

Non terminal classes inherit all the methods and attributes implemented in abstract non terminal classes they implement as known from object oriented programming. Every non terminal class also knows its sub components. This means it has access to the production elements and offers type save access methods allowing typed traversal of AST parsing trees. This access methods are generated automatically and they return the type of the AST class of the production element they correspond to. Their signature is *ClassTypeOfSubcomponent.getSumcomponentname()* [10]. It may be, that some terminal or non terminal occurs several times in a production. In this case the symbol must be named. This is done by writing *UniqueNameInProduction:* before the symbol, where *UniqueNameInProduction* is the new unique name for the symbol. The grammar production *A ->S S S* will be handeled like *A ->S1 S2 S2* if the AST specification for it is something like *A ::= S1:S S2:S S3:S;*. Of course the naming of symbols doesn't change the abstract grammar at all. If it is unknown how often a subcomponent may occur [11], a method *int getNumSymbolname()* is offered, which returns how often the symbol occured. The access method for the sub components has an additional parameter of type *int*, telling which sub component to access. The method signature in this case is *Symbol getSymbol(int number)*. If a production contains token, for every token contained a method *String getTokenname()* is generated, which returns the token's lexem in the input stream the parser parsed. If a production element is optional, it has an additional method *boolean hasSymbolname()*, which returns *true*, if the symbol occurred while parsing and *false* otherwise.

3.2 Semantic module specifications

[10]Most times the sub component class is the *Sumcomponentname*. This may only be different if a sub component got renamed.
[11] * operator

The semantic of an attribute grammar is specified by associating attributes with grammar symbols and defining their evaluation functions. In JastAdd attributes have to be bound to AST-classes and evaluation functions are specified as Java expressions or methods. The semantic of a complete language can be separated in different semantic aspects. We call the semantic aspects just aspects. Every aspect has its own module, that's a *.jadd or *.jarg file and is a part of the specification for the semantic of an abstract grammar. Aspects can be developed independently from each other and don't have to know about each other. This also means, that an aspect may change, evaluate and use attributes and evaluation functions defined in another aspect.

An aspect may consist of attribute declarations, declarative or imperative evaluation functions and declarative rewriting rules. Every Java package can be imported. A JastAdd project consists of any number of different aspects. All this aspect modules will be automatically translated into ordinary Java code and than be weaved into the Java AST-class skeletons generated out of the AST specification.

Let us now examine step by step, in a recursive way how the syntax of semantic aspects look like and what the syntactical constructs do. An aspect module has the following structure :

```
JavaImportStatements

aspect firstAspectName {
     //aspect body for first aspect
}
aspect secondAspectName {
     //aspect body for second aspect
}

...

aspect lastAspectName {
     //aspect body
}
```

JavaImportStatements can be imports of any number of Java packages used in the aspect module. The aspect module itself can contain as many aspects as the user desires but most times it will be one aspect. Every aspect is specified by his aspect body.

The aspect body may contain usual Java class and interface implementations, as well as static attributes and methods. More interesting is, how attributes are bound to AST-classes and how their evaluation is specified. We have to distinguish between declarative and imperative parts in the aspect body, which can be mixed in any order. Let us start to look at the declarative way to specify semantics.

Attributes are distinguished in inherit and synthetic attributes. This is shown by the two declarative keywords *inh* and *syn*. To bind an attribute at some AST-class and all its subclasses the notation *ASTClassName.attributeName(AttributeParameterList)* is used. The *AttributeParameterList* is just any kind of parameter list as known from Java. Of course an attribute has some type. This can be any Java standard type or even a self designed class. Additional an attribute may be computed lazy, that means its value will be cached and only computed once. To do so the keyword *lazy* can be used. A short example of attribute declarations and bindings :

```
syn lazy String RomanNumeral.value();
syn int ArabicNumeral.value();
syn boolean Block.
     localEnvironment(String identName);
inh lazy A.x();
```

In attribute grammars, the evaluation of attributes is bound to contexts. A context is a grammar rule. Such a context can be displayed as a graph of depth one, where the left side of the grammar rule is the parent node and the grammar production symbols are the child nodes, occurring in the same order. As the attributes are bound to grammar symbols, the question arises, which attributes have to be evaluated in a context. All the synthetic attributes of the parent node have to be evaluated and all the inherit attributes of the children. [12]

Attributes declared and bound to grammar symbols have to be computed. To specify an attribute evaluation in a declarative way the keyword *eq* followed by the context, the attribute name and the evaluation description is used. The context for a synthetic attribute is just an AST-class, as not abstract AST-classes represent a non terminal and one of the productions for this non terminal and abstract AST-classes are just generalizations of concepts specified for all their implementations. The evaluation description can be a = followed by a Java expression or a Java block containing a return statement at the end, which returns an object of the same type the attribute evaluated is. The second method can be used, if the attribute evaluation is more complex, but its also a more imperative one.

```
eq ASTClass.AttributeOfASTClass =
     Java-Expression;
eq ASTClass.AttributeOfASTClass {
     //Java code
     return AttributeOfASTClassType;
}
```

The context for a inherit attribute is an AST-classes sub component. So declarative attribute evaluation specifications for inherit attributes look like :

```
eq ASTClass.getSubcomponent.
     AttributeOfSubComponentASTClass =
     Java-Expression;
eq ASTClass.getSubcomponent.
     AttributeOfSubComponentASTClass {
     //Java code
     return AttributeOfSubComponentASTClassType;
}
```

[12]For german readers : The attributes to calculate are the innen Attribute.

Of course declaration and evaluation of attributes can be specified in one step :

```
syn lazy Type ASTClass.attribute() =
     Java-expression;
inh Type ASTClass.ASTClassSubcomponent.
     attribute() {
     //Java code
     return Type;
}
```

Also refining semantics defined in other aspect modules is possible. This is done by the keyword *refine*. So methods can be overwritten. Of course the refined method can be called, similar to the Java keyword *super*. The syntax is :

```
aspect B {
     refine OtherAspect void ASTClass.method() {
          //Java code
          OtherAspect.ASTClass.method(); //super
          //Java code
     }
}
```

Circular attribute evaluation can be specified by using the keyword *circular* followed by a starting value in *[* and *]* brackets while attribute declaration :

```
syn Type ASTClass.
     circularAttributeName(ParameterList)
     circular [starting value of type Type];
eq ASTClass.circularAttributeName(Type value) =
     direct/indirect recursive Java-Expression;
```

Graph rewriting is done in a declarative manner. The keyword *rewrite* is used followed by the AST-class node A in the parsing tree to rewrite. In the following block possible rewriting processes for the specified AST-class A are listed, with the conditions when they are applied, as well as the AST-class the original AST-class A node will be replaced with. Every rewriting process consists of a block with Java statements, which are mainly used to rewrite the sub components of the original AST-class A node. Such a block has to return the new AST-class, A will be replaced with. The keyword *to NewASTClassNode* is used to specify which AST-class type A is rewritten to.

```
rewrite A {
     when (Java-condition1)
     to B {
          //Java code
          return exp1;
     }
     when (Java-condition2)
     to C {
        //Java code
        return exp2;
    }
}
```

Keep in mind, that rewriting rules are applied as soon as possible. So if parsing trees are traversed manually, all rewriting processes will already be applied. After a rewriting process is done, it is checked if another rewriting process can be applied. The rewriting conditions are checked in the following order :

1. conditions in superclasses are evaluated before conditions in subclasses
2. conditions within an aspect file are evaluated in lexical order
3. conditions in different aspect files are evaluated in the order the files are weaved into the AST-class skeletons. That's the order the files are listed in the jastadd command.

It is also possible to do unconditional rewrites. In this case the *condition* keyword and its Java-condition is left.

Imperative specifications are based on the fact, that the parsing tree is a composition of AST-classes, where every AST-class knows its sub components and its parent node, if it isn't the root. They are done, by implementing imperative attribute evaluation methods directly, like the attribute access methods and most times using the access methods or imperative evaluation methods for subcomponents and parent nodes. The implemented methods can have any Java visibility keyword like *private* and *public*. The visibility is interpreted in the way that an aspect corresponds a class. The imperative attribute evaluation methods may have any kind of parameter list. They are associated with AST-classes in the same way declarative specifications are, by writing *ASTClass.methodName*. Imperative specifications have the following structure :

```
visibility Type ASTClass.methodName(ParameterList) {
    //Java code
    return Type;
}
```

As imperative specifications are based on traversing the parsing tree (the AST-class composition) they can also be implemented by using the visitor pattern. But the visitor pattern has a few disadvantages compared to declarative specifications. The visit method has only one return value and its parameter list is always the same. Its not possible to adjust the visit method for different contexts. Also the attributes associated with parsing tree nodes can not just be bound into the visit method. If aspect oriented weaving and declarative specifications are used this drawbacks don't occur.

4. CONCLUSIONS[13]

[13]THE FOLLOWING PART IS A SUBJECTIV IMPRESSION.

The JastAdd system provides a modern way to implement semantic specifications and evaluations based on attribute grammars for languages. It is powerful enough to solve all the common problems and easy to use. The specifications look very familiar and the tool at all doesn't seem cryptic. Its easy to get started with and while using it, additional features introduced will be explored, without to worry about them if not used at all. Of course a few of the additional features have their drawbacks. So its not easy to debug not terminating evaluations in bigger projects. Are it not terminating declarative recursive attribute evaluations or a cycle of rewriting rules? Additional features and power also means more complexity to understand how to use the tool and introduces a more theoretical background. It doesn't mean (in the case of JastAdd) that the specifications become more complex. They are less complex in JastAdd and small, but they do a lot. JastAdd allows nearly to specify as known from paper or at least as close as possible. The advantages of a modern programming language like Java with its huge class library are added.

The advantages mentioned above are introduced in a natural way, just by using the existing Java environment and syntax. Referenced attributes are no hard task to implement and understand when Java is used, a language based on references. Object oriented design is common as well and Java only supports this design method. Additional nice development environments for Java exist like eclipse. The development of JastAdd specifications can be integrated into eclipse with a few configuration steps. The JastAdd developer not only made the implementation of JastAdd more easy by using existing Java solutions, they also made it more easy for new users to understand the system and to experiment around with it.

JastAdd fulfils its task to support semantic reuse and makes it easy. Aspect oriented weaving of semantic tasks is the way to go instead only module based imperative reuse. But it may be a bit better if a common aspect oriented system, which allows the full power of aspect oriented programming is used instead an own implementation. For example AspectJ could be used. The JastAdd developers already mentioned this and announce they are looking into this possibility.

The JastAdd developer focused on the tools main task : specifying and evaluating semantics using attribute grammars. There is no tool function not supporting this task. This is fine because the system is not going to become a big addle apple. There are a few attribute grammar tools out there, which try to do everything : lexing, parsing, evaluating, interpreting, modularization, specifying, prototyping and whatever you might think about. Well, they are just going to smash you, taking a lot of time to get started with, only to recognize you would do much better using specified tools for the different tasks.

At least JastAdd is an active project. New versions are still released. That's always a good sign and gives a good feeling about its future.

5. ACKNOWLEDGMENTS

This paper was written for the lecture "Methoden zur Spezifikation der Semantik von Programmiersprachen" during winter semester 2006 at TU-Dresden, Germany. You are free to copy it, as long, as its original realease and author are referenced.

6. REFERENCES

[1] Donald E. Knuth: *Semantics of Context-Free Languages,* California Institute of Technology, 1967

[2] Torbjörn Ekman: *Extensible Compiler Construction,* Department of Computer Science, Lund University, 2006

[3] Görel Hedin, Eva Magnusson: *JastAdd-an aspect oriented compiler construction system*, Department of Computer Science, Lund University, 2002

[4] Torbjörn Ekman, Görel Hedin: *Rewritable Reference Attributed Grammars*, Department of Computer Science, Lund University

[5] Lothar Schmitz: *Syntaxbasierte Programmierwerkzeuge,* B. G. Teubner, Stuttgart, 1995

[6] Sven Karol: *An Introduction To Attribute Grammars,* TU-Dresden, 2006

[7] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann: *Compilerbau Teil 1*, Oldenbourg Verlag München Wien, 2. Auflage, 1999

[8] Uwe Aßmann: *Invasive Software Composition*, Springer, 2003

APPENDIX

A. HEADINGS IN APPENDICES

A.1 Introduction

A.2 Attribute grammar tools goals and JastAdds features

A.2.1 Attribute grammar tools design goals

A.2.2 JastAdds design goals

A.2.2.1 Main design goals and their realisation

A.2.2.2 Implementation of basic attribute grammar concepts

A.2.2.3 Extensions of the basic attribute grammar concepts and summary of JastAdds features

A.3 Implementing JastAdd specifications and using JastAdd

A.3.1 AST specifications

A.3.2 Semantic module specifications

A.4 Conclusions

A.5 Acknowledgments

A.6 References

Advanced Seminar Softwareengineering - Denotational semantics

Klemens Muthmann
Tutor Steffen Zschaler

1. INTRODUCTION

This text gives some basic knowledge about *denotational semantics*. It is intended as a rough but hopefully broad overview of the topic.

While reading it, one might acquire the needed background knowledge. To get this knowledge it might be necessary to consult additional material. Everytime this is needed a hint will be given.

The reader of the text should definitely know about the concepts of *syntax*. He should have heard about *semantics*.

2. WHAT IS DENOTATIONAL SEMANTICS

There are three main formalisms for defining the semantics of *programing languages*. These formalisms are the *axiomatic semantics*, the *operational semantics* and the *denotational semantics*. The axiomatic and the operational semantics are also called definitional semantics. The denotational semantics is the one you will read about now. It tries to translate mathematical objects into programs.

2.1 What is it needed for?

Denotational semantics is needed by programmers and mostly by *compiler* and *compiler-compiler* builders. The semantics of many programming languages were and are only informally specified in natural language. Because of natural language being sometimes abigous, this leads to problems while implementing these languages. Formal notations, like the BNF-Notation, make the syntax of code easy to describe. Formal notations of the semantics would also make the proving of the correctness more easy. Denotational semantics gives a mathematical formalism to specify the meaning of syntax and thus solves these issues.

Allision (All89) describes briefly how to implement a *parser* for denotational semantics. He shows practical implementations of *compiler-compilers* for a subset of PASCAL and PROLOG.

Denotational semantics worth for programmers is limited since the rules are very complex applied to a specific program. It is however conceivable to run proves of programs. This way you can guarantee the correctness of critical programs or program parts and find difficult bugs in complex programs.

2.2 What is the meaning of denotational semantics

To get an impression of what denotational semantics is you have to analyze know what the two words 'denotational' and 'semantics' stand for.

2.2.1 Semantics

What someone wants if he talks about semantics is to assign a meaning to an otherwise meaningless expression. Lets give an example. I could say: “Das Ergebnis von vier plus vier ist acht.” and because I am german I would know the meaning of this sentence. But a person only speaking english needs me to give him the semantics which actually would be '4 + 4 = 8'. Of course this is only another notation and strictly speaking we still have syntax here. Semantics are what something means in your head or in the circuits of a computer but somewhere we need to make a point. This point should be made, when everyone who needs to, knows about what is meant by the expression. The rest I will leave to the philosophers.

2.2.2 Denotational

The word 'denotational' comes from 'to denote' but there is not much more you can take from this.

But what are they denoting? Well if you want to analyze the semantics of a computable program such a program in the simplest case takes some input, processes it and produces some output. If one of these three steps is missing you have a very boring program either doing nothing, producing always the same or not giving any information.

So you get - with some borrowing from axiomatic notation - something which Allison (All89) notes the following way:

$$\{I\}Program\{O\}$$

Well but if you rewrite this to $Program(I) = O$ you get a very familiar notation, which can be abstracted to $f(x) = y$. So a program is nothing else than an algorithm for a mathematical function. The problem with mathematical functions is that they are usually infinite mappings of input to output, which is not usable by computers.

To express that a program is only a partial mapping and not the whole function, which is expressed by its semantics, we can say that the total mathematical function denotes the partial function of the algorithm implemented by a program.

A program is a partial function denoted by a total function from input to output. Now you know why it is called denotational semantics.

This can be driven even further. A program consists of statements and expressions and the like. Each of these are also simple mappings or functions. They map from one state of the program, before the execution of the statement to another after the execution. What a state is is not important now and will be explained later. You should only understand that everything, that you write down while you are writing a program is - in terms of denotational semantics - a function and even functions are composed of other functions.

3. THE λ CALCULUS

Before delving now into the depths of denotational semantics you should know about another concept, very often used to write down its rules. As *meta language* for denotational semantics, the basics of the *lambda calculus* will be presented here. But note that lambda calculus is, though the most widely spread, only one meta language for denotational semantics.

Informally saying lambda calculus is a very simple but also very powerful method of writing down functions.

More formally spoken it is a formal model similar to *functional programming languages*. This relation is similar to that of the *Turing Machine*'s to *imperative programming languages*.

Well but what is a calculus? A calculus consists according (aut) to out of the following two parts:

- A *language* for the calculus, which means:
  - an *alphabet* of the basic symbols
  - a definition of the *well-formed expressions*
- A *deduction framework*, consisting of:
  - *axioms* out of the well formed expressions of the language (terms)
  - *deduction rules* for transforming expressions

The language formulated as *abstract syntax* in BNF looks the following way as presented in (Feh89):

$$\begin{aligned} \Lambda ::= & C| && (1)\\ & \mathrm{V}| && (2)\\ & \Lambda\ \Lambda| && (3)\\ & \lambda \mathrm{V}.\Lambda && (4) \end{aligned}$$

The C is an arbitrary constant (c,d,e. . .), V a variable $(x, y, z, \ldots)$ and Λ an arbitrary λ-*term* (M,N,L,. . .).

The term (4) is called *abstraction*. This abstraction can be understood the following way. The function

$$f = (2 + 2)^2$$

is the constant function with the value 16. It is very concrete because there is only one value as solution of the computation. To express a more general function which also represents this computation you simply would write:

$$f(x) = (x + x)^2$$

But for λ-calculus there is no difference between x and 2. They are only symbols without any meaning[1]. To do the abstraction step above you have to bind the desired variable to a λ. So that the term:

$$(x + x)^2$$

can be abstract the following way:

$$\lambda x.(x + x)^2$$
$$\text{or}$$
$$\lambda 2.(2 + 2)^x$$

Looks a little weird but you will get familiar with this soon.

The term (3) is called *application* and means that some argument is applied to a λ-term. For instance:

$$f2 \equiv (\lambda x.(x + x)^2)2$$

applies 2 to f. Now let us see some other examples of valid λ terms:

$$c \quad x \qquad (xc) \qquad (\lambda x.(xc)) \quad (y(\lambda x.(xc))) \quad ((\lambda v.(vc))y)$$

Such terms, which you can define yourselves, are the axioms of the calculus.

3.1 Currying

A problem you get with the basic λ-calculus is that you can not define functions of more than one variable.

Consider

$$plus(x, y) = x + y$$

which is only expressible with two nested λ-terms.

$$plus = \lambda x.(\lambda y.x + y)$$

But was does this mean? To calculate the sum of 2 and 3 with the function *plus* you can directly add the two numbers or you can construct a function $p1$ which returns a function $p2$ that adds a variable to a constant. Now you can call $p1$ with the argument 2 and get a function that adds something to 2. The second function called with 3 now adds 2 and 3 produces the desired output 5.

$$plus : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$$
$$p1 : \mathbb{N} \mapsto (\mathbb{N} \mapsto \mathbb{N})$$

$$p1(2) = p2(y) = 2 + y$$
$$p2(3) = 5$$

This concept is called *currying* after Haskel B. Curry and was invented by the mathematician Schönfinkel.

More formally spoken currying works the following way:

$$(A \times B) \mapsto C \equiv A \mapsto (B \mapsto C)$$

[1]To simplify notation we will assume that literals such as 2 will have the usual meaning but strictly spoken they are only meaningless symbols in λ calculus.

Such functions that do return other functions are called *higher-order functions*. They are seen very often while working with denotational semantics. But for the λ-calculus this is a little bit annoying so for simplicity we define:

$$\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M))) \equiv \lambda x_1\ x_2 \ldots \lambda x_n.M$$

3.2 Bound and free variables

λ-expressions contain two types of variables. One type is *bound* by λ's and the other is *free*. The free variables are like global ones in programming languages, while the bound are the parameters of a function.

Consider

$$(\lambda x.xy)x$$

where y is a free variable and x is both. Inside the brackets x is bound by λx and outside it is free because there is no λ to bind it.

Definition 1. *The Set of free variables FV(M) of a λ-term M is inductively defined by the form of M:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(c) &= \emptyset \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

3.3 Reduction rules

The following basic *reduction rules* are used to reduce λ-terms to *normal form*. This means they are evaluated as far as possible. They define the deduction rules of the calculus.

The most important ones are *α-conversion* and *β-reduction*. The δ- and *η-reduction* are seldomly applied and are not mentioned in most publications. They will only be mentioned in brief.

3.3.1 β-reduction

The first rule is used to apply application. It substitutes all variables bound by a λ in a term which are free variables in the remaining subterm to witch it is applied. Look at the following example:

$$\begin{aligned} (\lambda y.(\lambda x.x + y))2\ 3 &\xrightarrow{\beta} (\lambda x.x + 2)3 \\ &\xrightarrow{\beta} 3 + 2 \\ &\xrightarrow{\beta} 5 \end{aligned}$$

Before the first reduction step y is bound by the first λy but it is a free variable in the remaining term so 2 is applied to all occurrences of y in this term and the λy at the beginning is discarded. The second step does the same for x and 3.

3.3.2 α-conversion

The second rule is called alpha conversion[2]. It is used to rename variables that have the same name but different meanings in one lambda term. It is based on the fact that the names of variables in λ calculus are only symbols and can be changed without changing the semantics of the λ-term.

[2]Actually this is the first rule. Presenting it as second one it is easyer to understand.

Consider:

$$(\lambda y.(\lambda x.x\ y))x$$

if you apply β-reduction to this lambda term the count of free variables in the remaining term drops from 1 to 0. Abruptly x is occurring two times inside the λ-term. This should not happen because the x outside the brackets is not the same as the one inside.

Using α-conversion each variable that is used two times in one λ-term is renamed to show their different meanings. The above term would be reduced the following way:

$$\begin{aligned} (\lambda y.(\lambda x.x\ y))x &\xrightarrow{\alpha} (\lambda y.(\lambda x1.x1\ y))x \\ &\xrightarrow{\beta} \lambda x1.x1\ x \end{aligned}$$

This results in the correct solution.

The notation for replacing one symbol in a λ-term with another is the following:

$$\mathrm{M}[x/y]$$

This means that in M each occurrence of y is replaced with x. It will be needed later again.

3.3.3 δ-reduction

The δ-reduction also called constant-reduction is applied in two cases. The first is when some base operations have to be calculated on constants like:

$$2 + 2 \xrightarrow{\delta} 4$$

The second is the application of *combinators*[3], which are actually only high-order constants. Principly they are functions of high-order but they got names and can now be handled like base operations. For example look at the following expression:

$$id\ t \xrightarrow{\delta} t$$

3.3.4 η-reduction

Each λ-term

$$\lambda x.(\mathrm{M}\ x)$$

can be reduced the following way, if x is not free in M

$$\lambda x.(\mathrm{M}\ x) \xrightarrow{\eta} \mathrm{M}$$

The interesting fact with η-reduction is that one can handle every expression as a function. This is reached because η-reduction proofs that

$$y \equiv \lambda x.(y\ x)$$

3.4 Combinators

Combinators or closed λ-terms are those terms which do not have free variables. This means $FV(M) = \emptyset$.

[3]They are described in the next section.

One example might be the identity combinator:

$$\mathrm{Id} = \lambda x.x$$

Applied to an arbitrary λ-term it produces the same λ-term again.

$$(\lambda x.x)M \equiv M$$

There are much more combinators, which realize boolean values, integer numbers and arithmetic and logical constructs. Often they are given a name so one has not to write them down over and over again.

3.4.1 boolean combinators

To express conditions a λ-term for if-then-else constructs can be given.

At first functions for the values *true* and *false* are needed. These two values are not really functions but remember that everything can be expressed as a function and the basic λ-calculus has no method of expressing literal values. The function for *true* is

$$\lambda x\ y.x = true$$

a projection on the first argument and *false* is

$$\lambda x\ y.y = false$$

a projection on the second argument.

Now an if-construct can be defined the following way:

$$\lambda z\ x\ y.z\ x\ y$$

The variable z is *true* or *false* while x and y give the 'then' and the 'else' branch.

Consider the following example form (aut) to get a feeling for the usage:

$$\begin{aligned}
&(\lambda z\ x_1\ x_2.z\ x_1\ x_2)(\lambda x\ y.x)y_1\ y_2\\
&\xrightarrow{\beta}(\lambda x_1\ x_2.(\lambda x\ y.x)x_1\ x_2)y_1\ y_2\\
&\xrightarrow{\beta}(\lambda x_2.(\lambda x\ y.x)y_1\ x_2)\ y_2\\
&\xrightarrow{\beta}(\lambda x\ y.x)y_1\ y_2\\
&\xrightarrow{\beta}(\lambda y.y_1)y_2\\
&\xrightarrow{\beta}y_1
\end{aligned}$$

This construct can now be used in a more readable form in every λ-term like

$$\text{if } z \text{ then } x \text{ else } y$$

3.4.2 Fixed-point combinator

Because in λ-calculus you can not use a function f before it is defined there is some problem with using recursion. But to define the semantics of an arbitrary programming language we will most likely need recursion, if only to define the semantics of while statements but probably also that of recursive functions and procedures.

Fortunately this problem is solvable. What does a recursive function do? It takes the result of itself to compute its result. The result of the whole call is called *fixed-point* because the function can only have a valid solution if there is a point where it produces no new values. The value is fixed at this point.

Consider

$f(x) = 5$	The only fixed-point is 5.
$g(x) = x^2 - 2$	The fixed-points are 2 and -1.
$h(x) = x$	This has an infinite fixed point.

So it is clear that the formal definition for a fixed point looks the following way:

DEFINITION 2. *Fixed-point*

$$fix(f) = f(fix(f))$$

In λ-calculus this would be expressed as:

$$fix\ f = f\ fix\ f$$

The only remaining problem is the structure of fix. For this the interesting fact is used that the λ-term:

$$\lambda x.(x\ x)$$

Produces - if applied to itself - itself as output:

$$(\lambda x.(x\ x))(\lambda x.(x\ x)) \xrightarrow{\beta} (\lambda x.(x\ x))(\lambda x.(x\ x))$$

The only thing that has to be added to get fix or Y as it is usually called is a parameter that takes the function f and reproduces this function:

$$Y = \lambda y.(\lambda x.y(x\ x))(\lambda x.y(x\ x))$$

And now it is possible to express recursion with the λ calculus.

4. A SIMPLE DENOTATIONAL SEMANTIC

4.1 Syntactic domains

At first, look at the the input values for the *semantic functions.* Since you cannot assign a meaning to anything you do not know, you have to build the sets of syntactic constructs. For theses sets you can and will build functions which will assign a sense to the syntax later.

These sets of syntactic constructs are called *syntactic domains.* To get a feeling for these constructs and how they will be evaluated by denotational semantics you will see a simple example.

One type of syntactic domain might simply be all integer constants.

$$\mathbf{Num} = \ldots, -2, -1, 0, 1, 2, \ldots$$

They form a language constructed by the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} and the following *context-free grammar* [4]:

$$\begin{aligned}
\nu &::= \nu\delta | \delta\\
\delta &::= 0|1|2|3|4|5|6|7|8|9
\end{aligned}$$

[4]Because we want to talk about semantics there will be not much explanation of syntax and grammars here. Please refer some other material if you do not know these concepts.

If you do not have any semantics for them something like '395' means nothing more then the word build of concatenating the three digits 3, 9 and 5 one after the other. The numerals of the alphabet are usually standing for integer constants. This means there is a function **V**:

$$\begin{array}{ll}
\mathbf{V} : \mathbf{Num} \mapsto \mathbf{Int} & \\
\mathbf{V}[\![\nu\delta]\!] = 10 * \mathbf{V}[\![\nu]\!] + \mathbf{V}[\![\delta]\!] & \\
\mathbf{V}[\![0]\!] = \mathit{0} & \mathbf{V}[\![1]\!] = \mathit{1} \\
\mathbf{V}[\![2]\!] = \mathit{2} & \mathbf{V}[\![3]\!] = \mathit{3} \\
\mathbf{V}[\![4]\!] = \mathit{4} & \mathbf{V}[\![5]\!] = \mathit{5} \\
\mathbf{V}[\![6]\!] = \mathit{6} & \mathbf{V}[\![7]\!] = \mathit{7} \\
\mathbf{V}[\![8]\!] = \mathit{8} & \mathbf{V}[\![9]\!] = \mathit{9}
\end{array}$$

These rules give the semantics for each syntactic construct. The syntax is surrounded by $[\![$ and $]\!]$ to distinguish it from the elements of the meta language. The numerals in between the brackets are strings while the ones outside are in *italics* and are standing for the real integer constants.

You might say that this is really obvious but the strength of good formalisms is that they are correct, even for the most simple example and that you can depend on them when the examples are not that obvious.

The point is that you should have understood, that any programming language would correctly report an error when you try to do something like $0 =' 0'$

4.2 Semantic domains

One application of the function **V** results in an integer value computable by a machine which is one out of the big domain of all computable integer values which approximate all mathematical integers. They only do approximate them because the mathematical integers are an infinite domain and a computer can only work with finite numbers, as was already mentioned.

Such a domain is called *semantic domain.* Each element of the semantic domains defines the meaning of certain constructs from the syntactic domains. But semantic domains are not only discrete values like integers or booleans, but also the basic functions over the integers and the partial functions approximating the basic functions on a finite interval.

Semantic domains form a *lattice* or *complete partial order* which means among other thing that they all have a relation $\sqsubseteq$ called 'approximates'.

Lets take the function $f(x) = x$ out of the semantic domain of all functions over the integers. The function $g(x) = x$ where $x \epsilon \mathbb{Z}$ is an approximation of f because it is limited to the positive integers but in the domain it is defined for it corresponds with f. This means $g \sqsubseteq f$.

For each semantic domain there is also a least element called $\perp$ 'bottom'. This is the element that is according to $\sqsubseteq$ smaller than any other element in the semantic domain. It gives no information and is used for things like infinite loops that do not even produce some information while looping.

Knowing all this a formal definition for semantic domains can be given like Fehr (Feh89) and others did.

Definition 3. *The structure $A = (\underline{A}, \sqsubseteq_A)$ is a semantic domain (cpo) (complete partial order) if:*

1. *The relation $\sqsubseteq_A$ is a partial order on $\underline{A}$ (called approximation relation), meaning $\sqsubseteq_A$ is reflexive, antisymmetric, and transitive. If A is obvious from the context $\sqsubseteq_A$ is shortened to $\sqsubseteq$.*

2. *The set $\underline{A}$ contains regarding to the relation $\sqsubseteq$ a minimal element $\perp_{\mathrm{A}}$, meaning that the relation $\perp_A \sqsubseteq a$ is valid for each $a \epsilon\ \underline{A}$. The element $\perp_{\mathrm{A}}$ is also shortened to $\perp$ if A is obvious from the context.*

3. *Each chain of elements $K \subseteq \underline{A}$ has regarding to the order $\perp_{\mathrm{A}}$ a least upper bound $\bigsqcup K$ in $\underline{A}$, where K is called chain if for each two elements $k_1, k_2 \epsilon\ K$ applies: $k_1 \sqsubseteq k_2$ or $k_2 \sqsubseteq k_1$.*

4.2.1 State

For denotational semantics one of the most important semantic domains is the domain of all *states* a program can get in. Since not all semantics are as simple as integers a more complex syntactic construct needs a state to work on.

Such a state is a function from variable identifiers ξ to corresponding values

As was already mentioned, every expression, every command and every program, consisting out of commands and expressions, transforms one state into another thereby changing the allocation of values to identifiers.

The semantic domain of states S is the set of all functions from identifiers **Ide** to values **Value**.

$$\mathbf{S} = \mathbf{Ide} \rightarrow \mathbf{Value}$$

One particular state:

$$\sigma : \mathbf{S}$$

defines the current values of the variables of one program run. The notation means that the state is one out of the set of states or of the data-type of states.

4.3 Semantic functions

The final element for a complete denotational semantics are the functions, which map the syntactic domains into the semantic ones. They are called semantic functions. The function **V** from section 4.1 was one. Usually the λ calculus is used to express these functions. In the following the semantics for a simple programming language, actually a subset of PASCAL is given, like Allison (All89) did. As values it only has the integers like they are specified by **V**.

The first thing to do is presenting the syntactic domains in form of abstract syntax:

$$\begin{aligned}
&\textbf{Cmd}:\\
&\quad \gamma ::= \xi := \varepsilon |\\
&\qquad \textbf{if } \varepsilon \textbf{ then } \gamma \textbf{ else } \gamma |\\
&\qquad \textbf{while } \varepsilon \textbf{ do } \gamma |\\
&\qquad \gamma ; \gamma\\
&\qquad \textbf{skip}\\
&\textbf{Exp}:\\
&\quad \varepsilon ::= \varepsilon\ \mathbf{\Omega}\ \varepsilon | - \varepsilon | \nu | \xi\\
&\textbf{Opr}:\\
&\mathbf{\Omega} ::= = | \neq | < | > | \leq | \geq | + | - | \times | /\\
&\textbf{Bexp}:
\end{aligned}$$

After having done this, the constructs that semantics should be defined for are clear.

4.3.1 Expression semantics

At first the semantics for the simple expressions are defined. To do this it is necessary to construct the semantic domains into which the syntactic constructs will be mapped.

Expressions and later on also commands map an expression onto a value which can only be an integer in this simple case. They are also dependent onto the current state of the program as was mentioned under Section 4.2.1. This would mean the semantic function for expressions would look like:

$$\mathbf{E} : (\mathbf{Exp} \times \mathbf{S}) \mapsto \mathbf{Int}$$

But because it is not the intention to get the semantics for tupels of expressions and states a little trick is used to get a function that only takes an expression. This function can return another function that maps the current state onto the value that the expression gets under this state. How this works was already mentioned under currying (Section 3.1).

$$\mathbf{E} : \mathbf{Exp} \mapsto \mathbf{S} \mapsto \mathbf{Int}$$

A second semantic domain is needed for all the operations used inside of the expressions. It maps the operations onto the integer functions and the comparators onto the boolean values.

$$\begin{aligned}
\mathbf{Ifns} &= \mathbf{Int} \times \mathbf{Int} \mapsto \mathbf{Int}\\
\mathbf{O} &: \mathbf{Opr} \mapsto \mathbf{Ifns}
\end{aligned}$$

There are no functions for defining the boolean operations yet. For this simple example consider *0* to be *false* and all other values to be *true*. Now the semantic functions for expressions are constructed:

$$\begin{aligned}
&\mathbf{E}[\![\xi]\!] = \lambda\sigma.\sigma[\![\xi]\!]\\
&\mathbf{E}[\![\nu]\!] = \lambda\sigma.\mathbf{V}[\![\nu]\!]\\
&\mathbf{E}[\![\varepsilon\ \mathbf{\Omega}\ \varepsilon']\!] = \lambda\sigma.\mathbf{O}[\![\mathbf{\Omega}]\!]\langle \mathbf{E}[\![\varepsilon]\!]\sigma, \mathbf{E}[\![\varepsilon']\!]\sigma\rangle\\
&\mathbf{E}[\![-\varepsilon]\!] = -\mathbf{E}[\![\varepsilon]\!]\sigma\\
&\mathbf{O}[\![+]\!] = + : \mathbf{Int} \times \mathbf{Int} \mapsto \mathbf{Int}\\
&\text{etc. for the other arithmetic operations}\\
&\mathbf{O}[\![<]\!] =\\
&\qquad \lambda\langle v_1, v_2\rangle.\text{if } v_1 < v_2 \text{ then } \mathit{true} \text{ else } \mathit{false} : \mathbf{Int} \times \mathbf{Int} \mapsto \mathbf{Int}\\
&\text{etc. for the other boolean operations}
\end{aligned}$$

It should be clear what is happening. The semantics of an identifier ξ is the value of ξ under a given state. The value of a value was given by the function **V** and the state is ignored. An Operation is a function which takes a tupel of **Int**-values and produces a resulting **Int**-value. The pointed brackets mark a tupel here. The tupel is evaluated by evaluating the two expressions surrounding the operator symbol. The value of a negative expression is the negative value of this expression. The boolean operations use the 'if-then-else'-combinator.

4.3.2 Command semantics

The semantics of commands is to take a statement and return a function which expresses how this statement transforms one state into another.

$$\mathbf{C} : \mathbf{Cmd} \mapsto \mathbf{S} \mapsto \mathbf{S}$$

The semantic functions for the example language look the following way:

$$\begin{aligned}
&\mathbf{C}[\![\xi := \varepsilon]\!] = \lambda\sigma.\sigma[\mathbf{E}[\![\varepsilon]\!]\sigma/\xi]\\
&\mathbf{C}[\![\textbf{if } \varepsilon \textbf{ then } \gamma \textbf{ else } \gamma']\!] =\\
&\qquad \lambda\sigma.(\text{if } \mathbf{E}[\![\varepsilon]\!]\sigma \text{ then } \mathbf{C}[\![\gamma]\!] \text{ else} \mathbf{C}[\![\gamma']\!])\sigma\\
&\mathbf{C}[\![\textbf{while } \varepsilon \textbf{ do } \gamma]\!] =\\
&\qquad \lambda\sigma.(\text{if } \mathbf{E}[\![\varepsilon]\!]\sigma \text{ then } \mathbf{C}[\![\textbf{while } \varepsilon \textbf{ do } \gamma]\!] \circ \mathbf{C}[\![\gamma]\!] \text{ else Id})\sigma\\
&\mathbf{C}[\![\gamma;\gamma']\!] = \mathbf{C}[\![\gamma']\!] \circ \mathbf{C}[\![\gamma]\!]\\
&\mathbf{C}[\![\textbf{skip}]\!] = \mathit{Id}
\end{aligned}$$

This is again readable very intuitively with some thought. The semantics of an assignment under a given state is to update the state by setting the identifier ξ to the value expressed by the expression ε. The 'if'-construct simply takes on the semantics of the 'if-then-else'-combinator executing either the command γ or the command γ'. The 'while'-statement is a recursion, like it can be defined by the fixed-point combinator. It executes its body if the expression ε is *true* and then calls itself again or does nothing if the expression is not *true*. The next function simply splits command sequences and **skip** does actually nothing.

Now it is time to present a simple example how a short program can be evaluated by denotational semantics. Consider the following little program part:

```
x:=1;
y:=2;
if x<y then x:=x+y else x:=y-x
```

For simplification σ_1 is defined to be $\sigma[\mathit{1}/x, \mathit{2}/y]$. The evaluation by denotational semantics works now by applying the

6

semantic functions to the program P:

$$\begin{aligned}
&\mathbf{C}[\![P]\!]\sigma_0\\
&= \mathbf{C}[\![y := 2;\\
&\quad \textbf{if } x < y\\
&\quad \textbf{then } x := x + y\\
&\quad \textbf{else } x := y - x]\!] \circ \mathbf{C}[\![x := 1]\!]\sigma_0\\
&= \mathbf{C}[\![\\
&\quad \textbf{if } x < y\\
&\quad \textbf{then } x := x + y\\
&\quad \textbf{else } x := y - x]\!] \circ \mathbf{C}[\![y := 2]\!]\sigma_0[\mathbf{E}[\![1]\!]\sigma_0/x]\\
&= \mathbf{C}[\![\\
&\quad \textbf{if } x < y\\
&\quad \textbf{then } x := x + y\\
&\quad \textbf{else } x := y - x]\!]\sigma_0[\mathbf{V}[\![1]\!]\sigma_0/x, \mathbf{E}[\![2]\!]\sigma_0[\mathbf{V}[\![1]\!]\sigma_0/x]/y]\\
&= \mathbf{C}[\![\\
&\quad \textbf{if } x < y\\
&\quad \textbf{then } x := x + y\\
&\quad \textbf{else } x := y - x]\!]\sigma_0[\mathit{1}/x, \mathbf{V}[\![2]\!]\sigma_0[\mathit{1}/x]/y]\\
&= \mathbf{C}[\![\\
&\quad \textbf{if } x < y\\
&\quad \textbf{then } x := x + y\\
&\quad \textbf{else } x := y - x]\!]\sigma_1\\
&= (\text{if } \mathbf{E}[\![x < y]\!]\sigma_1 \text{ then } \mathbf{C}[\![x := x + y]\!] \text{ else } \mathbf{C}[\![x := y - x]\!])\sigma_1\\
&= (\text{if } \mathbf{O}[\![<]\!]\langle \mathbf{E}[\![x]\!]\sigma_1, \mathbf{E}[\![y]\!]\sigma_1\rangle \text{ then}\\
&\quad \mathbf{C}[\![x := x + y]\!] \text{ else } \mathbf{C}[\![x := y - x]\!])\sigma_1\\
&= (\text{if } \mathbf{O}[\![<]\!]\langle \mathit{1}, \mathit{2}\rangle \text{ then } \mathbf{C}[\![x := x + y]\!] \text{ else } \mathbf{C}[\![x := y - x]\!])\sigma_1\\
&= (\text{if } (\text{if } \mathit{1} < \mathit{2} \text{ then } \mathit{true} \text{ else } \mathit{false}) \text{ then}\\
&\quad \mathbf{C}[\![x := x + y]\!] \text{ else } \mathbf{C}[\![x := y - x]\!])\sigma_1\\
&= (\text{if } \mathit{true} \text{ then } \mathbf{C}[\![x := x + y]\!] \text{ else } \mathbf{C}[\![x := y - x]\!])\sigma_1\\
&= \mathbf{C}[\![x := x + y]\!]\sigma_1\\
&= \sigma_1[\mathbf{E}[\![x + y]\!]\sigma_1/x]\\
&= \sigma_1[\mathbf{O}[\![+]\!]\langle \mathbf{E}[\![x]\!]\sigma_1, \mathbf{E}[\![y]\!]\sigma_1\rangle/1]\\
&= \sigma_1[\mathbf{O}[\![+]\!]\langle 1, 2\rangle/x]\\
&= \sigma_1[\mathit{3}/x]
\end{aligned}$$

This should show how the specifications work but also that it is not practicable to apply this manually to big programs.

4.3.3 Program semantics

It might seem magical where the first σ in the example is coming from. As you can see the example is no whole program so be assured that there is some function P that produces that first σ out of the programs input. In a complete denotational semantics there has to be such a function P for program semantics. It usually takes an input, produces an empty starting state where no identifiers are bound to any values and finally projects onto the output of the program, which is what the result of a program should be.

$$\mathbf{P} : \mathbf{Inp} \mapsto \mathbf{Out}$$
$$\mathbf{P}[\![\gamma]\!] = \lambda\text{inp}.(\pi_3\ \mathbf{C}[\![\gamma]\!]\langle \sigma_0, inp, out_0\rangle)$$

For completely specifying this function one needs semantics for input and output commands and an extension of the basic semantic functions specified earlier.

$$\mathbf{C} : \mathbf{Cmd} \mapsto (\mathbf{S} \times \mathbf{Inp} \times \mathbf{Out}) \mapsto (\mathbf{S} \times \mathbf{Inp} \times \mathbf{Out})$$

Please read further literature like (All89) for the detailed new specification of **C**.

5. FURTHER ISSUES

For the definition of the whole semantics of a programming language there are much more rules, constructs and definitions some of which I want to mention shortly for your reference.

The λ calculus also has a typed form in contrast to the untyped which was presented here. For the specification of the semantics in section 4.2.1 it was used implicitly. But it would have gone to far to present this here because this also needs some extensions for keeping the ability of recursion of λ-terms.

As was already mentioned there are still more combinators. With them it is possible to completely define numbers and the operations on them. This is based on the so called *Church numerals.*

There is a big mathematical apparatus behind the theory of semantic domains which has something to do with the fact that everything in λ-calculus is a function, even values but that a set of values can generally not be isomorphic to the set of total functions from the values to some other set. Try to find informations about Scott's theory of lattices also called complete partial orders.

The complete specification of directly evaluable semantics was not given here. The so called 'direct semantics' do also contain constructs for declarations, error-semantics, procedures and functions.

Continuation semantics make it possible to specify the semantics of gotos[5], functions and procedures. They add a new semantic domain of continuations which are something like a return adress and allow to break the order of statements.

As already mentioned one interesting field of application for denotational semantics is the creation of parsers. For practical minded people I suggest to try implementing the rules above or look in (All89). All the rules above and much more are implemented there in Pascal and Algol-68.

References

Lloyd Allison. *A practical introduction to denotational semantics.* Cambridge Computer Science Texts 23, 1989.

Unknown author. Das gefürchtete Lambda-Kalkül. Internet. Good understandable introduction into the λ calculus. But in german.

Henk P. Barendregt. *Handbook of Theoretical Computer Science*, volume B, chapter 7, pages 321–360. Elsevier, 1990.

[5]This may be one reason why I do not present this here.

Elfriede Fehr. *Semantik von Programmiersprachen*. Springer-Verlag, 1989.

Mia Minnes James Worthington. Denotational Semantics of IMP. Script, Octobre 2005.

Benjamin Weitz. Die Semantik von Programmiersprachen: Der denotationelle Ansatz. Internet. Very minimalistic german introduction.

Denotational semantics and XML

Jan Tank
Fakultät Informatik
Technische Universität Dresden
Dresden, Germany
s1500847@mail.inf.tu-dresden.de

1. INTRODUCTION

Being an example to the use of denotational semantics this text should be an overview on the paper 'A Prototype of a Schema-Based XPath Satisfiability Tester' by Groppe and Groppe [3]. It will show all the basic functions needed to compute the examples at the end of these paper.

For better understanding the reader should read before 'Denotational semantics' by Klemens Muthmann. Also the concepts of XML [5], XML Schema [2] and XPath [1] should not be new to the reader.

In this paper the concepts of the satisfiability tester will be shown. Therefore we will compute some examples. For that we need some knowledge about XML Schema, which will be introduced in section 3. Because of the bad readability of XML Schema for humans a XML example file has been written. This file will be used to explain some XPath basics. We then also need some additional functions to be able to compute the semantic rules. After that we can analyse the three examples. At the end there will be some points of criticism and a conclusion.

Because it should only be an overview on the paper the loop detection and the computation of predicate expressions will not be described.

1.1 A satisfiability test for XPath

Examples for the use of this satisfiability tester can't be given (due to the fact that no implementations could be found), but the intent of this tester is the analysis of XPath queries.

XPath itself is a query language for XML data. So the satisfiability tester should help to avoid unnecessary submission and evaluation of unsatisfiable queries. Therefore it evaluates XPath queries in presence of XML Schema files. The key benefits of testing with the presence of schemas are faster testing (because we only need the XML Schema file and not a XML instance), less processing time and less query costs.

Thus testing only with a schema the tester can only return unsatisfiable, if the XPath query evaluates as false, but can **not** return satisfiable, if the query returns true on the schema. On the XML instance the query can even be unsatisfiable. So in this fact the tester could only return *maybe satisfiable*.

2. XML AND XML FEATURES

2.1 XML Schema

```
<schema>
<group name='pages'>
    <sequence>
        <element name='page' minOccurs='0'
             maxOccurs='1'>
        <complexType>
            <sequence>
                <element name='title' minOccurs='0'
                     maxOccurs='1' type='string'/>
                <element name='link' minOccurs='0'>
                <complexType>
                    <group ref='pages' minOccurs='0'
                         maxOccurs='unbounded'/>
                </complexType>
                </element>
        </sequence></complexType></element>
    </sequence>
</group>
<element name='web'>
    <complexType>
        <group ref='pages' minOccurs='0'
             maxOccurs='unbounded' />
        <attribute name='id' type='integer' />
    </complexType>
</element>
</schema>
```

As you can see XML Schema is a XML language which describes XML itself with specified rules.
In this schema file we have a group with name 'pages' that has one child. The child element 'page' has 3 children: 'title', 'link' and a reference to 'pages'. This reference is a loop. Thus we can infinitely repeat this construct. But the reference points only to the style definition in the schema file and not to the specific tag in the XML instance.

The second definition is an element 'web' which has the children id as an attribute and the element page due to the reference on pages. So like the first example it refers not to the element itself but its children.

So in this example we have a path to the title element with '/web/page/title'.

2.2 XML

Because of the difficult readability of XML Schema files this example XML file has been writen which refers to the schema file and validates through the rules given in the schema file.

You should remember that the semantic rules (they will be introduced in section 3.3) work on the XML Schema and not on this XML instance.

```
<?xml version='1.0' encoding='UTF-8'?>
<web id='1'
 xsi:schemaLocation='http://www.example.org/web.xsd'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

Here we can match the rules of the XML Schema file against the XML tags and see clearly the reference, which is described before.

2.3 XPath

Because we need some basic XPath queries in the examples a quick overview of the syntax will be presented.

The '/' represents an absolute path beginning from the root node. If we want to return a specific node, we can reference to it by building a path from the root node to the node itself by separating each node with a slash.

Because to the fact, that 'child' is the standard axis in XPath, we can rewrite it.

XPath: /web $\widehat{=}$ /child::web

```
<?xml version="1.0"?>
<web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

XPath: /web/page/title

```
<?xml version="1.0"?>
<web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

'//' refers to all nodes, which accomplish the following rules. As you can see, '//page' returns all nodes, that accomplish the term 'page'. '//' is the short form of referencing, but it differs between the notation. '/page' will not return the same like '//page', only if the node 'page' exists only one time.

XPath: //page

```
<?xml version="1.0"?>
<web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

With the '@' prefix we refer the attributes. In the case of '//@attribute' it will return all appearances of the attribute with the name 'attribute'. Due to the rewriting rules we can replace 'attr::' with '@'.

XPath: //@id $\widehat{=}$ //attr::id

```
<?xml version="1.0"?>
<web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

With the square brackets we can make a node test. If we write '//[@attribute]', we test all nodes for the attribute 'attribute'. Only if the node contains the attribute, the node will be returned. We can also write '1' in square brackets to determine the first element of a set. The function 'last()' will then return the last element of a set.

XPath: //[@id]

```
<?xml version="1.0"?>
<web id='1'>
  <page>
    <title>Outer</title>
    <link>
      <page>
        <title>Inner</title>
      </page>
    </link>
  </page>
</web>
```

Out of the original paper the tester supports the following expressions:

$e ::= e \mid e \;\; | \;\; /e \mid \; e/e \;\; | \;\; e[q]$ |axis::nodetest
$q ::= e \;\; | \;\; e = C \;\; | \;\; e = e \;\; | \;\; qandq \;\; | \;\; qorq \;\; | \;\; not(q) \;\; | \;\; (q) \;\; | \;\; true() \;\; | \;\; false()$
$axis ::= child \mid attr \mid desc \mid self \mid following \mid preceding \mid parent \mid ances \mid DoS \mid AoS \mid FS \mid PS$
$nodetest ::= label \mid * \mid node() \mid text()$

To solve the examples and to make it simpler for this paper we only need a subset:

$e ::= /e \mid e/e \mid axis :: nodetest$
$axis ::= child \mid attr$
$nodetest ::= label$

3. XPATH SATISFIABILITY TESTER

3.1 Prerequisites

We also need some additional functions.

The function $NT(x, label)$ is a node test. It checks, if a schema node and a given label are identical.

The function $iAttr(x)$ returns a set of all attributes of a given schema node.

The function $iChild(x)$ computes all children of a given schema node, returns a set and is defined as following:

$$\begin{aligned} iChild(x) = \{z \mid y \in S(x) \quad &\wedge \quad z \in succe(y) \\ &\wedge \quad (isiElem(z) \vee isiText(z))\} \end{aligned}$$

The auxiliary function S(x) relates the node x to the self node and all the descendant nodes of x, which occur before the instance child nodes of x in the document order. Now all successors of y will be calculated and returned to z. Due to the definition of XML z needs to be checked to be an element or a text, because also attributes are children of a node.

In the paper by Groppe and Groppe [3] some functions like iAttr are only described in prose. This makes it hard to determine the output of these functions. Thus we can only try to interpret and hope to get the right output.
You also can come to the conclusion, that there can be some inaccuracies. For this academically paper the authors should have described all functions with mathematical terms.

3.2 Schema Path

A schema path 'p' is a sequence of pointers to the schema path records $< XP', N, z, lp, f >$

- XP' is an XPath expression,
- N is a node in an XML Schema definition,
- z is a set of pointers to schema path records,
- lp is a set of schema paths,
- f is a schema path list or a predicate expression q'

$\vartheta(r, g)$: The function generates a new schema path record $e =< xp', r, g, -, - >$, adds a pointer to e at the end of the given schema path p and returns a new schema path.

For the examples we only need the first three entries of a schema path. 'lp' holds the information about detected loops and f will only be computed if there is a predicate expression q'. The loop detection will not be used, so it will also not be explained here.
Because also ϑ is only described in prose, it is really hard to determine its output. Especially the pointer to prior entries makes it difficult to show on paper, but a line based method with identifiers in the front will be used to refer to these identifiers for the results.
ϑ will always generate a new schema path record, if all prerequisites are fulfilled. It will generate xp', which is the actual part of the XPath query we process, r, which is the actual node in the XML Schema and g, which is set of pointers to prior schema path records.

3.3 Denotational Semantics

The denotational semantics in the paper are not in λ - Notation, although it is the most common method in denotational semantics. The functions in this paper get a syntactic expression and a parameter. The syntactic expression is in this case an XPath expression and the parameter is a set of schema paths. They use a set of schema paths because through the evaluation of the XPath expressions it can happen, that we get two different schema paths, which must be evaluated separately.
We also need only a subset of the given semantic expressions to solve the examples:

L : XPath expression $\times$ schema path $\rightarrow$ set(schema path)

$$\begin{aligned} L[\![/e]\!](p) &= L[\![e]\!](p1) \wedge \ p1 = (< /, /, -, -, - >) \\ L[\![e1/e2]\!](p) &= \{p2 | p2 \in L[\![e2]\!](p1) \wedge p1 \in L[\![e1]\!](p)\} \\ L[\![child :: n]\!](p) &= \{\vartheta(r, p(S)) | \\ &\qquad r \in iChild(p(S).N) \wedge NT(r, n)\} \\ L[\![attr :: n]\!](p) &= \{\vartheta(r, p(S)) | \\ &\qquad r \in iAttr(p(S).N) \wedge NT(r, n)\} \end{aligned}$$

The first rule tells us, that we can compute 'e' when we substitute ε with $< /, /, -, -, - >$. This schema path record is the root schema path record, because the third parameter 'z' points to no other schema path record.
The second rule splits the XPath expression and computes each component of the expression beginning on the right side, due to the fact, that we need 'p1' to compute the left function.
The third rule computes the standard axis of XPath. Before we can calculate ϑ we have to compute the prerequisites. First iChild and then the node test. The auxiliary function 'p(S)' gets the last schema path record out of the schema path. The letter 'S' represents the size of the schema path, thus 'p(S)' means the last one, 'p(S-1)' means the pre-last and so on.
'p(S).N' refers to the specific entry in the schema path record and gets us the node of the XML Schema.
The last rule is similar to the third one, only iChild is substituted by iAttr.

In the original paper [3] the authors could also in this case try to improve these expressions. In an article [4] (cited also by the authors) is described how XPath expressions can be rewritten to a simpler form by eliminating some axis. This improvement could save two third of the expressions and would make it in some ways much easier to understand.

4. EXAMPLES

4.1 Example 1

We will start with a simple XPath expression '/web':

$$\begin{aligned}
L[\![/web]\!](\emptyset) &= L[\![web]\!](</,/,-,-,->) \\
&= L[\![child::web]\!](</,/,-,-,->) \\
&= \{\vartheta(r,</,/,-,-,->)|r \in iChild(/) \wedge \\
&\quad NT(r,web)\} \\
iChild(/) &= \{z|y \in S(/) \wedge z \in succe(y) \wedge \\
&\quad (isiElem(z) \vee isiText(z))\} \\
&= \{web\} \\
L[\![/web]\!](\emptyset) &= \{\vartheta(r,</,/,-,-,->)|r \in \{web\} \wedge \\
&\quad NT(r,web)\} \\
&= \{\vartheta(web,</,/,-,-,->)\}
\end{aligned}$$

Result:
(R1) $\{(</,/,-,-,->,$
(R2) $</web,web,\{R1\},-,->)\}$

As you can see, we can use the first rule of our semantic expressions to start the process. 'p1' will always be $</,/,-,-,->$ for that rule. Due to the standard axis in XPath we then can rewrite 'web' to 'child::web'. We now can use the third rule. 'r' is the next part that needs to be calculated for the use in NT and ϑ. For that we execute the function iChild with '/' as parameter, because p(S).N matches '/' in our last schema path entry.
In iChild we get all successors of the root node '/'. There we only get 'web' as a result. This result is also an element so we can return 'web'.
Now we can process NT(web, web) and get back 'true'. The next step is to calculate ϑ. For 'r' we can write 'web' and p(S) is our latest schema path entry.
As you can see this XPath expressions is satisfiable due to the XML Schema.

4.2 Example 2

The next example will show an unsatisfiable XPath expression '/page':

$$\begin{aligned}
L[\![/page]\!](\emptyset) &= L[\![page]\!](</,/,-,-,->) \\
&= L[\![child::page]\!](</,/,-,-,->) \\
&= \{\vartheta(r,</,/,-,-,->)|r \in iChild(/) \wedge \\
&\quad NT(r,page)\} \\
iChild(/) &= \{z|y \in S(/) \wedge z \in succe(y) \wedge \\
&\quad (isiElem(z) \vee isiText(z))\} \\
&= \{web\} \\
L[\![/page]\!](\emptyset) &= \{\vartheta(r,</,/,-,-,->)|r \in \{web\} \wedge \\
&\quad NT(r,page)\} \\
&= \emptyset
\end{aligned}$$

In this case most of the example is identically to the prior one. Only the node test will fail. As result, we get back an empty set.

4.3 Example 3

A little bit longer XPath expression '/web/@id':

$$\begin{aligned}
L[\![/web/@id]\!](\emptyset) &= \{p2 \mid p2 \in L[\![@id]\!](p1) \wedge \\
&\quad p1 \in L[\![/web]\!](\emptyset)\} \\
L[\![/web]\!](\emptyset) &= (R1)\ \{(</,/,-,-,->, \\
&\quad (R2)\ </web,web,\{R1\},-,->)\} \\
p1 &= (R1)\ </,/,-,-,->, \\
&\quad (R2)\ </web,web,\{R1\},-,-> \\
L[\![/web/@id]\!](\emptyset) &= \{p2|p2 \in L[\![@id]\!](p1)\} \\
&= \{p2|p2 \in L[\![attr::id]\!](p1)\} \\
L[\![attr::id]\!](p1) &= \{\vartheta(r,p1) \mid r \in iAttr(web) \wedge \\
&\quad NT(r,id)\} \\
L[\![attr::id]\!](p1) &= \{\vartheta(r,p1) \mid r \in \{id\} \wedge NT(r,id)\} \\
&= \{\vartheta(id,p1)\} \\
&= (R1)\ \{(</,/,-,-,->, \\
&\quad (R2)\ </web,web,\{R1\},-,-> \\
&\quad (R3)\ </web/@id,id,\{R2\},-,->)\} \\
L[\![/web/@id]\!](\emptyset) &= \{p2 \mid p2 \in L[\![@id]\!](p1)\} \\
p2 &= (R1)\ </,/,-,-,->, \\
&\quad (R2)\ </web,web,\{R1\},-,-> \\
&\quad (R3)\ </web/@id,id,\{R2\},-,-> \\
L[\![/web/@id]\!](\emptyset) &= \{(p2)\}
\end{aligned}$$

Result:
(R1) $\{(</,/,-,-,->,$
(R2) $</web,web,\{R1\},-,->,$
(R3) $</web/@id,id,\{R2\},-,->)\}$

As you can see we use the second semantic rule to split the calculation. The calculation of $L[\![/web]\!](\varepsilon)$ is identically to the first example. We then can match 'p1' and our schema path of the result of the calculation. Due to that we compute $L[\![@id]\!](p1)$. We rewrite '@id' to 'attr::id' and can use the fourth expression. iAttr(web) will return all attributes of 'web'. In this case we get back 'id'. Now our node test will succeed and we can calculate ϑ.

5. CRITICISM

In previous sections I have mentioned some points that could have been written better.

The most annoying part is the description of functions only in prose. Because of that for this paper I can't give the assurance that all output is correct. These functions were tried to interpret and the right conclusions have been reached hopefully. And although we used only a small part, functions like iAttr, ϑ or p(S) are some examples.

Furthermore there are sometimes functions that were double defined, for example iAttr, isiAttr and iAttribute, isRoot and root. Here the authors could have written some better methods.

Another point is the definition of the semantic rules. Overall there exists 25 rules, but due to [4] it is possible to cut away at least two third of them. Perhaps they don't use that paper because they complain that they have written an incomplete (but fast) satisfiability test.

The semantic rules are another factor. If you don't know the specification of XPath and don't know the standard axis and the rewriting you can't compute any XPath query. But that fact is nowhere to read in the paper. Another problematic fact is the enormous complexity of some rules. Sometimes it can't be avoided but then a good explanation would help. This paper sometimes lacks of explanations in general.

6. CONCLUSION

In the end one can't recommend the paper by Groppe and Groppe. All in all also no area of application for this tester is seen. It could perhaps be used for testing XPath queries during the development, but in every day usage the need is probable marginal, due to the fact the developer should use only queries that are satisfiable.

The main fact for this little need is mainly because of the fact, that the tester could only return the result not satisfiable definitely. If the tester would test also a XML instance it possibly could be a good alternative for actual XPath query parser.

In the paper is also mentioned a performance analysis between the prototype, Saxon Evaluator and Qizx Evaluator. But that analysis compares only not satisfiable queries and due to that one can assume that in every day usage such speedup factors can't be achieved.

Overall one have to say that they showed an interesting way of processing XPath queries and this approach could return a good way for processing XPath in general.

7. REFERENCES

[1] S. Boag, M. F. Fernández, D. Chamberlin, J. Siméon, J. Robie, A. Berglund, and M. Kay. XML path language (XPath) 2.0. W3C proposed reccommendation, W3C, Nov. 2006. http://www.w3.org/TR/2006/PR-xpath20-20061121/.

[2] D. C. Fallside and P. Walmsley. XML schema part 0: Primer second edition. W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[3] J. Groppe and S. Groppe. A Prototype of a Schema-Based XPath Satisfiability Tester. In S. Bressan, J. Küng, and R. Wagner, editors, *DEXA*, volume 4080 of *Lecture Notes in Computer Science*, pages 93–103. Springer, 2006.

[4] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of the EDBT Workshop on XML Data Management (XMLDM)*, volume 2490, pages 109–127. Springer, 2002.

[5] F. Yergeau, E. Maler, C. M. Sperberg-McQueen, J. Paoli, and T. Bray. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, Aug. 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

Rewriting

Andreas Rümpel
Faculty of Computer Science
Technische Universität Dresden
Dresden, Germany
s9843882@inf.tu-dresden.de

ABSTRACT

This is an overview paper regarding the common technologies of rewriting. Rewrite systems (or term rewriting systems), rewrite rules and the structure of the terms that are included in the system's rewrite rules are discussed. Rewriting provides methods of replacing subterms of a formula with other terms. It has its applications in mathematics, computer science and logic. The important properties of term rewriting systems, termination and confluence, are covered here.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*Algebraic language theory*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms

Algorithms, Languages, Theory

Keywords

Completion, confluence, normal form, reduction, rewriting, rewriting rules, rewriting systems, termination, terms, well-foundedness, word problem

1. INTRODUCTION

When we talk about rewriting, in the majority of cases term rewriting is intentioned, although there are other "branches" like graph rewriting[1]. This paper introduces some basics around the structure of terms, term rewriting systems and some very important properties, like confluence and termination. Furthermore, decidability is playing a big role. Some practically relevant issues, like completing a non-confluent term rewriting system, are contained here, too. Computing a term's normal form will provide some information about its "meaning" encoded in the rewrite rules of the system.

[1]Graph rewriting is covered in an extra paper in this course.

We will see, how terms can be built, and analyze their behavior in a term rewriting system, surrounded with some theoretical background. Then the "word problem" is addressed and, thereafter, it is shown, when we can decide, whether a special term rewriting system has a specific property like confluence. "Improving" a given term rewriting system that is missing such an important property using algorithms will play a significant role.

Main fields of application are operational semantics of abstract data types, functional programming languages and logic programming languages. Rewrite systems provide a method of automated theorem proving. This makes the concept of term rewriting important for theoretical computer science as well as for tools for proving and completing. In logic, the procedure for determining normal forms like the negation normal form[2] of a formula can be conveniently written as a rewriting system:

$$\begin{array}{rcl} \neg\neg H & \rightarrow & H \\ \neg(G_1 \wedge G_2) & \rightarrow & (\neg G_1 \vee \neg G_2) \\ \neg(G_1 \vee G_2) & \rightarrow & (\neg G_1 \wedge \neg G_2) \end{array}$$

The negation normal form is reached, if none of the tree rules apply. In this case, each negation symbol $\neg$ occurs only immediately before a variable. Your can see, that $\neg$ takes one argument (we call this a unary function symbol), but the brackets are missing. Such a prefix notation is quite common in many contexts.

2. TERMS

The expressions on both sides of the arrows in the above example are called terms. A term can be built from variables, constant symbols and function symbols (or operations). Each function symbol has an arity. The arity is the count of parameters, the function accepts. $f_1(x_1, x_2)$ would have the arity 3 and $f_2(x_1, x_2, x_3, x_4)$ the arity 4. The signature Σ is a set of function symbols, where each $f \in \Sigma$ has an arity, a non-negative integer value. The arity can classify the signature's elements by grouping all n-ary elements of Σ in $\Sigma^{(n)}$. Constant symbols are function symbols of arity 0 (this would be $\Sigma^{(0)}$).

Examples: The successor-function $succ(t)$ is a unary func-

[2]See [4] page 82.

tion symbol. The binary function symbol $+$ can be written in infix form, e.g. $x + (y + z)$ instead of $+(x, +(y, z)$. With this knowledge, we can say, that $\neg$ of the example in the introduction is a unary function symbol and $\wedge$ and $\vee$ are binary ones. If the n-fold application of a unary function symbol f to a term t should be expressed, it can be abbreviated by writing $f^n(t)$ instead of $f(f(...f(t)...))$.

Let V be a set of variables with the signature Σ disjoint from V ($\Sigma \cap V = \emptyset$). We can define the set $T(\Sigma, V)$ of all terms over V inductively:

- $V \subseteq T(\Sigma, V)$
- $\forall n \geq 0, f \in \Sigma^{(n)}, t_1, ..., t_n \in T(\Sigma, V)$: $f(t_1, ..., t_n) \in T(\Sigma, V)$

This means, that every variable is a term and every application of function symbols to terms results in terms, too. With this inductive definition tree-illustrations can be made with nodes representing function symbols and edges pointing to the argument terms of them. Leaf nodes are represented by variables and constant symbols. The example term $t = f(g(x, x), x, h(x))$ is illustrated in Figure 1. Such trees are very useful to determine a subterm of the term t at a specific position p.

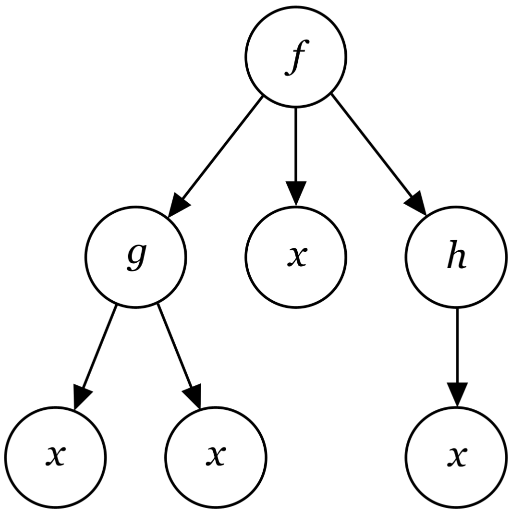

Figure 1: The term t in tree-illustration.

The variables in the introductionary example are H, G_1 and G_2. There are terms, that do not contain any variables. They are called ground terms. An example for a ground term is $succ(0)$ for Ackermann's function (see 3.4), containing a unary function symbol $succ$ and a constant symbol 0.

3. TERM REWRITING SYSTEMS

The notion "term rewriting system" implies that there must be a system containing something. In our case, it is a set whose elements are called rewrite rules. Some literature, e.g. [1], considers the term rewriting system itself as the set containing the rewrite rules. Apart from that, it is common to regard these rewrite rules contained in a set called reduction system (or rule system). The term rewriting system (TRS) is then a pair (T, R) with a set of terms T and a reduction system R, a collection of rewrite rules over those terms in T and a reduction relation. In this paper, the first definition, considering a term rewriting system as a simple set of rewrite rules is used.

3.1 Identities

But before defining rewrite rules, we first have a look at identities. Identities can be regarded as pairs of terms over a given set of terms, more formally: $(t_1, t_2) \in T(\Sigma, V) \times T(\Sigma, V)$, where V denotes the set of variables and Σ the signature. We are using the notation $t_1 \approx t_2$, according to [1]. An identity has a right-hand side (rhs) and a left-hand side (lhs), each side containing a term $t \in T(\Sigma, V)$.

In the context of rewriting, the identities from a set of identities E are regarded as rewrite rules of a term rewriting system that are applied unidirectionally. The difference is, that identities can be used to transform the instances of the left-hand side to instances of the right-hand side of the identity and vice versa.

3.2 Substitutions and Instances

The notion "instance" is introduced now, because this term will occur in this paper furthermore. Assume, you have the following rewrite rule in a term rewriting system

$$f(x, y) \to x$$

with variables x and y, You want to use this rule to transform the expression $f(2, 3)$. In this case, you have to substitute the variables x and y through 2 and 3. This is done by a substitution

$$\sigma = \{x \mapsto 2, y \mapsto 3\}$$

that is able to map any occurrence of variables to terms. If we have two terms t_1 and t_2 and a substitution σ, t_2 is called an instance of t_1, if $\sigma(t_1) = t_2$. The prefix notation σt is also widely-used for $\sigma(t)$. Two substitutions σ_1 and σ_2 may be composed by writing $\sigma_1(\sigma_2(t))$ or $\sigma_1\sigma_2(t)$. Unification is the process of applying a substitution to several terms. The substitution is then called unifier. Whole terms can be unified as well as subterms of terms.

3.3 Rewrite rules

We want our rewrite rules to be unidirectional, i.e. when we have a subject to rewrite with a term rewriting system, we only want to look on the left-hand sides of the rules, whether any rule applies to our expression. For this purpose, we need some kind of rewrite relation to connect the left-hand side and the right-hand side of a rewrite rule. If the left-hand side of an identity $t_1 \approx t_2$ is not a variable and every variable from t_2 occurs in t_1 as well, then it is a rewrite rule and is denoted $t_1 \to t_2$. The instance of the left-hand side of such a rewrite rule is called reducible expression[3]. $\to$ (or $\to_R$, if the corresponding TRS R is not clear from the context) is called rewrite relation. If the system's rewrite relation has a specific property like confluence or termination, we

[3]The short form redex for reducible expression is also quite common.

also assign this property to the term rewriting system itself and then say, that the TRS is confluent or terminating. A TRS is finite, if it contains finitely many rewrite rules. In the following, we talk about finite term rewriting systems without explicitly mentioning it.

3.4 Example system

Consider the following term rewriting system:

$$\begin{array}{rcll} ack(0,y) & \rightarrow & succ(y) & \text{(r1)} \\ ack(succ(x),0) & \rightarrow & ack(x,succ(0)) & \text{(r2)} \\ ack(succ(x),succ(y)) & \rightarrow & ack(x,ack(succ(x),y)) & \text{(r3)} \end{array}$$

This term rewriting system computes Ackermann's function (see [1] page 110) with the constant symbol 0, the function symbols *succ* (unary) and *ack* (binary) and the variables x and y. Now we want to use this system to compute $ack(1,1)$:

$$\begin{array}{rcl} ack(succ(0),succ(0)) & \xrightarrow{r3} & ack(0,ack(succ(0),0)) \\ & \xrightarrow{r1} & succ(ack(succ(0),0)) \\ & \xrightarrow{r2} & succ(ack(0,succ(0))) \\ & \xrightarrow{r1} & succ(succ(succ(0))) \end{array}$$

The final value is encoded in the number of successors of 0, e.g. $succ(succ(succ(0)))$ would be the number *tree.*

4. THE WORD PROBLEM

We will now discuss the approach that is managing the word problem in the context of rewriting. Assume you have two terms t_1 and t_2, a set of identities R and $\equiv_R$ denoting the equivalence generated by R. The question of the word problem is now the following:

$$t_1 \equiv_R t_2?$$

In other words, it should be determined, whether t_1 can be transformed into t_2 by using a set of equations in R.

One is now interested in making this undecidable problem decidable for the special situation. For this purpose, the bidirectional identities in R are transformed into rewrite rules of a term rewriting system and sometimes must be regarded as unidirectional term rewrite rules from left to right and sometimes vice versa. In this matter, it is important to check, whether the derived rewrite rules meet their definition from 3.3. The question of equivalence can then be reformulated to the following: Is there a path from t_1 to t_2 using the rewrite rules in both directions? An example situation is illustrated in Figure 2. The path $t_1 \rightarrow t_3 \leftarrow t_4 \rightarrow t_5 \rightarrow t_2$ has to be traversed with and against the direction of the arrows. Formally, we could write $\equiv_R \;=\; (\rightarrow_R \cup \leftarrow_R)^*$ or say, that we create the reflexive transitive symmetric closure of $\rightarrow_R$. In the case of convergent term rewriting systems we just have to compute the normal forms of t_1 and t_2 and compare them to decide $t_1 \equiv_R t_2$.

Because the word problem is, in general, undecidable, a universal algorithm cannot always terminate successfully (see [1] page 59). A nice approach of solving the word problem for special cases is the Knuth-Bendix algorithm. We will get back to it later, when we talk about completion.

5. TERMINATION

$$\begin{tikzcd} t_1 \arrow[dr] & & t_4 \arrow[dl] \arrow[dr] & & \\ & t_3 & & t_5 \arrow[dr] & \\ & & & & t_2 \end{tikzcd}$$

Figure 2: The word problem for t_1 and t_2.

In computer science termination is an important property. If we want a machine to compute something, it will be of interest to know, whether the computation terminates or not. But unfortunately, the property of termination for term rewriting systems is in general undecidable, i.e. there is no universal decision procedure, that can say whether the given term rewriting system terminates for arbitrary term rewriting systems. The undecidability is shown e.g. in [1] (pages 94–99) by reducing a touring machine to a finite term rewriting system. They conclude the undecidability of termination for arbitrary term rewriting systems from the undecidability of the halting problem for the touring machine.

If we want to transform an expression using the rewrite rules of a term rewriting system, then zero, one or more rules may apply. If an expression is obtained to which no more rules apply, a normal form is reached and the rewrite procedure terminates. Is this always the case after finitely many rule applications, the system is called terminating. More formally: A term rewriting system terminates, if there is no infinite chain of rule applications $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots$. Single rewrite rules as well as the combination of applications of different rules may cause the system to be non-terminating. A popular example of a non-terminating rewrite rule is that for the commutativity of addition

$$a + b \rightarrow b + a$$

that obviously may generate an infinite chain. A system (or one single rule), that generates cycles, i.e. by the reoccurrence of a just transformed term later in the reduction chain, even as a subterm, is not terminating.

An approach to make termination decidable is to take only a subset of finite term rewriting systems instead of arbitrary ones. These can be the finite right-ground term rewriting systems. Right-ground means, that all right-hand sides are ground terms (contain no variables). For this special subcase, a decision procedure[4] for termination exists.

5.1 Reduction orders

There is still the need for deciding the termination of a given term rewriting system, even if there is no general algorithm available. For this purpose, the given TRS can be analyzed

[4]For the algorithm see [1] page 100.

using reduction orders. To define a reduction order, we will need the notions well-foundedness, strict order and rewrite order.

Let us shortly mention the property of well-foundedness, because all terminating relations are enjoying it. The property says, that there must not be any infinitely descending reduction chain

$$t_1 > t_2 > t_3 > \cdots$$

The properties "well-founded" and "terminating" are often used interchangeably in the context of orders. A rewrite order $>$ is a strict order[5] on the set of terms T, if it has the following two properties: It has to be

1. compatible with operations:
   $t_1 > t_2 \Rightarrow f(\cdots_1, t_1, \cdots_2) > f(\cdots_1, t_2, \cdots_2)$ and
2. closed under substitution: $t_1 > t_2 \Rightarrow \sigma(t_1) > \sigma(t_2)$.

Closed under substitution signifies, that an application of a substitution σ does not change the order, that held for two terms before. The first property means the same, but for application of a function f to the two terms t_1 and t_2 for that e.g. $t_1 > t_2$ held. The terms occur as arguments at the same position in f (indicated by the indexed ellipses). A good counterexample is the order over the size of terms, which may change by applying a substitution to the terms and thus is not closed under substitution. A reduction order is a well-founded[6] rewrite order.

A term rewriting system is terminating, if for all of its rules $t_1 \rightarrow t_2$ the order $t_1 > t_2$ holds. That means to us: To show termination of a given TRS, we have to find an appropriate reduction order that applies. It is now desirable to have a pool with as many reduction orders as possible to choose one for the TRS in the given situation. The goal is to make the finding of those reduction orders as automatable as possible for a given TRS.

5.2 Examples of reduction orders

In the following, some examples of possible classes of reduction orders are shown. Simplification orders are suitable for doing automated termination proofs. They are rewrite orders $>$ with the subterm property, that $t > t|_p$ for all terms t and all positions p in this term except the root position, with $t|_p$ denoting the subterm of t at the position p. Examples are

- polynomial simplification orders,
- recursive path orders and
- Knuth-Bendix orders.

A brief description of the recursive path orders: The point is, that they compare first the root symbols of two terms and then look at the direct subterms. The collection of subterms may be ordered, yielding the lexicographic path order, or unordered multisets. This procedure goes on recursively.

[5]A strict order is a transitive, irreflexive and asymmetric binary relation.

[6]It is also called Noetherian order.

6. CONFLUENCE

Assume, we have a term t and more than one rewrite rule of a term rewriting system is applicable, e.g. there are the rules $t \rightarrow t_1$ and $t \rightarrow t_2$. One is interested in the consequences that are caused by the non-determinance of the choice of rules to apply next. If the two terms t_1 and t_2 are again joinable, i.e. if there is a term t' such that each term can be transformed into it using the rewrite rules, the system is called confluent. Joinability is written like this: $t_1 \downarrow t_2$. The exact definition of the property confluence is the following: A TRS R is confluent, iff[7]

$$\forall t, t_1, t_2 \in T : t_1 \overset{*}{\leftarrow} t \overset{*}{\rightarrow} t_2 \Rightarrow t_1 \downarrow t_2$$

Thus, confluence is describing that terms in a system can be rewritten in many ways, to yield the same result (Figure 3).

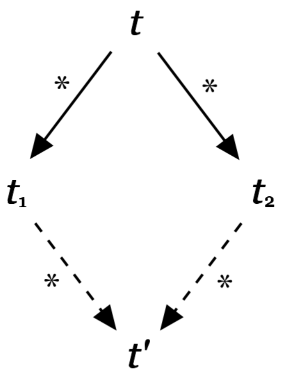
$$\begin{tikzcd} & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\ t_1 \arrow[dr, dashed, "*"'] & & t_2 \arrow[dl, dashed, "*"] \\ & t' & \end{tikzcd}$$

Figure 3: The confluence property.

There is another property, that is called the Church-Rosser property. A system is Church-Rosser (or has the Church-Rosser property), iff $t_1 \overset{*}{\leftrightarrow} t_2 \Rightarrow t_1 \downarrow t_2$. The conditions the "TRS is confluent" and the "TRS is Church-Rosser" are equivalent[8].

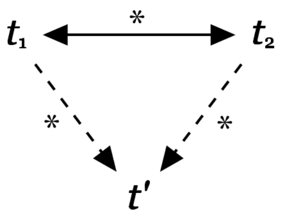
$$\begin{tikzcd} t_1 \arrow[rr, leftrightarrow, "*"] \arrow[dr, dashed, "*"'] & & t_2 \arrow[dl, dashed, "*"] \\ & t' & \end{tikzcd}$$

Figure 4: The Church-Rosser property.

Confluence is undecidable in the general case. A term t is irreducible, if no rules apply to it. We denote it by writing $t\downarrow$. $t'\downarrow$ is a normal form of t, if t can be transformed into t' using the rewrite rules of a term rewriting system. Such a normal form may not necessarily exist for a term t. There can also be multiple normal forms for one term. A system is called normalizing, if such a normal form exists in every

[7]The notation "iff" abbreviates "if and only if".

[8]The proof can be found in most of the literature that covers confluence of term rewriting systems, e.g. [1]

case. One of the main goals is, to have such unique normal forms.

What critical situations could appear, that would compromise confluence? The obvious situation is that more than one rule applies to a term t. But the critical situation only can occur, if the affected subterms, to which the rules may apply are not disjoint, for example, if the application of one rule prevents the application of the other one. The rules are then called overlapping (see Figure 5).

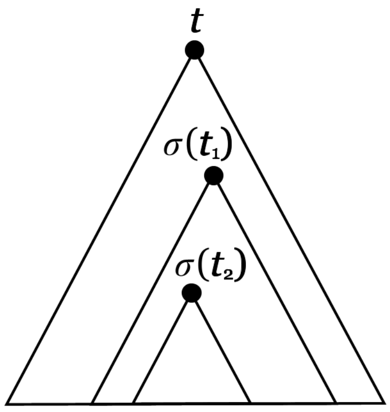

Figure 5: Overlapping situation for the rules $t_1 \to t_1'$ and $t_2 \to t_2'$.

6.1 Critical pairs

A critical pair is created by unifying the left-hand side of one rule with a non-variable subterm of the left-hand side of another one and reducing the gained term using both rules. Assume, that all of those critical pairs are joinable. In this case, a terminating TRS is confluent. Because a finite term rewriting system can have only finitely many critical pairs, the confluence of a finite and terminating term rewriting system becomes decidable.

We will have a look at a one-rule term rewriting system now:

$$R := \{f(f(x)) \to g(x)\}$$

By transforming the term $f(f(f(x)))$ we obtain a critical pair resulting from an overlapping of the rule with itself. The following transformations are possible:

- $f(f(f(x))) \to g(f(x))$ and
- $f(f(f(x))) \to f(g(x))$.

Because the two resulting terms are not joinable (both are in normal form and the one rule cannot transform them further) R is not confluent. Critical pairs can be used for completion (making a TRS convergent), if they are used in one rewrite rule to join the previously not joinable terms. By adding the rule $f(g(x)) \to g(f(x))$, the term rewriting system R becomes convergent (confluent and terminating):

$$\begin{aligned} f(f(x)) &\to g(x) \\ f(g(x)) &\to g(f(x)) \end{aligned}$$

There are also ways of determining the confluence of non-terminating term rewriting systems. This is handled, e.g. in section 6.3 in [1] with orthogonality.

6.2 Completing systems

Let's have a look at term rewriting systems, where some of the named properties are combined. If a TRS is confluent, all normal forms of a term t are identical. This is very nice, because one has not to be afraid of which rule to choose, if there is more than one applicable. The result is the same, unique normal form.

A terminating and confluent TRS is called complete (or convergent, canonical). One could be now interested in having a procedure, that makes non-confluent terminating TRS complete. Convergent term rewriting systems are important for solving the word problem, i.e. a convergent term rewriting system can decide the word problem for the underlying set of identities.

There are completion procedures, that take a finite set of identities and a reduction order and can produce an equivalent term rewriting system. In chapter 7 in [1] some completion procedures are introduced, e.g. Huet's completion procedure. The basic procedure may

- terminate successfully, if all critical pairs are joinable,
- fail in several cases or
- may run forever.

The completion described there is widely known as Knuth-Bendix completion. It tries to orient the identities from the input set (if $t_2 < t_1$ then $t_1 \to t_2$ can become a rule). This initial set of rewrite rules is completed with rules gained by detected critical pairs like shown in the example in 6.1.

Applications of completion algorithms:

- operational semantics of abstract data types (see [3] page 41)
- completion of given axiom systems (solving the word problem)

7. EXTENSIONS AND BRANCHES

This section will name some important extended rewriting concepts and special cases of rewrite systems.

- Rewriting modulo equational theories (dealing with non-terminating rules like commutativity)
- Higher-order rewriting, e.g. the map function

  $$\begin{aligned} map(f, empty) &\to empty \\ map(f, cons(x, xs)) &\to cons(f(x), map(f, xs)) \end{aligned}$$

  is no legal TRS, because f appears as a variable and as a function symbol in the second rule (see [1] page 270).
- Conditional rewriting (having conditional identities)

- String rewriting (with strings instead of terms)
- Graph rewriting (see the corresponding paper in this lecture)

Note

This paper was created during the course “Hauptseminar Methoden zur Spezifikation der Semantik von Programmiersprachen” at the Faculty of Computer Science at Technische Universität Dresden in winter semester 2006/2007.

8. REFERENCES

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[2] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 243–320. 1990.

[3] R. Hartwig. *Algorithmen für Termgleichungen – Termersetzungssysteme.* `http://www.informatik.uni-leipzig.de/~rhartwig` – Script to the lecture at the faculty of mathematics and computer science at the “Universität Leipzig”, 1998.

[4] S. Hölldobler. *Logik und Logikprogrammierung.* Dritte, durchgesehene Auflage. Synchron, Wiss.-Verl. der Autoren, Heidelberg, 2003.

Graph Rewriting

Applications of Graph Rewriting and the Graph Rewriting Calculus

Jan Lehmann
University of Technology Dresden
Department of Computer Science
Jan.Lehmann@inf.tu-dresden.de

ABSTRACT

This paper will give an overview over different approaches for graph rewriting, namely term graph rewriting, the graph rewriting calculus and an algebraic approach presented in [3]. Whereas term graph rewriting is little more than a slightly improved version of term rewriting, the graph rewriting calculus is a powerful framework for manipulating graphs.

1. INTRODUCTION

Rewriting in general is widely used. Not only in computer science but also in everydays life. Assume you want to buy two pieces of chocolate and each piece costs $0,70$€. In order to calculate the amount of money to pay, you write down (or think of) $2 * 0,70$€. The result of this, not really difficult, computation is $1,40$€. How this is correlated with rewriting? Formally your computation is the application of the following rewrite rule:

$$2 * 0,70 \rightarrow 1,40 \tag{1}$$

As presented in [4] term rewriting is a quite powerful tool to rewrite terms and can be widely used, for example in functional languages or logical languages. So why do we need an additonal theory for graph rewriting? The reason is, that terms can be represented as trees and trees have some severe limitations like the lack of sharing and cycles. Especially sharing can be useful when optimizing programming languages. Sharing means, that a node in the graph is referenced more than once. This is not possible in trees, because in a tree every node has only one incoming edge. However in a graph a node can have several incomming edges, allowing to point to this node from several other nodes in the graph. For example, when building the abstract syntax graph of a programming language, one only needs to reference a variable once, regardless of the number of occurences in the source code.

The possiblility to use cycles allows you to define rewrite rules like this:

$$x \rightarrow f(x) \tag{2}$$

As one can see here, the rule produces a right hand side which regenerates the left hand side again. The application of this rule would result in an endless cycle of function f calling itself:

$$f(f(f(\ldots))) \tag{3}$$

Another advantage of the graph rewriting approaches is, that they are higher order, what means that rewrite rules can be generated with other rewrite rules. This also means that one can match a variable to a function symbol. Imagine the rewrite rule from above is held more general:

$$2 * x \rightarrow double(x) \tag{4}$$

Lets also assume that the price of our piece of chocolate is given without VAT (value added tax). The VAT will have to be added by a function $vat()$. Due to the ability to assign function symbols to variables, one can match the left hand side of 4 with

$$2 * vat(0,59) \tag{5}$$

In order to be able to buy more than one piece of chocolate you need the higher order capabilities of term graph rewriting or the graph rewriting calculus, both presented in this paper. A nice side effect of being able to match variables with functions is, that you can share whole subgraphs in graph rewriting. In addition to saving memory, you need to evaluate the shared subgraph only once.

2. TERM GRAPH REWRITING

This section will provide an operational approach on graph rewriting, defining a graph as set of nodes, a successor function and a labeling function.

2.1 Notation

Term graphs are graph representations of (algebraic) terms. They can be represented as tuple $G = (N, s, l)$ where N is a set of nodes in the graph. s is defined as function returning the direct successor of a given node: $s : N \rightarrow N^*$ and l provides a label for each node in N. There is a special label for empty nodes: $\perp$. Typically empty nodes are nodes representing variables.

Term graphs may contain roots and garbage. If a term graph has a root, it will be denoted as $G = (r, N, s, l)$ where $r \in N$ is the root node of the graph. Nodes are called garbage if

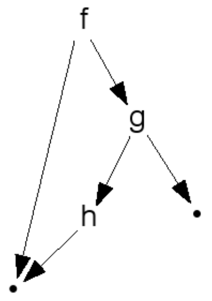

Figure 1: $f(x, g(h(x), y))$

there is no path between the root and these nodes. Garbage can be removed via garbage collection.

2.2 Representation

This leads to the question, how one can represent term graphs graphically. Lets take algebraic expressions as example. Every part of the expression becomes a node in the graph. Constant values or algebraic variables like x in $f(x)$ are seen as empty nodes and drawn as $\cdot$.

*E*xample 1: The term $f(x, g(h(x), y))$ will lead to the following term graph: $G_1 = (n_1, N, s, l)$ where:

- $N = \{n_1, n_2, n_3, n_4, n_5\}$
- $s : s(n_1) = n_2n_3, s(n_3) = n_4n_5, s(n_4) = n_2, s(n_2) = s(n_5) = e$
- $l : l(n_1) = f, l(n_3) = g, l(n_4) = h, l(n_2) = l(n_5) = \perp$

The rules above define a graph with 5 nodes. The nodes n_2 and n_5, representing the variables x and y are leafes of the graph. The resulting graph looks like in figure 1.

2.3 Rewriting

To be able to rewrite term graphs, we need some kind of formalism to define that two term graphs are equal. This formalism is called graph homomorphism and means a function that maps the nodes of one graph G_1 to the nodes of a graph G_2. A variable in one graph may be mapped to any kind of subgraph of the other graph. That means that the following homomorphism is possible:

$$G1 : add(x, x) \rightarrow G2 : add(s(y), s(y)) \qquad (6)$$

Figure 2 shows which parts of G_1 are matched against which parts of G_2.

The actual rewriting is defined by rewrite rules. A rewrite rule is a triple of form (R, l, r). R is a term graph and l, r are the roots of the left and right hand sides respectively. Therefore $gc(l, R)$, which means the term graph R with l as root node after garbage collection, is the state of the term graph to rewrite (or a subgraph of it) before the rewriting and $gc(r, R)$ after the rewrite step.

The rewriting itself is done as follows:

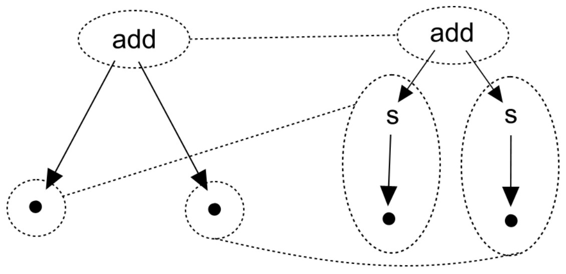

Figure 2: $G_1 \ll G_2$

- Find a subgraph that matches to the left hand side.
- Copy the parts of the right hand side to the graph to rewrite when there are no occurences yet.
- Redirect all pointers to the root of the left hand side to the root of the right hand side.
- Run garbage collection to remove parts of the left hand side, that are not reachable anymore.

Example 2 ([2]) shows the application for rewriting a graph defining the addition of two natural numbers. The term to rewrite is $add(s(0), s(0))$ where add is the addition of two natural numbers and s denotes the successor of a number. In fact this term simply formalizes the addition 1+1. This term as graph will look like the left graph in figure 3. To rewrite this term we use the rule $add(s(x), s(y)) \rightarrow s(add(x, s(y)))$. This rewrite rule is drawn as graph 2 in figure 3. This notation is a little bit confusing because both, the left hand side and the right hand side are drawn into one graph. To see the left hand side one has to set the left upper node as root and run garbage collection, which removes all nodes that are not reachable from this root. The same could be done for the right hand side. Now we can match the left hand side of the rule to the graph to rewrite. the left $\bullet$ will be mached to x and the right $\bullet$ will be mached to $s(y)$. Now we copy all nodes of the right hand side of our rewrite rule to graph 1 and reset the root to the root of this right hand side. After doing this, we remove all nodes, which are not reachable anymore. This are the left nodes labeled add and s. As result of this operation we get a graph looking like graph 3 in figure 3 representing the term $s(add(0, s(0)))$. In order to finish the addition one would define a second rewrite rule like $add(0, x) \rightarrow x$ to declare that the addition of 0 and a number equals to the number.

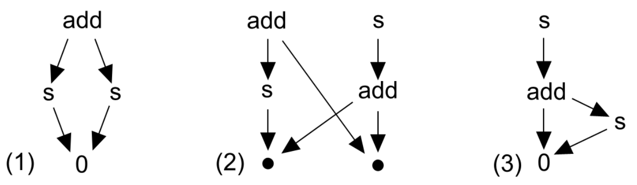

Figure 3: Example 2

3. THE GRAPH REWRITING CALCULUS

The approach of term graph rewriting is rather simple and understandable, but also rather weak. Weak means in this case, that one has no control about the application of the rewriting rules. Rules are always applied where possible. This is because the application of rewrite rules is done at meta level, not at object level. There exist more sophisticated methods to deal with graph rewriting. One is the extension of the ρ calculus, the so called ρ_g calculus or Graph Rewriting Calculus as presented in [2].

3.1 Overview

The ρ_g calculus is a higher order calculus, which main feature is to provide an abstraction operator, that is capable of handling more complex terms on the left hand side, than just simple variables. An example of an abstraction is $f(x) \to x$. As one can see, an abstraction is denoted by "$\to$". Such an abstraction can be seen as rewrite rule. The application of an abstraction is written as follows:

$$(f(x,x) \to 2x)f(a,a) \tag{7}$$

This means that the abstraction inside the first paranthesis is applied to the term $f(a,a)$. As in ρ calculus the application of such an abstraction is done via a constraint. The application in (7) would result in

$$2x[f(x,x) \ll f(a,a)] \tag{8}$$

Every application of a rewrite rule to another term is done like this. First the right hand side of the rewrite rule is written as term and than the matching of the left hand side of the rewrite rule and the term, the rule was applied to, is written as constraint to this right hand term. The operator $\_\,[\,\_\,]$ is called a constraint application. These constraint applications are useful for matching terms, that will be explained in the subsection about reducing terms. It also allows us to describe sharing of constants, as mentioned in the section about graph rewriting. There is not nessessarily only one constraint in ρ_g calculus but a list of constraints, which are separated by comma. Such a constraint list can be generated when a constraint is applied to a term inside another constraint or if a function of arity ¿ 1 is matched. In the last case every parameter of the left hand side of the matching is matched against the corresponding parameter of the right hand side of the matching.

Another operator I did not consider so far is $\wr$. It is the so called structure operator which by default has no fixed semantics. This fact shows, that the graph rewriting calculus is more framework than a ready to use tool.

3.2 Further formalisms

Before actually showing how the example above is rewriten, I have to give some additional formalisms. One referes the operator $\_ \ll \_$. This operator is called matching operator, which means it tries to match the left and the right hand side. A special form of matching operator ist $\_ = \_$, that can be seen as association, for example $x[x = a]$ means that variable x is associated with value a.

Another important thing is the theory of bound and free variables. A bound variable is a variable that occurs on a left hand side of a matching constraint. The table below shows how the free ($\mathcal{FV}$) and bound variables ($\mathcal{BV}$) are determined.

G	$\mathcal{BV}$(G)	$\mathcal{FV}$(G)
x (var)	$\emptyset$	$\{x\}$
k (const)	$\emptyset$	$\emptyset$
$G_1 G_2$	$\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$
$G_1 \wr G_2$	$\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$
$G_1 \to G_2$	$\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_2) \setminus \mathcal{FV}(G_1)$
$G_0\,[E]$	$\mathcal{BV}(G_0) \cup \mathcal{BV}(E)$	$(\mathcal{FV}(G_0) \cup \mathcal{FV}(E)) \setminus \mathcal{DV}(E)$

In a constraint E there is an additional type of variables, the defined variables $\mathcal{DV}$.

E	$\mathcal{BV}(E)$	$\mathcal{FV}(E)$	$\mathcal{DV}(E)$
ϵ	$\emptyset$	$\emptyset$	$\emptyset$
$x = G_0$	$x \cup \mathcal{BV}(G_0)$	$\mathcal{FV}(G_0)$	$\{x\}$
$G_1 \ll G_2$	$\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$	$\mathcal{FV}(G_2)$	$\mathcal{FV}(G_1)$
E_1, E_2	$\mathcal{BV}(E_1) \cup \mathcal{BV}(E_2)$	$\mathcal{FV}(E_1) \cup \mathcal{FV}(E_2)$	$\mathcal{DV}(E_1) \cup \mathcal{DV}(E_2)$

The distinction of free and bound variables is important for the so called α-conversion. This means renaming of variables when evaluating a term. Renaming might become nessessary when applying a rewrite rule to a term which has equaly named variables as these rewrite rule. In order to prevent free variables to become bound accidently they are renamed before the actual matching. Lets assume you have a ρ_g term of form $G \to H$ where G and H are ρ_g terms themselfes. In case of α-conversion every free variable in G gets another, not yet used, name in order to prevent it from becoming bound accidently. This is similar for constrainted terms. There the term where the constraint is applied to will get it's free variables renamed if there is a equaly named variable in one of the left hand sides in the constraint list.

3.3 Graphical Representation

As long as there are no constraints in the graphical representation of ρ_g -terms is just as defined in the section about term graph rewriting above. Recursion can be represented as (self-)loops and sharing as multiple edges. A little bit problematic is the representation of matching constraints. These constraints are terms which can be drawn as graphs themselfes but they do not really belong to the main graph. [2] suggests drawing of these matching constraints as separate boxes. The boxes can be nested, due to the fact, that matching constraints can be nested, too. The roots of the boxes and sub-boxes are marked with $\Downarrow$. Figure 4 visualizes the rewrite rule $mult(2, s(x)) \to add(y, y)[y = s(x)]$ An interesting question on this example is: Why isn't the $s(x)$ on the left hand side shared, too? This was not done in order to keep the term wellformed. The condition of wellformedness demands that there is no sharing between the left and the right hand side of a ρ_g-term.

3.4 Rewriting

So how to do the rewriting in ρ_g calculus? To demonstrate this lets take our example from the term graph rewriting section. The rewrite rule we want to apply is

$$add(s(x), y) \to s(add(x, y)) \tag{9}$$

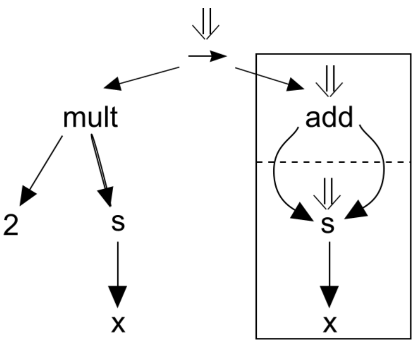

Figure 4: $mult(2, s(x)) \rightarrow add(y, y)[y = s(x)]$

and it is applied to the term

$$add(s(0), s(s(0))) \tag{10}$$

which is in fact the very complicated computation 1+2. The rewriting goes as follows. I will describe the steps after the example.

$(add(s(x), y) \rightarrow s(add(x, y)))add(s(0), s(s(0)))$
$s(add(x, y))[add(s(x), y) \ll add(s(0), s(s(0))]$
$s(add(x, y))[s(x) \ll s(0), y \ll s(s(0))]$
$s(add(x, y))[x \ll 0, y = s(s(0))]$
$s(add(x, y))[x = 0, y = s(s(0))]$
$s(add(0, s(s(0))))$

The first two lines follow the claims I made in the Overview part of this section. It is the application of the rewrite rule in (9) to the term in (10). In line three the function *add* was matched and splits up into two constraints which are applied as list to the term before [_]. In the fourth step this is done a second time with the successorfunction. The matching operator between y and $s(s(0))$ is replaced by the application operator. This can be done because y is a variable in the term and there is no need or possiblity to further match it in any way. The same is done in line 5 where x equals to 0. The last line is finally the result of the rewriting. Unlike the pure ρ-calculus the ρ_g-calculus does not require the last step, which may be an advantage if one wants to preserve the sharing of equal parts.

3.5 Confluence and Termination

As presented in [4] there is no guarantee that the application of rewriting steps in term rewritings terminates. One can easily see that this also holds for the ρ_g calculus. First of all, graphs may contain cycles. If a rewrite rule matches one of these cycles but does not eliminate it, it will be applied over and over again. A second example to show that a rewriting may not terminate is the following set of rules:

$f(x) \rightarrow x$
$x \rightarrow y$
$y \rightarrow f(x)$

The reduction always comes to it's starting point and restarts again.

In general there is also no clue that (graph-)rewriting is confluent. However the graph rewriting calculus restricts the left hand sides of it's rewrite rules in order to achieve this property. The restriction is that the left hand sides have to appliy to linear patters. A linear pattern is formally defined as follows:

$$\mathcal{L} := \mathcal{X} \mid \mathcal{K} \mid (((\mathcal{K}\ \mathcal{L}_0)\mathcal{L}_1)\ldots)\mathcal{L}_n \mid \mathcal{L}_0[\mathcal{X}_1 = \mathcal{L}_1, \ldots, \mathcal{X}_n = \mathcal{L}_n] \tag{11}$$

where 2 patterns $\mathcal{L}_i$ and $\mathcal{L}_j$ are not allowed to share free variables if $i \neq j$. Furthermore a constraint of form $[L_1 \lll G_1, \ldots, L_n \lll G_n]$, with $\lll$ is either $\ll$ or $=$, is called linear if all patterns L_i are linear. The complete proof of confluence in the linear ρ_g calculus can be found in [2].

4. AN ALGEBRAIC APPROACH

Another approach for rewriting of graphs was presented in [3]. The paper describes graphs as logical structures and their properties with help of logical languages. Graphs are described as classes $D(A)$, where D is the class and A is an alphabet containing the labels for the edges of graphs in D. Such a graph can be represented as follows:

$|G|_1 := \langle V_G, E_G, (edg_{aG})_{a \in A}\rangle$
$|G|_2 := \langle V_G, E_G, (lab_{aG})_{a \in A}, edg_G\rangle$

Where V_G and E_G are the sets of vertices and edges respectively, $lab_{aG}(x)$ means an edge x labeled with a, $edg_G(x, y, z)$ is an edge x from vertex y to vertex z and $edg_{aG}(x, y, z)$ describes the combination of $lab_{aG}(x)$ and $edg_G(x, y, z)$.

4.1 Properties of graphs

In order to match graphs for rewriting, one has to define isomorphisms between graphs. [3] defines two graphs as isomorphic if there exist bijections from V_G to $V_{G'}$ and from E_G to $E_{G'}$. A property of a graph is a predicate over a class of graphs. Such predicates may be expressed by logical languages like first order logic, second order logic and so on. The more powerful a language is, the better is it's expessiveness. A discussion about the expressiveness of several logical languages can be found in [3]. The following example expresses the property that there are no isolated vertices in a graph, that means that every vertex is linked somehow with the other parts of the graph.

$$\forall x \exists y \exists z \left[\bigvee edg_a(z, x, y) \vee edg_a(z, y, x) | a \in A \wedge \neg(x = y)\right] \tag{12}$$

4.2 Graph manipulations

In this approach graphs are not manipulated directly but overlayed with so called hypergraphs. Each hypergraph consists of hyperedges which are defined through a label and zero to many edges of the graph. There are three operations defined over these hypergraphs. The first is the addition of two hypergraphs denoted by $\oplus$. This operation merges the edges and vertices of two hypergraphs into one. The operation $\theta_{\delta,n}$ describes the merge of two vertices, i.e $\theta_{1,3}$ means that vertices 1 and 3 are merged into one vertex. Finally the operation $\sigma_{\alpha,p,n}$ describes renaming of vertices. The expression $\sigma_{1,4}$ means that vertex 1 is renamed to 1 and vertex 4 is renamed to 2, according to their positions in the subscript.

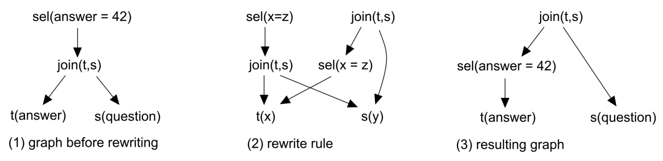

Figure 5: Rewrite of a SQL execution graph

With these basic operations one can define derived operations like the series-composition of hypergraphs:

$$G \cdot G' = \sigma_{1,4}(\theta_{2,3}(G \oplus G')) \tag{13}$$

First the hypergraphs G and G' are added together. Then the θ-operation merges the second vertex of G (2) with the first vertex of G' (3) and finally the first vertex and the last vertex of the resulting hypergraph get new labels. The vertex in the middle is not labeled at all.

5. APPLICATIONS

Now we know what possiblities we have to rewrite graphs, but where is this useful? Graph rewriting is especially useful for optimization purposes. Everything that can be represented as graph can be optimized with help of graph rewriting, where the optimizations are encoded as rewrite rules (see also [1]). An example for this may be an SQL statement. Each statement can be transformed into a query execution plan, which essentially is a tree or graph. There exist several rules of thumb, like the rule, that a selection should be executed as early as possible in order to keep the amount of tuples as small as possible. This kind of optimization is called static because it does not use any knowledge of the data to transform. Lets assume we have a simple SQL statement like

```
SELECT *
FROM table1 t, table2 s
WHERE t.id = s.id;
AND t.answer = 42;
```

A query execution plan may look like in figure 5.1. The application of the rule in figure 5.2 would result in the graph of figure 5.3.

Another example, similar to the one above, is the execution graph of functional languages. One can imagine that there are similar rules as in the SQL example. The representation as graphs will save a severe amount of space because variables (in the second case) or whole subqueries (in the first example) can be shared.

The third application of graph rewriting, i would like to mention, is optimization and analysis of syntax graphs in languages like C♯ or Java. Rewriting can assist you in refactorings, too. The desired refactoring can be encoded as graph rewrite rule and will be applied to the syntax graph.

6. CONCLUSION

As we have seen, there exist several approaches for graph rewriting. All of them have some properties, like sharing and the possibility to handle cycles, in common, which make them to powerful tools for changing the structure of graphs. However, these approaches use different ways for defining rules and graphs and therefor differ in power and flexibility. The most intuitive way is term graph rewriting, but it's also a bit limited. The graph rewriting calculus is a powerful framework, which allows to build an own calculus on top of it. Common to all approaches is, that you can finally buy more than one piece of chocolate.

7. REFERENCES

[1] U. Assmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, 2000.

[2] C. Bertolissi. The graph rewriting calculus : confluence and expressiveness. In G. M. P. Mario Coppo, Elena Lodi, editor, *9th Italian conference on Italian Conference on Theoretical Computer Science - ICTCS 2005, Siena, Italy*, volume 3701 of *Lecture Notes in Computer Science*, pages 113–127. Springer Verlag, Oct 2005.

[3] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 193–242. Elsevier, Amsterdam, 1990.

[4] A. Rümpel. Rewriting. 2007.