Code Selection Techniques:
Pattern Matching, Tree Parsing, and Inversion of Derivors

Robert Giegerich
Karl Schmal

FB Informatik - Lehrstuhl V
Universität Dortmund
Postfach 500500
D-4600 Dortmund

1. Pattern Matching Approaches to Code Selection

1.1 Summary

Significant progress in the area of formal tools to support the construction of code generators in recent years has come along with a severe terminological confusion. Closely related techniques appear in different styles, further obscured by ad-hoc extensions. We try to alleviate this situation by suggesting that the code selection task should be understood as the problem of inversion of a hierarchic derivor. This understanding leads to several interesting generalizations. More expressive formalisms - heterogeneous tree languages, regular tree languages, derivor images - can be used to define the code selectors input language. In all cases, we retain the ability to decide the completeness of the code selector specification as a side-effect of code selector generation. The extension to nonlinear matching, in combination with matching relative to a subsignature M with a nontrivial equational theory, allows to express the non-syntactic conditions formerly associated with a production in a Graham-Glanville style code generator description. Due to space restrictions, such extensions can only be sketched here, while the emphasis of this paper lies on motivating and demonstrating our reformulation of the classical pattern matching approach to code generation.

1.2 A Short Review of Recent Approaches to Code Selection

Initiated by the work of Graham and Glanville [GrGl77], many approaches to retargetable code generation have been presented in recent years. Commonly and correctly, they are subsumed under the phrase "pattern matching techniques". Some form of pattern matching is used to guide code selection, while other subtasks of code generation, such as register allocation, cost comparison of different coding alternatives, or evaluation ordering must be organized in some way along with the matching process. Henry [Henr84] has carefully and extensively demonstrated the limitations inherent in the original Graham-Glanville approach, which used LR-parsing techniques for pattern matching. We are particularly concerned here with work that attempts to overcome these limitations. While Graham and Glanville had demonstrated how the syntax-directed translation paradigm could be beneficially applied to code selection, it became clear that two kinds of improvements were desirable: using a more flexible kind of pattern matching for the "syntactic" aspects, and new techniques to describe other subtasks of code generation that have to be performed along with and are directed by the matching process. Several related suggestions have been made with respect to the first task, while the second has received less systematic treatment. As most workers in this area have observed, off-the-shelf tree pattern matching in the "classical" sense of [Kron75] or [HoOD82] is close to providing a solution, but is not quite expressive enough to serve as an adequate technique for code selection. Unfortunately, all approaches developed their own terminology and extensions, and the concepts used to formulate the individual pattern matching techniques have not been adequately separated from their particular application to the code generation task. As a result, the relative virtues of the different approaches can hardly be evaluated, as a comparison can only be made at a most technical or empirical level. Let us give a short discussion of the approaches we are referring to.

First variations of the Graham-Glanville approach still used string parsing techniques, e.g. [Henr84] and [CHK85]. [Turn86] uses a technique called up-down parsing, and actually parses trees, but sees his grammars still as string grammars. [Gieg84] and [Benk85] use parsing with (regular) tree grammars, resorting to classical terminology from formal language theory. [HaCh86], [WeWi86] and [AhGa85],[AGT87] use the name pattern matching, but with different meanings. Recent approaches, yet to be worked out further, try to embed code generation in a formalism of algebraic equational specifications and term rewrite systems [Gieg85], [MRSD86].

Let us further exemplify the terminological inhomogenity by a look at corresponding notions in different approaches. Maybe the best-understood terminology is that of nonterminals, terminals and productions of a tree grammar, as everyone can understand a tree grammar as a context free grammar where the righthand sides of productions are trees. (This view is used frequently, but it does have a pitfall, which we will address later.) [Turn86] is closest to this terminology, speaking of nonterminals, operators and prefix (string) expressions. [AGT87], [WeWi86] and [HaCh86] use "patterns" for productions. [AGT87] uses "labels" and "operators" for nonterminals and terminals, while [WeWi86] uses "labels" for both concepts. [HaCh86] calls terminals "node-type", and it seems at the first glance, that nonterminals show up here as "renaming symbols". But in fact, the correct counterpart of nonterminals is an index into some table, which may either be a "renaming symbol" or some proper sub-pattern. [Chas87], contributing a significant improvement of the classical pattern matching algorithm of [HoOH82], also discusses extensions necessary for code selection applications. He uses the most creative naming, calling nonterminals "introduced wildcards". Finally, with algebraically oriented approaches, we know (e.g. from [ADJ78]) that nonterminals correspond to the sorts of some signature, with terminals denoting the operators.

Of course, if this was only an inconsistency of namings, it would not be worth bothering about. But to the extent that formal language terminology is abandoned, the concept of a derivation disappears - and this concept is in fact a very useful one in the given context - although not quite sufficient. In spite of all similarities, it does make an important difference for the expressive power of an approach, whether it uses "trees over a ranked alphabet" or "terms from a given signature" as its basic concept. It is one of our goals in this paper to explicate these differences, which are often considered negligible.

We conclude this little survey by another observation with a similar lesson. Glanville [Glan77] originally addressed the problem of completeness of the code generator description (the "machine grammar"), partly ensuring it by imposing the condition of "uniformity". Much later work has been designed to remove this restriction. Interestingly, the more these approaches deviate from formal language terminology, the less inclined they are to address the completeness aspect.

2. A Sketch of an Algebraic Model of Code Generation

The overall goal of this work is not to suggest *another* pattern-directed technique for code selection. Our goal is a reformulation of such techniques, in a way suitable to handle code generation as well as other applications, combining the virtues of three areas:

- Efficiency and known generative techniques from classical pattern matching [Kron75], [HoOD82], [Chas87];
- clean concepts and decidability results from formal language theory [Brai69], [Benk85], together with a modest gain in expressive power;
- powerful specification, implementation and proof techniques available in equational algebraic specifications [ADJ78], [HuOp80], [HuHu80].

In order to show how this can be achieved, we must first sketch our understanding of code generation. This subsection is an excerpt of a more substantial investigation in the theory of code generation (unpublished at this point). A predecessor of the model sketched here can be found in [Gie85]. The goals of this work are shared by the approach of [MRSD86], which describes work on the design of a code generation tool based on term rewriting techniques.

To arrive at a model of code generation with the desired properties, we must break with two paradigms prevalent in previous approaches to code generation. The first is the "code emission paradigm". Typically, in code generator descriptions there are "actions" or parameterized code

strings associated with the patterns, which, upon a match of the template, trigger the emission of target machine or assembly code to some file. The problem with this is that when code is emitted right away, it has to be perfect from the beginning. This leads to a tendency to overfreight the pattern matching with other tasks such as register allocation or peephole optimization, which should preferably be described separately, at least on the conceptual level. Instead, the code we generate will be machine code in abstract syntax, and we disregard the task of writing a linearization of it to some file.

The second paradigm we abandon is that "machine description" and "code generator description" have traditionally been treated as synonyms. It turns out to be very important to formally distinguish these two notions. The machine description says what (abstract) target programs are, the code generator specification says how they are related to source programs. We will now discuss approaches to code generation as if they had always been using our conceptual model. We focus on the central task of instruction selection for arithmetic and addressing calculations.

Let Q and Z be many-sorted signatures. Source (= intermediate) and target (= machine) language programs are terms in the term algebras T(Q) and T(Z), respectively. Code generation requires (among other tasks) to specify and implement a code selection morphism γ: T(Q) --> T(Z).

Two ways have been used to obtain γ. In handcrafted compilers, as well in systematic approaches striving for retargetability like [ACK83], one considers all relevant operator/ operand combinations in T(Q), and specifies for each some term from T(Z) as its target code. If good code is desired, the necessary analysis of special cases becomes intricate and error-prone. It was an important observation of [GrGl77], [Catt78], [Ripk77], that it may be more convenient to describe the target machine instructions in terms of the intermediate language, rather that vice versa. Hence, for each Z-operator (typically representing a machine instruction), one specifies some semantically equivalent term from T(Q). Let us call this description δ. With some right, it could be called a machine description rather than a code generator description. It specifies a homomorphism δ: T(Z) --> T(Q). (But traditionally it has not been understood this way, due to the two paradigms discussed above.) Algebraically, δ is a derivor [ADJ78] from Z to Q. Now the arrow goes the "wrong" way, but to obtain γ from δ, pattern matching techniques can be used. So, the gains of this approach are twofold - descriptive ease, and generative support for an efficient and correct implementation of the required case analysis. (Ignoring this aspect leads to the argument in [Hors87] that Graham-Glanville style code generation goes the "wrong way".)

A third advantage of the latter approach is the following. Inherent in the task of code generation there is some freedom of choice, usually exploited to optimize target code quality. The specification δ preserves this freedom. Correspondingly, to obtain the desired γ, one must not only invert δ, but also supply a choice function ξ wherever for some $q \in T(Q)$, there are several $z \in T(Z)$ with $\delta(z)=q$. For example, dynamic programming accoording to [AhJo76] has been used to implement this choice. On the conceptual level, we would like to have the functionality $T(Q) \xrightarrow{\delta^{-1}} 2^{T(Z)} \xrightarrow{\xi} T(Z)$, while on the technical level, we want to interleave ξ with the construction of δ^{-1}.

The same holds for other subtasks of code generation, such as register allocation, evaluation ordering, or machine specific data type coercions. In current approaches, these tasks have not found a formal specification. In our algebraic framework, they are described along with Q, Z, and δ by equational specifications for a so-called semantic subsignature M of Q and Z. ([Gieg85] provides an example of such a specification.) The main concern of our framework is that *all* aspects of code generation can be described formally and in a modular way, thus allowing proofs of completeness and correctness. But for the moment, this is still an open promise, and not the subject of the current paper.

With formal definitions still postponed, we illustrate the above by a small example.

Example 1

"semantic" subsignature M =

sorts Number Type

operators
word: --> Type
long: --> Type
0, 1, 2, ...: --> Number
scale: Type --> Number

equations
scale(word) = 2
scale(long) = 4.

"source" signature Q = M +

sorts E

operators
const: Number --> E
+, *: E E --> E
mem: Type E --> E
reg: Type Number --> E.

"target" signature Z = M +

sorts Exp Adr

operators
ADD: Exp Exp --> Exp
ADDI, MULI: Exp Number --> Exp

R: Type Number --> Exp
M: Type Adr --> Exp

mk_adr: Exp --> Adr
bdx: Type Type Number Number Number --> Adr.

◊

The above target signature is a little contrived, in order to demonstrate both nonlinearity and the use of nontrivial semantic subterms. Z-operator bdx denotes the addressing mode "base-displacement-addressing with index", where the first argument indicates the word length to be used in the address calculation, while the second indicates the word length of the addressed memory cell, used to determine an automatic scaling factor. The other arguments are base and index register number, and the displacement.

We can now explain the role of M. It serves a threefold purpose.

i) Usually there are several variants of instructions or addressing modes like our bdx - depending on the available choices of operation, address and operand length. In Graham-Glanville style descriptions, this led to a phenomenon called "type crossing" [Henr84]: The size of the machine description is (essentially) multiplied by the number of machine data types, leading to extremely large descriptions. At least for the sake of readability, parameterization of the description is called for (which may be expanded automatically). In our approach, we just use machine data types as extra arguments to Z-operators, keeping the description concise without extra parametrization mechanisms, and avoiding expansion.

ii) From the beginning in [GrGl77], patterns have been augmented by semantic attributes, which were instantiated by concrete register numbers or constant values, and used to test semantic restrictions on the applicability of a pattern. Again, we need no special attribute concept for this purpose - the register number is just another argument (with a sort from S_M) of the reg operator.

iii) Functions calculating and predicates testing semantic attribute values in Graham- Glanville style descriptions are defined outside the formal specification. Here, they are defined as operators of OP_M by equations in E. "Testing a semantic predicate" is thus elegantly subsumed in the notion of matching modulo $=_E$.

This is further demonstrated by the second part of Example 1, the derivor specification. Note that the equation for bdx is nonlinear in t - our target machine uses word or longword addresses, but both base address and index must be of the same length. The M-subterm scale(tt) will match the numbers 2 or 4, but no others - according to the equations specified with M.

Example 1 - continued

hierarchic derivor δ:Z --> Q

(By definition, δ is the identity on M, and hence this part of δ is not shown.)

sort map δ: Exp --> E
Adr --> E

operator implementation equations -- using infix notation for + and * --

δ(M(t, a)) = mem(t, a)

δ(R(t,i)) = reg(t, i)

δ(ADD(e, f)) = e+f

δ(ADDI(e,n)) = e+const(n)

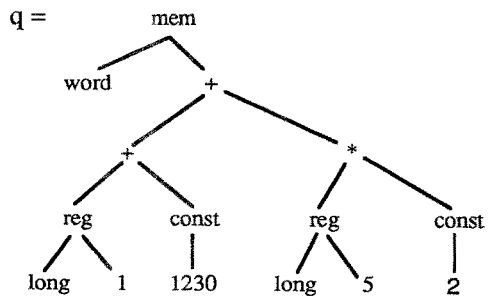
δ(MULI(e, n)) = e*const(n)

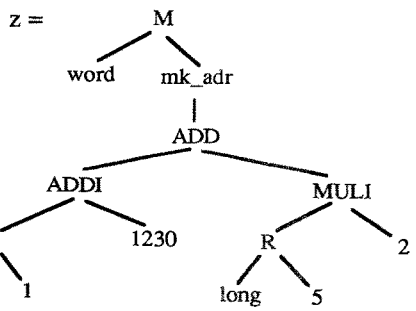
δ(bdx(t, tt, i, j, n)) = (reg(t, i) + const(n)) + reg(t, j) * const(scale(tt))

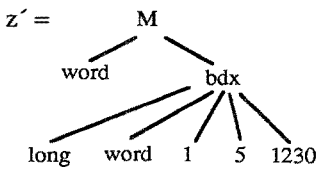
δ(mk_adr(e)) = e

◊

Example 2







Example 2 shows z, z´ such that $\delta(z) =_E q$ and $\delta(z') =_E q$, which can be verified formally from the definition of δ. The terms describe a memory word addressed via base, displacement and scaled index, which could be written (in a VAX-like notation) either as operand 1230(R1.L)[R5.L] in a word instruction, or as operand (R1) preceded by the code sequence ADDI.L R1, 1230; MULI.L R5, 2; ADD.L R1, R5.

The reader is invited to attempt to construct z or z´ from q by performing the appropriate pattern matching, for the moment on an intuitive basis. Note also that if we replace the constant 2 in q by (say) another register, there is no target term for the modified q. The specification is incomplete.

3. Homogeneous Tree Languages, Heterogeneous Term Algebras, and Regular Tree Languages

This chapter introduces familiar concepts and notes some straightforward correspondencies and differences.

3.1 Homogeneous Tree Languages

Definition 1: Ranked Alphabet, Homogeneous Tree Languages

A ranked alphabet is a finite set A of symbols, together with a function rank, such that

- rank(a)≥0 for each a∈A.

The homogeneous tree language trees(A), also called the set of trees over the ranked alphabet A , is defined by

- a∈trees(A), if rank(a)=0,
- $a(t_1, ..., t_n) \in trees(A)$, if $t_i \in trees(A)$ for $1 \le i \le n$, a∈A, and rank(a)=n.

◊

Example 3

Let A1 be the alphabet {0, 1, cons, nil}, with ranks 0, 0, 2, 0, respectively. Elements of trees(A1) are, for example: 0, nil, cons(nil, 1), cons(cons(0, 1), cons(nil, nil)).

◊

We shall also depict trees in graphical form, as in Example 2.

Observation 1

Consider the subset lists(A1) of trees(A1) containing the constants 0 and 1, plus all linear lists of binary digits, such as nil or cons(1, cons(0, nil)). It cannot be described as a homogeneous tree language trees(A) for any A.

◊

3.2 Regular Tree Languages

Definition 2: Regular Tree Grammars and Regular Tree Languages

A regular tree grammar G is a triple (N, A, P), where

- N is a finite set of nonterminal symbols,
- A is a ranked alphabet of terminal symbols, A∩N=Ø,
- P is a finite set of productions of the form X --> t, with X∈N and t∈trees(A∪N), with nonterminals given rank 0.

For t, t´∈trees(A∪N), t immediately derives t´, written t -> t´, if there is a p∈P, say X->t´´, such that t´ results from t by replacing a leaf labelled X by t´´. Where relevant, we indicate the p used in the derivation step, writing $->_p$. The relations $->^+$ and $->^*$ are the transitive and the transitive and reflexive closure of ->.

L(X) := {t | t ∈trees(A) and X-->*t}, and
L(G) := {t | t∈L(X) for some X∈N} is the language of G.
◊

In our context, we are not interested in a particular root symbol, and so it was omitted from Definition 2.

Example 4

Let G1 be the regular tree grammar ({D, R}, A1, P1} with A1 as before, and
P1 = { R --> nil
R -->cons(D, R)
D -->0
D -->1}.
Clearly, L(G1) = lists(A1) of Observation 1.
◊

We call these grammars *regular* (following [Brai69]), since in their formal properties, they are a generalization of regular string grammars, rather than of context free string grammars. The usual view of Graham-Glanville style machine descriptions as *context free* grammars, with *prefix expressions* on the righthand sides of productions, does not adequately express how strong the restriction to "prefix expressions" really is. For example, language containment is undecidable for context free string grammars, but decidable for regular string grammars.We note:

Observation 2

For regular tree grammars G_1 and G_2, $L(G_1) \subseteq L(G_2)$ is decidable.
◊

A proof of this fact is given in [Benk85], although this result is probably much older. We will see a very important application of this result later. In the sequel, we shall need a few more technical notions:

Definition 3

A production of the form X -->Y, with X, Y ∈ N is a chain rule. Other productions are called structural rules.

A derivation containing a subderivation of the form X -->$^+$X (using chain rules only) is called a circular derivation.

A sentential form of G (for X) is a tree t∈trees(A∪N) such that X -->*t.
◊

Observation 3

t∈L(G) has infinitely many derivations iff it has a circular derivation.
◊

Of course, regular tree grammars may be ambigous, but observation 3 tells us, that if we restrict our attention to noncircular derivations, there will be only finitely many such derivations for any t∈L(G).

3.3 Heterogeneous Term Algebras

For the standard terminology we need from algebraic specifications, we do not give full formal definitions (see [HuOp80] for that matter), but only explain the notation we are going to use.

Definition 4: Signatures, Specifications

For a signature Σ = (S, OP), T(Σ) denotes the set of terms over Σ, while the set of terms of a

particular sort s is denoted $T(\Sigma)$:s. $T(\Sigma, V)$ is the set of Σ-terms over an S-sorted set V of variables. A specification (Σ, E) is a signature Σ together with a set E of equations of form $t = t'$, with $t, t' \in T(S,V)$. It induces the equality relation $=_E$ on $T(\Sigma,V)$.

Substitutions are mappings $\sigma: V \rightarrow T(\Sigma, V)$, the application of σ to t is denoted by $t\sigma$.

The replacement of a subterm t' in t by t'' is written t[t' <-- t'].
◊

Example 5

Let $\Sigma 1$ be the signature with sorts {D, R} and operators
{0: --> D, 1: --> D, cons: D R -->R, nil: -->R}.

Clearly, $T(\Sigma 1)$ is isomorphic to L(G1), and $T(\Sigma 1, V)$, when restricted to a single variable for each sort, is isomorphic to the sentential forms of G1.
◊

In the sequel, we will mainly be concerned with syntactic aspects - recognizing terms for which $=_E$ is the syntactic identity relation. However, we set the stage for a convenient generalization, allowing a subsignature for which nontrivial equations may hold. The idea is that equality relation of this subsignature can be implemented by some other means, for example by providing a canonical term rewrite system for it. The concept of a hierarchic signature here is the same as in algebraic compiler specifications with attribute coupled grammars [GaGi84]. There, the subsignature is called the "semantic" subsignature, while its complement is called "syntax". In this terminology, terms can be seen as syntactic trees with semantic terms at their leaves.

Definition 5: Hierarchic Signature

A signature Σ is hierarchic, if it contains a subsignature Σ' such that all operators of Σ with result sorts from Σ' are from Σ'.
◊

We shall restrict our attention to specifications where equations may only be given between terms of those sorts which belong to the subsignature Σ'. The particular significance of the subsignature M and its equational theory in the application to code generation was explained following Example 1 in section 2.

3.4 Simple Correspondencies

Let η be the grammar morphism that merges all nonterminals into a single one, modifying productions accordingly.

Observation 4

$L(G) \subseteq L(\eta(G)) \subseteq trees(A)$.
◊

In general, the inclusion is proper. This is illustrated by Example 6.

Example 6

Let grammar G2 =({D, R}, A1, P2} with
P2= { R --> nil
R --> cons(0, R)
R --> cons(1, R)
D --> 0
D --> 1}.

We have L(G2) = L(G1). We have L(G2) ⊂ L(η(G2)) ⊂ trees(A1), proved by considering cons(0, 0) and cons(cons(0, 0), nil). Although L(G2) = L(G1), the second inclusion is not proper in the case of G1.
◊

Observation 5

For any A, trees(A) = L(G) for a regular tree grammar G in the following restricted form:
(RF1) G =({X}, A, P}, with

P = {X--> a(X, ..., X) | for a∈A and according to rank(a)}.

◊

Now we denote by η the signature morphism that identifies all sorts, yielding one-sorted signatures.

Observation 6

T(Σ) ⊆ T(η(Σ)) = trees(OP)
◊

To see that the above inclusion is proper, consider cons(0, 0) ∈ T(η(Σ1)) \ T(Σ1). More precisely, the inclusion above would have to be expressed as T(Σ) being isomorphic to a subset of T(η(Σ)), and the equality as isomorphism. Besides this, the equality in Observation 6 is justified below.

Observation 7

Any T(Σ) for Σ=(S, OP) is isomorphic to L(G) of a regular tree grammar G in the restricted special form
(RF2) G = (S, OP, P) with

P = {X_0 --> a(X_1, ..., X_n), for any a: X_1 ... X_n --> X_0 ∈ OP}.

◊

In G, we simply use sort symbols as nonterminals. If G has only one nonterminal, RF2 and RF1 coincide. Hence, the grammar corresponding to η(Σ) is in RF1, which proves T(η(Σ)) = trees(OP) of Observation 6.

According to the isomorphisms stated here, heterogeneous term algebras are rightfully called heterogeneous tree languages, and we will consider terms as trees and trees as terms, as convenient. For example, we will speak of a Σ-derivation of t as a way to sucessively construct some t∈T(Σ) from Σ-operators.

Summing up, we shortly address the question of what concept to use when designing an intermediate language. Expressive power is an important issue here. It is undesirable to specify an intermediate language which is a superset of what will actually occur, and then base subsequent compiler phases on assumptions on "that subset of the intermediate language actually produced by the front-end". (An interesting lesson is to be learned here from Henry's experience with the portable C-Compiler [Henr84].) Summing up our observations, homogeneous tree languages are less powerful than heterogeneous tree languages, i.e. term algebras, which are in turn less powerful than regular tree languages. However, in section 4 we shall add another concept from algebraic data type specifications, which makes many-sorted term algebras more expressive than regular tree grammars.

3.5 Tree Parsing

Let G = (N, A, P) be a regular tree grammar. We now define what a G-parse for some t∈trees(A) is. Permitting t∈trees(A) rather than t∈L(G) means of course that a G-parse does not necessarily exist. Since in general, G is ambiguous, we define the notion of a parse such that it comprises all

possible derivations of the given tree.

Definition 6

Let t∈trees(A), G = (N, A, P).

The G-semi-parse of t is the function ϕ' associating with each subtree t' of t the set of all productions p of form X -->t'' such that X -->$_p$ t'' -->* t'.

The G-parse of t is the function ϕ associating with each subtree t' of t the set of all productions p of form X-->t'' such that for any p: X-->t'' ∈ ϕ(t'), there is some derivation X' -->* t[t'<--X] -->$_p$ t[t'<--t''] -->* t.

◊

Usually, the grammar G of concern will be clear from the context, and we speak just of semi-parses and parses. Clearly, a semi-parse may associate productions with some subtree that cannot be part of a derivation of the overall tree.

Observation 8

Let ϕ be the parse, and ϕ' the semi-parse of t.

i) ϕ and ϕ' coincide at the root: $\phi(t) = \phi'(t)$.

ii) t∈L(G) iff $\phi(t) \neq \emptyset$.

iii) If $\phi(t') = \emptyset$ for some proper subtree t', this does not imply t ∉ L(G).

◊

In case iii), the root of subtree t' may still be derived as an inner node of the righthand side of some production.

Given ϕ(t), we can enumerate all derivations for t. If G is noncircular, or if we restrict our interest to noncircular derivations, their number is finite, and a simple backtracking traversal of t is sufficient to

enumerate them. But generally, although ϕ can clearly be represented in space linear in the size of t, there may be exponentially many noncircular derivations, and a complete enumeration or explicit representation is not what one is interested in. We shall return to the problem of what to do with a parse in chapter 8.

4. Derivors and Their Inversion

Derivors [ADJ78] are an implementation concept in algebraic data type specifications: By a derivor δ:Q-->Z , the operators of some signature Q are implemented by composite operators in some other signature Z, which are specified as terms of T(Z, V). Sometimes a derivor in the opposite direction, δ:Z-->Q, is used to (partially) specify a morphism γ: T(Q) --> T(Z), which one is interested in. For q∈T(Q), we require γ(q):= some z∈T(Z) such that $\delta(z) =_E q$. The task, then, is to construct in some way an inverse of δ. Code selection is an important and natural example for this phenomenon when understood in the way outlined in section 2).

4.1 Derivors and Linearity

Definition 7

A derivor δ: Z --> Q is specified by

- a sort implementation map δ: S_Z --> S_Q,
- an operator implementation map δ: OP_Z --> T(Q, V), specified for each operator

$a{:}s_1 \ldots s_n \dashrightarrow s_0 \in OP_Z$ by operator implementation equations of the form

$\delta(a(v_1{:}s_1, \ldots, v_n{:}s_n){:}s_0) = t_a(v_1{:}\delta(s_1), \ldots, v_n{:}\delta(s_n)){:}\delta(s_0)$.
(Not all variables of the lefthand side must actually occur in t_a.)

A derivor is linear, if for all $a \in OP_Z$, none of the variables v_i occurs more than once in the term t_a.

When Q and Z are hierarchic signatures (cf. Definition 5) over the same subsignature Σ', we call δ hierarchic if it is the identity on Σ'.
◊

On the righthand side of these equations, the variable v_i of the Q-sort $\delta(s_i)$ denotes the δ-image of the i-th argument (whose sort is s_i) of operator a. For an example, return to Example 1 in section 2.

The terms t_a are called derived operators in Q, and forgetting the original operators of Q turns T(Q) into a Z-algebra. The Z-homomorphism defined by δ is readily implemented by reading the operator implementation equations left to right as rewrite rules over $T(Q \cup Z)$, where the elements of T(Q) are the normal forms. As there is, by definition, one equation for each Z-operator, there are no critical pairs, and the rules trivially form a canonical rewrite system.

However, while δ translates target into intermediate language programs, our interest goes in the opposite direction. Simply orienting the equations right to left to obtain a rewrite system for γ, an inverse of δ, will generally fail for two reasons:

(1) If δ is not injective, the rewrite system so obtained will not be confluent.

(2) If δ is not surjective, some $t \in T(Q)$ will have normal forms that are not in T(Z).

Injectivity of δ cannot be enforced (- at least when a certain freedom of choice in γ is inherent in the problem under consideration -), so standard rewrite techniques cannot be used for implementing γ. Surjectivity (modulo $=_E$) of δ will be required - as δ should specify γ for any input term -, and for practical matters we need a way to verify this requirement.

Definition 8

Let there be given a derivor δ: Z --> Q, specifying (as outlined in section 1.3) some inverse mapping γ: T(Q) --> T(Z). We call this δ complete, if δ is surjective modulo $=_E$, i.e.

for all $q \in T(Q)$, there is some $z \in T(Z)$ such that $q =_E \delta(z)$.
◊

4.2 Derivor Inversion by Tree Parsing

Observation 9

If δ is linear, $\delta(T(Z)) \subseteq T(Q)$ can be described by a regular tree grammar Δ_δ of the following form:

$\Delta_\delta = (S_Z, OP_Q, \{s_0 \dashrightarrow t_a(s_1, \ldots, s_n)$ for each operator implementation equation

$\delta(a(v_1{:}s_1, \ldots, v_n{:}s_n){:}s_0) = t_a(v_1{:}\delta(s_1), \ldots, v_n{:}\delta(s_n)){:}\delta(s_0)$ of $\delta\}$).

◊

Where δ is clear from the context, we just write Δ for Δ_δ. Note that Δ constructs terminal trees over OP_Q, using as nonterminals the sorts of S_Z. Single variables on righthand sides of operator implementation equations give rise to chain productions in Δ, while proper terms yield structural productions. Equations with the same righthand sides and the same result sort s_0 give rise to the same production in Δ, in spite of the fact that they implement different Z-operators.

As one easily verifies, for a linear δ, L(s) as defined by Δ is the set of δ-images of T(Z):s. This means that, for a linear δ, the problem of constructing γ, the inverse of δ, is solved by constructing Δ-parses. The reader is invited to construct Δ_δ for δ as given in Example 1, (ignoring its nonlinearity for the moment), to construct the Δ_δ-parse of q, and to obtain from it z and z´ as shown in Example 2.

Theorem 1

Let E = Ø. If γ is specified to be an inverse of a linear derivor δ, then

i) completeness of the specification is decidable,

ii) any choice for γ(q) can be obtained from a Δ-parse of q∈T(Q).

Proof:

i) follows from Observations 2, 7 and 9: Completeness now means verifying that, with Q seen as a grammar according to Observation 7, L(Q) ⊆ L(Δ).

ii) We construct a Z-derivation of z´ "in parallel" to a Δ-derivation of δ(z´) = q: For each application of $s_0 \dashrightarrow t_a(s_1, ..., s_n)$ in a Δ-derivation of q , apply $s_0 \dashrightarrow a(s_1, ..., s_n)$ for some a∈OP_Z with $\delta(a(v_1{:}s_1, ..., v_n{:}s_n){:}s_0) = t_a(v_1{:}\delta(s_1), ..., v_n{:}\delta(s_n)){:}\delta(s_0)$ being an operator implementation equation, thus obtaining a Z-derivation of z´∈T(Z, S_Z). Obtain z from z´ by substituting different variables for all occurences of sort-symbols in z´. For any substitution σ, δ(zσ) = q.

◊

Note that several terms z can be constructed from the same Δ-derivation of t according to the above proof. One reason is that there is a choice of a, if several operators with the same result sort s_0 have implementation equations with the same righthand side t_a. The other reason is that if there are equations where some variable of the lefthand side does not occur on the righthand side, the corresponding Z-subterm is "forgotten" by δ, and hence may be chosen arbitrarily by σ.

4.3 Expressive Power Revisited

We now remark that many-sorted signatures together with the concept of linear derivors have the same expressive power as regular tree languages. Observation 9 showed that they do not have more expressive power, as every linear derivor image can be generated by a regular tree grammar. It is easy to see that the reverse also holds:

Observation 10

Let Q be a many-sorted signature. For any regular tree grammar G with L(G)⊆T(Q), there exist a signature Z and a linear derivor δ such that L(G)=δ(T(Z)).

Proof:

1. For any G with L(G)⊆T(Q), there exists a G´ such that for all nonterminals X of G´, there is a unique Q-sort s such that for all t∈L(X), sort(t)=s. We construct this G´ as follows: Let $X \to_p t:s$, $X \to_{p'} t':s'$ with $s \neq s'$, and both p and p´ be structural productions. X cannot occur on the righthand side of any structural production, as this would violate L(G)⊆T(Q) by generating a non-wellsorted term. We replace X by a new nonterminal symbol X´ in p´, and add for each chain rule Y->X a further chain rule Y->X´. The same argument applies to Y with Y -->X, Y -->X´, and $X \to_p t:s$, $X' \to_{p'} t':s'$ with $s \neq s'$. Successive application of this step yields the desired G´, with L(G´)=L(G).

2. Now w.l.o.g. let G be such that for all nonterminals X of G, there is a unique Q-sort s such that for all t∈L(X), sort(t)=s. Let us denote this s as sort(X). We define Z and δ as follows:
$S_Z = N_G$. OP_Z has an operator $p:X_1...X_n \to X_0$ for each production p: $X_0 \to t_p(X_1...X_n) \in P_G$.
The derivor δ is given by the sort map

$$\delta(X) = sort(X),$$

and the operator implementation equations

$$\delta(p(v_1, ..., v_n)) = t_p(v_1, ..., v_n).$$

◊

Note that the derivor constructed in Observation 10 is linear. Hence, regular sub-languages of a term algebra can be seen as homomorphic images of some other term algebra, specified by a linear derivor. Images of nonlinear derivors can define sub-languages that cannot be specified by regular tree grammars.

5. Two Notions of Matching

While "matching" (between terms) has a precise and uniform meaning in the field of term rewrite systems (cf. Definition 4), the phrase "pattern matching" is sometimes used more vaguely - for matching a single pattern against a tree, for determining all matches of a pattern set in a given tree, or sometimes for purposes that would more rightfully be called parsing of trees. This section clarifies the correspondencies by describing parsing as a slightly generalized form of matching between terms.

Definition 9

Let R be some subset of T(Σ, V). A substitution σ is a substitution from R, or an R-substitution for short, if $v\sigma \neq v$ implies $\sigma(v) \in R$.
◊

In the application we have in mind, R will be described by a grammar, but for Definitions 9 and 10, this is irrelevant.

Definition 10

Let (Σ, E) be a specification, let p, t∈T(Σ,V), R⊆T(Σ,V).

- p matches t, if there exists a substitution σ such that $p\sigma =_E t$.
- p R-matches t, if there exists an R-substitution σ such that $p\sigma =_E t$.

◊

Note that if p R-matches t, this does not imply t∈R!

Theorem 2

Given Σ, let G be such that $L(G) \subseteq T(\Sigma)$, and $t:s \in T(\Sigma):s$, $v:s \in V$. Then we have:

$t \in L(G)$ iff v L(G)-matches t.

◊

This is in fact the trivial case of Definition 10, with p being a single variable. The point is that we can formulate it in a more constructive way by separating out the "topmost production" of t:

Corollary

$t \in L(G)$ iff, for some production p of form $X \rightarrow t'$, with $t' \in T(\Sigma, V)$, t' L(G)-matches t.

◊

Matching a term p against t (in the sense of Definition 10) is a local task - only the non-variable part of p must be compared against the outermost portion of t. The required substitution is determined automatically by associating subterms of t to variables of p in corresponding positions. (If p is nonlinear, subtree comparison must also be performed, but we ignore nonlinearity for the moment.) The expensive part of this, if any, is testing the equality relation $=_E$ underlying the comparison. In tree pattern matching, as in [Kron75], [HoOD82], no equality other than syntactic equality is considered, and matching a single pattern against a tree is still a local task. But in typical applications, one is interested in all matches of a set of patterns at all subtrees of t, and this is the reason why the matching algorithms studied there perform a complete top-down or bottom-up traversal of t.

With matching relative to L(G), this is different: In order to determine a single match of p, say at the root of t, we must not only compare p and the appropriate portion of t, but also ensure that the substitution so obtained is an L(G)-substitution. This in turn requires L(G)-matching the variables of p to the corresponding subtrees of t, and so forth. So in any case, a complete analysis of t is required.

6. Algorithms for Derivor Inversion

We formulate algorithms for derivor inversion, retaining the classical pattern matching algorithm as a special case.

6.1 Problem Statement and Outline of Solution

Given:

- Two hierarchic signatures $Q=(S_Q, OP_Q)$ and $Z=(S_Z, OP_Z)$ over a common subsignature $M=(S_M, OP_M)$.
- A set E of equations of form $m=m'$, with $m, m' \in T(M, V)$, defining the relation $=_E$ (equality modulo E) on T(M, V), T(Q, V) and T(Z, V), such that unification, equality and matching in T(M,V) (and hence T(Q, V) and T(Z, V)) modulo E are decidable (and presumably efficiently implementable).
- $\delta:Z \rightarrow Q$, a hierarchic derivor.

Desired:

- For $q \in T(Q)$, some representation of the set $\delta^{-1}(q) := \{z \in T(Z) | \delta(z) =_E q\}$.
- A criterion for "specification completeness", i.e. for $T(Q) \subseteq_E \delta(T(Z))$.

Special cases:

For space reasons, we can consider here in detail only the case where M is empty and δ is linear. For $|S_Z| = |S_Q| = 1$ this is the case of classical pattern matching according to [HoOD82] or [Kron75]. In this section we will generalize the notion of matching sets appropriately to handle the heterogeneous case, and shortly comment on the other generalizations.

Outline of the solution:
Let $\Delta=(N^{\Delta}, OP^{Q}, P^{\Delta})$ be the regular tree grammar for δ, constructed according to Observation 9. $L(\Delta)$ describes $\delta(T(Z))$, if δ is linear, and else a superset of $\delta(T(Z))$. For $X\in N^{\Delta}$, $sort(X)\in S_Q$ is uniquely defined as in Observation 10, which follows from the way in which Δ is constructed.

Definition 11: Pattern, Pattern Forest, Matching Sets

- Let $P \subseteq T(Z, V)$ be the set of terms occuring on the righthand sides of the operator implementation equations of δ. P is called the set of "patterns".
- Let $PF = \{p' \mid p' \text{ is a subterm of some } p\in P\}$, the "pattern forest". (Note that $P\subseteq PF$.)
- For $q\in T(Q)$, let the "matching set" of q be
  $MS(q) = \{p\in PF \mid p\ \delta(T(Z))\text{-matches } q\}$.
- Let $MSs = \{M \mid M = M(q) \text{ for some } q \in T(Q)\}$.

◊

These notions are analogous to those used in pattern matching in homogeneous tree languages, but significantly more general. Basing the definitions on $\delta(T(Z))$-relative matching of terms (with variables) accomodates heterogenity, nonlinearity, and a nontrivial equational theory of M.

Definition 12: Algorithm Building Blocks

$build_a$: For $m_1, ..., m_n \subseteq PF$, $a \in OP_Q\backslash OP_M$,
$build_a(m_1, ..., m_n) = \{p\in PF \mid p = a(p_1, ..., p_n), p_i\in m_i\}$.

prod: For $p\in P$ with $p = t(v_1{:}\delta(s_n), ..., v_n{:}\delta(s_n))$, $prod(p) = t(s_1,..., s_n)$.
(Remember: $N^{\Delta} = S_Z$.)

$nonlin_q$: For any $q \in T(Q)$, $m \subseteq PF$,
$nonlin_q(m) = \{p\in m \mid p= t(v_1{:}s_1, ..., v_n{:}s_n)$ such that
i) $prod(p) \rightarrow^* q'$ for some $q\in T(Q)$ with $q' =_E q$.
Let q_i' be the subtree derived from s_i in the derivation of q'.
ii) ($v_i = v_j$ implies $q_i' =_E q_j'$ for any $i, j \in 1..n$)$\}$.

chain: Let $N' \subseteq N^{\Delta}$, and $closure(N') := \{X \in N^{\Delta} \mid X \rightarrow^* Y , Y \in N'\}$.
For $R \subseteq prod(PF)$, $chain(R) = closure(\{X \in N^{\Delta} \mid (X \rightarrow t) \in P^{\Delta}, t \in R\})$.

vars: For $X\in N^{\Delta}$, $vars(X) = \{v \in PF\cap V \mid sort(v)=sort(X)\}$.

prod and vars are extended to sets element-wise .
◊

$build_a$ just constructs the set of patterns rooted by a, given choices of subpatterns for the $n\geq 0$ arguments of a. chain determines nonterminals from which certain righthand sides can be derived by (possibly empty) chains, ending in a structural production whose righthand side is in R. prod and vars are used to translate between the algebraic-view and the grammar-view of δ. For $p \in P$,

prod(p) is the righthand side of the corresponding Δ-production(s). Ignoring $=_E$ for the moment, if δ is linear, the notions of derivation and L(Δ)-relative matching coincide. In this sense, prod(p) can also be thought of as a rewritten pattern p with variables renamed to make it linear. δ(T(Z))-relative matching of p against some q can thus be separated into two "parts": derivation of q from prod(p) with $q_i \in \delta(T(Z))$, and verifying the nonlinearity condition. $nonlin_q$ checks nonlinearity of patterns p which - if they were linear - would match at the root of q. (In an implementation, one uses the functionality nonlin(p, q) instead of $nonlin_q(p)$.)

Algorithms calculating matching sets will be composed from these functions. MS(q) and the matching sets for all subterms of q will be the desired representation of $\delta^{-1}(q)$. The following theorem shows that matching sets are an extension of the notion of a semi-parse ϕ'.

Theorem 3:

Let M be empty and δ linear. We have the following correspondence between the matching sets and a semi-parse ϕ' of $q \in T(Q)$ (and for all subterms of q):

$\phi'(q) = \{X \dashrightarrow prod(p) \in P^{\Delta} \mid p \in MS(q) \cap P\}$.

◊

We now have to show - for the general case - that MS(q) in fact represents $\delta^{-1}(q)$ in a precise way. The following two definitions are mutually recursive:

Definition 13a

For $p \in P$, let Z-op(p) = $\{f(v_1, ..., v_n) \mid f \in OP_Z$ and $\delta(f(v_1, ..., v_n)) = p$ is an operation implementation equation of δ$\}$.

For $q \in T(Q)$, define Z-terms(q) $\subseteq T(Z,V)$ as follows:

i) for $q \in T(M)$: Z-terms(q) = $\{t \in T(M) \mid t =_E q\}$.

ii) for $q \in T(Q) \setminus T(M)$:

Z-terms(q) = $\{f\rho \mid f \in$ Z-op(p) for some $p \in MS(q) \cap P$, and ρ a matching substitution for p over q$\}$.

◊

Case i) comes from the fact that δ by definition is the identity on T(M). One may choose to represent Z-terms(q) by q in this case. Case ii) constructs a Z-term rooted by f, taking as its arguments Z-terms from the corresponding variable positions in p, as matched against q. This correspondence is defined formally as the notion of a "matching substitution", which substitutes Z-terms for variables, and hence applies to $f(v_1, ..., v_n)$ rather than p.

Definition 13b

For $p \in MS(q)$, we say that ρ: V->T(Z, V) is a matching substitution for p over q, iff one of the following applies:

i) p = v (a single variable) and $v\rho \in$ Z-terms(q);

ii) $p \in T(M,V)$ and $p\rho \in$ Z-Terms(q);

iii) v does not occur in p and vρ is a new unique variable;

iv) $p = a(p_1, ..., p_k)$, $a \in OP_Q \setminus OP_M$, $p_i \in T(Q,V)$, (and hence $q = a(q_1, ..., q_k)$), and ρ is a matching substitution for p_i over q_i for $1 \le i \le k$.

◊

If p is nontrivial, case iv) applies and takes us down to the positions in q that correspond to the variables in p. It ensures that variables occuring in several leaf positions are consistently substituted. Cases ii) and iii) should be clear, but case i) contains a subtlety that needs further explanation.

On the one hand, case i) is just the terminating case in the parallel structural induction over p and q as specified by case iv). However, considering the situation where the pattern p considered in case ii) of Definition 13a is a single variable, Definitions 13a and 13b seem circular. When $\delta(f(v))=v$, and $v \in MS(q)$, they say that $f(t) \in$ Z-Terms(q) if $t \in$ Z-Terms(q). When t and f(t) happen to be of the same sort, this even implies $f^i(t) \in$ Z-Terms(q) for $i \geq 0$. This is exactly what we need here: it models the situation where δ plainly forgets Z-structure, which is recovered by finding (potentially circular) derivations using the chain productions in Δ that stem from operation implementation equations of the form $\delta(f(v_1, ..., v_n) = v_i$.

Note that Z-terms(q) may contain terms of different sorts. They contain variables for subterms "forgotten" by δ. Finally, we observe:

Theorem 4

$\delta^{-1}(q) = \{t\sigma \mid t \in$ Z-terms(q) and σ a ground substitution$\}$.
◊

Now we are left with the task of efficiently calculating MS(q) for a given $q \in T(Q)$.

6.2 The Basic Dynamic Algorithm

With the functions of Definition 12, it is straightforward to describe our basic algorithm for determining matching sets for a given input term q. Subsequently, we shall derive table driven versions from this Algorithm 1.

Algorithm 1:

Input: $r \in T(Q)$, PF.

Output: For each subterm q of r (including r itself): MS(q).

(1) for all subterms q of r with sort(q) $\in S_M$ *)
do MS(q)= {p ∈ PF| p matches m (modulo E)} od;
(2) i:= 1;
(3) while i ≤ height(r)
(4) do for all subterms $q = a(q_1, ..., q_n)$, $n \geq 0$, of r with height(q) = i and sort(q) $\notin S_M$
(4a) do MS(q) := let $\underline{m} = (MS(q_1), ..., MS(q_n))$ in
$(nonlin_q \circ build_a)(\underline{m}) \cup (vars \circ chain \circ prod \circ nonlin_q \circ build_a)(\underline{m})$
od;
(4b) i:= i+1;
od
◊

*) Actually it suffices to compute the matching sets of "semantic" subterms m (i.e. sort(m)$\in S_M$),which are not subterms of a semantic subterm themselves, because on the one hand $\delta^{-1}(m) = m$, and on the other hand we need the matching sets of such "complete" subterms m, in order to be able to compute the matching sets of the "syntactic" superterms q´ (i.e. sort(q´) $\notin S_M$) containing them.

6.3 Table-Driven Algorithm for M empty and δ linear

For this section, we let M be empty and δ linear. In step 4a) of Algorithm 1, all functions except $nonlin_q$ are independent of the actual input term q. When δ is linear, $nonlin_q$ is the identity on pattern sets, for any q. Hence, the matching set of a term $q = a(q_1, ...,q_n)$ is computed (only) from the operator "a" and the matching sets of the subterms $q_1, ..., q_n$.

As noted earlier, the set of all matching sets, MSs, is finite. These facts gave rise to the idea (in the treatment of the homogeneous case in [Kron75] and [HoOD82]) to precompute the information which is dynamically computed in step 4 of Algorithm 1. This information is represented by tabulating the following functions f_a, for each $a \in OP_Q$:

$$f_a(\underline{m}) = build_a(\underline{m}) \cup (vars \circ chain \circ prod \circ build_a)(\underline{m}).$$

The functionality of f_a could be seen as f_a: $(MSs)^{rank(a)}$ ---> MSs, but this would be excessively expensive in table size and generation time. Instead, f_a should only be tabulated for those combinations of arguments that can actually occur. A restriction of the possible combinations is observed by exploiting the heterogenity of the input language: All patterns matching a term q must have the same sort as q. We may partition MSs according to S_Q:

$$MSs = \cup \text{ N-MSs, for } N \in S_Q, \text{ with } MS(q) \in \text{N-MSs iff } sort(q)=N.$$

It suffices to precompute the following (generally smaller) tables:

For $(a: N_1...N_n \text{ --> } N_0) \in OP_Q$: f_a: N_1-MSs × ... × N_n-MSs ---> N_0-MSs.

(As observed in [Kron75] for the one-sorted case, the N-MSs as carriers and f_a as functions form a Q-algebra. Another such algebra could be defined over the carriers $C_N = 2^{\{p \in PF \mid sort(p)=N\}}$, with f_a extended accordingly. This algebra is the worst case of our approach with respect to size of the precomputed tables. Work on the homogeneous case has shown that the tables for f_a as defined above are significantly smaller in many practical applications.)

Provided that we have precomputed f_a for all $a \in OP_Q$, we get the following table driven version of Algorithm 1:

Algorithm 2:

Input: $r \in T(Q)$; for all $a \in OP_Q$: f_a
Output: as Algorithm 1

```
(1)  -- step 1 is omitted as we assume M to be empty --
(2)  i:= 1;
(3)       while i ≤ height(r)
(4)       do    for all subterms q = a(q1,...,qn), n≤0, of r with height(q) = i
(4a)            do   MS(q):= f_a (MS(q1),...,MS(qn)) od;
(4b)            i:= i+1
          od
◊
```

Obviously, this matching algorithm is linear in the size of r, as it consists of a single table-lookup per node of the input term r.

The principle idea in the following table generating algorithm is to compute successively the matching sets of all terms from T(Q) of height 0,1,2, etc. until MSs, the set of all matching sets, converges. Terms are not enumerated explicitely. Rather, a term of height i is represented by its top operator and the possible matching sets for its arguments. The first iteration (for nullary a) is taken out of the repeat loop, as its repeated calculation cannot yield further matching sets.

Algorithm 3:

Input: PF, OP_Q, S_Q.

Output: for all $a \in OP_Q$: f_a.

for all $N \in S_Q$ do $N\text{-}MSs^0 := \emptyset$ od;

for all $(a : \text{--> } N) \in OP_Q$

do tabulate f_a;

$N\text{-}MSs^0 := N\text{-}MSs^0 \cup f_a$

od;

i:=1;

repeat for all $(a : N_1 ... N_n \text{ --> } N) \in OP_Q$, $n \geq 1$,

do for all $N \in S_Q$ do $N\text{-}MSs^i := \emptyset$ od;

for all $(R_1,...,R_n) \in N_1\text{-}MSs^{i-1} \times ... \times N_n\text{-}MSs^{i-1}$ *)

do tabulate $f_a(R_1,...,R_n)$;

$N\text{-}MSs^i := N\text{-}MSs^i \cup f_a(R_1,...,R_n)$

od

od;

for all $N \in S_Q$ do $N\text{-}MSs^i := N\text{-}MSs^i \cup N\text{-}MSs^{i-1}$ od;

i:= i+1

until for all $N \in S_Q$, $N\text{-}MSs^{i-1} = N\text{-}MSs^{i-2}$

◊

*) Here we can demand at least one R_j to be computed in the last iteration step of the repeat-loop, in order to ensure that there will be computed matching sets of terms of height i indeed.

Observation 12:

The specification is complete, i.e. $T(Q) = L(\Delta)$ $(= \delta(T(Z)))$, iff

for all table-entries $f_a(R_1,...,R_n) = R$ holds: $R \cap P \neq \emptyset$.

◊

6.4 Extensions

Space does not allow to explicate the extension of the generator algorithm to the cases where M is non-empty, and the derivor may be non-linear. This will be done in an extended version of this paper, planned to appear elsewhere. We only sketch here the particular problems to be solved for each of these, and one further extension.

Extension to non-empty M

A particular initial step is needed to calculate the s-MSs for $s \in S_M$. Generally, not all subsets of $2^{PF:s}$ can occur, since some patterns in PF:s may not be independent, as for example the patterns car(cons(1, v)) and 1 in the presence of the usual axioms. Here we need the prerequisite of our problem statement that unification modulo E be decidable.

Extension to nonlinear δ

Consider Algorithm 1. In the linear case, $nonlin_q$ is the identity, and the composite effect of step 4a) can be precomputed. $nonlin_q$, however, can only be evaluated dynamically, and so we have to generate separate tables representing $build_a$ and $f(\underline{m}) := \underline{m} \cup (vars \circ chain \circ prod)(\underline{m})$. Now the central step (4a) in Algorithm 2 becomes

$MS(q) :=$ let $m' = nonlin_q \circ build_a(MS(q_1), ..., MS(q_n))$ in $m' \cup f(m')$,

where $build_a$ and f are tabulated. $nonlin_q$ introduces twofold complications: In order to know for which arguments $build_a$ and f must be precomputed, we must anticipate the effect of $nonlin_q$. Furthermore, for applying $nonlin_q$ dynamically, information about matching substitutions must be calculated along with the matching sets.

Extension to More Refined Input Languages

Let IL be the input language to our pattern matcher. So far, we have assumed that it is a term algebra, IL=T(Q) for some Q. But an actual IL may be some subset of T(Q), being the output of earlier compiler phases. IL may be $L(G) \subseteq T(Q)$ for some regular tree grammar G, or it may be $\rho(T(P))$ for some other signature P and a (possibly nonlinear) derivor ρ: P --> Q. In these cases, the table driven algorithm with tables generated as if IL = T(Q) will still work correctly, but the tables may contain matching sets that cannot actually occur for the more restricted IL. Besides tables being larger than necessary, our completeness criterion is only a sufficient, but no longer a necessary condition. But in both cases, our generative algorithms can be adapted to generate the precise matching sets for the given IL. There is no need to study further generalization to input languages such as

$IL = (\rho_2 \circ \rho_1)(T(P))$, as derivors are closed under composition.

7. Table Size and Generation Effort

Measuring space and time efficiency in terms of the size of the given pattern set P, it is known from work on the homogeneous case [HoOD82], that there is an exponential worst case behaviour. Fortunately, it has also been experienced that this behaviour does not occur for many practical situations, in particular when the compacting technique of [Chas87] is used. As our algorithms include the homogeneous case when $|S_Q| (= |S_Z|) = 1$, these worst case observations are still valid. On the other hand, one can construct a specification with $|S_Q| = 1$ and $|S_Z| > 1$ that yields a linear number of matching sets, but turns into the worst-case example of [Chas87] when applying the sort-identifying morphism of section 3.4 to the target signature Z.

The generator algorithm (Alg. 3) is a 360-line PROLOG program. It uses the compacting generation technique of [Chas87], and some care was given to the way in which PROLOG´s backtracking is used. The largest example it has been run with is a fairly complete description for the MC68000 processor, containing 37 sorts and 92 operators in the target signature. The (compiled) generator produces 76 matching sets in about 32 seconds on a SUN-3 (25MHz) workstation, and the generated tables (in the form of PROLOG facts) occupy 53K bytes of storage. (Without Chase´s compacting technique, generation time is about 2 hours!) With the present data, it seems that space and time requirements of the code selector and its generator will no longer be a problem. But experiments comparable to those of [Henr84] have not yet been performed.

8. Conclusion and Future Work

We expect that more general machine specific aspects or code generation subtasks such as register allocation, that were previously treated in an ad-hoc manner, can be expressed by extending the target signature Z by equational specifications. Such code generator specifications may look rather different from the ones in most of the approaches discussed here (with the exception of [MRSD86]), but the underlying implementation technique will still be pattern matching as developed here. The long-term goal of this work is to make code generator specifications more formal and complete, such that proof methods from the area of term rewrite systems [HuOp80], [HuHu80], [RKKL85] can be used to verify the correctness of code generators.

The recent approaches we have discussed shortly in the introduction should be evaluated in terms of the formalism presented here. Besides by peculiarities in their pattern matching mechanisms, they are characterized by the way in which the evaluation of the "choice function" ξ is interleaved with the construction of δ^{-1}. Both ideas which have been used - dynamic programming at matching time,

and reduction of generated tables at generation time according to cost criteria - can be incorporated with our approach. Most interesting may be an hybrid scheme, using table reduction where it retains completeness and optimality, and matching-time cost comparison otherwise.

Finally, a conceptually interesting and technically demanding problem spared out in this paper is the following: Having implicitly represented $\delta^{-1}(q)$ in a compact way (cf. Theorem 4), how do we extract from it an interesting subset according to cost minimality or other well-formedness criteria that express machine properties not covered by δ itself? Some progress has been achieved [Weis87] by work subsequent to [WeWi86], but an eventual solution to this problem also depends on what further subtasks of code generation are to be integrated into the overall approach.

References

[ACK83] *A. Tanenbaum, H. van Staveren, E. Keizer, J. Stevenson:* A Practical tool Kit for making Portable Compilers. CACM 26 (9), pp. 654-660, 1983]

[ADJ78] *J.A. Goguen, J.W. Thatcher, E.G. Wagner:* An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh (ed.): Current trends in programming methodology, Vol. IV, Prentice Hall, 1978.

[AhGa85] *A.V. Aho, M. Ganapathi:* Efficient tree pattern matching: an aid to code generation. Proceedings POPL 12, pp.334-340, 1985.

[AGT86] *A.V. Aho, M. Ganapathi, S.W.K. Tjiang:* Code Generation Using Tree Matching and Dynamic Programming. Report , Bell Laboratories, Murray Hill, 1986.

[AhJo76] *A.V. Aho, S.C. Johnson:* Optimal Code Generation for Expression Trees. JACM 23(3), pp. 488-501, 1976.

[Benk85] *M. Benk:* Tree grammars as a pattern matching mechanism for code generation. Report TUM-I8524, Technical University München, 1985.

[Brai69] *W.S.Brainerd:* Tree generating regular systems. Information and Control 14, pp. 217-231, 1969.

[Catt77] *R.G.G. Cattell:* Formalization and Automatic Derivation of Code Generators. Dissertation, Report CMU-CS-78-117, Carnegie-Mellon-University, Pittsburgh 1978.

[Chas87] *D.R. Chase:* An improvement to bottom-up tree pattern matching. Proceedings POPL 14, 1987.

[CHK84] *Th.W. Christopher, Ph.J. Hatcher, R.C. Kukuk:* Using dynamic programming to generate optimized code in a Graham-Glanville style code generator. Proceedings SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices 19, 6, 1984.

[GaGi84] *H. Ganzinger, R. Giegerich:* Attribute coupled grammars. Proceedings 2nd SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 19 (6), pp.70-80, 1984.

[Gieg84] *R. Giegerich:* Code generation phase models based on abstract machine descriptions. Report TUM-I8412, Technical University München, 1984.

[Gieg85] *R. Giegerich:* Logic specification of code generation techniques. In: H. Ganzinger, N.D. Jones (Eds.): Programs as data objects. LNCS 217, Springer Verlag, 1985.

[Glan77] *R.S. Glanville:* A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers. Dissertation, Report UCB-CS-78-01, University of California, Berkeley 1977.

[GrGl77] *R.S. Glanville, S.L. Graham:* A new method for compiler code generation. Proceedings 5th ACM Symposium on Principles of Programming Languages, pp. 231-240, 1977.

[HaCh86] *Ph. J. Hatcher, Th. W. Christopher:* High quality code generation via bottom-up tree pattern matching. Proceedings SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices 21, 6, 1986.

[Henr84] *R.R. Henry:* Graham-Glanville code generators. Dissertation, Report UCB-CSD-84-184, Berkeley 1984.

[HoOD82] *Ch. Hoffman, M. O'Donnell:* Pattern matching in trees. JACM , pp.68-95, 1982.

[Hors87] *N. Horspool:* An alternative to the Graham-Glanville code-generation method. IEEE Software, pp. 33-39, May 1987.

[HuHu80] *G. Huet, J.-M. Hullot:* Proofs by induction in equational theories with constructors. Proceedings 21st SFCS, Lake Placid, pp 96-107, 1980.

[HuOp80] *G. Huet, D.C. Oppen:* Equations and rewrite rules: A survey. In R. Book (ed.): Formal language theory: Perspectives and open problems. Academic Press, 1980.

[Kron75] *H. Kron:* Tree templates and subtree transformational grammars. Dissertation, UC Santa Cruz, 1975.

[MRSD86] *M. Mazaud, R. Rakatozafy, A. Szumachowski-Despland:* Code Generation Based on Template-Driven Target Term Rewriting; Rapport de Recherche, INRIA, 1986.

[Ripk77] *K. Ripken:* Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinencodeerzeugung aus attributuerten Programmgraphen. Dissertation, TUM-INFO-7731, Institut für Informatik, TU München, 1977.

[RKKL85] *P. Rety, C. Kirchner, H. Kirchner, P. Lescanne:* NARROWER: a new algorithm for unification and its application to logic programming. Proc. 1st Conference on Rewriting Techniques and Applications, LNCS 202, Springer Verlag, pp. 141-157, 1985.

[Turn86] *P.K. Turner:* Up-down parsing with prefix grammars. SIGPLAN Notices 21, (12), 1986.

[Weis87] *B. Weisgerber:* Private communication.

[WeWi86] *B. Weisgerber, R. Wilhelm:* Two tree pattern matchers for code generation. Internal Report, University Saarbrücken, 1986.