DISSERTATION

Specialization of Prolog Programs with Partially Static Goals and Binarization

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftlichen Fakultät

von

Ulrich Walther Neumerkel
A-9020 Klagenfurt, Krassniggstraße 42
Matr.-Nr. 8425843
geboren am 25. Juli 1965 in Wien

Wien, im September 1992

Originalversion: 30.9.1992. Last date of revision: May 18, 1993

Kurzfassung

In dieser Dissertation werden zwei neue Methoden zur Spezialisierung von normalen und binären Prologprogrammen vorgestellt und mit den bisherigen Techniken verglichen.

Die eine Methode verwendet partiell statische Ziele, welche die herkömmlichen Bindungsumgebungen verallgemeinern und dadurch mehr Information zur Spezialisierung insbesondere in auf Expandierung und Komprimieren basierenden Transformationssystemen weitergeben können.

Die andere Methode, gleichheitsbasierte Transformation von 'Continuations' ('Equality-based continuation transformation'), dient zur Entfernung der beim Durchreichen von Kontextargumenten und Akkumulatoren entstehenden Existenzvariablen und anderer überflüssiger Strukturen, für die aus der Literatur keine Verfahren bekannt sind. Aus Definiten Klausen-Grammatiken (DCGs) entstandene Programme können so von sämtlichen zum Durchreichen der Differenzliste verwendeten Existenzvariablen befreit werden.

Diese Methode bereitet durch die Entfernung von Existenzvariablen Programme nicht nur für weitere Programmtransformationen auf, sondern erhöht auch die Ausführungsgeschwindigkeit und vermindert den dynamischen Speicherbedarf. Sie ist besonders für binäre Prologprogramme geeignet, was durch experimentelle Untersuchungen belegt wird.

Es werden einige Anwendungen unserer Methoden vorgestellt: Der der „Vienna Abstract Machine" eigene Unifikationsalgorithmus kann durch die Verbindung der beiden neuen Methoden hergeleitet werden. Weiters werden einige Erweiterungen üblicher Programmiertechniken vorgestellt, für die gleichheitsbasierte Transformationen von Nutzen sind.

Abstract

This thesis presents two new methods that specialize ordinary and binary Prolog programs and compares them with current approaches.

The first method is based on the new notion of partially static goals. Partially static goals generalize binding environments. More information can be propagated for specialization. The method is well suited for application in fold/unfold transformation systems.

The second method, EBC-transformation (equality-based continuation transformation) is able to remove unnecessary, existential variables and structural redundancies caused by difference lists, context arguments, short-circuits or accumulator passing that cannot be removed by the methods known from the literature. In particular, for programs representing Definite Clause Grammars EBC-transformations are able to remove all unnecessary variables related to the difference list to be parsed or generated.

EBC-transformation does not only make programs amenable to subsequent program transformations by removing unnecessary variables, but also improves execution speed and reduces dynamic memory consumption. The method is of particular interest for binary Prolog implementations. Experimental investigations support this view.

Some applications of our methods are given. We were able to derive the unique unification mechanism of the Vienna Abstract Machine by applying partially static goals and EBC-transformations. Extensions to usual programming techniques are presented that benefit from EBC-transformations.

Contents

Chapter 1

Introduction

The history of Prolog and logic programming has been reported by many researchers involved. There are different viewpoints how the developments of preceding research culminated in Prolog. Kowalski reports about the early development of Prolog and logic programming [Kow88] and his cooperation with Alain Colmerauer. J.A. Robinson [Rob92] presents the development that culminated in logic programming from the broader viewpoint of machine-oriented logic.

Another perspective is given by Cohen [Coh88], who was in the 1960's a colleague of Colmerauer in the research group on compilers at IMAG in Grenoble (Institut d'Informatique et Mathematique Appliquè de Grenoble). He stresses the fact that the development of Prolog was influenced by the concepts developed in compiler construction. The mechanism of Prolog's backtracking is traced back to backtracking parsers in the 1960's. The influence of W-grammars developed for the definition of Algol 68 [vWMP$^+$75] is outlined. De Chastellier and Colmerauer considered the application of W-grammars for various other purposes as syntax-directed translation and translation of English sentences into French [CC69]. In the following years, Colmerauer concentrated in Montreal on the development of a natural language translation system from English to French. The resulting system — système Q — employed a formalism that went beyond W-grammars. To some extent, logical variables were already present although not in their full generality. Grammars were usable both for parsing and generating sentences. System Q used in contrast to Prolog a bottom up computation rule for parsing. It is still commercially available and is used in Canada for the automatic translation of English weather-reports into French.

Prolog. The insight that logic is able to encode parsing in an elegant way which simulates a nondeterministic top down parser when SL-resolution is applied was for Colmerauer the key for the development of Prolog. Later, this encoding became known under various names as difference list and accumulator passing. It is still considered as one of the most important techniques a Prolog programmer must master.

Prolog itself was first implemented in 1972 in Algol W by Phillipe Roussel. This implementation, known as Prolog 0 (see [Can86] for a more detailed description of the development of Prolog systems), already possessed inequality constraints over terms. The subsequent implementation, Prolog I, was written by G. Battani and H. Meloni in 1973. The system was able to execute about 200 unifications per second on an IBM 360-67 (Current systems on stock hardware are about a factor of 10 000 faster yielding more than 2MLIPS). In 1974 David H. Warren visited Marseille to write a plan generation system in Prolog. In the following years, he implemented a compiler [War77] on a DEC10, which founded the family of the Edinburgh-Prologs. Since difference lists are an effective but rather tedious way to implement parsers, Colmerauer developed the formalism of Metamorphosis Grammars ([Col75] published in English: [Col78]) which have a very simple mapping into Prolog. This formalism was popularized by Warren under the name Definite Clause Grammars [PW80]. We will consider them in the sequel in detail.

Grammars in Prolog. The principal decision to be made when implementing grammar rules in Prolog is how a string, or a sequence in general, and the associated relation of concatenation should be represented in Prolog. Beside the approach that represents sequences with relations as in [Kow79], ('Terms versus relations as data structures'), sequences are represented as lists. The simplest approach is to represent every sequence of terminals with a list. When a grammar rule has only terminals in the right hand side, facts are used.

```
q ⟶ "ab".

q([a,b]).
```

Concatenation of two sequences is established by the predicate append/3. A grammar rule

```
p ⟶ q, r, s.
```

is implemented by the following predicate:

```
p(I) ←
    append(I0,I12,I),
    q(I0),
    append(I1,I2,I12),
    r(I1),
    s(I2).
```

Operationally, the list I to be parsed is split into two lists I0 and I12. The list I0 is handed over to the first goal q/1 which in turn attempts to parse I0. The remaining list I12 is split again yielding two lists for the last two goals. It is evident that this representation is not very efficient compared with the traditional way a top down parser is implemented. All possible splittings between I0 and I12 have to be considered, as well as all splittings for some I12. In turn, a similar operation will happen for all nonterminals.

The classical approach to parse a sequence of tokens top down (and without backtracking) is called recursive descent. It consists in mapping every rule for a nonterminal to a procedure. Nonterminals on the right hand side are implemented by procedure calls. Therefore, a procedure implementing a nonterminal p will only access the sequence of strings if a terminal is to be read. In our example the operations connected with p will consist only of calling the three other procedures in sequence. No operations that depend on the length of the string have to be performed in p using recursive descent.

Using the technique of difference lists in this situation, a sequence L is represented by a pair of two sequences Es0 and Es. Es must be a subsequence of Es0. The difference between these two lists, i.e., that sequence whose concatenation with Es results in Es0, denotes the sequence to be represented. To describe the situation in Prolog, a given list L can be transformed into a difference list Es0 and Es as follows:

```
list_difflist(L,Es0,Es) ←
    append(L,Es,Es0).
```

I.e., the conversion is nothing more than usual concatenation. A grammar rule can be represented with difference lists as follows. The concatenation operation append/3 is now immaterial.

$$\mathrm{p}(\overset{\mathrm{I}}{\widehat{\mathrm{X0},\mathrm{X}}}) \leftarrow \mathrm{q}(\overset{\mathrm{I0}}{\widehat{\mathrm{X0},\mathrm{X1}}}), \mathrm{r}(\overset{\mathrm{I1}}{\widehat{\mathrm{X1},\mathrm{X2}}}), \mathrm{s}(\overset{\mathrm{I2}}{\widehat{\mathrm{X2},\mathrm{X}}}).$$

In summary, the following operations where performed: The concatenation of three lists

$$I = I0 + I1 + I2$$

was transformed by introducing the following equations:

$$I = X0 - X, I0 = X0 - X1, I1 = X1 - X2, I2 = X2 - X$$

$$X0 - X = (X0 - X1) + (X1 - X2) + (X2 - X)$$

The implementation method is already very close to the traditional way of implementing a top-down parser. When considering the process of parsing a given list, the list is propagated as follows. First, the variable X0 is bound to the complete list in which a prefix should be parsed by the rule. Later, the suffix of the list that was not parsed by p/2 will be bound to the variable X.

$$\mathrm{p}(\mathrm{X0}, \mathrm{X}) \leftarrow \overbrace{\mathrm{q}(\mathrm{X0}, \mathrm{X1})}^{\mathrm{X0}}, \mathrm{r}(\mathrm{X1}, \mathrm{X2}), \mathrm{s}(\mathrm{X2}, \mathrm{X}).$$

After q/2 has parsed a part of the list X0, the remaining string is passed back in the variable X1. The next goal r/2 is called, passing over the variable X1 with the actual string and X2 which in turn serves as a 'return parameter'.

$$\mathrm{p}(\mathrm{X0}, \mathrm{X}) \leftarrow \mathrm{q}(\mathrm{X0}, \overbrace{\mathrm{X1}}^{\mathrm{X1}}), \mathrm{r}(\mathrm{X1}, \mathrm{X2}), \mathrm{s}(\mathrm{X2}, \mathrm{X}).$$

Finally, s/2 is called. s/2 instantiates X to the remaining string, which is also the remaining string of p/2. In summary, we have to consider the following dataflow:

$$\mathrm{p}(\overbrace{\mathrm{X0}}^{\mathrm{X0}}, \underbrace{\leftarrow \mathrm{q}(\mathrm{X0}, \overbrace{\mathrm{X1}}^{\mathrm{X1}}), \mathrm{r}(\mathrm{X1}, \overbrace{\mathrm{X2}}^{\mathrm{X2}}), \mathrm{s}(\mathrm{X2}, \mathrm{X})}_{\mathrm{X}}).$$

This method is already very close to traditional recursive descent, there are still some deficiencies: The logic program requires much more variables to implement recursive descent, because logical variables can only be bound to a single value. The variable in a procedural language is simulated by the variables X0, X1, X2, and X. Each variable represents the value of the variable in a procedural language at a different time. This is reflected by the fact that the lifetimes of the variables X0, X1, and X2 are disjoint.

$$\mathrm{p}(\overbrace{\mathrm{X0}}^{\mathrm{X0}}, \underbrace{\leftarrow \mathrm{q}(\mathrm{X0}, \overbrace{\mathrm{X1}}^{\mathrm{X1}}), \mathrm{r}(\mathrm{X1}, \overbrace{\mathrm{X2}}^{\mathrm{X2}}), \mathrm{s}(\mathrm{X2}, \mathrm{X})}_{\mathrm{X}}).$$

Recursive descent in a procedural language uses a single global variable to represent the input string and procedures that manipulate the global variable accordingly. An assignment-free method would be the implementation of every nonterminal as a function instead of a procedure. The function's argument is the input string to be read; the function evaluates to the remaining input string. In ML this would read as:

```
fun parse_p(string) = parse_s(parse_r(parse_q(string)))
```

EBC-transformation. In all cases, the internal representation of the parser in a procedural or functional language is still simpler than in Prolog. There are many programming techniques in Prolog beside grammars and difference lists that use variables in a similar way. Techniques that are close to those in other declarative languages, namely monads in pure functional programming languages [Wad92], use variables in a similar way. We will propose a program transformation technique called equality-based continuation transformation that is able to overcome the described deficiencies. Equality-based continuation transformation is able to transform a program into another equivalent program that behaves much closer to a corresponding procedural implementation. In the example above all variables that serve to represent the current state of the input string can be collapsed into a single register.

Overview

In Chapter 2, we will discuss several execution models for Prolog. The development of execution models in the form of meta-interpreters is the basis of considering specialization strategies for Prolog. In Chapter 3, the need for specialization is motivated. Specialization strategies, partial evaluation, and fold/unfold transformations for Prolog are considered. Some drawbacks of current systems are discussed. In Chapter 4, we are presenting an extension to fold/unfold transformations based on the notion of partially static goals. Chapter 5 discusses the current state in the development of binary programs. Chapter 6 introduces equality-based continuation transformations. Chapter 7 presents some applications of our techniques.

Chapter 2

Execution Models for Prolog

The investigation of execution models for a programming language may bear considerable insight into the structure of this programming language. Execution models reflect properties of a language that would otherwise remain uncovered. A concrete implementation will be largely destined by the execution model the implementor had in mind. To formalize such execution models, e.g., to implement them in a representation independent way supports the development of an actual implementation. In particular, the structure of program specializers or partial evaluators is largely determined by the corresponding execution model. Again, it will be an advantage to state the intended execution model as unambiguously as possible.

When considering Prolog, early attempts in defining execution models used standard specification methods as the Vienna-Definition-Method (VDM) [Nil84] or META IV [Kom81,Kom82]. In particular, Komorowski obtained an operational semantics that served for both, a specification of an implementation and a formal description of program optimization techniques based on the principle of partial evaluation. In [Kom81,Kom82], he describes a pure Prolog interpreter using META IV and corresponding transformations. Partial evaluation (partial deduction) is defined as pruning of clauses as well as forward and backward propagation of data.

2.1 Meta-interpreters

A meta-interpreter or meta-circular interpreter [ASS85] for a language L is an interpreter for L written in L. If we will refer to meta-circular interpreters we underline that they are able to interpret themselves. Meta-interpreters will be seen, as is current practice in the context of Prolog, from a broader viewpoint. A meta-interpreter written in L interprets a language L' that is 'close' to L. In fact, most commonly used meta-interpreters for Prolog are not even able to interpret themselves.

For a given language L, there are many different meta-interpreters, varying considerably in their sizes. They range between minimal two clause meta-interpreters, the three clause 'vanilla' meta-interpreters, and up to 30 pages Prolog code meta-interpreters written in a pure stratified subset of Prolog with negation [DF92]. Evidently, these programs do not serve the same purpose although they all can be seen as members of the same class of programs.

Reification and absorption. The various meta-interpreters for a language differ in what aspects of the execution model they reify[1], that is, make explicit, and what aspects they absorb. Typically meta-interpreters will absorb unification or the search rule for backtracking.

[1] **reify** /'ri:ɪ,faɪ/ *v.tr.* (**-ies, -ied**) convert (a person, abstraction, etc.) into a thing; materialize. *The Concise Oxford Dictionary of Current English, 1990*

Layout of data structures. Reification of language elements is done by representing them in data structures. The simplicity and clarity of a meta-interpreter relies heavily on an appropriate layout. To underline this aspect, we will use predicates of arity one with the prefix is_/1 to define the used data-structures. These predicates can be seen as 'type declarations'. In most cases, deficiencies can be already discovered by simply considering their definition. A central design decision in meta-interpreters that is closely related to the layout of the data structures concerns the way the meta-interpreter is connected with the program to be interpreted.

2.2 Examples of meta-interpreters

The vanilla meta-interpreter. The vanilla meta-interpreter is still the most popular meta-interpreter. It originates from the syntax (and abstract syntax of terms) of Prolog 1 (and DEC10-Prolog). The infix operator ,/2 is used for denoting conjunction. The definition of the data structures the meta-interpreter relies on, already gives some hints on its deficiencies. The predicate is_body/1 that describes the layout of the list of conjunctions has to assume that is_goal/1 does not succeed for the functor ,/2.

```
mi(true).
mi((A,B)) ←
    mi(A),
    mi(B).
mi(Goal) ←
    clause(Goal,Body),
    mi(Body).

is_body(G) ←
    is_goal(G).
is_body((A,B)) ←
    is_body(A),
    is_body(B).
```

The definition assumes implicitly that clause/2 does not succeed for true/0 or ,/2 and will not yield an error. Further, in order to be tail-recursive, the meta-interpreter needs some cuts or negated goals. The introduction of cuts prevents meta-circularity, because the meta-interpreter is not able to interpret cuts correctly. We have either the choice of extending this meta-interpreter or restricting our considerations to meta-interpreters that do not need any cuts but are still meta-circular. A meta-circular meta-interpreter treating cuts correctly is significantly longer. Although Colmerauer corrected these deficiencies within Prolog II about ten years ago, this model of a meta-interpreter is still employed by recently developed logic programming languages like Gödel [HL92].

A decorated vanilla meta-interpreter. The most obvious correction to the vanilla meta-interpreter is to distinguish explicitly between the conjunction, the special goal true/0, and ordinary goals. The distinction is encoded by wrapping a new functor g/1 around ordinary goals. The resulting meta-interpreter has the same structure, but is now able to interpret itself. In addition, it is tail recursive and determinate with respect to the control operations.

```
mi(true).
mi((A,B)) ←
    mi(A),
    mi(B).
mi(g(Goal)) ←
    mi_clause(Goal,Body),
    mi(Body).

is_body(true).
is_body(g(G)) ←
    is_goal(G).
is_body((A,B)) ←
    is_body(A),
    is_body(B).
```

The linear meta-interpreter. The approach Colmerauer took in Prolog II was to change the syntax in order to linearize the body of a rule. The body is represented by a list of goals (Hs). Since every body has to be a list, the case of the empty list will be encountered for every body. If

a singe goal pred(X) is to be proved, the goal ?- mi_list([pred(X)]) has to be proved instead. In the next inference, this goal is reduced to the conjunction "mi_list(Body), mi_list([])". For every goal, the empty list [] has to be proved as the last goal. The meta-interpreter is no longer tail recursive. I.e., a tail recursive program is not interpreted in a tail recursive manner.

```
mi_list([]).
mi_list([G|Gs]) ←
    mi_lclause(G,Hs),
    mi_list(Hs),
    mi_list(Gs).
```

A tail recursive variation specializes the case of a goal list with a single element.

```
mi_list([]).
mi_list([G]) ←
    mi_lclause(G,Hs),
    mi_list(Hs).
mi_list([G,H|Gs]) ←
    mi_lclause(G,Hs),
    mi_list(Hs),
    mi_list([H|Gs]).
```

A linear tail recursive meta-interpreter. For the computation rule of Prolog that selects the leftmost atom, we are able to encode the resolvent explicitly. The list the meta-interpreter works on represents the complete resolvent. While the meta-interpreters given so far still rely implicitly on Prolog's computation rule, the following meta-interpreter mi_applist/1 interprets a program by always selecting the leftmost atom, even if mi_applist/1 is executed with another selection rule. The preceding meta-interpreters contained a conjunction of two recursive goals, whereas mi_applist/1 contains only one recursive goal. The selection rule has therefore not much choice for selecting an atom: The goal mi_lclause/2 contains only facts which must be selected in any case, while append/3 spawns only a finite branch bounded by the length of Hs.

```
mi_applist([]).
mi_applist([G|Gs]) ←
    mi_lclause(G,Hs),
    append(Hs,Gs,Is),
    mi_applist(Is).
```

```
always_infinite ←
    always_infinite,
    fail.
```

The predicate always_infinite/0 will possess only infinite failure trees when interpreted by the interpreter mi_applist/1 regardless of the computation rule. The preceding meta-interpreters possessed still a finite failure tree. The fact that no fair computation rule for ?- mi_applist([always_infinite]). exists indicates that we have reified Prolog's computation rule.

In mi_applist/1, there are still some redundancies: The goal append/3 receives the body Hs. The length of Hs can be determined statically. Thus append/3 can be executed statically. We fold the goals mi_lclause/2 and append/3 in order to overcome the redundant computations performed by append/3. The body of a clause is represented by a difference list.

```
mi_dllist([]).
mi_dllist([G|Gs]) ←
    mi_dlclause(G,Gs0,Gs),
    mi_dllist(Gs0).
```

```
mi_dlclause(h(X),[g(X)|Gs],Gs).
```

Meta-interpreters with ground representations. The meta-interpreters discussed so far reified the AND-control of Prolog. Unification was still absorbed. Prolog provides some predicates to reason about variables in Prolog, thereby mixing the variables of the object and the meta-level. In general, however, we have to resort to a ground representation of variables [Bar88]. One of the earlier meta-interpreters that uses a ground representation is presented by Kowalski [Kow79].

```
demonstrate(_Prog,Goals) ←
    empty(Goals).
demonstrate(Prog,Goals) ←
    select(Goal,Goals,RestGoals),
    member(Procedure,Prog),
    renamevars(Procedure,Goals,ProcedureR),
    parts(ProcedureR,Head,Body),
    match(Goal,Head,Sub),
    add(Body,RestGoals,InterGoals),
    apply(InterGoals,Sub,NewGoals),
    demonstrate(Prog,NewGoals).
```

This meta-interpreter still reuses implicitly some properties of Prolog. In particular, the atom member/2 which implements search uses still backtracking of Prolog to implement it's own choices. In the meta-interpreter above substitutions are applied to the whole resolvent.

Another encoding represents terms and variables by a binding environment. In the case of Prolog, the binding environment is a list of unifications that were encountered.

```
mi_be(true,E,E,N,N).
mi_be((A,B),E0,E,N0,N) ←
    mi_be(A,E0,E1,N0,N1),
    mi_be(B,E1,E,N1,N).
mi_be(g(Goal),E0,E,N0,N) ←
    mi_be_clause(Goal,N0,Head,Body),
    add_equation(Goal=Head,E0,E1),
    N1 is N0 + 1,
    mi_be(Body,E1,E,N1,N).
```

The binding environment is now a separate entity of the meta-interpreter. It can be extended arbitrarily. In addition to the unifications we are adding annotations of the form !(Goal) into the binding environment after a goal has been proven. The proven goals are represented by the structure (!Goal).

```
mi_be(true,E,E,N,N).
mi_be((A,B),E0,E,N0,N) ←
    mi_be(A,E0,E1,N0,N1),
    mi_be(B,E1,E,N1,N).
mi_be(g(Goal),E0,E,N0,N) ←
    mi_be_clause(Goal,N0,Head,Body),
    add_el(Goal=Head,E0,E1),
    N1 is N0 + 1,
    mi_be(Body,E1,E2,N1,N),
    add_el(!(Goal),E2,E).
```

```
is_binding_environment(Env) ←
    empty(Env).
is_binding_environment(Env1) ←
    add_el(A=B,Env0,Env1),
    is_binding_environment(Env0).
is_binding_environment(Env1) ←
    add_el(!(Goal),Env0,Env1),
    is_goal(Goal),
    is_binding_environment(Env0).
```

This meta-interpreter will be the basis for the introduction of partially static goals.

Chapter 3

Program Specialization

3.1 Programming techniques exploiting partial evaluation

We will review some programming techniques in Prolog that profit considerably from partial evaluation. Some of them are very well known, others do not seem to be in widespread usage in spite of their advantages. In the area of functional programming languages similar programming techniques were developed [Wad92].

Interpreters. The most frequently mentioned applications of partial evaluation are in the area of language implementation. Programming languages tailored to very specific needs cannot be implemented with the classical implementation techniques due to the high development costs. Instead, an interpreter is implemented in the language L. With a partial evaluator for L, the interpretation overhead is removed. In the setting of interpreter-driven compiler-generators [Neu88,Neu86] or self-applicable partial evaluators, the process of specialization can be accelerated considerably.

If a language has simple (meta-)interpreters, then a convenient way to enhance the language, or implement related languages, is by starting from a meta-interpreter and extending it. On the other hand, if a language proves too complex to provide a meta-interpreter we have to resort to other programming techniques to structure software.

One of the disadvantages of the programming technique of interpretational abstraction is that a completely new language is designed. Existing code is not directly reusable — or, must be linked together with the interpreter. Programmers must learn the new language. This may be acceptable for some specialized areas as expert-systems. It is not acceptable in general. Furthermore, the implementor of the new language must be very experienced to avoid design errors. Prolog as it stands (in the Prolog 1, Edinburgh, ISO-Prolog tradition) is such an example: The design of the abstract syntax tree was rather ad hoc and driven by completely peripheral considerations such as the layout of the concrete syntax.

Another disadvantage descends from the fact that meta-interpreters cannot be composed in general. There are some approaches to this problem [LS88], however, the composition of meta-interpreters itself must be expressed with new language constructs which have no resemblance in Prolog. E.g., in [LS88] new constructs called joins are introduced. Such constructs cannot be implemented with a straight forward meta-interpreter. However, they are easily implemented by some transformations.

Generic intermediate data structures. Often in programming, we already have some predefined predicates working on some data-structures. When similar predicates are desired for other data-structures, a very simple approach is to transform the data in such a way that it is directly usable by the predefined predicates. Typically, the generic data structure is a list. Many predicates are already defined on lists — e.g., a predicate for computing the sum of the list's elements. The

```
ground(T) ←
    nonvar(T),
    struct_make(T,State),
    ground_elements(State).

ground_elements(State) ←
    struct_done(State).
ground_elements(State0) ←
    struct_next(El,State0,State),
    ground(El),
    ground_elements(State).
```

Figure 3.1: Implementation of ground/1

predicate working on lists can be reused by transforming our specific data-structure into a list. The list itself is only needed by this transformation. Partial evaluation is sometimes able to remove such intermediate data structures. This approach is usable in Prolog as long as the number of elements to be transformed is finite. In the case of an infinite number of elements, we would need to extend the execution model of Prolog by delaying the process of generating new elements. This technique is especially popular in functional languages with lazy evaluation.

Although the approach of using some generic intermediate data structures is already rather abstract, it relies heavily on the efficient implementation of a partial evaluator. Furthermore, this approach does not *hide* the representation of sequences. The separation of abstraction and representation is not manifested. The typical outcome is that lists are used for all data to be represented.

Abstract datatypes and sequences. In other programming languages, especially in object oriented languages, an abstract data type would be defined to settle an interface for, e.g., an iteration once and forever. Since in Prolog many predicates working on sequences can be defined very easily, the need for abstraction is not realized so often. Still, there are many cases where abstraction would be desirable.

make/next/done-interface. An often suitable interface for defining abstractions for iterations is the make/next/done-interface which defines the basic operations on sequences. In close compliance to the convention in [O'K90], the abstract interface to deal with sequences is defined as follows:

1. make_v($A_1, \ldots, A_n$,S0): The arguments are mapped to the initial state.
2. next_v(E,S0,S1): The next element in the sequence S0 is given, the state S1 is the next state to be considered. Usually $\not\vdash$ next_v(E,S0,S0) is true.
3. done_v(S): Is true if there are no more elements in the list. Under the assumption that E is a free variable, done_v(S) should be equivalent to $\not\vdash$ next_v(E,S,_).

Implementation of ground/1**.** We are presenting a simple example that demonstrates the interface's potential of abstraction. The predicate ground(T) is true if T contains no variables. I.e., there is no subterm of T which is a variable. To access the subterms we are either converting the arguments into a list with the help of =../2 or the subterms can be accessed via functor/3 and arg/3. The make/next/done-interface is able to abstract the difference between these two implementation alternatives (see Fig. 3.1). For every non-variable term, some state is created (struct_make/2). The arguments of a term are accessed with struct_next/3. Finally, when all elements have been visited, struct_done/1 is true. The original version using =../2 contained the list of arguments as the state to be passed further. The version using functor/3 and arg/3 used two arguments representing the state: The whole term as a context argument and a counter to access the argument.

The version using =../2 (Fig. 3.2) shows that the make/next/done-interface is very similar to the way one works on lists. All predicates to represent argument access are trivially defined.

```
struct_make(Term,State) ←
    Term =.. [_|State].

struct_next(El,[El|State],State).

struct_done([]).
```

Figure 3.2: Representation with =../2

```
struct_make(Term,'Opaque'(N,Term)) ←
    functor(Term,_,N).

struct_next(El,'Opaque'(N0,Term),'Opaque'(N,Term)) ←
    N0 > 0,
    arg(N0,Term,El),
    N is N0 - 1.

struct_done('Opaque'(0,_Term)).
```

Figure 3.3: Representation with functor/3 and arg/3

In the other case, when we are interested in using functor/3 and arg/3 for the implementation we have to map two different variables into the single variable representing an abstract state. Both, the context argument containing the whole structure and the counter for accessing the arguments are passed on. The simplest solution is to introduce a new structure 'Opaque'/2 (Fig. 3.3) that contains as arguments both states. Note that we are introducing this structure only because the interface demands that the state is represented in a single variable. The structure is therefore completely redundant for the computation itself. The Mixtus-evaluator [Sah91] is able to remove the overheads induced by the more abstract interface in this case. I.e., the resulting code is identical to the hand-written specialized variant.

In the predicate ground/1, we were interested in abstracting some details in the representation of the iterators. The make/next/done-interface allows even more than that. In the example above, the iteration was basically performed by forward recursion; a backtracking variant may be derived in the same style. This is one of the most remarkable advantages of the make/next/done-interface. It provides the essential representation-dependent elements to define backtracking as well as forward recursive predicates. Take for example the subterm(Sub,T) relation which is true if Sub is a subterm of T. The program in Fig. 3.4 defines this predicate reusing the variety struct defined above. In the case of lists, the predicate struct_element/2 is nothing more than the predicate member/2 in disguise. Partial evaluation unmasks the original member predicate.

When programming with the make/next/done-interface, we are already reusing existing abstractions. The difficult but rewarding effort of finding the right abstractions has already been performed. In the course of usual programming, we are often faced with the problem of rewriting existing predicates to make them more useful. Often the backtracking variant is the most easy to define. In fact, it is less specific than the forward recursive variant: It does not even require that the atom nil/0 is used as a terminator. We consider it therefore as a starting point for creating an

```
subterm(T,T).
subterm(Sub,T) ←
    struct_make(T,S0),
    struct_element(S0,U),
    subterm(Sub,U).

struct_element(S0,Sub) ←
    struct_next(T,S0,S1),
    (   T = Sub
    ;   struct_element(S1,Sub)
    ).
```

Figure 3.4: Subterm relation with make/next/done

```
tree_element( t(TA,X0,TB), X) ←
    (   tree_element( TA, X)
    ;   X = X0
    ;   tree_element( TB, X)
    ).

tree_elementlist(nil) ⟶
    [].
tree_elementlist(t(TA,X,TB)) ⟶
    tree_elementlist(TA),
    [X],
    tree_elementlist(TB).
```

Figure 3.5: Accessing elements via backtracking and forward recursion.

abstraction. In Fig. 3.5, a predicate tree_element/2 is defined describing the relation between an element and the tree the element is in[1]. Although this backtracking version is easy to write, we cannot reuse it for other purposes. It is impossible in a pure Prolog program to define the sum of the elements of the tree with the help of tree_element/2. In fact, the backtracking version contains even less information than a corresponding forward recursive predicate because empty trees nil/0 are not considered. Although a corresponding forward recursive version is as well very easy to write (Fig. 3.5), there are many different forward recursive programming schemes that may be of interest. The DCG in (Fig. 3.5) maps the tree to a list. Iterations may then be performed on the list. Many partial evaluation schemes try to remove the intermediate list. This technique is known under the name deforestation [Wad88]. Also Pettorossi's techniques [PP91] focus on this problem.

The efficiency of using an intermediate list largely depends on the abilities of the partial evaluator. In fact, if the tree to be considered is very large, only the residual may be able to execute at all. Furthermore some seemingly unrelated computations may stall the executability of the partial evaluator. If we are working on an infinite sequence, the approach of converting the elements into a list is not very useful. We would need a lazy evaluation scheme to perform the operations.

At first, writing a more general make/next/done-version is more complex in the beginning. Nevertheless, the incentives in further transformations may be rewarding. Under certain circumstances, the make/next/done-version may also be generated automatically.

Both the backtracking and the forward recursive predicates in Fig. 3.5 are transforming a given tree-node into new goals. The backtracking version spawns an OR-tree consisting of three alternatives; the forward recursive version spawns an AND-tree consisting of the very same goals. Since we would like to serve both, we are not able to span a tree be it an AND-tree or OR-tree. As a compromise, we create a sequence that may be interpreted later as an AND-tree or an OR-tree. The symbol ∘ denotes concatenation of sequences:

$$\text{tree(t(TA,X,TB))} \Rightarrow \text{tree(TA)} \circ \text{node(X)} \circ \text{tree(TN)}$$

When a node/1 is encountered, the iteration is stopped to yield the element. If the sequence is empty, the iterations stops yielding the empty sequence. Both Prolog's AND-control and OR-control are selecting the leftmost node in their tree for proceeding further on. Concatenation can hence be implemented with lists. In Fig. 3.6, the predicate ptree_expand/2 implements this rewriting process. The predicate maps a sequence of nodes into another sequence which is either empty or starts with the node node/1. Note that other strategies of expansion are easily implemented by, e.g., reordering the nodes in the list of the last clause or by considering the other nodes that are in Ts0.

The interface is now easily settled in Fig. 3.7. The initial state is the root of the tree in make_ptree/2. The next element is found by expanding the tree with ptree_expand/2 and if the first element is a node(_). The iteration is finished if the sequence of nodes can be reduced to

[1]This example was mentioned by P. Singleton in article <2073@keele.keele.ac.uk> in a response to the author's article <1992Feb3.191603.22322@email.tuwien.ac.at> in *comp.lang.prolog*, USENET News, Internet. He remarked *that writing a make/next/done version of this is quite hard (it seems to involve passing around a list of yet-to-be-dealt-with snapped-off branches and node values).* Singleton suggested to used the forward recursive variant yielding generic intermediate data structures.

```
ptree_expand([],[]).
ptree_expand([node(X)|Ts],[node(X)|Ts]).
ptree_expand([tree(nil)|Ts0],Ts) ←
    ptree_expand(Ts0,Ts).
ptree_expand([tree(t(TA,X,TB))|Ts0],Ts) ←
    ptree_expand([tree(TA),node(X),tree(TB)|Ts0],Ts).
```

Figure 3.6: Generic expansion of goals.

```
make_ptree(T,[tree(T)]).

ptree_done(Ts) ←
    ptree_expand(Ts,[]).

ptree_next(node(X),Ts0,Ts) ←
    ptree_expand(Ts0,[node(X)|Ts]).
```

Figure 3.7: Tree walks with make/next/done

the empty list. The new predicate tree_element_2/2 accomplishes the same purpose as the original predicate tree_element.

Abstract programming interfaces as the make/next/done interface share many common properties that are of importance for program specialization. A successful specialization of programs using such abstract interfaces has to take these properties into account.

- Auxiliary data-structures are used to comply to the simple interface. In particular, if the state of the iterator is more complex consisting of n simple states, these states are wrapped with an auxiliary structure $s(a_1, \ldots, a_n)$.
- Data structures contain often redundancies in their layout. In the example in Fig. 3.7, we have used a list containing only the two structures: tree/1 and node/1. A less redundant representation would merge these structures with the list.
- The interface forces to perform redundant operations. E.g., skip_trees/2 is used in both done/1 and next/2. Removing such redundancies before further operations are performed is highly desirable.

3.2 Fold/Unfold-Transformations

Fold/unfold transformations are a well-known method for program transformation. Burstall and Darlington demonstrated the effectiveness for function definitions in the form of recursion equation schemes [DB76,BD77]. Their work was inspired by [Min70] in which he recommended programming as a good application area of artificial intelligence. The underlying structure of a fold/unfold transformation system separates the knowledge about equivalence preserving transformations and the tactics and strategies guiding the transformation.

```
ptree_member(E,Ts0) ←
    ptree_next(F,Ts0,Ts1),
    (   E = F
    ;   ptree_member(E,Ts1)
    ).

tree_element_2(T,E) ←
    ptree_make(T,Ts),
    ptree_member(E,Ts).
```

Figure 3.8: Elements via backtracking with make/next/done

```
generic_multiply(integer-A,integer-B,integer-R) ←
    R is A * B.
generic_multiply(float-A,float-B,float-R) ←
    R is A * B.
generic_op(A,B,R) ←
    generic_multiply(A,A,I),
    generic_add(I,B,R).

← peval generic_op(A,B,C).
```

```
generic_add(integer-A,integer-B,integer-R) ←
    R is A + B.
generic_add(float-A,float-B,float-R) ←
    R is A + B.
```

Figure 3.9: Example of an 'unevaluable' program

Strategies. The transformation rules are defined on a low level of abstraction. Strategies are therefore required to guide the application of the basic transformations. Mostly the problem consists in finding appropriate goals to unfold such that a later fold may be performed. Most of the work on fold/unfold systems is therefore concerned with finding procedures to apply transformations successfully.

Logic Programming. In the area of logic programming, Clark [CS77] and Hogger [Hog81] have pioneered the work on program transformation[2]. Tamaki and Sato proposed an influential fold/unfold framework for Prolog [TS84]. Finding fold/unfold transformation sequences is undecidable, for a proof for systems used for logic programs see Pettorossi [PP89].

Partial evaluation vs. partial deduction. Due to the many research efforts in functional languages, the notion of partial evaluation is also used for logic programming languages. Often partial evaluation is used for systems preserving procedural equivalence while partial deduction and fold/unfold transformations refer to equivalence based on the declarative semantics of logic programs. E.g., the Mixtus system [Sah91] is called a partial evaluator for full Prolog. On the other hand, Mixtus is already able to optimize programs without any data given.

Given the usual definition of partial evaluation in functional languages, partial evaluation is only applicable if some data for a function are static. If, on the other hand, no data are given, nontrivial partial evaluation is not possible. The program in Fig. 3.9 contains such an 'unevaluable' example. The predicate generic_op(A,B,C) implements $C = A^2 + B$ for two 'tagged' datatypes represented by the functor -/2 in the form Type-Value. In a similar manner, polymorphic predicates can be implemented which are close to the methodology of object-oriented programming. Although termed as a partial evaluator, the residual (Fig. 3.10) is rather the result of fold/unfold transformations. Mixtus is able to unfold generic_multiply/3 and generic_add/2 which permits to optimize generic_op/3 removing the 'tag' -/2 for the internal computations.

3.3 Problems with current specialization techniques

Lack of incrementality and interfaces. Most partial evaluators and fold/unfold-systems take a given program and specialize it accordingly. At the time of specialization the complete program must be present. In most systems part of a program may be left unknown. This will cause the system to make very conservative assumptions about the parts that are absent.

[2]Kowalski has considered in [Kow79], Chapter 9, 'Global Problem-Solving Strategies' another approach that is close to Burstall and Darlington's techniques. Instead of applying transformations on logic programs, i.e., statically, he considers similar transformations on goals during the course of attempting to solve them. This, in turn, is very close to plan generating systems, another area of artificial intelligence.

```
generic_op(A, B, C) ←
    generic_op1(A, B, C).

generic_op1(integer-C, A, B) ←
    D is C*C,
    A=integer-E,
    B=integer-F,
    F is D+E.
generic_op1(float-C, A, B) ←
    D is C*C,
    A=float-E,
    B=float-F,
    F is D+E.
```

Figure 3.10: Residual generated by Mixtus

Unnecessary variables. A class of variables which indicate potential inefficiencies is identified. These variables called *unnecessary variables* (according to [PP91]) or internal variables ([ST89]) are classified as follows.

- Existential variables (internal variables in the terminology of [ST89]) are variables which occur only in the body of a clause. Existential variables are representing intermediate data. Pettorossi assumes that this intermediate data can be completely removed when unfolding the predicates. But there is still a large class of programs with intermediate values stored in existential variables that cannot be removed with his techniques.
- Multiple variables occur more than once in the body of a clause. Multiple variables are often used to connect several independent computations depending on the same original structure.

In order to remove unnecessary variables new predicates are defined to fold all goals were the variable occurs in. The following example shows that removal of unnecessary variables is not always possible. We consider a simple context free expression described by the predicate prefixexpression/1. The first two arguments of the predicate prefixexpression/2 form a difference list. The predicate is essentially the expanded form of the DCG-program below.

The variable Xs1 is the only existential variable that may cause inefficiencies. The algorithm of Pettorossi and Proietti suggests to fold the two atoms in the body because they contain the existential variable. The program neither belongs to the class of tree-like programs (the atom prefixexpression(Xs1,Xs) depends on prefixexpression(Xs0,Xs1) via Xs1) nor is it a non-ascending program.

```
prefixexpression ⟶
    [n(_)].
prefixexpression ⟶
    [+],
    prefixexpression,
    prefixexpression.

prefixexpression(Xs) ←
    prefixexpression(Xs,[]).

prefixexpression([n(_Num)|Xs],Xs).
prefixexpression([+|Xs0],Xs) ←
    prefixexpression(Xs0,Xs1),
    prefixexpression(Xs1,Xs).
```

Applying the definition rule we define a new predicate new/2:

```
new(Xs0,Xs) ←
    prefixexpression(Xs0,Xs1),
    prefixexpression(Xs1,Xs).
```

Unfolding of the atoms and moving the second alternatives into the first yields:

```
new(Xs0,Xs) ←
    (   Xs0 = [n(_)|Xs1]
    ;   Xs0 = [+|Xs0_1],
        prefixexpression(Xs0_1,Xs1_1),
        prefixexpression(Xs1_1,Xs1)
    ),
    (   Xs1 = [n(_)|Xs]
    ;   Xs1 = [+|Xs0_2],
        prefixexpression(Xs0_2,Xs1_2),
        prefixexpression(Xs1_2,Xs)
    ).
```

```
new(Xs0,Xs) ←
    (   Xs0 = [n(_)|Xs1],
        (   Xs1 = [n(_)|Xs]
        ;   Xs1 = [+|Xs0_2],
            prefixexpression(Xs0_2,Xs1_2),
            prefixexpression(Xs1_2,Xs)
        )
    ;   Xs0 = [+|Xs0_1],
        prefixexpression(Xs0_1,Xs1_1),
        prefixexpression(Xs1_1,Xs1),
        (   Xs1 = [n(_)|Xs]
        ;   Xs1 = [+|Xs0_2],
            prefixexpression(Xs0_2,Xs1_2),
            prefixexpression(Xs1_2,Xs)
        )
    ).
```

Folding

```
new(Xs0,Xs) ←
    (   Xs0 = [n(_)|Xs1],
        (   Xs1 = [n(_)|Xs]
        ;   Xs1 = [+|Xs0_2],
            new(Xs0_2,Xs)
        )
    ;   Xs0 = [+|Xs0_1],
        new(Xs0_1,Xs1),
        (   Xs1 = [n(_)|Xs]
        ;   Xs1 = [+|Xs0_2],
            new(Xs0_2,Xs)
        )
    ).
```

Under the assumption that we may reorder clauses freely, we obtain again

```
new(Xs0,Xs) ←
    (   Xs0 = [n(_),n(_)|Xs]
    ;   Xs0 = [n(_),+|Xs0_2],
        new(Xs0_2,Xs)
    ;   Xs0 = [+|Xs0_1],
        new(Xs0_1,[n(_)|Xs])
    ;   Xs0 = [+|Xs0_1],
        new(Xs0_1,[+|Xs0_2]),
        new(Xs0_2,Xs)
    ).
```

```
new([n(_),n(_)|Xs],Xs).
new([n(_),+|Xs0_2],Xs) ←
    new(Xs0_2,Xs).
new([+|Xs0_1],Xs) ←
    new(Xs0_1,[n(_)|Xs]).
new([+|Xs0_1],Xs) ←
    new(Xs0_1,[+|Xs0_2]),
    new(Xs0_2,Xs).
```

The new predicate again contains a double recursion with the existential variable Xs0_2. Again we could try to define a new predicate by means of new/2 which tries to remove the double recursion. Although folding will always apply, there is no hope to remove these variables at all.

Consider we are at the n-th unfolding step and we have (beside $2^{n+1} - 1$ clauses that are facts or contain only a single recursion) a clause new_n as follows:

```
new_n(Xs0,Xs) ←
    new_{n-1}(Xs0,Xs1),
    plusseq(n,Xs1,Xs2),
    new_{n-1}(Xs2,Xs).
```

```
plusseq(0,Xs,Xs).
plusseq(s(N),[+|Xs0],Xs) ←
    plusseq(N,Xs0,Xs).
```

plusseq/3 is evaluable to a partial list containing n elements. The next unfolding step will yield $2^n - 1$ simple clauses and one clause that contains again a double recursion. The procedure to remove the unnecessary variables that are shared between the two atoms will therefore never terminate producing more and more specialized predicates that cannot be folded at all.

The same argument applies for other uses of prefixexpression/2. Consider the predicate prefixexpression_d/2 which is nothing more than duplicating the atoms of prefixexpression/2.

```
prefixexpression_d(Xs0,Xs) ←
    prefixexpression(Xs0,Xs),
    prefixexpression(Xs0,Xs).
```

Note that in this predicate the first atom prefixexpression/2 already ensures that the second atom will have precisely the same proof tree. Therefore, the second atom should be eliminable independently of the original modes of Xs0 and Xs. The fold/unfold process should therefore be able to define a new predicate which is equivalent to prefixexpression/2. The unfolding process will immediately generate existential variables that cannot be removed by further fold/unfolding. The second goal prefixexpression/2 can therefore never be removed by the usual transformation process.

Chapter 4

Partially Static Goals

Consider the computation to be performed in the following predicate p/2:

```
p(S0,S) ←
    q(S0,S1),
    r(S1,S).
```

We are usually interested in obtaining an optimized version of the predicate p/2. The goal q/2 performs some computation transforming S0 to S1, then r/2 will take S1 and perform computation depending on the result of q/2 via S1 and S. Some data for S0 will be given, therefore enabling to evaluate q/2. Still r(S1,S) will be not evaluable in general depending on S. As long as q/2 is dynamic, we can only concentrate on optimizing r/2 with respect to q/2.

To give the predicates useful names assume that q/2 is a compiler or 'preformatter' generating data in some well defined way. r/2 can be seen as an interpreter that interprets some general data-structures. The goal q/2 therefore restricts the possible values in some way, without yielding concrete values as long as S0 is not given.

In this setting partial evaluation which depends on concrete data cannot be applied. Current approaches concentrate on describing in some way the domain of the connected variable S1. The formalisms they employ for description constitutes the bottleneck in the propagation of static information about q/2. E.g., the domain used in the Mixtus-system uses only terms and free variables. The information that is propagated from q/2 to r/2 contains only concrete data and the annotations that some variables are free. There are several existing approaches that describe the domain of S1 in more detail without the knowledge of S0: We may use abstract interpretation, data-flow analysis, or type inference, to describe the domain in more detail. In our view there are several disadvantage to these approaches:

Complexity of algorithms and runtime. The algorithms are very complex even if the precision of the domain is not very high. Type inference algorithms become undecidable very soon. When typing is of interest, user defined declarations have to be provided. In [Deb92] it is shown that the worst case complexity for dataflow analysis for even extremely simple programs is exponential in the program size. The same argument applies to any algebraic formulation [FN88,FNT91].

Generality. All of these approaches need to introduce a new formalism to describe what can be assumed after q/2 has been proven. It is therefore very probable that the formalism itself needs to be extended from time to time.

Intuitivity. There are many optimization schemes that are very simple to understand, but that are not expressible in current transformation systems. The unification algorithm in the Vienna Abstract Machine explained in a later chapter may serve as such an example. On the other

hand, current transformation systems are able to perform derivations that are difficult to understand even if a programmer is able to inspect the transformation process. One reason is that in many cases the code generated by a partial evaluator is over-specialized.

Many programs are based on structural induction. The 'desirable' properties can be derived by similar techniques i.e., fold/unfold rules.

In this thesis we propose a different approach: Due to the computation rule of Prolog, we are able to assume after the goal q(S0,S1) nothing more than that q(S0,S1) has been executed successfully, i.e.: there exists a finite computation (proof tree) for the goal q(S0,S1). We are able to assume this *statically* but the concrete values for q(S0,S1) may not be given. The goal is therefore specified only *partially*. This knowledge is annotated by introducing a *partially static goal* after the actual goal. The partially static goal is marked with a !. Please note that the !/1 has nothing to do with cuts. It is defined as a prefix-operator with ← op(900, fy, !).

```
p(S0,S) ←
    q(S0,S1),
    !q(S0,S1),
    r(S1,S).
```

We are now able to define a new predicate 'r_!q'/2 that can be specialized with respect to p/2 — without any concrete data given.

```
'r_!q'(S1,S) ←
    !q(_,S1),
    r(S1,S).
```

Partially static goals can be seen as a way to describe the binding environment in a more general form. Partially static goals describe a part of the history of performed computations. With our framework we have therefore settled a very general domain which introduces a formalism that is very close to ordinary Prolog programs. The formalism was chosen in order to be able to reuse existing fold/unfold transformations. We will first give a small example to demonstrate the capabilities of this approach. The new transformation rules that need to be incorporated into a fold/unfold framework are given thereafter.

A first example. The following predicates are defining lists with elements that form simple regular expressions. The predicate is_ab_xy/1 describes a list of the form ((a|b)(x|y))*, the predicate is_bc_/1 describes a list of the form ((b|c)z)*, where z denotes an arbitrary token.

```
is_ab_xy([]).
is_ab_xy([AB,XY|L]) ←
    (AB = a; AB = b),
    (XY = x; XY = y),
    is_ab_xy(L).
```

```
is_bc_([]).
is_bc_([BC,_|L]) ←
    (BC = b; BC = c),
    is_bc_(L).
```

Let us assume that some facts f/1 have been stored into the database which satisfy is_ab_xy/1. These facts may represent programs that are to be interpreted later on. The predicate is_b_xy/1 is defined as follows:

```
is_b_xy(L) ←
    f(L),
    is_bc_(L).
```

At specialization time the facts may not be given, or the size of the data is that large that a specialization of all facts f/1 is impossible due to space restrictions. We are still able to specialize is_b_xy/1 with respect to the database since all facts satisfy is_ab_xy/1. In current frameworks we would need a domain that is able to describe — in this case — regular expressions of lists. With our approach, we reuse the knowledge that is_ab_xy/1 has been executed directly:

```
is_b_xy(L) ←
    f(L),
    !is_ab_xy(L),
    is_bc_(L).
```

We are now able to specialize the goal is_bc_(L) with respect to the partially static goal !is_ab_xy(L). A new predicate is defined accordingly:

```
is_b_(L) ←
    !is_ab_xy(L),
    is_bc_(L).
```

The transformations we apply are a slight extension to fold/unfold transformations. In most of the cases static goals are treated like unification goals in fold/unfold transformations. The data obtained by the unfolding of dynamic goals may be freely propagated into the static goal. The inverse is only true if the static goal has precisely one solution. Instead of unfolding only dynamic goals, we are also able to unfold static goals. Unfolding a static goal does not have any direct impact to the operational semantics of the program. Note that the unfolding process is driven by the dynamic goal.

```
is_b_([]) ←
    !is_ab_xy([]).
is_b_([BC,XY|L]) ←
    !is_ab_xy([BC,XY|L]),
    (BC = b; BC = c),
    is_bc_(L).
```

```
is_b_([]).
is_b_([BC,XY|L]) ←
    % !is_ab_xy([BC,XY|L]),
    !((BC = a; BC = b)),
    !((XY = x; XY = y)),
    !is_ab_xy(L),
    (BC = b; BC = c),
    is_bc_(L).
```

Because static goals are only some annotations that have no effect on the execution of a program, we are allowed to discard them at any time. In this example, the variable XY is only of interest for the static goal. Furthermore, since static goals belong to the binding environment, we are allowed to propagate them to the right as far as desirable.

```
is_b_([]).
is_b_([BC,_|L]) ←
    !((BC = a; BC = b)),
    (BC = b; BC = c),
    !is_ab_xy(L),
    is_bc_(L).
```

Folding is performed similar to fold/unfold transformations. I.e., if the body of the definition is found, we are able to replace the body by the defined predicate.

```
is_b_([]).
is_b_([BC,_|L]) ←
    (   !((BC = a; BC = b)),
        BC = b
    ;   !((BC = a; BC = b)),
        BC = c
    ),
    is_b_(L).
```

In the last step the static goals are interacting with the dynamic goals. Due to the properties of unification, we can propagate unifications as often as desired. Therefore a static goal that describes only a unification can be transformed into a dynamic goal at any time. Conversely, we are able to propagate a dynamic binding into a static goal. In our case, we are not able to find an alternative of the static disjunction that is true for the dynamic goal BC = c. The static goal is therefore transformed into a static failure. Similar to static unifications, also static failures are converted into dynamic failures. The second alternative is therefore deleted.

```
is_b_([]).
is_b_([BC,_|L]) ←
    (   BC = b
    ;   c = c,
        !((c = a; c = b))
    ),
    is_b_(L).
```

The final predicate is more efficient than the original predicate. Yet, no concrete data was necessary for its derivation.

```
is_b_([]).
is_b_([b,_|L]) ←
    is_b_(L).
```

Comparison with usual fold/unfold-techniques. In the usual setting given for fold/unfold-transformations, we are only able to specialize a conjunction of dynamic goals. In our example this would mean that the computations performed by the goal is_ab_xy/1 are manifest in the residual.

```
is_b_traditional(L) ←
    is_ab_xy(L),
    is_bc_(L).
```

Furthermore, it is impossible to apply fold/unfold-transformations that preserve the program's operational semantics for the predicate is_b_traditional/1: Fold/unfold transformation requires the *reordering* of dynamic goals. A different proof tree is therefore constructed. In our case, the goal ?- is_b_traditional(L). will yield only one solution (L = []), thereafter an infinite failure branch is encountered. Applying fold/unfold-transformations must not yield a program with different behavior. In the unfolded predicate is_b_traditional/1, it is impossible to reorder unifications in such a way, as to make a successive fold, because left propagation may prune some infinite branches.

```
is_b_traditional([]).
is_b_traditional([BC,XY|L]) ←
    (BC = a; BC = b),
    (XY = x; XY = y),
    is_ab_xy(L),
    (BC = b; BC = c),
    is_bc_(L).
```

4.1 Transformation rules

The transformation rules to be defined are an adaptation of the usual fold/unfold setting. We are using some elementary transformations and some derived transformations. Derived transformations are only shortcuts that shorten the transformation process. The new contribution is the introduction of partially static goals and their associated transformation.

The following transformations preserve all SLD-trees, whether they are finite or not, that are computed with a fixed left to right computation rule. Equivalence is preserved for all search rules

which are responsible for selecting an OR-branch. Therefore, Prolog-equivalence is included. By a goal we refer to both a dynamic goal and a static goal. The transformation rules are valid for pure Prolog without negation. Using impure predicates reduces the applicability of our transformation rules.

Creation and propagation of bindings

A binding is represented by

- a unification goal $a = b$. A unification goal may be created after a corresponding static unification goal $!a = b$
- a static goal. A static goal $!G$ may be created *after* a dynamic goal G.

Bindings can be always propagated to the right side. Unification goals may be propagated to the left side, over a goal G, if G has finitely many solutions. A static goal $!G$ may *recreate* a dynamic goal, if the predicate $H \leftarrow G, G.$ can be transformed into G. This is especially true for unification goals and for the predicate fail/0.

A static goal $!S$ yields a generalized static goal $!G$, if the definition of G can be obtained from the definition of S by replacing some goals in S by true/0. If we can assume that S has a finite success branch, we may as well assume that G has a corresponding finite success branch that will be shorter.

Unfolding

Unfolding is the process of replacing a goal by its definition. Unfolding for Prolog has to retain the order of solutions. Therefore the usual transformation rule [TS84] which unfolds a goal G in a clause by generating a new clause containing the body of every clause whose head is unifiable with G is not applicable. Unfolding is in this case only Prolog-equivalent if the first goal is chosen for unfold. The unfolding procedure has to be restricted to the first goal. We are assuming that Clark's completion procedure [Cla78] has been applied to every predicate:

Dynamic goals: Replace the goal G by the right hand side of the corresponding completed procedure "$p(X_1, \ldots, X_n) \leftrightarrow E_1 \vee \ldots \vee E_m$" as follows:

$$\ldots, G, \ldots$$
$$\Downarrow$$
$$\ldots, p(X_1, \ldots, X_n) = G, (E_1 \vee \ldots \vee E_m), \ldots$$

Static goals: Replace the static goal $!S$ by the right side of the corresponding definition as follows.

$$\ldots, !S, \ldots$$
$$\Downarrow$$
$$\ldots, !(p(X_1, \ldots, X_n) = S, (E_1 \vee \ldots \vee E_m)), \ldots$$

Therefore, static goals are replaced by static disjunctions like dynamic goals are replaced by dynamic disjunctions.

Definition

A new predicate (definition predicate) is added to the program. It consists of a conjunction of existing goals referring to existing predicates and some bindings. Since the predicate is not used in any other context it is considered as a definition. No recursive definition is allowed.

Folding

A conjunction of goals $G_1, \ldots, G_n$ in the body is replaced by the head of a definition predicate. Unfolding the definition predicate must result in the original goals. No recursive dependencies are allowed during folding.

Functionality

A predicate $p(a_1, \ldots, a_i, a_j, \ldots, a_n)$ is functional with respect to the arguments $a_1, \ldots, a_i$ if the predicate $q/n + i$ defined below

$$\begin{array}{l} q(a_1, \ldots, a_i, b_1, \ldots, b_i, a_j, \ldots, a_n) \leftarrow \\ \quad p(a_1, \ldots, a_i, a_j, \ldots, a_n), \\ \quad p(b_1, \ldots, b_i, a_j, \ldots, a_n). \end{array}$$

can be transformed into

$$\begin{array}{l} (q(a_1, \ldots, a_i, a_1, \ldots, a_i, a_j, \ldots, a_n) \leftarrow \\ \quad p(a_1, \ldots, a_i, a_j, \ldots, a_n))\theta. \end{array}$$

A goal G can be replaced by θ, if there is a partial static goal $!S$ for the same predicate and the predicate is functional.

Goal deletion

Goal deletion is a special case of functionality. A goal G may be deleted if there is a partial static goal $!S$ before G, and the predicate P defined as $P \leftarrow S,G$ can be transformed into S respecting OR-equivalence. Although goal deletion is applicable for logic programs in any case, this is not true for pure Prolog programs. The following predicate select_select/3 is not equivalent to select/3 because it cannot be transformed into the form of select/3.

```
select(E,[E|Es],Es).
select(E,[F|Es],[F|Fs]) ←
    select(E,Es,Fs).
```

```
select_select(E,Es,Fs) ←
    select(E,Es,Fs),
    select(E,Es,Fs).
```

The predicates are not equivalent because after the process of unfolding the two goals of predicate select_select/3, we cannot apply indexing to the resulting four goals. This can also be seen from the fact that the two heads of select/3 are unifyable (select(E,[E|Es],Es) = select(E,[F|Es],[F|Fs])). In other words, while the goal ?- select(E,Es,Fs). yields solutions of the form

Es = [A1,...,AN,E|L], Fs = [A1,...,AN|L],

the goal ?- select_select(E,Es,Fs). yields solutions of the form

Es = [E,...,E|L], Fs = [E,...|L].

Ideally, select/3 selects an element in A1, while A2 is the list with the selected element and A3 the list without it. As we have shown in [Neu92], such cases may lead to the creation of infinite failure branches. Note that goal deletion is also applicable for potentially nonterminating predicates as the following predicate p_nont/1 shows. The predicate may loop if the first rule is selected.

```
p_nont(k(N)) ←          ... ←
    p_nont(k(N)).           ...,
p_nont(null).               !p_nont(N),
p_nont(s(N)) ←              p_nont(N),
    p_nont(N).              ....
```

Still, we are able to remove the dynamic goal if a corresponding static goal is present. The static goal ensures that there is a finite success branch for the goal. If there is not a finite success branch for !p_nont(N), the position where p_nont(N) stood will never be reached. The second goal can neither loop nor fail, nor does it contribute to the actual bindings.

Goals with finitely many solutions

In fold/unfold-transformation we often obtain during the 'case analysis' after unfolding a conjunction of goals, a conjunction of the form P, fail. In most approaches like [Sah90,Sah91] such branches are eliminated by default, otherwise such a branch cannot be removed. Prolog-equivalence is therefore only provided for terminating programs. The determination of terminating goals involves in general a termination proof. There are, however, many simple cases that do not need the full generality of [Plü91,Plü90].

Depth bound predicate. A predicate is a depth bound predicate if it contains an argument whose size constitutes an upper bound for the number of inferences to be performed. Every predicate p/n can be transformed into a depth bound predicate by adding a new argument:

- In case of a rule, the argument contains a term $s^n(N)$ in the head. In every goal the argument contains a term $s^k(N)$ with $k < n$. The variable N is a newly introduced variable.
- In case of a fact, the argument contains the term $s^n(0)$.

A goal G is terminating if

- "G, fail" can be reduced to fail.
- there are bindings expressed by the static goal $!S$ and the depth bound versions S' and G' can be folded such that the arguments r_s and r_g for determining the depth of the computation are the same. I.e., in the resulting predicate the 'counters' must be the same.

  $$t(a_1, \ldots, a_n, r_s, b_1, \ldots, b_m, r_g) \leftarrow$$
  $$\quad !s(a_1, \ldots, a_n, r_s),$$
  $$\quad !g(b_1, \ldots, b_m, r_g).$$

  can be transformed into a new predicate (a folded predicate) that contains both arguments in any clause, and the size of the terms (the number of structures s/1) is for every argument the same.

Deletion of finitely failing goals

First some goals may be reduced by other means than termination proofs: Goal deletion and functionality may remove the goal. Other goals must be terminating in the above sense.

Deletion of overshadowed alternatives

The following transformation applies only to Prolog. Note that it may transform predicates in an sometimes unintended way: Alternatives that are never reached are discarded. This transformation only reduces the size of a program, and, maybe, some choice-points at runtime. Usually, applicability of this rule is an indication of a programming error. A clause is overshadowed by an infinite branch if a preceding clause will always yield an infinite number of solutions. In the current setting we are only detecting this case if a clause can be reduced by other transformations to: c ← c.

Treatment of built-ins

Many built-ins are sensitive to left propagation of bindings. Furthermore, static goals may be propagated to the right only if they contain built-ins whose properties will remain invariant. E.g., the built-ins ==/2, >/2, nonvar/1 may be safely propagated to the right as static goals. Other built-ins like
==/2, var/1 cannot be propagated as static goals. Some built-ins cause further failures, e.g., "!X > Y, !Y > X". In general, some constraint solver would be desirable. Cuts in static goals are ignored. Some information may be lost, because the domain described by the related predicate without cuts is larger. Techniques as they are proposed by [BR89] or [Sah91] might be adaptable.

Further transformations

In current partial evaluators there is only one level of bindings. If a structure is propagated to the right, a dynamic unification is propagated. In most cases unifications propagated to the right can be absorbed by the following predicates. In some circumstances this is not possible [Sah91]. Using partially static unification goals for right propagation overcomes this problem. Static unifications do not need to be materialized into a dynamic unification.

4.2 Strategies

One of the advantages of partially static goals is that they fit into the general framework of fold/unfold transformations developed for logic programming languages. When adopting general strategies, e.g., [PP91] we have to be aware of the correct reordering of goals: Dynamic goals are not allowed to be reordered but static goals may be propagated freely to the right hand side. Many fold/unfold transformations require that, after unfolding of two goals, we are allowed to reorder the goals. The typical pattern that occur after unfolding two goals A, B is as follows:

$$\ldots\ A_1,\ A_2,\ B_1,\ B_2,\ \ldots$$

Let the pair A_2, B_2 be those goals that may be folded. To preserve Prolog-equivalence, we must be able to reorder the goals A_2 and B_1. In general this is not possible because A_2 may loop if B_1 fails or because the order of solutions produced will be exchanged. If the original goal A was a partially static goal, also A_1 and A_2 are static goals. A_2 can be propagated freely to the right.

A simple 'proof' without the occur-check. Sometimes partially static goals are able to transform a non-recursive program, in the following program we consider the goal L = [_|L], into a recursive program.

```
inflist(L) ←          is_list([]).
    L = [_|L].        is_list([_|L]) ←
                          is_list(L).
```

The predicate inflist/1 will attempt to create an infinite list for L. In a Prolog with occur-check the unification can fail statically removing the predicate. But, as it is often the case in current Prolog systems, infinite terms [Col84] are supported instead of the finite first order terms. Therefore the unification might succeed. The decision has to be kept dynamic. On the other hand infinite lists are very seldom used intentionally. It is very probable that we might be able to derive a static partial goal for L like !is_list/1 which guarantees that L is already a list of fixed length. If a partially static goal like !is_list/1 is given, the goal "?- !is_list(L), inflist(L)" will always fail — with or without occur-check.

A new predicate is defined which is specialized to the case of L being a list. I.e. there is already a static finite SLD-branch for the goal is_list(L) (or bindings equivalent to its existence). By using the techniques of fold/unfold we are able to propagate the 'bindings' of the static goal.

```
inflist_is_list(L) ←
    !is_list(L),
    L = [_|L].
```

Note that the whole disjunction obtained after unfolding the static goal is again a static goal.

```
inflist_is_list(L) ←
    !((  L = []
    ;    L = [_|L_1],
         is_list(L_1)
    )),
    L = [_|L].
```

Case analysis is able to remove the first alternative.

```
inflist_is_list(L) ←
    ! L = [_|L_1],
    !is_list(L_1),
    L = [_|L].
```

Propagating !is_list/1 further, changing the static goal ! L = [_|L_1] into a dynamic goal.

```
inflist_is_list(L) ←
    L = [E|L_1],
    !is_list(L_1),
    [E|L_1] = [E,E|L_1].
```

```
inflist_is_list(L) ←
    L = [E|L_1],
    !is_list(L_1),
    L_1 = [E|L_1].
```

For reasons of simplicity, we generalize this predicate by making E again an anonymous variable. (An alternative approach would define a new predicate inflist_is_list(E,L), the result is in any case the same). Then folding the definition of inflist_is_list/1 is applicable.

```
inflist_is_list(L) ←
    L = [_|L_1],
    inflist_is_list(L_1).
```

Therefore, the predicate will never succeed. Showing termination of inflist_is_list/1 constitutes in showing that any branch in !is_list(L) determines a branch in inflist_is_list/1. In this case this is very simple. In general such termination proofs are only applicable if the static goal can be folded completely into the dynamic goal.

4.3 Related Work

4.3.1 Bossi's method of specializing logic programs

The following basic operation are used: Unfold, fold, pruning derivable clauses, thinning clauses, and fattening clauses. Additionally, atoms may be constrained and hidden. In addition to the logic program predicates are used for constraining predicates. Our technique is closely related to the work of [BCD90]. In contrast to their approach partially static goals are applicable to Prolog, while their methods can only be used for logic programs. Prolog-equivalence is not preserved by their operations.

4.3.2 Type systems

Static goals share a similar idea with the type system proposed by Naish [Nai92]. Naish proposes to prescribe types by using type declarations which are similar to ordinary Prolog rules. Instead of a rule *Head* ← *Body*. a type declaration is defined as *Head* type *Body*. Any goal in the type specifications of Naish can be used in the body. This goes beyond the traditional view of types [CW85]. Naish gives examples that are similar to traditional pre- and postconditions that have been treated separately from type systems:

```
merge(A,B,C) type
    sorted(A),
    sorted(B),
    sorted(C).
```

Such a 'constraint' cannot be expressed with ordinary type systems. Polymorphism is expressed by sharing variables in the body of a type declaration:

```
append(A,B,C) type
    list_of(T,A),
    list_of(T,B),
    list_of(T,C).

list_of(_,[]).
list_of(T,[E|Es]) ←
    is_a(T,E),
    list_of(T,Es).
```

```
is_a(boolean,B) ←
    is_boolean(B).
is_a(integer,I) ←
    integer(I).
is_a(list_of(T),Es) ←
    list_of(T,Es).

is_boolean(true).
is_boolean(false).
```

This system has been criticized of being too general. Indeed, the type declarations are computationally intractable. Statically and even at runtime as any program is. However, it permits to express many 'assertions' than cannot be expressed with usual type systems, but which are statically decidable.

```
p_while(X,_) type
    invariant(X).
```

```
p_while(X,X) ←
    not cond(X).
p_while(X0,X) ←
    cond(X0),
    update(X0,X1),
    p_while(X1,X).
```

Showing partial correctness consists in optimizing the type checks away. I.e. it must be proved somehow that if invariant(X0) and cond(X0) hold invariant(X1) will hold after update(X0,X1).

The theoretical argument of undecidability has discouraged many language implementors from considering such expressive constructs. Even in procedural languages where the concepts of assertions are well known since twenty-five years [Flo67] and have been refined continuously [Hoa69]

[Dij76,DS90], the argument of undecidability and the 'overheads' of runtime checks have prevented the development of safer languages. It is only a rather recent development that invariants, pre- and postconditions have found their way into commercial viable programming languages. The specification is seen as a contract between the environment, which has to comply to the preconditions, and the program, which must comply to the postconditions. The resulting theory of programming by contract — see [Mey85] and [Mey90] for related literature — forms the basis of assertion mechanisms in the programming language Eiffel [Mey92].

Our proposed method of program transformation may be applied to reduce the overheads induced by the assertion like type declarations. If the residual program has no calls to errors anymore then there is no possibility that the type declarations are violated. The program has thus been verified implicitly with respect to the specification.

4.3.3 Partial evaluation for CLP-languages

CLP-languages [JL87,JLM86] are logic programming languages that provide instead of syntactic unifications, some other unification theories. CLP(Bool), CLP(FT) (Prolog as defined in the Prolog 0-version. I.e., with disequalities allowed as constraints), CLP(R), CLP(N), CLP(Q) (in [Col87]) are the most prominent examples. Recent work on partial evaluators for CLP-languages can be found in [SH90,HS91,Smi91].

4.3.4 Futamura's Generalized Partial Computation

In addition to the static knowledge of a partial evaluator, the technique of Generalized Partial Computation [FN88,FNT91]. uses information about the operating environment. This information is expressed with the help of some logic which can be propositional logic, predicate logic, or informal logic, depending on the predicate evaluation power.

The difference to ordinary partial computation lies in the treatment of conditionals. Due to the information i which does not only describe simple bindings of the form $v = \text{const}$, but also arbitrary logic descriptions about variables, conditionals can be made statically even if the expression contains dynamic parts. In an expression with p dynamic "if p then x else y", a usual partial computation function peval_{pc} will reduce the expression to "if $\text{peval}_{pc}(p, e)$ then $\text{peval}_{pc}(x, e)$ else $\text{peval}({}_{pc}y, e)$". A generalized partial evaluator peval_{gpc} uses, instead of the binding environment e, the more complex environment i. It will first use a theorem prover to check if the resulting value is statically known, i.e., whether or not $i \vdash p$ although there are some dynamic variables in p. In the case that the conditional is static and can be substituted by the appropriate branch. In the case that p is not statically decidable or too costly to compute, the generalized partial evaluator behaves similar to the original evaluator. The only difference is that the environment i refined depending on the branch taken. Either p or $\neg p$ is added to i.

$$\text{if } \text{peval}_{gpc}(p, i) \text{ then } \text{peval}_{gpc}(x, i \wedge p) \text{ else } \text{peval}_{gpc}(y, i \wedge \neg p).$$

Note that generalized partial computation can be generalized even further: While Futamura is using the theorem prover to determine some conditionals statically, one may as well determine statically using algebraic reasoning the value of other expressions with still dynamic arguments.

4.3.5 Lavrov

In [Lav88], Lavrov has criticized the current partial evaluation enthusiasm. As the annotated bibliography [SZ88] puts it:

> Some reflections on ordinary and mixed computation are given. A number of possible pitfalls in speaking about mixed computation are pointed out.

Lavrov did not only criticize current habits, he also pointed out how narrow the view of partial evaluation is. We believe that our work fits into his suggestions. The following quote is taken literally from [Lav88].

> More interesting though is the case when the value [the input to a function] is defined by more complex expression up to the case when the program (procedure) calculating the value is given. Such an approach is rather natural in functional programming, where any program is a function having some value. Nevertheless in common (procedural) programming this approach makes sense too. Let G be a program, x is the intersection of (the set of variables representing) the results of the program with the arguments of another program F. One can consider the program $G; F$ and state the problem of simplifying the part F of the program taking into account that the value of x is supplied by the program G and is therefore not an arbitrary one but is bounded to some restrictions. Essentially it is a way to specify an areal of the value of x (in the above mentioned sense).
>
> Following the analogy with inverse predicate transformers in denotational semantics the problem of transformation of program G may be stated as well. This time one has to start from the fact that the results of the program G are used not in arbitrary way but as the initial values for the program F.
>
> In other words any composition of algorithms naturally leads to the optimization (mixed evaluation, if you like) problem for the resulting algorithm.

4.3.6 Kasyanov's annotated programming

Kasyanov [Kas91] (original reference [KP82]) considers the specialization ('concretization') of PASCAL programs with respect to some annotations. He suggests to construct various tools based on this principle: source-to-source optimizers, 'instrumentation tools' that add checks into the program to test the given annotations similar to the approach Eiffel [Mey92] follows, verification tools that try to find out implausibility properties [KP79]. Kasyanov's transformation machine discerns four kinds of elementary transformations:

Schematic transformations: removing and inserting inaccessible/unnecessary fragments, computations, and branches; replacement of variables and terms according to their properties; copying fragments; composing fragments; fold/unfold. These transformations correspond directly to the transformation performed in the usual setting of an fold-/unfold system.

Elementary transformations: reflecting the semantics of language constructs. Again, this part fits into a fold-/unfold system

Elementary transformations using domain knowledge: e.g. algebraic properties of datatypes. They are also mostly present in fold-/unfold systems.

Property transformations: *New* annotations are generated by extracting information from basic program constructions.

Annotations are split in assertions which are represented by a predicate about the memory state and directives which are statements on the memory. The assertions in the annotated programming framework could permit to use the common pre- and postcondition formalism. The approach employs at least three levels: The PASCAL programs, the directives (which are not further discussed) and the assertions. Linguistically two formalisms are needed: The PASCAL programs and the assertions. Furthermore, some annotations given do not fit well into a pre- and postcondition

framework. E.g., DEAD(Y) to assert that Y will no longer be used. However, neither is a detailed description given which assertions are possible, nor is it discussed how new annotations are derived, nor how programs and directives are related to one another.

Partially static goals could be mapped into this framework. However, compared to Kasyanow, we are not using a new different formalism for describing properties which evidently complicates the system. Furthermore, we are able to reuse the existing strategies for fold-/unfold systems directly. Partially static goals correspond rather to the notion of directive than to the notion of assertion in annotated programming, because partially static goals are able to describe concrete bindings. Assertions are rather close to the formalism by Bossi [BCD90].

4.3.7 Turchin's super-compiler

Supercompilation is another technique for specializing in particular functional programs. Turchin notes that super-compilation is more general than partial evaluation. In partial evaluation one considers only the potential redundancy caused by fixed values of variables. Super-compilation is also able to reduce redundancy induced by nested loops and repeated variables.

In [Tur86] (Section 2, Historical and comparative remarks) Turchin compares the process of super-compilation with Burstall and Darlington's program transformation system. He notes that the unfold rule corresponds to the basic driving step in super-compilation. The action of looping back and declaring the recurrent configuration corresponds to the fold rule. Since Turchin does not mention other transformation rules we have the impression that super-compilation can obtain at most what a fold-/unfold system is able to derive. The examples given by Turchin [Tur86] are explicable within an fold/unfold-system. Nevertheless, we underline the advantages of super-compilation compared with a general transformation system: The process of super-compilation may be faster than a general transformation system, because the strategy of applying transformation rules is highly restricted. On the other hand, many current fold/unfold-based algorithms are also restricting the application of transformation to accelerate the speed of transformation trading transformation speed for execution speed.

Chapter 5

Binary Programs

It seems to be a general tendency in current language implementation techniques to reduce the implementation effort by defining an appropriate sub-language or subset of a language that is easier to implement than the original language. The language implementor can therefore concentrate on the efficient implementation of the essential mechanisms. The remaining mapping of the underlying language is handed over to a higher level that mostly employs source-to-source transformations. Such has been successfully done in the area of functional languages with the introduction of continuation passing style [App92], in the area of object oriented languages by replacing classes and inheritance by prototypes and delegation [Ung87], in the area of computer architecture by defining processors with a simplified and more regular instruction set, in the area of concurrent logic programming languages [Sha89] and in the area of logic programming with the introduction of binary programs. An often observed interesting outcome is that the resulting simplified languages turn out to be subject to a better formal treatment than the original languages. In most cases these sub-languages existed already in some theoretical framework before their relevance for practical implementations was discovered. The sub-languages are also better suited to express optimization schemes in the form of source to source transformations. In functional languages the continuation passing style is closely related to Church's λ-calculus and denotational semantics. The quadruple code was commonly used by compilers as an intermediate language for code generation even if the computer architecture itself used another encoding. Furthermore, the sub-languages are encouraging new programming styles that contribute to the expressibility of the original language. First class closures allow the programmer to define their own control structures such as iterators over abstract data types representing containers (or collections). Although such higher level constructs are expressible in the original language as well, a considerable execution overhead has to be paid for.

Binary programs are the equivalent in logic programming to the continuation passing style (CPS) [Sto77,Wan80] in functional languages. The first approaches to continuation passing date back to 1963. Coroutines as introduced by M.E. Conway [Con63] required the usage on a continuation passing regime. The implementation technique acquired more attention by the development of the programming languages SIMULA I [DN66] which used the technique for implementing coroutines and ISWIM [Lan66], an implementation of strict λ-calculus, the ancestor of most modern applicative lazy languages. Probably the first mention of the technique of binary programs at the implementation level is due to David Warren [War83]. At least Caneghem reports 1986 [Can86] that he is unaware that anybody has used this technique, citing Warren's report as the only one that has considered the idea. The explicit notion of binary programs is probably due to Tarau [TB90].

Binary definite programs are a subset of definite programs. They consist of binary clauses and the unit clause true/0, which is a simple fact. A binary clause is a rule with a single goal. Queries posed to a binary program consist of a single goal.

```
member(X,[X|_]) ←          true.
    true.
member(X,[_|Xs]) ←
    member(X,Xs).
```

Figure 5.1: A naïve binary program

```
mi_maher([],[]).
mi_maher([G|Gs],[]) ←
    mi_maher([],[G|Gs]).
mi_maher(Hs,[G|Gs]) ←
    mi_clause(G,Hs0,Hs),
    mi_maher(Hs0,Gs).
```

Figure 5.2: Binarizable breadth first meta-interpreter

The addition of the unit clause corresponds to the end case in the meta-interpreter mi_dllist/1 that uses difference lists in Section 2.2. In a practical implementation, the predicate true/0 may be omitted because we usually pose queries within some top-level shell. When true/0 is encountered, the system would stop completely. In Fig. 5.1, a simple valid binary program is given. The program is, however, not very useful, because we are only able to pose queries on this single predicate. After true/0 has been encountered, i.e., after a goal member/2 has been proved, no further computation can take place. We cannot reuse this predicate in another more complex situation.

This small example shows what binary programs are able to represent directly: unification, direct recursion, backtracking (the OR-tree). AND-trees (conjunctions) are not directly supported. The problem of mapping definite programs into binary form comes down to the problem how AND-trees are represented.

5.1 Binarization: from definite programs to binary form

The implementation of definite programs by means of binary definite programs is usually motivated by a more or less *ad hoc* transformation. Binary definite programs have in contrast to ordinary definite programs no equivalent to the selection rule for AND-control. This means that binary definite programs must reify the computation rule. There is therefore also the danger that translations to binary definite programs, are not equivalent to the original programs, in that equivalence with respect to SLD-resolution is not preserved. In that sense, binary definite programs do not have much choice for their control regime. However, since we are only interested in Prolog-semantics, we can ignore these shortcomings.

Meta-interpreter based implementation. Another way to implement definite programs in binary programs is to implement a meta-interpreter in binary form. Let us reconsider the meta-interpreters given in Chapter 2.1. All meta-interpreters given contain a clause with at least two goals. By using partial evaluation for the given data, we are able to 'unmask' the meta-interpreters that are of interest. The meta-interpreter working with difference lists can be easily transformed into binary form.

Maher's transformation. Maher [Mah88] presents a transformation giving a stratifiable binary program that uses two implicit goal stacks. He proposes a translation scheme which corresponds to a fair selection strategy for atoms. His transformation preserves operational semantics but does neither ensure equivalences based on the logic semantics nor Prolog's semantic. His method

```
inf ←                          bin_inf(Cont) ←
    inf.                           bin_inf(Cont).

p ←                            bin_p(Cont) ←
    inf,                           bin_inf(fail(Cont)).
    fail.
                               % ← bin_p(true).
% ← p.
```

Figure 5.3: bin_p/1 does not possess a finite failure SLD-tree

contains two 'goal-stacks' that implement a breadth-first control for the AND-control. Instead of discussing his transformation scheme in detail we present in Fig. 5.2 his transformation scheme in the form of a meta-interpreter that can be transformed into a binary program.

Binarization. Binary programs enforce that SLD-resolution will always be performed with the goal selection that has been reified. In some sense, Prolog's selection rule of the leftmost atom is compiled into the continuation. The declarative semantics with respect to SLD-trees is therefore changed. A simple example showing that semantics is not preserved is given in Fig. 5.3. However, because we are interested only in Prolog-execution, this observation can be ignored.

A definite program is binarized by translating every clause as follows [Dem92,TB90]. Every predicate p/n is transformed into a new predicate $p/n+1$. The new argument represents the continuation. Facts are transformed into rules calling the continuation. Rules are transformed into binary rules. The goal in the binary rules is the first goal of the original rule. Subsequent goals of the original rule are put into the continuation. For example:

```
perm([],[]).                   del(X,[X|Xs],Xs).
perm([X|Xs],[Y|Zs]) ←          del(X,[Y|Xs],[Y|Ys]) ←
    del(Y,[X|Xs],Ys),              del(X,Xs,Ys).
    perm(Ys,Zs).
```

is translated into

```
perm([],[],Cont) ←                      del(X,[X|Xs],Xs,Cont) ←
    Cont.                                   Cont.
perm([X|Xs],[Y|Zs],Cont) ←              del(X,[Y|Xs],[Y|Ys],Cont) ←
    del(Y,[X|Xs],Ys,perm(Ys,Zs,Cont)).      del(X,Xs,Ys,Cont).
```

CPS-Conversion by Sato and Tamaki. The starting point of the CPS-conversion [ST89] for existential continuation (AND-continuations) is a definition of mode patterns for every predicate called a predicate partition π. The definition of the mode patterns is given on the level of arguments distinguishing input and output arguments. For every predicate a new predicate is introduced with an additional argument for passing the continuation. Input and output arguments are treated differently in the newly defined predicates. Input arguments are directly passed to the original predicate, while output arguments are passed to another new predicate that handles the continuation. With these initial definitions given, Sato and Tamaki derive a binary program. Since output arguments are not directly passed to a goal, the goal may be specialized.

In contrast to Tarau's method, the structure of the programs may be changed significantly. Tarau reports in [TB90] that binary programs obtained by his simpler transformation are more efficient in practice than binary definite programs obtained by CPS-conversion which uses more sophisticated fold/unfold techniques. He remarks in particular that clause indexing for the first argument of the original program is not preserved which is one of the most important optimizations in Prolog systems. For permutation/2 given by Sato and Tamaky he reports a 30%-40% speedup

in comparison to the corresponding program obtained by CPS-conversion. We have experienced that the programs that result from Sato and Tamki's CPS-conversion have to be simplified by unfolding trivial definition of clauses in order to obtain acceptable performance.

But there is still another reason, why Sato and Tamaki's work is of limited interest for a binary Prolog implementation: CPS-conversion does not preserve Prolog-equivalence. While Sato and Tamaki insist that *our mode pattern has nothing to do with instantiation pattern of arguments at runtime* and that a clause head may be invoked even with a different pattern at runtime, their method does not preserve Prolog-equivalence. If the instantiations at runtime do not comply to the given patterns we are able to encounter infinite failure branches instead of finite failure in the original program. The first step in their transformation yields already clauses that are no longer Prolog-equivalent to the original program. A minimal example is the predicate inf_loop/2 below. Clearly, inf_loop/2 loops if the second argument is free. The conversion into the existential continuation form with the mode pattern inf_loop(+,-) yields a new predicate where the second argument is no more directly connected to the predicate.

```
inf_loop(A,s(N)) ←          inf_loop_1(A,Cont) ←
    inf_loop(A,N).              inf_loop(A,B),
                                cont_inf(B,Cont).
```

The same argument applies to the permutation program presented in [ST89]. A goal like ?-perm(_,const). will, e.g., cause an infinite loop while the original program fails. In order to use CPS-conversion when Prolog-equivalence is desired, we have to resort to some data-flow analysis in order to receive correct results.

5.2 Implementation aspects

In the report presenting the Warren Abstract Machine [War83], Chapter 10, David Warren compares two different techniques for implementing the resolvent. The usual structure sharing technique (environment stacking) is compared to another technique (goal stacking). At the moment there are at least three different Prolog systems that use this or a similar environment stack free technique: Prolog by BIM, Prolog-Mali and as the most recent BIN-Prolog.

An implementation of binary Prolog can be seen as a simplified Warren Abstract Machine with a split stack model where the environment stack has been abandoned completely. Compared to the usual implementation technique in the WAM, there is a considerable overhead in space consumption. In particular, a continuation (represented by a Prolog-term) has to be created for every subgoal except for the first. Furthermore, goals are usually sharing some data. In most implementations, a variable needed after the first goal is represented once in the environment. In binary notation, we have to represent that variable for every subgoal it occurs in. Nevertheless, there are several advantages of binary Prolog beside the simpler implementation techniques. Especially, calling continuations is faster due to several reasons.

- In the ordinary WAM based model, the continuation is represented as a pointer to the continuing goal code. When a continuation is fetched, one has to fetch first the goal code which consists of several initialization instructions for the argument registers. The next goal is called thereafter. In a binary WAM, the continuation is a pointer to a structure which represents directly the next goal. Therefore there is no dereferencing.

- When calling the continuation in binary WAM (i.e., when executing a goal after some first goal), the initialization code for the argument registers depends only on a predicate's arity. It is straight forward to implement this code once for all predicates with a generic routine (e.g., '$demo'/1 in BIN-Prolog). In an emulated system this will result in less instruction

decoding, while a compiled system profits at least from the code space reduction and the increased memory locality.

- As long as there are only two goals in a clause, the size of the continuation equals the size of the newly allocated environment in the WAM.
- No deallocate instruction has to be executed in the binary version. Note that this corresponds to the 'proof' of Appel [App87] that a heap based memory allocation scheme with a garbage collector will be always faster (with respect to instructions executed, neglecting actual memory locality) than a stack based scheme provided that we are able to use a sufficiently large size of memory: The heap based scheme does not perform any pop-operation to trim the stack. As long as the pop-operation requires additional execution time, we are always able to find a sufficiently large memory size to make the heap based scheme faster.
- As long as all arguments starting from the second goal in a clause are distinct, the space overheads are only present for clauses with more than two goals: If we have $n, n > 2$, goals in a clause, we have $2(n-2)$ additional cells to write. This overhead is due to the further continuations written in advance.

A compiled binary program will be slightly more compact than a corresponding WAM-program. The memory consumption is in this case equal, if we ignore stack related optimizations.

```
u(X0,X) ←                u(X0,X,Cont) ←
    v(X0,X1),                v(X0,X1,w(X1,X,Cont)).
    w(X1,X).
```

BINWAM-code	*WAM-code*
put_structure w/3,A4	allocate
set_variable A5	get_variable Y1,A2
set_value A2	put_variable Y2,A2
set_value A3	call v/2,2
put_value A5,A2 *	put_value Y2,A1
put_value A4,A3 *	put_value Y1,A2
execute v/3	deallocate
	execute w/3

Note that the call instruction in WAM-code implicitly contains a further memory move (continuation pointer) and that at the entry of the next clause we have to set up the correct environment.

The continuations in binary Prolog will be considerably larger if the goals starting from the second goal contain identical arguments. A special case of such arguments are unnecessary variables. But also identical constants cause a similar overhead:

```
p(X) ←
    c(X),
    draw(1,1).
```

In this case no environment has to be allocated in the WAM while the binary WAM has to allocate space for both arguments of draw.

In a compiled binary Prolog the initialization code for continuations is a single basic block since no branches occur which is favorable on typical modern processors. The usual WAM on the other hand has split the initialization code before every goal. See [MD91] for head unification and [Mar88] for goal unification. Our observations show that although binary Prolog has some space overheads most overheads are avoidable by simple source-to-source transformations. The arguments of a binary program can be implemented with registers. This follows immediately from the implementation model of the WAM [War83].

Chapter 6

Equality-Based Continuation Transformation

Equality based continuation transformations (EBC-transformations) are a new technique for removing redundancy in functors and arguments of predicates. In particular unnecessary variables are removed that cannot be removed with the methods known from literature. The removal of these unnecessary variables allows us to fold more goals. The removal of unnecessary variables also reduces the dynamic space consumption of programs. EBC-transformations preserve a program's original structure. They only transform arguments and function symbols in predicates. The overall structure, the number of clauses, and the number and the sequence of goals is left untouched. However, in a further pass, some auxiliary predicates (consisting of simple definition rules which only depend on the number of predicates and not on a program's size) may be introduced.

Since Tarau's simple transformation scheme already yields encouraging results, we will take it as the basis for a more complex transformation scheme. In 3.3 we presented a program whose unnecessary variable cannot be reduced with traditional techniques.

6.1 Transformation rules

The basic idea of EBC-transformations is to introduce new function symbols and connect them to the existing symbols with an equality relation $=_E$. There are several approaches to deal with the extended unification algorithm:

- Extend syntactic unification to support the new equality relation. With meta-structures [Neu90b] resp. attributed variables [Hol92] we are able to implement the new equality relations.
- Allow new function symbols only if all occurrences can be reduced to syntactic unification.

The two approaches serve different purposes. When using a new extended unification algorithm, we are able to introduce arbitrary equations that do not affect the behavior of the original terms. Old terms and their new corresponding representations may be mixed arbitrarily. Our transformations will be applicable for any program. The second approach is not always applicable. In the sequel, we will define when it is possible to reduce the extended unification algorithm to pure syntactic unification. Currently, we are preferring the second approach since it is implementable in any Prolog without overheads. Further, we are primarily interested in optimizing continuations in binary programs.

6.1.1 Localizing the continuation

Only parts of a program that contain no full syntactic unification for their terms are amenable to EBC-transformations. We call these parts continuations (more precise: continuation-like), although they do not need to be the result of binarization. These parts are derived for a program as follows:

Continuation: A term t is a continuation if t occurs in a continuation argument and

- t is a continuation variable or
- $t = f(t_1, \ldots, t_n), n > 0$, is a continuation functor with continuations in all continuation arguments or
- $t = \text{true}/0$

Continuation variable: A continuation variable is a variable that occurs only as a subcontinuation. A continuation variable occurs exactly once in the head and at least once in the body of a clause. This restriction guarantees that full syntactic unification is never needed.

Continuation predicate: A predicate p/n that has at least one continuation argument is a continuation predicate.

Continuation functor: A functor f/n that has at least one continuation argument is a continuation functor.

Continuation argument: A continuation argument is an argument of a continuation predicate or continuation functor. It must contain a continuation.

Subcontinuation: A continuation that is a subterm of a continuation argument in head or goal of a continuation predicate.

Continuation prefix: For a given continuation, a continuation prefix is the continuation with some continuation argument replaced by a new continuation variable.

Continuation part: For a given continuation, the continuation prefix of a subcontinuation is a continuation part.

Matching continuation: Is a continuation in the head that contains free variables in all non-continuation arguments. A continuation ensures that only matching is needed for unification.

The properties above guarantee that a continuation is always a rigid term at runtime. I.e. the size of a continuation defined as the number of its subcontinuations remains invariant under substitution. Since a query is a clause with an empty head, a query must not contain a continuation variable. A continuation functor may also be used in the other parts of the program, only occurrences as subcontinuations are of interest for the following transformations.

A binary program P_{bin} obtained by Tarau's binarization has at least the following continuation part: For every predicate $p(a_1, \ldots, a_n)$ the argument a_n is a continuation argument. Every functor $f(a_1, \ldots, a_n)$ in P_{bin} with $f/n = p/n$, is a continuation functor with continuation argument a_n. Many programming schemes like the make/next/done-interface use 'hidden continuations' (e.g., lists used as stacks for traversals etc.).

6.1.2 Introduction of new function symbols

A set of equations E is introduced for a given program P containing equations e of the form $t_{old} \doteq t_{new}$ where

- t_{old} is a term constructed of variables and function symbols in P and t_{new} is a term with, maybe, new function symbols that do not occur in P.
- Every variable in e has to occur at least twice.
- $\mathrm{VAR}(t_{new}) \subseteq \mathrm{VAR}(t_{old})$. I.e., new variables must not be introduced in the new term.
- Continuation variables have to occur once in t_{old} and once in t_{new}.
- The set of equations E must ensure that for two different terms $t_1 \neq t_2$ the inequality $t_1 \neq_E t_2$ holds. For our purposes, it is sufficient that every t_{new} contains as subterm a new functor f/n that does not occur in other equations.

  Examples:
  f(X,f(X,Cont)) $\doteq$ nf(Cont)
  expr(T,X0,X,Cont) $\doteq$ i(X0,expr(T,o(X,Cont)))

6.1.3 Compiling function symbols into the program

We are using the equations e as rewrite rules to replace every occurrence of old terms t_{old} by new terms t_{new} in all goals of P. For all clauses that contain occurrences of t_{old} in the head, new clauses are generated containing all possible substitutions. There are very few occurrences of non-variable continuations in the head. In programs that are the result of binarization, there is only one place where non-variable continuations occur in the head: the \$demo/1-predicate. If all occurrences of t_{old} and all occurrences of its (non-variable) subterms were replaced successfully in the goals, the corresponding equation can be deleted from E. Although new clauses are generated as alternatives, no additional nondeterminism is obtained at runtime.

6.1.4 Removal of redundant matching subcontinuations

In ordinary binary programs, the continuation argument in the head is always a continuation variable. The introduction of new function symbols may have produced some new subcontinuations. As an example, the equality relation expr(T,X0,X,Cont) $\doteq$ i(X0,expr(T,o(X,Cont))) rewrites the head of the predicate expr/4 yielding the subcontinuations expr(...) and o(...). The introduction of new function symbols ensures that a new continuation in the head is only a matching continuation. The splitting of arguments induced by the equality relation may uncover redundant passings of arguments from the head to the last continuation. A clause like (for some n)

$$\begin{aligned} &\mathrm{p}(\ldots, \ldots g(a\_1, \ldots, a_{n_a}, \mathrm{Cont}) \ldots) \leftarrow \\ &\quad \mathrm{q}(\ldots, \ldots g(a\_1, \ldots, a_{n_a}, \mathrm{Cont}) \ldots). \end{aligned}$$

is transformed (generalized) into

$$\begin{aligned} &\mathrm{p}(\ldots, \ldots \mathrm{NCont} \ldots) \leftarrow \\ &\quad \mathrm{q}(\ldots, \ldots \mathrm{NCont} \ldots). \end{aligned}$$

6.1.5 Generalizing variables in goals

While unnecessary variables are mostly blocking the process of specialization, they have to be introduced if a goal is too specific to be optimized successfully with EBC-transformation. This is the case for predicates that pass a single context argument around. In Fig. 6.1, an example is given. Variables that occur 'too often' in a goal are split into several pairs of unnecessary variables. It must be ensured that the program still remains the same; this is done by investigating all continuations within reach. For every subcontinuation of a goal we must be able to determine the predicate that will read this continuation. Since this is in general undecidable, the class of predicates to be considered is restricted to continuations in the last argument. I.e., only the last argument of predicates and functors are taken as continuation arguments for this optimization.

Continuation association: A given continuation is split into disjoint continuation parts covering all subcontinuations. A continuation association is established if for every continuation part a unique continuation predicate can be found that uses the continuation.

Goal with generalized variables: The renaming of some occurrences of the variable v in a goal yields a goal with generalized variables if at least one occurrence of v is not renamed.

Predicate unification prefix: For a predicate p/n a sequence s of unification goals is a predicate unification prefix if the goal $p(a_1, \ldots, a_n)$ always implies s. By antiunification of all heads of a predicate, a (suboptimal) predicate unification prefix is obtained. The goal "$p(a_1, \ldots, a_n)$" is always equivalent to "$s, p(a_1, \ldots, a_n)$".

Equivalent goal with generalized variables: A generalized goal p_g/n is equivalent (with respect to Prolog semantics) to a goal $p(a_1, \ldots, a_n)$ if for every continuation part d_{a_n} of a_n there is a continuation association $q_1, \ldots, q_k$ and "$s_{q_1} \ldots s_{q_k}, p(a_1, \ldots, a_{n-1}, d_{a_n})$" is equivalent to "$s_{q_1} \ldots s_{q_k}, p(a_1, \ldots, a_{n-1}, a_n)$." A goal with generalized variable is hence equivalent to the original goal if all possible predicate unification prefixes of the corresponding continuation parts yield the same results. I.e., all unifications that will always be executed to reach a subcontinuation are considered.

Applications. If there are multiple occurrences of a single variable in a goal, these variables can be split by transforming the goal into a generalized one, provided that we cannot see any effect at runtime. Goals that are too specific for further optimizations can be generalized. Constant context arguments that pass a single variable through several layers of recursion are the major target of the transformation.

Example. The predicate all_els_gt/2 in Fig. 6.1 is true if all elements of a binary tree are greater than a given value. The variable K in the second argument represents the value to be compared with all nodes. Variable K is an unnecessary variable because it has two occurrences in the body. The techniques for removing unnecessary variables in the fold/unfold framework [PP91] cannot be applied in this predicate: The predicate has a nontrivial recursion and the unnecessary variable is just in between them. In a previous transformation using Tarau's method, the predicate was already transformed into the binary form bin_all_els_gt/2. This form was further transformed by the

```
all_els_gt(nil,_).
all_els_gt(t(E,A,B), K) ←
    E > K,
    all_els_gt(A, K),
    all_els_gt(B, K).

bin_all_els_gt(nil, _, Cont) ←
    Cont.
bin_all_els_gt(t(E,A,B), K, Cont) ←
    E > K,
    bin_all_els_gt(A, K, bin_all_els_gt(B,K,Cont)).

i(K, bin_all_els_gt(nil,o(K,Cont))) ←
    Cont.
i(K, bin_all_els_gt(t(E,A,B),o(K,Cont))) ←
    E > K,
    i(K, bin_all_els_gt(A,o(K,i(K,bin_all_els_gt(B,o(K,Cont)))))).
```

Figure 6.1: A predicate with a constant context argument

```
i(K,bin_all_els_gt(nil,o(K,Cont))) ←
    Cont.
i(K,bin_all_els_gt(t(E,A,B),K_Cont)) ←
    E > K,
    i(K, bin_all_els_gt(A,o(K,i(K,bin_all_els_gt(B,K_Cont))))).

i(K,bin_all_els_gt(nil,o(K,Cont))) ←
    Cont.
i(K,bin_all_els_gt(t(E,A,B),K_Cont)) ←
    E > K,
    i(K, bin_all_els_gt(A,o(KNew,i(KNew,bin_all_els_gt(B,K_Cont))))).
```

Figure 6.2: Generalization of a continuation part and variables

equality relation bin_all_els_gt(T,K,Cont) ≐ i(K,bin_all_els_gt(T,o(K,Cont))). The primitive built-in predicate >/2 is not written in binary notation to simplify readability. The rule is generalized by removing the redundant continuation o(K,Cont) in Fig. 6.2. The variables in the goal are generalized. The new predicate now contains a continuation part in the rule that can be further transformed by introducing a new equality relation. E.g., o(K,i(K,Cont)) ≐ oi(Cont).

6.2 Transformation strategies

We have identified the following situations where EBC-transformation is useful:

1. Accumulator pairs. As they occur in e.g., Definite Clause Grammars. According to the lifetime continuations containing accumulator pairs are split into three nested continuations: An external or input part, the remaining continuation, and the output part as the innermost continuation. The names of the functors of the external and the internal continuations are for all continuations the same. The remaining continuation still preserves the original name.

   A new functor is defined to collapse neighbouring continuations: The output continuations are combined with the input continuations. The newly defined continuation functor possesses only a single argument for the continuation itself. E.g.: output(A,B,input(A,B,Cont)) ≐ f(Cont)

2. Context Arguments. Our current approach is to transform context arguments into accumulator pairs. The context argument is split into two shared variables. By generalizing the variables we are able to separate the shared variables. The further steps are the same as for accumulator pairs.

3. Nesting of function symbols. After the preceding transformations, redundant nestings are removed by introducing new function symbols.

6.3 Efficiency evaluation

The initial motivation for the development of EBC-transformations came from problems in program transformation. Unnecessary variables are removed with their help in order to make folding of clauses possible. During the development of EBC-transformation, we realized that they may also be an interesting optimization technique since they are reducing the size of environment frames and continuations. In this section, we will report on some experiments to quantify the effect of EBC-transformation to the WAM and the binary WAM. The programs considered are parts of benchmarks where our optimization is applicable. They were not written with EBC-transformation in mind. Programming techniques that explicitly exploit the properties of EBC-transformations yield better speedups.

It is difficult to estimate the effective space consumptions of languages with implicit dynamic allocation and implicit storage management. Very little work is known from the literature. Appel as a notable exception devotes a whole chapter (Chapter 12 [App92]) to the impact of optimizations to space complexity. Yet, many implementation aspects blur a rigorous analysis: sharing of terms that are identical for the system, reference counting schemes etc. Since we want to compare our transformed programs only with the traditional implementations on WAM-models, and since EBC-transformations do not perform 'deep transformations' (I.e., the residuals resemble algorithmically so much the original programs that it is difficult to call them different algorithms) we use the following measure. The size of cells produced per inference and the size of cells still alive. Ideally, after every memory operation the garbage collector should be activated to get a precise result.

The measurements were performed on a SPARCstation ELC, (CPU 33MHz Cypress, 8Mb RAM). We used BIN-Prolog Version 1.39 with the default parameters for memory configuration and Sicstus-Prolog 2.1 with the flag compiling set to fastcode. All measurements were performed in failure-driven loops.

Notation for program versions. We compared the following versions of programs: 'DS': The original programs (written in *direct style*). 'EBC': EBC-transformation applied. Since there are also other program transformation techniques, in particular, folding of goals in the body we compared also these transformations (DS^f) with the corresponding EBC-versions (EBC^f). Programs that allowed further fold/unfold-transformations are denoted by 'DS^x' and (EBC^x)

Because BIN-Prolog supports a special built in meta-call '$demo'/1, we have compared predicates against the direct implementations (DS) using this built-in call as well as against versions that emulate the built in call via simple linking predicates (DS_n). The effective differences where quite considerable (more than 10%) and are probably due to the fact that the emulated version has to decode some WAM-instructions for linking a continuation to the actual predicate, while a direct implementation can load continuations directly into the WAM-registers. The linking predicates simulating the meta-call consist of at least three indexable clauses in order to prevent a special optimization of BIN-Prolog for two clauses. In larger programs, this optimization is not applicable for implementing the emulated meta-call. For EBC-versions we attempted to simulate the corresponding speedup gained by the speedup due to '$demo'/1 in direct style versions by an optimized linking predicate that avoids some of the indexing overheads (EBD_p).

	V_{DS}	$V2_{\mathrm{DS}}$	V_{DS^f}	V_{EBC}	V_{EBC^f}
t[s]	18 762	21 357	14 254	12 830	11 229
V_{DS}	0%	13%	-25%	-32%	-41%
$V2_{\mathrm{DS}}$	-13%	0%	-34%	-40%	-48%
V_{DS^f}	31%	49%	0%	-10%	-22%
V_{EBC}	46%	66%	11%	0%	-13%
V_{EBC^f}	67%	90%	26%	14%	0%

Figure 6.3: adcg//2 on BinProlog

	V_{DS}	$V2_{\mathrm{DS}}$	V_{DS^f}	V_{DCG}	V_{EBC}	V_{EBC^f}	V_{EBC_p}	V_{EBC^x}
t[s]	3 848	4 570	3 818	3 589	3 127	2 884	2 805	2 754
V_{DS}	0%	18%	-1%	-7%	-19%	-26%	-28%	-29%
$V2_{\mathrm{DS}}$	-16%	0%	-17%	-22%	-32%	-37%	-39%	-40%
V_{DS^f}	0%	19%	0%	-7%	-19%	-25%	-27%	-28%
V_{DCG}	7%	27%	6%	0%	-13%	-20%	-22%	-24%
V_{EBC}	23%	46%	22%	14%	0%	-8%	-11%	-12%
V_{EBC^f}	33%	58%	32%	24%	8%	0%	-3%	-5%
V_{EBC_p}	37%	62%	36%	27%	11%	2%	0%	-2%
V_{EBC^x}	39%	65%	38%	30%	13%	4%	1%	0%

Figure 6.4: adcg//2 on SICStus

6.3.1 A simple DCG

The following definite clause grammar was taken to estimate the benefits that we can expect from EBC-transformations. Definite Clause Grammars are a particularly well suited candidate because they hide a difference list that can never be accessed directly by the programmer.

```
adcg(tok(T),tok(T)) ⟶
    [].
adcg(prefix,tree(A,B)) ⟶
    [X],
    adcg(X,A),
    [Y],
    adcg(Y,B).
```

The first 2 551 trees that can be generated from the grammar were used as data for the benchmark. The benchmark consisted in parsing every list 10 000 times. In Fig. 6.3 the results for BIN-Prolog are presented. The program corresponding to the traditional DCG-translation (V_{DS}) was taken and compared against the EBC-binarized (V_{EBC}) versions. Note that we did not use the translator for DCGs integrated in BinProlog since BinProlog does not translate into list-unification directly. An additional inference would have been needed. The versions of V_{EBC} and V_{DS} were directly compared, $V2_{\mathrm{DS}}$ has an additional argument pair to pass further. For the EBC-binary versions no measurable difference could be found by adding new arguments. Every version was also further folded to minimize the continuation. In Fig. 6.4 the results for compiled SICStus are given. Additional measurements were performed with V_{DCG}, the built-in translation of grammar-rules, and V_{EBC^x}, a handwritten EBC-program that uses a list to represent the continuation.

6.3.2 Predicate numbered/2

There are many programs that contain unnecessary variables which are not eliminable by the usual transformation technique. As an example, consider the predicate numbered/2 in the standard benchmark serialize/2.

```
numbered(tree(T1,pair(_,N1),T2),N0,N) ←
    numbered(T1,N0,N1),
    N2 is N1+1,
    numbered(T2,N2,N).
numbered(void,N,N).
```

The purpose of numbered/2 is to attach to each node pair/2 a unique number in the second argument. In a procedural language, the variables N* could be represented by a variable global to the recursive procedure. The variables holding the numbers are not a typical example of the technique of passing arguments: The variable N1 which is in the middle of the chain has an occurence in the head. This implies that we will not be able to remove this variable completely. In numbered/2, the final value of the argument depends always on the second argument. First, the predicate is transformed into its binary equivalent:

```
numbered(tree(T1,pair(_,N1),T2),N0,N,Cont) ←
    numbered(T1,N0,N1,+(N1,1,N2,numbered(T2,N2,N,Cont))).
numbered(void,N,N,Cont) ←
    call(Cont).
```

The binary form is rewritten by splitting the functor into an external part e/2 and an interal part i/2 and into a part in the middle that is not affected by the transformation. The external and the internal part correspond to the input output pattern that will occur at runtime. The arguments in the middle can be both.

```
numbered(T,N0,N,Cont) ≐ e(N0,numbered(T,i(N,Cont))).
+(N0,K,N,C) ≐ e(N0,+(N0,K,i(N,C))).

numbered(T,N0,N,Cont) ←
    e(N0,numbered(T,i(N,Cont))).

e(N0,numbered(tree(T1,pair(_,N1),T2),i(N,Cont))) ←
    e(N0,numbered(T1,i(N1,e(N1,+(N1,1,i(N2,e(N2,numbered(T2,i(N,Cont)))))))))).
e(N,numbered(void,i(N,Cont))) ←
    call(Cont).

e(N0,+(N0,K,i(N,C))) ←
    +(N0,K,N,C).
```

We are able to generalize the continuation. The innermost structure i(N,Cont) is not needed within the rule.

```
e(N0,numbered(tree(T1,pair(_,N1),T2),NCont)) ←
    e(N0,numbered(T1,i(N1,e(N1,+(N1,1,i(N2,e(N2,numbered(T2,NCont))))))))).
e(N,numbered(void,i(N,Cont))) ←
    call(Cont).
```

Structures are folded that share a common variable, which must not occur somewhere else.

V_{DS}	compiled by BIN-Prolog (clause wise compilation, no optimizations)
V_{DS^f}	fold of the last two goals (+/3 and numbered/3)
V_{DS^x}	unfolding to propagate addition
$V_{\mathrm{DS_n}}$	with emulated meta-call
$V_{\mathrm{DS}^f_{\mathrm{n}}}$	with emulated meta-call
$V_{\mathrm{DS}^x_{\mathrm{n}}}$	with emulated meta-call
V_{EBC}	direct clause wise compilation
V_{EBC_p}	optimizing one emulated meta call
V_{EBC^f}	defining a new predicate for the last two goals (similar to V_{DS^f})
V_{EBC^x}	similar to V_{DS^x}

Figure 6.5: Versions for benchmark serialize/2

```
i(N,e(N,Cont)) ≐ ie(Cont).

e(N0,numbered(tree(T1,pair(_,N1),T2),NCont)) ←
    e(N0,numbered(T1,ie(+(N1,1,ie(numbered(T2,NCont)))))).
e(N,numbered(void,i(N,Cont))) ←
    call(Cont).
e(N,numbered(void,ie(Cont))) ←
    e(N,Cont).

e(N0,+(N0,K,i(N,C))) ←
    +(N0,K,N,C).
e(N0,+(N0,K,ie(C))) ←
    +(N0,K,N,C),
    e(N,C).


plus_ie(N,K,Cont) ≐ +(N,K,ie(Cont)).
e(N,numbered(T,Cont)) ≐ e_numbered(N,T,Cont).

e_numbered(N0,tree(T1,pair(_,N1),T2),NCont) ←
    e_numbered(N0,T1,plus_ie(N1,1,numbered(T2,NCont))).
e_numbered(N,void,NCont) ←
    e(N,NCont).

e(N,i(N,Cont)) ←
    $demo(Cont).
e(N0,plus_ie(K,N0,C)) ←
    +(N0,K,N,C),
    e(N,C).
e(N,numbered(T,NCont)) ←
    e_numbered(N,T,NCont).
```

Due to the rewrite process, we are no more creating the functor +/3. Clauses containing it in the head can therefore be removed (marked with asterisk). Further simplifications can also be done in the usual context of fold/unfold transformations.

```
e_numbered(T,N,Cont) ←
    e(N,numbered(T,Cont)).
```

By defining a more compact continuation (f and x versions) due to definition, both compilation schemes can be improved. The EBC-versions are consuming always less space. If the sizes of continuations do not differ a lot, other aspects (indexing, the number of emulated instructions executed, different trailings) are more important. However, even in this case the difference ($V_{\mathrm{DS}^x_{\mathrm{n}}}$

Predicate		numbered/3				serialize/3			
			speedup w.r.t.				speedup w.r.t.		
Version	c/s	t [ms]	V_{DS}	$V_{\mathrm{DS_n}}$	V_{-}	t [ms]	V_{DS}	$V_{\mathrm{DS_n}}$	V_{-}
V_{DS}	9/2	3 720	0.0%	13.2%	—	49 490	0.0%	1.2%	—
V_{DS^f}	5/1	2 900	28.3%	45.2%	28.3%	48 670	1.7%	2.9%	1.7%
V_{DS^x}	5/1	2 760	34.8%	52.5%	5.1%	48 620	1.8%	3.0%	0.1%
$V_{\mathrm{DS_n}}$	9/2	4 210	-11.6%	0.0%	—	50 070	-1.2%	0.0%	—
$V_{\mathrm{DS_n}^f}$	5/1	2 990	24.4%	40.8%	40.8%	48 860	1.3%	2.5%	2.5%
$V_{\mathrm{DS_n}^x}$	5/1	2 870	29.6%	46.7%	4.2%	48 690	1.6%	2.8%	0.6%
V_{EBC}	6/2	3 440	8.1%	22.4%	—	49 220	0.6%	1.73%	—
V_{EBC_p}	6/2	3 390	9.7%	24.2%	1.5%	49 140	0.7%	1.89%	0.2%
V_{EBC^f}	4/1	2 960	25.7%	42.2%	14.5%	48 700	1.6%	2.81%	0.9%
V_{EBC^x}	4/1	3 010	23.6%	39.9%	-1.7%	48 740	1.5%	2.73%	-0.1%

Figure 6.6: numbered/3 and serialize/2

	space	$\mathrm{std_{DS}}$	$\mathrm{std_{EBC}}$	$\mathrm{std_{EBC^f}}$	$\mathrm{dcg_{DS}}$	$\mathrm{dcg_{EBC}}$	$\mathrm{dcg_{EBC^f}}$
t[ms]	—	6 950	6 960	6 720	7 740	6 980	6 750
$\mathrm{std_{DS}}$	11/3	0%	0%	-4%	11%	0%	-3%
$\mathrm{std_{EBC}}$	8/3	-1%	0%	-4%	11%	0%	-4%
$\mathrm{std_{EBC^f}}$	7/2	3%	3%	0%	15%	3%	0%
$\mathrm{dcg_{DS}}$	13/3	-11%	-11%	-14%	0%	-10%	-13%
$\mathrm{dcg_{EBC}}$	8/3	-1%	-1%	-4%	10%	0%	-4%
$\mathrm{dcg_{EBC^f}}$	7/2	2%	3%	-1%	14%	3%	0%

Figure 6.7: qsort/2 for BinProlog

is 3% faster, than V_{EBC^f}) is neglectable and might be due to the noise caused by the indexing mechanism which relies on a hash function not described in detail [Tar92a]. The reduced space consumption which causes better memory locality will outweight at least the differences in larger programs.

6.3.3 Predicate qsort/2

qsort/2 is a benchmark by D.H.D. Warren sorting a list of 50 given integers. The sort predicate uses difference lists for construction of the list. The resulting list is written in a rather unnatural manner: First, the end of the list is computed and then the beginning. In this manner no trailing operations are necessary since only new lists are appended at the head of the list and not within the list.

```
qsort([X|L],R0,R) ←
    partition(L,X,L1,L2),
    qsort(L2,R1,R),
    qsort(L1,R0,[X|R1]).
qsort([],R,R).
```

Another, in our opinion more natural way uses DCGs. The DCG notation cannot express the original qsort/2 because difference lists are treated in the usual way. I.e., the two goals must be exchanged.

```
qsort([E|Es]) ⟶
    {partition(Es,E,L1,L2)},
    qsort(L1),
    [E],
    qsort(L2).
qsort([]) ⟶
    [].
```

Our results are presented in Fig. 6.7. The times were measured by executing qsort/3 for 1000 times. The standard versions (std) and DCG-versions (dcg) are compared. The binarized versions are obtained by applying EBC-transformation only to the predicate qsort/3. Including the predicate partition/4 into the transformation yielded unfavorable results, because clause indexing is not implemented in BinProlog for arithmetical comparison. BinProlog created larger choice points due to the increased number of argument registers. The single list element [E] was merged with the continuation qsort/3 partition in a different version ($_f$). The original version was also folded but without any measurable success. The standard version (std) is 11% faster than the version using a DCG. However, the difference between the binarized and optimized binarized versions are neglectable.

6.3.4 A little compiler

In [BF92], an implementation of a restricted subset of a procedural language is given in Prolog. The implementation consists of a tokenizer scan/2 (which for the sake of simplicity reads from a given list of characters instead from the input stream), a parser generating a checked AST compile/2 and an interpreter working on the AST. Usually, one would apply partial evaluation on such a program. A given program will then be compiled into a program faster than the interpreter. However, partial evaluation will only be of interest during the last 'pass' — the interpreter. The other passes will have to be executed anyway. Executing them with a partial evaluator would slow down the tokenization and the generation of the AST. In order to test our method for binarization, all three passes were optimized. The resulting programs are completely procedurally equivalent to the original programs. Furthermore, we mapped each clause to an equivalent clause; no general fold/unfold optimizations were performed. Further speedup would be possible. However, we are showing only these very local optimizations, since they are in our opinion more practicable.

Predicate	t_{DS}[ms]	t_{EBC}[ms]	speedup	s_{DS}	s_{EBC}	saving
scan/2	263 490	174 720	51%	8 400	7 968	5%
compile/2	32 800	18 900	74%	1 588	1 144	28%
interpret/1	869 130	787 630	10%	516 364	544 388	-5%

The typical compiler passes yielded the biggest speedup. Even if we would have unfolded this very simple tokenizer, there is still an advantage of using the transformed binary program because the number of arguments is reduced by one. Therefore the size of the choice-points is reduced by one argument register. The compiler yielded the biggest speedup, because it was possible to propagate another argument containing the declared identifiers. For the largest part of the token-stream (parsing the statements after the declarations), the continuations are reduced by 3 cells (containing the two lists for the DCG and the 'global' list of declared identifiers). In compensation, the predicates searching for declared entries stalled the direct propagation of registers and caused the creation of additional continuations. Within the interpreter, the propagation of the actual binding environment was optimized. In comparison to the compiler, there are much more calls to predicates stalling the register propagation. Namely, the lookup predicate and the predicate that changes the current binding environment. Therefore the optimization nearly annihilates the optimizations.

```
a(X,Cont) ←                    ...
    det(X,rest(Cont)).         det(...,Cont) ←
                                   $demo(Cont).
                               ...
```

Figure 6.8: Usual way of defining a leaf-predicate

Summary

From our experiences so far, EBC-transformations seem worth to be incorporated into a binary Prolog system. For an ordinary stack based WAM-implementation only a compiled system may take advantage of our optimization.

6.4 Further optimizations

6.4.1 Forced propagation of registers

Predicates that are determinate but that have no redundant arguments to be propagated further on are stalling our optimization. As a typical example the partition/5-predicate in qsort/2 above, interrupts the propagation of values in registers.

6.4.2 Leaf predicates

In a binary Prolog system, continuations are written if the body contains more than two user defined goals. In a clause "a(X) ← det(X), rest" the goal det/1 is a determinate predicate, maybe some facts that are used for determinate table lookup. In any case this is a predicate that does not create any choice points. Still we have to write the continuation (Fig. 6.8). The usual approach is to define a new predicate and unfold it accordingly. In the case of det/1 being a (let us assume big) table, the whole table is specialized, duplicating its memory consumption. Instead of calling the generic $demo/1, the specific rest/1 is called directly in the bodies of the facts. In general, this technique is not very practical, especially for tables and other very large predicates.

Leaf predicate: A leaf predicate p/n is a predicate that does not write new continuations. The continuation in the head must be the same as in the goal. Predicates that are called by p/n must be primitive (built-in) or leaf predicates.

Examples. facts, append/3, member/2. For reasons of efficiency it is preferred that the predicate is determinate.

We propose another approach in avoiding to write the continuation rest/1 and many other continuations: A new predicate det/1+n is defined with n additional arguments. Instead of calling $demo/1 we are calling a similar predicate $demo/1+$n$. All entries of $demo/1 of the form $demo(C) are also present in $demo/1+$n$ in the following form $demo(C,_,_, ...). I.e. the argument registers are simply ignored. New entries in $demo/1 fill more arguments (Fig. 6.9). These optimizations do make sense in binary Prolog as long as the number of arguments is not much larger than the number of used arguments. (The remaining arguments are best initialized with some constant value.) The internals of the implementation may be changed to avoid the unnecessary initialization of the unused argument registers. Furthermore the table $demo/1+$n$ might be shared for different n.

```
a(X,Cont) ←                    ...                    ...
    det(X,rest1,Cont, ...).    det(...,Cont,A1, ...) ←    $demo(rest1,Cont, ...) ←
                                   $demo(Cont,A1, ...).       rest(Cont).
                               ...                    ...
```

Figure 6.9: Avoiding the heap for leaf-predicates

6.5 Related Issues

6.5.1 Sato and Tamaki's CPS-conversion

At first sight, the splitting of a functor (and goal) of the form $f(a_1, \ldots, a_n)$ into the structure of the form $i(i_1, \ldots, i_{n_i}, f(b_1, \ldots, b_{n_b}, o(o_1, \ldots, o_{n_o})))$ with $n_i + n_b + n_o \geq n$ resembles to the input/output partition of Sato and Tamaki [ST89]. There is, however, a crucial difference: Our transformations are introduced by equations over the functors. The introduced equations can never alter the Prolog-semantics of the program. Sato and Tamaki start with defining new predicates which are no longer equivalent. If an argument is 'declared' as output, but is never a variable, and 'drives' the computation, an infinite failure branch can be encountered in CPS-transformed programs. If, on the other hand, we are putting similarly an argument into the output-structure that is in reality an input argument, we will fail to apply further transformations. But it is never the case for our transformation that Prolog-equivalence is questioned. The predicate inf_loop/2 is used to demonstrated this fact.

```
inf_loop(A,s(N)) ←
    inf_loop(A,N).
```

The binary equivalent is:

```
inf_loop(A,s(N),Cont) ←
    inf_loop(A,N,Cont).
```

Using the equation inf_loop(A,N,Cont) $\doteq$ i(A,inf_loop(o(N,Cont))) in the vain hope that the first argument is an input argument and the second an output argument, we obtain:

```
i(A,inf_loop(o(s(N),Cont))) ←
    i(A,inf_loop(o(N,Cont))).
```

This program is still Prolog-equivalent, no continuations can be detected for generalization. We are therefore not able to optimize this program. Conversely, if we are taking the second argument as input, expressed by the following equation we obtain:

```
inf_loop(A,N,Cont) ≐ i(N,inf_loop(o(A,Cont))).

i(s(N),inf_loop(o(A,Cont))) ←
    i(N,inf_loop(o(A,Cont))).
```

which *can* be 'optimized' to

```
i(s(N),inf_loop(ACont)) ←
    i(N,inf_loop(ACont)).
```

and transformed back:

```
inf_loop(A,N,Cont) ←                    i_inf_loop(s(N),Cont) ←
    i_inf_loop(N,inf_loop_o(A,Cont)).       i_inf_loop(N,Cont).
```

The new predicate uses only two argument registers, while the original used three. On the other hand, the program now writes a continuation, while the original program did not. Whether this transformation can be considered as an optimization is not our major point. At least it preserves Prolog-equivalence in contrast to Tamaki and Sato. Guiding heuristics for EBC-transformations are indeed desirable, some dataflow analysis or abstract interpretation is definitely useful. However, EBC-transformations ensure Prolog-equivalence. They do not rely on the correctness of the results obtained by an abstract interpretation but ensure Prolog-equivalence. Simpler, incomplete, or completely heuristical algorithms may be used to guide EBC-transformations.

6.5.2 Calling conventions, interprocedural register allocation

The implementation of Prolog with the WAM-model bears a lot of resemblances to the traditional calling conventions in assembly languages. In the WAM, the argument registers can be seen as scratch registers and argument registers in machine language. The creation of choice-points corresponds to the `longjump()`-primitive. I.e., all relevant registers are saved. Since the WAM has no other registers (for terms to be passed around), the calling conventions in the WAM are very simple: All registers are caller saved, there are no callee saved registers at all. Registers are saved by the caller by putting them into the environment or into a continuation. All registers in the WAM are invalid after returning from a predicate call even if this predicate is very simple. For this reason, the WAM has favorable performance only as long as the registers are propagated further on. I.e., until a second goal in a clause is called. Binary Prolog makes these assumptions even more explicit. Arguments can always be seen as registers. Programming binary Prolog clauses is therefore very close to programming in assembly languages. This view is also shared in the functional language community. As Appel [App92] says: [...] *the CPS language is meant to model the program executed by a von Neumann machine, which likes to do just one thing at a time, with all the arguments to an operation ready-at-hand in registers.*

With the introduction of leaf-predicate optimization, we were able to regain the optimization of leaf-procedures implemented in compilers for RISC-machines. The leaf-predicate optimization can be applied as well for any predicate even if it is not a leaf. In this case, the delayed saving of registers corresponds to callee-saved registers.

Compared to traditional calling conventions for languages like C or PASCAL, the binary-WAM has some more possibilities left: Since in the binary-WAM there is never a return from a procedure, but only a calling to the next, return parameters can be propagated without any new conventions. In traditional calling conventions, we have usually only one or two registers that are dedicated to hold the return values. After a function call in traditional languages, only callee-saved registers and registers holding the return values are valid. More complex return values cannot be passed over registers; global variables accessed and modified by both the caller and callee cannot be propagated further on. Compare this situation to the optimizations that were able with EBC-transformations. During the whole computation all values of temporary interest were held in registers. Traditional compilers need a separate optimization pass (interprocedural register allocation) to obtain similar results to the binary-model.

In summary, the binary-WAM model is able to perform the same optimizations as advanced register allocation algorithms for procedural languages. The implementation of *volatile* data represented in registers can be implemented equally well in the declarative and procedural settings. Nevertheless, we have to make still a concession: The abolition of the stack based regime.

6.5.3 Structure sharing

The equations introduced during EBC-transformation are very similar to the way structure sharing is implemented. In fact, we are able to mimic to some extent the optimizations of structure sharing.

Implementing EBC-transformations with structure sharing. Equations $d \doteq f(a_1, \ldots a_n)$ with

- $\mathrm{VAR}(d) = \mathrm{VAR}(f(a_1, \ldots a_n))$
- $\mid \mathrm{VAR(d)} \mid = n$
- f/n a newly introduced function symbol

have a direct correspondence in a structure sharing implementation: All variables are retained in the new function symbol which corresponds to a molecule. However, some other equations are usually not implemented in structure sharing. E.g.: f(X,X,Cont) $\doteq$ fxx(Cont). In this case, EBC-transformation is able to remove the redundant occurrence of the variable X.

Implementing structure sharing with EBC-transformations. Structure sharing can be simulated by introducing equations that contain only all distinct variables on the right hand side. However, these terms must be used as continuations only; general unification is not possible. If we would like to use general unification as well, we have to extend the system's unification algorithm. Using meta-structures [Neu90b], such could be done.

6.5.4 λ-Prolog

The logic programming language, λ-Prolog [NM88], extends Prolog by polymorphic typing, higher-order programming (via quantification over predicates), λ-term as data structures, unification of simply typed lambda-terms (higher-order unification), and several other features. The general higher-order unification algorithm is rather complex in addition to the fact that the unification problem is semi-decidable. Existing implementations are implementing only some part of Huet's unification algorithm. The representation of terms with λ-terms may reduce the space and time complexity of programs considerably [BR91]. The major difference is that the equations we introduced are always trivially decidable, while in λ-Prolog one has to be aware of the internal implementation since in difficult cases of unification, unification is simply delayed for further instantiations. Furthermore, we are able to map continuations to syntactic unification yielding a very fast implementation that does not interfere with the machine's internals.

6.5.5 Prolog-optimizations on WAM-level

The only way iteration can be expressed in Prolog is by recursion. There are many tail recursive thus iterative schemes of Prolog-programs. This has motivated work on optimizations on the level of the underlying Prolog machine. Micha Meier identifies in [Mei91] such cases of recursive programs. His approach concentrates on optimizations on the WAM-level. Furthermore, he considers the classical WAM which has an additional data area for AND-continuations (called the environment stack). The reuse of complete environments of same size, single values stored already in environments, delayed environments, and choice-points that are no longer needed is considered. Cases are identified, where destructive assignments are possible in environments. To be safe in general, a global analysis of the program is needed. Concerning the reuse of deallocated environments, he speculates: *... however, this might be easier done using source transformations, e.g., partially evaluating the last call.* [...] *The tail call has to be recursive. If a recursion is indirect, partial evaluation can be used to convert it into a direct one.*

The work concentrates on the reuse of memory areas located on stacks. Our work concentrates on propagating the lifetime of registers by separating arguments from their proper continuation. As we have shown the claim — that indirect recursion can be converted into direct recursion — is

true, however, not by classical partial evaluation but by EBC-transformations. Concentrating only on registers and not on addressable memory objects where pointers representing shared variables as well as trail entries may cause dangling references for destructive updates has also the advantage that we do not need to consider determinism for preserving the correctness of EBC-transformations. Our optimization is completely independent of whether the predicates are deterministic or not. Even more, we are able to optimize choice points in passing: Every accumulator pair in the arguments is reduced to a single argument.

Another low-level approach. By using a value trail in an environment based implementation, we could also reduce the size of the environments by different means. The lifetime of variables is analyzed and variables that have disjoint lifetimes are allocated in the same place of the environment. Furthermore, aliasing must not take place. In this case, we would need only two cells in the environment. Overwriting has to be performed by trailing: If a choice point has been created, the old value of the overwritten cell has to be written onto the trail. Therefore, the value is copied on the trail occupying twice the space as it would have needed within the environment.

$$\mathrm{p}(\overbrace{\underbrace{\mathrm{X0},\mathrm{X}}_{\mathrm{X}}\leftarrow \mathrm{q}(\mathrm{X0})}^{\mathrm{X0}},\overbrace{\mathrm{X1}}^{\mathrm{X1}},\mathrm{r}(\mathrm{X1}),\overbrace{\mathrm{X2}}^{\mathrm{X2}},\mathrm{s}(\mathrm{X2},\mathrm{X})).$$

6.5.6 Lexical scoping in functional and procedural languages

Equality based continuation transformations are able to optimize predicates that use some global state. They are able to optimize the propagation of constant arguments (also called context parameters). Here the technique closely resembles to compilation techniques proposed for lexically scoped functional and procedural languages. In both languages (e.g., some functional language or PASCAL), an implementation has to cope with the fact that a function or procedure may have free variables (i.e., variables that are not declared in the innermost scope) that refer to an external scope. Unless these variables are removed in some way, an environment-based implementation has to maintain linked environments, known in procedural languages as displays and more general as windows. Procedural languages are mostly using the relatively simple display techniques, because the usage of function parameters is restricted. Implementations of functional languages have to use the more general mechanism. In some implementation the additional implementation overhead is avoided by using supercombinators that pass the external variables directly into the function. This technique is known as lambda lifting [Joh85].

6.5.7 Relations to attribute grammars

Attribute grammars (AG) were developed by Knuth about 1968. Attribute grammars form an extension of context-free grammars by attaching attributes to the grammar symbols. Attribute values are defined by evaluation rules associated with the ordinary productions of a context-free grammar. The evaluation rules specify how to compute some values of attributes out of other attributes given. Since the language describes how the value of some attributes is determined, attributes are divided into synthesized and inherited attributes. The relation of DCGs and other logic formalisms and attribute grammars is well studied in the literature [Der83,DM85,DM88,Mal91,Rie91,RL88]. The additional arguments a nonterminal in a DCG can have correspond to the attributes in attribute grammars.

There are also some differences: Attribute grammars do not have the corresponding notion of a logical variable. In many cases, notably when using context arguments and accumulator pairs,

a DCG may degenerate to an attribute grammar if bidirectionality is not required. I.e., in the case where a DCG is used for generating a sentence, no correspondence to an AG can be found. DCGs employ a simple top down parsing strategy while AGs are first generating the parse tree and are then decorating it with the help of an attribute evaluator. On the other hand, unification of attributes may direct the parsing process in a DCG, while in AGs the parse tree is determined before any evaluation. AGs have single directed evaluation; cycles of dependencies may appear that cannot be resolved with primitive evaluation.

Due to the many concepts in common, also parts of the implementation strategies may be similar. Beside the evaluation strategy, implementation of AGs concentrate on the efficient implementation of attributes. Attribute evaluation is somewhat easier to implement than Prolog execution because no backtracking is involved. Attributed structure trees are implemented straight forward by tree nodes representing symbol instances. Since many attributes are passed around, much in the style of Prolog programming techniques, attempts are made to globalize such attributes saving all the space for all instances of attributes connected with one another [KS86,Kas87,Kas84]. The basic idea is to decide whether each attribute can be implemented into a global variable or into a stack. The choice is done by analyzing the overlappings of attribute instance lifetimes for all possible trees. The instances of an attribute can be stored into a single variable if all their lifetimes are non-overlapping, and this for any tree. Further optimizations [JP90] are using unstrict stacks, where not only the top element can be accessed but also an element below with a fixed offset. With the 'globalization' of WAM-registers, we can simulate the implementation of attributes whose lifetime is pairwise disjoint.

Chapter 7

Applications

In this chapter some applications of our techniques are given. After deriving the essential mechanism of a Prolog machine, we are presenting some programming techniques that benefit from transformations.

7.1 Re-inventing the Vienna Abstract Machine

Program transformation cannot only help us to improve the execution time of programs, it can also produce insights into the implementation architecture on its own behalf. Many researchers in the area of language implementation have tried to formalize and automate their work — writing compilers and interpreters — with the help of partial evaluation. In fact Futamura's original work was motivated by implementing Lisp, Ershov's work by Algol-languages and Komorowski's work by Prolog. Using Prolog as an executable specification for its own implementation has been used for various abstract Prolog machines. Bowen, Byrd and Clocksin [BBC83] were probably the first to use this approach. Their machine, called ZIP, used symmetric read and write modes and an intermediate list for passing arguments. The read and write mode is most easily expressed in the specification with the help of unification.

Kursawe's approach. In his influential paper [Kur86] Peter Kursawe explained the compilation of the KAP (Karlsruhe Prolog Machine) which is a very close relative to the WAM (Warren Abstract Machine). Kursawe identified the required instruction set of the WAM by considering the residual programs obtained by partial evaluation. Compilation is then performed as follows: Prolog's unification is defined with a general unification predicate that uses lower level operations — the instructions of the abstract machine (e.g., assignment to a free variable, access to arguments of structures etc.). The clauses are normalized — head unifications are made explicit using the new predicate as a replacement. With the help of partial evaluation he is able to compile a Prolog clause into these instructions. Kursawe notes that his approach is applicable to the compilation of any language and even better if *very complex implicit operations (i.e. unification in Prolog)* exist in the language.

We have therefore tried to derive the implementation details of another Prolog Machine — the VAM (Vienna Abstract Machine) — in a similar manner. (See Fig. 7.1 for a comparison with other abstract machines.) In [KN90] we presented a handwritten executable specification of the VAM and have speculated that:

> Taking the translation of Prolog clauses to VAM code and a simple meta-interpreter as input, the VAM could probably be derived automatically by partial evaluation (deduction)—being in the style of [Kur86].

Machine	*yr.*	*Operands* Head	Goal	*Decoding*	*Implicit operands*	*control transfer position*	*instruct. removal*
PLM	77	2	1	h [g]	none	prefix	n
ZIP	83	1	1	g, h	arg-stack	postfix	y
WAM	83	2	2	g, h	none	postfix	y
VAM_{2P}	86	1	1	h+g	none	prefix	n
VAM_{1P}	86	0	2	g	none	prefix	y

Figure 7.1: Comparison of instruction formats

However, it turned out that partial evaluation is too weak to tackle with our machine.

In the sequel we explain the essentials of the VAM and present then the derivation of the difficult parts of the VAM that cannot be derived with the usual transformation techniques. With the help of partially static goals and EBC-transformation we are able to transform the general meta-interpreter into the executable specification. We will first present an abstract machine for restricted clauses. The full description of the abstract machine and its implementation can be found in [KN90]. Initially, we restrict clauses to those containing no variables at all. While this is not a realistic subset of Prolog for practical applications, it serves to clarify the fundamental differences between WAM and VAM. The introduction of variables pose no problems for the derivation method.

7.1.1 Representation of clauses

The representation of clauses in VAM intermediate code is very close to their syntactic representation. In Fig. 7.2 the complete (bijective) mapping between ground clauses and VAM code is defined by a DCG. By and large, terms are translated to a flat prefix code. In a clause, three different kinds of code are used:

control codes, (c-Any) are used to embrace goals. A goal starts with c-goal $\langle p \rangle$ and either ends with c-call if another goal is thereafter or ends with c-lastcall if it is the last goal in a clause. If the clause is a fact, we have no goal at all denoted by c-nogoal.

head codes, (h-Any) are used to encode terms in the arguments of a clause's head. The arguments are translated into flat prefix code.

goal codes, (g-Any) encode terms in goals. The structure is the same as for head codes.

VAM instructions differ fundamentally from WAM instructions. They can be understood only by their combination at runtime. The *real* instruction set of VAM is the set of all valid combinations of instructions. Taking the translation of Prolog clauses to VAM code and a simple meta-interpreter as input, the VAM could probably be derived automatically by partial evaluation (deduction)—being in the style of [Kur86]. However, the abstract interpreter as well as the complete VAM was designed by hand.

In Fig. 7.3 an abstract interpreter for VAM code is given. The specification describes the process of unification and (determinate) control in detail, but—similar to [BBC83]—it does not explicitly cover backtracking aspects. A program to be interpreted is represented by the vam_clause/1 facts. A fact vam_clause([PredName|Cs]) consists of the predicate name and the VAM code translated in Fig. 7.2. If a predicate consists of several clauses the goal ...,vam_clause([NextPred|NHs]),... will yield several solutions. Backtracking is therefore implicit in the specification. For further discussions of nondeterminism in the VAM refer to [KN90].

```
clause(Head,Goals) ⟶
    head(Head),
    body(Goals).

body(true) ⟶
    [c-nogoal].
body((Goal,Goals)) ⟶
    goallist((Goal,Goals)).

goallist(true) ⟶
    [c-lastcall].
goallist((Goal,Goals)) ⟶
    goal(Goal),
    [c-call],
    goallist(Goals).

head(struct(F/N,L)) ⟶
    [F/N],
    argumentlist(h,L).

goal(struct(F/N,L)) ⟶
    [c-goal,F/N],
    argumentlist(g,L).

argument(X,struct(Const/0,[])) ⟶
    [X-const,Const].
argument(X,struct(F/N,[A|As])) ⟶
    [X-struct,F/N],
    argumentlist(X,[A|As]).

argumentlist(_X,[]) ⟶
    [].
argumentlist(X,[E|Es]) ⟶
    argument(X,E),
    argumentlist(X,Es).
```

Figure 7.2: Clause representation in VAM for ground clauses

The procedural behavior of VAM is described by the proof tree of the logic program. First a query is translated like the body of a clause, then the corresponding predicate is fetched and the interpreter is finally called. The interpreter consists of a (tail recursive) predicate vam_prove/3 which holds the interpreter state consisting of: the list of remaining head codes, the list of remaining goal codes and a continuation stack for nested calls. The process of proving a goal consists of two major steps corresponding to the different kinds of codes (Ch. 7.1.1): unification and resolution. By consequence an iteration in the interpreter vam_prove/3 (a recursive call) can be performed in two ways, either via unification/3 or resolution/5 respectively[1]. The state transitions are specified by facts in order to emphasize which states are changed. Before trying to prove these facts, the interpreter takes the first elements of both lists (head and goal code) and combines them (symbolized by the functor +/2) in order to pass them to the facts. The effective instructions HeadCode+GoalCode are derived by generating all valid combinations of head and goal codes.

unification(Instruction,DifflistHead,DifflistGoal) An attempt is made to unify corresponding arguments of head and goal; the remaining codes are passed back to the predicate vam_prove/3. Combinations such as h-struct+g-const are not stated, they simply fail. Note that unification/3 changes only the two code lists. An observation evident from the specification is that arbitrarily nested structures which occur both in the head and in the goal neither need a so called *push down stack* nor a counter to unify their arguments since the functors F/A do not insert their arity into vam_prove/3's state.

resolution(Instruction,HeadCode,DifflistGoal,DifflistStack,NextPred) A clause of NextPred is selected by the interpreter (see goal vam_clause/1). If a fact in the head was proved and if the body contains another subgoal (c-nogoal+c-call), the new goal is selected. The stack is not affected at all. If the head unifies and the goal was the last in the caller's clause (c-goal+c-lastcall), the head code will become the new goal code. Again, the stack is not altered (last-call optimization). If the head unifies and there is another goal in the caller's clause (c-goal+c-call), then the continuation is pushed onto the stack, the head code becomes the new goal code and the interpreter switches to the new clause's head. The stack needs to be popped if a fact unifies, and if the goal is the last in the caller's clause (c-nogoal+c-lastcall).

[1]Note that there is exactly one or no match for a correct goal vam_prove/3

```
% unification/3: The unification instructions
% unification(Head+Goal, HeadsIn-HeadsOut, GoalsIn-GoalsOut)

unification((h-const)+(g-const), [Const|Hs]-Hs, [Const|Gs]-Gs).
unification((h-struct)+(g-struct), [F/A|Hs]-Hs, [F/A|Gs]-Gs).

% resolution/5: Goal selection
% resolution(Head+Goal, Heads, GoalsIn-GoalsOut, StackIn-StackOut, NextPred)

resolution((c-nogoal)+(c-call), [], [c-goal,F/A|Gs]-Gs, St-St, F/A).
resolution((c-goal)+(c-lastcall), [F/A|Hs], []-Hs, St-St, F/A).
resolution((c-goal)+(c-call), [F/A|Hs], Gs-Hs, St-[Gs|St], F/A).
resolution((c-nogoal)+(c-lastcall), [], []-Gs, [[c-goal,F/A|Gs]|St]-St, F/A).

% vam_prove/3: Abstract interpreter
% vam_prove(HeadList,GoalList,Stack)

vam_prove([c-nogoal],[c-lastcall],[]).
vam_prove([H|Hs],[G|Gs],St) ←
    unification(H+G,Hs-NHs,Gs-NGs),
    vam_prove(NHs,NGs,St).
vam_prove([H|Hs],[G|Gs],St) ←
    resolution(H+G,Hs,Gs-NGs,St-NSt,NextPred),
    vam_clause([NextPred|NHs]),
    vam_prove(NHs,NGs,NSt).

query(Query) ←
    parse(body(Query),[c-goal,F/A|GoalCode]),
    vam_clause([F/A|HeadCode]),
    vam_prove(HeadCode,GoalCode,[]).
```

Figure 7.3: An abstract interpreter for VAM

Execution proceeds with the popped continuation. If the stack is empty, the interpreter halts successfully.

7.1.2 A derivation of the difficult parts of the VAM

At first sight, it may seem that the resolution part is more difficult to derive than the unification part, since more complex operations are involved. However, the opposite was true. The resolution part is merely another encoding of the typical iteration of a Prolog meta-interpreter with a linear body. The unification part differs radically from the ordinary unification algorithm: While general unification even of ground terms *must* use an algorithm that requires additional space that depends on the terms' sizes (usually a recursive definition, or an equation solver [MM82] is used), the unification part in VAM (for ground terms) requires constant space – it is therefore independent of the terms' sizes. The reason for the disappearance of the recursive definition lies in the flat representation of the terms in head and body. Furthermore, the interpreter relies on the well-formedness of these codes.

To simplify our presentation, we will use the predicates term_vam/2 for mapping a general term to VAM-code, term_term/2 for the unification definition, and vam_vam/2 for defining unification between VAM-terms which is defined in terms of term_vam/2 and term_term.

```
is_term(const).
is_term(struct(A,B)) ←
    is_term(A),
    is_term(B).

term_vam(T,Vs) ←
    term_vam(T,Vs,[]).

term_vam(const,[const|Vs],Vs).
term_vam(struct(A,B),[struct|Vs0],Vs) ←
    term_vam(A,Vs0,Vs1),
    term_vam(B,Vs1,Vs).
```

```
term_term(const,const).
term_term(struct(AL,BL),struct(AR,BR)) ←
    term_term(AL,AR),
    term_term(BL,BR).

vam_vam(Vs,Ws) ←
    term_vam(TV,Vs),
    term_vam(TW,Ws),
    term_term(TV,TW).
```

All predicates working on the list representation contain unnecessary variables, while the parts using the usual term representation do not. With the usual fold/unfold-techniques we are able to specialize vam_vam/2 by removing the unnecessary variables TV and TW. In order to ensure procedural equivalence, we have to assume already that both lists are of fixed size.

```
vam_vam([const|Vs],Vs,[const|Ws],Ws).
vam_vam([struct|Vs0],Vs,[struct|Ws0],Ws) ←
    vam_vam(Vs0,Vs1,Ws0,Ws1),
    vam_vam(Vs1,Vs,Ws1,Ws).
```

The resulting definition works already without the intermediate variables representing the terms as trees. However, we have obtained new unnecessary variables for both lists. These variables cannot be removed within the classical framework. Indeed, the double recursion is not completely redundant: Our definition still requires that the well formedness of the two lists is still checked. In order to proceed further, we have to use partially static goals that ensure the well-formedness of both lists. The VAM-code was generated with the 'compiler' term_vam/2. We reuse this knowledge for specialization in is_vam/1. I.e., there exists a term _T which can be mapped onto the list Vs. Note that a static goal like !is_vam(Vs) cannot be expressed in any formalism given in the literature of partial evaluation. Binarizing the predicate and using EBC-transformation a simplified version of is_vam/1 is obtained.

```
is_vam(Vs) ←
    term_vam(_T,Vs).
```

```
is_vam([const|Vs],Vs).
is_vam([struct|Vs0],Vs) ←
    is_vam(Vs0,Vs1),
    is_vam(Vs1,Vs).
```

```
is_vam([const],true).
is_vam([const|Vs],s(Cont)) ←
    is_vam(Vs,Cont).
is_vam([struct|Vs],Cont) ←
    is_vam(Vs,s(Cont)).
```

The continuation argument is reduced to a simple 'counter'. EBC-transformation is also able to reduce the double recursion of the unification algorithm for VAM-code to a simple counter.

```
vam_vam([const],[const],true).
vam_vam([const|Vs],[const|Ws],s(Cont)) ←
    vam_vam(Vs,Ws,Cont).
vam_vam([struct|Vs],[struct|Ws],Cont) ←
    vam_vam(Vs,Ws,s(Cont)).
```

The remaining problem consists now in specializing the following predicate vam_vam_r/2.

```
vam_vam_r(Vs,Ws) ←
    !is_vam(Vs,true),
    !is_vam(Ws,true),
    vam_vam(Vs,Ws,true).
```

By definition, the following predicate is defined which makes the continuations more explicit:

```
vam_vam_r(Vs,Ws,C) ←
    !is_vam(Vs,C),
    !is_vam(Ws,C),
    vam_vam(Vs,Ws,C).
```

Unfolding and folding back again yields the original predicate vam_vam/3. Since it was possible to map the continuations of all three predicates into a single continuation, we define a new predicate that does not take the continuation of vam_vam/3 into account:

```
vam_vam_s(Vs,Ws,C) ←
    !is_vam(Vs,C),
    !is_vam(Ws,C),
    vam_vam(Vs,Ws,_).
```

```
vam_vam_s(Vs,Ws,C) ←
    !is_vam(Vs,C),
    !is_vam(Ws,C),
    vam_vam_s(Vs,Ws).

vam_vam_s([],[]).
vam_vam_s([const|Vs],[const|Ws]) ←
    vam_vam_s(Vs,Ws).
vam_vam_s([struct|Vs],[struct|Ws]) ←
    vam_vam_s(Vs,Ws).
```

Again, we are able to derive the same predicate vam_vam/3. Therefore, unification of the simplified Vienna Abstract Machine code can be implemented with the predicate vam_vam_s/2.

To summarize our strategy, the following operations were necessary:

- Unification of VAM-code was defined by mapping VAM-code to terms for which a unification algorithm was already given.
- The recursive predicates working exclusively on VAM-code (is_vam/1 and vam_vam/2) were mapped into tail recursive predicates using the EBC-transformation.
- A new unification predicate that is specialized to the case when both arguments are syntactically valid VAM-code was defined. The restriction was only expressible with the help of partially static goals.
- Another predicate was defined which generalized the continuation of the goal vam_vam/2. It was possible to show that this predicate is equivalent to the inefficient unification procedure vam_vam/2. This was possible because the partially static goals were unfolded and static unifications can be converted into dynamic unifications.

7.2 DCGs with error handling

The usual DCG formalism is incapable of handling errors. In case of an error the grammar simply fails giving no clue to where the error occurred. The parsing of incorrect sentences is also of interest for natural language parsers ([Ber91] pages 178-186). Current proposals for error-handling in parsers for formal languages, e.g., the system PROFIT [Paa91], restrict DCGs to deterministic cases. Error handling is performed in the traditional style using a panic mode. While DCGs might become a replacement for traditional parser generators in this way, the concessions made to obtain efficiency are rather high. In existing parsers written in Prolog, a similar panic-mode is often used. If an error is encountered, the error message is stored into the database. Since the

parsing process might be non-deterministic, the error messages in the database are only valid if the parsing process did finally fail. After an error has been stored in the database, the parser may recover by backtracking to an earlier point yielding a successful parse. The database may therefore contain invalid error messages. These error messages have to be removed from the database. There are several drawbacks using this technique: First, side effects are used to describe the errors encountered, making a program less declarative and more dependable on the actual execution strategy. Second, ambiguous grammars are very tricky to handle. If there was already a successful parse, we are only interested in further parses without errors. Third, syntactic error messages occur often long after the location of the actual error. This is especially true in languages which use abundantly parentheses for different purposes or overload many tokens (especially C and Prolog). For traditional well engineered languages as Modula, traditional error recovery yields more accurate results because the grammars distinguish several layers in programs that are already distinguishable at the token level. E.g., semi-colons are used only to delimit statements within a statement sequence. The corresponding 'parentheses' on the outer levels are represented with unique keywords. Current Prolog parsers yield only very terse error messages which do not help a lot. Especially beginners have considerable problems with error messages that are not precise. In a Prolog course, we experienced that more than 50% of the error messages (produced by SICStus Prolog 2.1) occurred at the end of a Prolog term, far away from the actual error.

We will discuss another way of parsing terms in Prolog which does not need the database for storing possibly invalid error messages and yields more accurate error messages. The idea is to add another difference list to the DCG that passes the sequence of encountered error messages further on. Parsing now consists in either trying to find the parses without error which is the usual case. If no successful parse can be encountered, we are searching for the 'best' explanation of the error which is, e.g., the smallest sequence of error messages.

A simple way to use the grammar is to generate all possible lists to store the error messages starting with the smallest sequence. Since the number of errors in programs is usually very small, it will be equally sufficient to parse the program text with some small numbers of errors that may occur. Otherwise, i.e., if there are too many errors in a text, the text is parsed with an unconstrained error list. A single cut is necessary in contrast to the usage of the database predicates.

```
parse_text(Text,Errors) ←
    (   length(MaxErrors,max_errors),
        append(Errors,_,MaxErrors)
    ;   true
    ),
    phrase_errors(program,Text,[],Errors,[]),
    !.
```

Error messages have to be encoded explicitly. I.e., for every desirable error message a new grammar rule has to be introduced. The grammar rules have now to be rewritten to take the new error classes into account. In such an approach, we are able to encode the 'most common' errors with ease.

7.3 Taming left recursion

DCGs are executed in a recursive descent mode. Left recursions can therefore not be handled directly. Usually grammars are rewritten to overcome such problems. The classical compiler construction techniques can be applied. The construction of the Abstract Syntax Tree is mostly complicated by such an approach. Another alternative is to change the resolution strategy to, e.g., OLD [TS86]. We are proposing yet another strategy to deal with left recursions in DCGs. In addition to the usual 'state' — the sequence of tokens still to be read, represented by a difference

```
cps_dcg([],Is,Is,Es,Es).
cps_dcg([L|Gs],Is0,Is,Es0,Es) ←
    list_difflist(L,Is0,Is1),
    cps_dcg(Gs,Is1,Is,Es0,Es).
cps_dcg([Goal|Gs],Is0,Is,Es0,Es) ←
    call(Goal),
    cps_dcg(Gs,Is0,Is,Es0,Es).
cps_dcg([error(E)|Gs],Is0,Is,[E-Is0|Es0],Es) ←
    cps_dcg(Gs,Is0,Is,Es0,Es).
cps_dcg([g(G)|Gs],Is0,Is,Es0,Es) ←
    cps_dcg_clause(G,Gs0,Gs),
    cps_dcg(Gs0,Is0,Is,Es0,Es).
```

Figure 7.4: A DCG-interpreter with error messages

```
cps_dcg_bin([],Is,Is,Es,Es,Cont) ←
    $demo(Cont).
cps_dcg_bin([L|Gs],Is0,Is,Es0,Es,Cont) ←
    list_difflist(L,Is0,Is1,cps_dcg_bin(Gs,Is1,Is,Es0,Es,Cont)).
cps_dcg_bin([Goal|Gs],Is0,Is,Es0,Es,Cont) ←
    call(Goal,cps_dcg_bin(Gs,Is0,Is,Es0,Es,Cont)).
cps_dcg_bin([error(E)|Gs],Is0,Is,[E-Is0|Es0],Es,Cont) ←
    cps_dcg_bin(Gs,Is0,Is,Es0,Es,Cont).
cps_dcg_bin([g(G)|Gs],Is0,Is,Es0,Es,Cont) ←
    cps_dcg_bin_clause(G,Gs0,Gs,cps_dcg_bin(Gs0,Is0,Is,Es0,Es),Cont).
```

Figure 7.5: A binary DCG-interpreter with error messages

list — we add another state for the number of tokens that can be used by newly encountered nonterminals. The number of tokens that will be read by the terminals within a single rule are therefore reserved in advance.

In the grammar in Fig. 7.6, expressions are represented in an ambiguous way. The grammar cannot be used to parse a given sequence of tokens if the AST is unknown, because the second rule of a//1 contains as its first entry a nonterminal that depends on a//1 — in this case a direct recursion. Nonetheless, given a finite sequence, finitely many solutions are contained in the SLD-tree. The solutions are only 'overshadowed' by an infinite failure branch. Every inference that goes through left recursion does not reduce the size of the token sequence. But the number of tokens left over for new nonterminals is reduced by one for every inference. The infinite failure branch can therefore be pruned after the numbers of tokens left over for new non-terminals is exhausted. The number of tokens left over for new recursions can be most simply represented by the list to be parsed. When encountering a new rule, the number of tokens contained in that rule is subtracted from the given sequence (Fig. 7.7). It is possible to reduce the number of useless inferences even more by left-propagation of statically known data (Fig. 7.8).

```
a(number(N)) ⟶
    [number(N)].
a(A+B) ⟶
    a(A),
    [+],
    a(B).
```

```
dcg_a(number(N),[number(N)|L],L).
dcg_a(A+B,L0,L) ←
    dcg_a(A,L0,L1),
    L1 = [+|L2],
    dcg_a(B,L2,L).
```

Figure 7.6: An ambiguous grammar with nonterminating left recursion

```
dcgl_b(T,L) ←
    dcgl_bi(T,L,[],L,[]).

dcgl_bi(number(N),[number(N)|L],L,[_|W],W).
dcgl_bi(A+B,L0,L,[_|W0],W) ←
    dcgl_bi(A,L0,L1,W0,W1),
    L1 = [+|L2],
    dcgl_bi(B,L2,L,W1,W).

?- dcgl_b(T,[number(1),+,number(2),+,number(3)]).
```

Figure 7.7: Terminating left recursive parsing

```
dcgl_c(T,L0,L) ←
    L0 = [_|L1],
    dcgl_ci(T,L0,L,L1,L).

dcgl_ci(number(N),[number(N)|L],L,W,W).
dcgl_ci(A+B,L0,L,[_,_|W0],W) ←
    dcgl_ci(A,L0,L1,W0,W1),
    L1 = [+|L2],
    dcgl_ci(B,L2,L,W1,W).
```

Figure 7.8: Optimized terminating left recursive parsing

7.4 Improving occur-check

In [Neu92], we have proposed an improved execution model for Prolog with occur-check. It is current practice that Prolog systems omit the occur-check. The standard arguments against the occur-check are efficiency reasons. We have experienced that unification with occur-check [Rob71] does not contribute a lot to the executability of logic descriptions. The reason is that most

```
dcgl_d(T,L0,L,Cont) ←
    L0 = [_|L1],
    dcgl_di(T,L0,L,L1,L,Cont).

dcgl_di(number(N),[number(N)|L],L,W,W,Cont) ←
    $demo(Cont).
dcgl_di(A+B,L0,L,[_,_|W0],W,Cont) ←
    dcgl_di(A,L0,L1,W0,W1,=(L1,[+|L2],dcgl_di(B,L2,L,W1,W,Cont))).

dcgl_e(T,L0,L,Cont) ←
    L0 = [_|L1],
    dcgl_ei(T,L0,L1,dcgl_e_true(L,Cont)).

dcgl_ei(number(N),[number(N)|L],W,LWCont) ←
    dcgl_ei_demo(LWCont,L,W).
dcgl_ei(A+B,L0,[_,_|W0],LWCont) ←
    dcgl_ei(A,L0,W0,'C'(+,dcgl_ei(B,LWCont))).

dcgl_ei_demo(dcgl_e_true(L,LWCont),L,L) ←
    $demo(LWCont).
dcgl_ei_demo('C'(+,LWCont),[+|L],W) ←
    dcgl_ei_demo(LWCont,L,W).
dcgl_ei_demo(dcgl_ei(B,LWCont),L,W) ←
    dcgl_ei(B,L,W,LWCont).
```

Figure 7.9: Binary verions without and with EBC-transformation applied

programs which need occur-checks fail to provide useful solutions because they end up in infinite failure branches. Although they will not yield an unsound solution, this behavior is not very satisfactory from a programmer's point of view. We believe that this may be another reason why the occur-check is not considered important by Prolog implementors. As has been noted by [Llo87], the occur-check is most vital to the correct implementation of difference list programs. Prolog's powerful notation of difference lists and similar techniques cannot be used to their full extent due to the lack of a reasonable implementation of the occur-check: In most cases, the programmer has to assume tacitly that the original list is complete. The possibility to deal with unknown data is therefore rather restricted. What a programmer would like to have are well-founded difference lists: If the length of the difference is known, we want to reason about difference lists in the same convenient way [Plü91] as we are used to reason about ordinary lists with fixed length. It should be possible to write a structurally similar program with general difference lists that behaves in the same way as its restricted counterpart.

Pitfalls of the occur-check. A case where the occur-check leads to an infinite failure branch is discussed. Consider the following simple recursive predicate. Note that suffixd/2 is the same as suffix/2 in [SS86] but with arguments exchanged in order to outline that the arguments form a difference list L0-L.

```
suffixd(L,L).
suffixd([_|L0],L) ←
    suffixd(L0,L).
```

Since in the fact suffixd(L,L) a variable occurs twice the occur-check may fail. The second clause will never fail because of an occur-check since all variables in the head are distinct. Queries to suffixd/2 where the occur-check causes the first clause to fail will therefore construct an infinite failure branch. E.g, the query ← suffixd(L,L) succeeds once and ends in an infinite failure branch on backtracking, trying to resolve at the $(n+1)$-st step suffixd(L,[$V\_1, ..., V\_n$|L]) and trying to produce the infinite term [$(V\_1, ..., V\_n)$*]. Note that also infinite trees [Col84] do not help: The system would yield an infinite number of unsound solutions. On backtracking, termination is again a problem. Failing completely cannot be a solution, because there may be further correct solutions although the occur-check failed: The query ← suffixd([a|L],L) will at first attempt to unify [a|L] = L; so the occur-check will fail, but will find thereafter the identity substitution, which is the only correct answer. Again, the predicate will fall into an infinite failure branch since the second clause is always applicable. We now identify the cases in which our example predicate has to fail completely: If the first argument is a variable and the second argument is structured and contains the first one, there is no possibility to remove the occurrence of the first argument within the second via forward recursion: Starting from ← suffixd(L,[a|L]), the next resolution step will yield a new subgoal ← suffixd(L1,[a,_|L1]). Here L has been substituted: L = [_|L1]. Note that in our example this is the only case where our program goes into an infinite loop although sufficient information is provided to fail safely.

A difference list is a term (or argument pair) of the form Xs-Ys denoting the difference between the longer list Xs and the shorter list Ys. Since difference lists are a pure syntactic convention in Prolog, there are also terms that do not describe a difference list, e.g.: []-[1]. The implicit restriction on correctly used difference list Xs-Ys is that Xs must not occur in Ys, and if ground, Ys must occur in Xs. While an incorrect difference list which consists of a pair of rigid terms [Plü91] poses no problem — a failure will occur anyway — open lists may show up the problem discussed. Since we are interested in performing the occur-checks as early as possible, we use the domain of subterm relations for abstract interpretation.

$$s \ll t \text{ iff } t = f(t_1, \ldots, t_n) \text{ and for some } t_i : s = t_i \text{ or } s \ll t_i$$

With the help of this analysis we were able to obtain a new and improved version of the predicate. In the following we use unify/2 —unification with occur-check— and the predicate occurs_check(Term,Var) which succeeds if Var is a variable and does not occur in Term[2].

```
suffixd(L0,L) ←
    var(L0),
    nonvar(L),
    ⊬ occurs_check(L,L0),
    !,
    fail.
suffixd(L0,L) ←
    L0 == L,
    !.
suffixd(L0,L) ←
    unify(L0,L).
suffixd([_|L0],L) ←
    suffixd(L0,L).
```

This example shows that our technique is applicable to general recursive predicates often used in DCGs. The following predicate constructs a flattened binary tree in prefix notation. Note that delaying the annotation cannot be performed directly. We have to introduce another argument to propagate the "bottom list". This predicate handles incomplete data structures safely, e.g.: ← tlist([...|L],L) will produce the same answers as ← tlist([...],[]), the usual constrained way of using difference lists. E.g. ← tlist([A,B,C|L],L) yields correctly solution A = t, B = a, C = a (and fails thereafter), while a system with infinite trees [Col84] will come up with the unsound solution A = a, B = _,C = _, [..inf..], produces then the correct one and continues with infinitely many unsound solutions.

```
tlist([a|L0],_L,X) ←
    var(L0),
    nonvar(X),
    ⊬ occurs_check(X,L0),
    !,
    fail.
tlist([a|L0],L,_) ←
    unify(L0,L).
tlist([t|L0],L,X) ←
    tlist(L0,L1,X),
    tlist(L1,L,X).
```

The implementation we have proposed has to carry an additional argument through the grammar. In fact, this argument is a context argument. It can be easily moved into the registers. The second argument is not used at all. The program transformed with EBC-transformations will therefore be comparable to the usual implementation: Two arguments are used, however no argument is written onto the continuation. The only overheads that must be removed with other means are the lack of indexing or a similar mechanism that could combine the occur-check, pruning and general unification. Such improvements, however, are out of our current scope.

[2]A public domain implementation `METUTL.PL` exists; created by R.A.O'Keefe

Bibliography

[App87] Andrew Appel. Garbage collection is faster than stack allocation. *Information Processing Letters*, 25:275–279, 1987.

[App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[AR89] Harvey Abramson and M. H. Rogers, editors. *Meta-programming in Logic Programming*. The MIT Press, see also [Llo88], revised edition, 1989.

[ASS85] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[Bar88] Jonas Barklund. What is a meta-variable in Prolog? In Lloyd [Llo88], pages 281–292.

[BBC83] D.L. Bowen, L.M. Byrd, and W.F. Clocksin. A portable Prolog compiler. In *Proceedings of the Logic Programming Workshop*, Albufeira, Portugal, 1983.

[BCD90] Annalisa Bossi, Nicoletta Cocco, and Susi Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[Bee88] Joachim Beer. The occur-check problem revisited. *The Journal of Logic Programming*, 5(3):243–262, September 1988.

[BEJ88] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[Ber91] Robert C. Berwick. Principle-based parsing. In Peter Sells, Stuart M. Shieber, and Thomas Wasow, editors, *Foundational Issues in Natural Language Processing*, chapter 4, pages 115–226. The MIT Press, 1991.

[BF92] Manfred Brockhaus and Andreas Falkner. Skriptum zur Vorlesung Übersetzerbau. Institut für Computersprachen, 1992.

[BR89] M. Bugliesi and F. Russo. Partial evaluation in Prolog: Some improvements about cut. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 645–660, Cleveland, Ohio, USA, 1989.

[BR91] Pascal Brisset and Olivier Ridoux. Naïve reverse can be linear. In Furukawa [Fur91], pages 857–870.

[Can86] Michel van Caneghem. *L'Anatomie de Prolog*. InterÉditions, Paris, 1986.

[CC69] G. de Chatellier and Alain Colmerauer. W-grammar. In *Proceedings of the ACM Congress*, pages 511–518, San Francisco, Calif., August 1969. ACM.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, page New York. Plenum Press, 1978.

[Coh88] R.A. Cohen. A view of the origin and development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[Col75] Alain Colmerauer. Les grammaires de metamorphose. Technical report, Groupe d'Intelligence Artificielle, Université de Marseille II, November 1975.

[Col78] Alain Colmerauer. Metamorphosis grammars, [Col75]. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–189. Springer-Verlag, 1978.

[Col84] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84), ICOT, Tokyo*, pages 85–99, 1984.

[Col87] Alain Colmerauer. Opening the Prolog-III universe. *BYTE Magazine*, 12(9), August 1987.

[Con63] M.E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6:396–408, 1963.

[CS77] K.L. Clark and S. Sickel. Predicate logic: A calculus for deriving programs. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 1977.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[Dan92] Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP '92)*, volume 582 of *Lecture Notes in Computer Science*, pages 130–150, Rennes, France, February 1992. Springer-Verlag.

[DB76] J. Darlington and R.M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[Deb92] Saumya K. Debray. On the complexity of dataflow analysis of logic programs. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 509–520. Springer-Verlag, July 1992.

[Dem92] Bart Demoen. On the transformation of a Prolog program to a more efficient binary program. Technical Report 130, K.U.Leuven Department of Computer Science, revised version LOPSTR92, 1992.

[Der83] Pierre Deransart. Logical attribute grammars. In R. E. A. Mason, editor, *IFIP '83*, pages 463–469. North-Holland, September 1983.

[DF92] P. Deransart and G. Ferrand. An operational formal definition of Prolog. *New Generation Computing*, 10(2):121–177, 1992.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DLM88] Pierre Deransart, Bernard Lorho, and Jan Małuszyński, editors. *First International Symposium, PLILP 88*, volume 348 of *Lecture Notes in Computer Science*, Orléans, France, May 1988. Springer-Verlag.

[DM85] Pierre Deransart and Jan Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[DM88] Pierre Deransart and Jan Maluszynski. A grammatical view of logic programming. In Deransart et al. [DLM88], pages 219–251.

[DM90] P. Deransart and J. Małuszyński, editors. *2nd International Symposium, PLILP 90*, volume 456 of *Lecture Notes in Computer Science*, Linköping, Sweden, August 1990. Springer-Verlag.

[DN66] O.-J. Dahl and K. Nygaard. Simula — an algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[DS90] Edsger W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York, 1990.

[FFS89] K. Furukawa, H. Fujita, and T. Shintani. Deriving an efficient production system by partial evaluation. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 661–676, Cleveland, Ohio, USA, 1989.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In Bjørner et al. [BEJ88], pages 133–151.

[FNT91] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991. Also in: D. Bjørner and V. Kotov: Images of Programming, North-Holland, 1991.

[Fur91] Koichi Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991. The MIT Press.

[GB90] John Gallager and Maurice Bruynooghe. Some low-level source transformations for logic programs. In Maurice Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 229–244. K.U. Leuven, Department of Computer Science, April 1990.

[HL92] Pat M. Hill and John W. Lloyd. The Gödel report. Technical Report TR-91/2, University of Bristol, June 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[Hog81] C. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(4):372–392, April 1981.

[Hol92] Christian Holzbauer. Metastructures vs. attribute variables in the context of extensible unification. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Languages Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 260–268, Leuven, Belgium, August 1992. Springer-Verlag.

[HS91] T.H. Hickey and D.A. Smith. Toward the partial evaluation of CLP languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 43–51. ACM, 1991.

[Hui90] Serge Le Huitouze. A new data structure for implementing extensions to Prolog. In Deransart and Małuszyński [DM90], pages 136–150.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[JLM86] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. A logic programming language scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Relations, Functions and Equations*, pages 441–468. Prentice Hall, 1986.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, Berlin, 1985.

[JP90] Catherine Julié and Didier Parigot. Space optimization in the FNC-2 attribute grammar system. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 29–45. Springer-Verlag, September 1990.

[Kas84] Uwe Kastens. The GAG-system—a tool for compiler construction. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 165–182. Cambridge University Press, 1984.

[Kas87] Uwe Kastens. Lifetime analysis for attributes. *Acta Informatica*, 24(6):633–652, November 1987.

[Kas91] V.N. Kasyanov. Tools and techniques of annotated programming. In D. Hammer, editor, *Compiler Compilers, Third International Workshop, Schwerin, Germany, October 1990.*, volume 477 of *Lecture Notes in Computer Science*, pages 117–131. Springer-Verlag, 1991.

[KB88] Robert A. Kowalski and Kenneth A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seatle, 1988. ALP, IEEE, The MIT Press.

[KN90] Andreas Krall and Ulrich Neumerkel. The Vienna Abstract Machine. In Deransart and Małuszyński [DM90], pages 121–135.

[Kom81] H.J. Komorowski. *A Specification of an Abstract Prolog Machine and Its Application to Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[Kom82] H.J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 255–267, 1982.

[Kow79] Robert Kowalski. *Logic for Problem Solving*. North Holland, New York, 1979.

[Kow88] R.A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[KP79] V.N. Kasyanov and I.V. Pottosin. Application of optimization techniques to correctness problems. In *Constructing Quality Software, Proc. IFIP TC2 Working Conference*, pages 237–248, Amsterdam, 1979. North-Holland.

[KP82] V.N. Kasyanov and I.V. Pottosin. Concretization systems: Approach and basic concepts (in russian). Preprint 349, Computing Center, Novosibirsk, USSR, 1982.

[KS86] Uwe Kastens and M. Schmidt. Lifetime analysis for procedure parameters. In B. Robinet and Reinhard Wilhelm, editors, *Proceedings of the 1st European Symposium on Programming (ESOP '86)*, volume 213 of *Lecture Notes in Computer Science*, pages 53–69. Springer-Verlag, March 1986.

[Kur86] Peter Kursawe. How to invent a Prolog machine. In E. Shapiro, editor, *Third International Conference on Logic Programming, LNCS, vol. 225*, pages 134–148. Springer-Verlag, 1986. Also in New Generation Computing 5:87-114.

[Lan66] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[Lav88] S.S. Lavrov. On the essence of mixed computation. In Bjørner et al. [BEJ88], pages 317–324.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[Llo88] John W. Lloyd, editor. *Proceedings of the First Workshop on Meta-programming in Logic*. University of Bristol, see also [AR89], June 1988.

[LS87] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. Technical Report CS-87-09, Department of Computer Science, University of Bristol, England, 1987. Revised version in [LS91b].

[LS88] A. Lakhotia and L. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2,3):211–225, 1988.

[LS91a] A. Lakhotia and L. Sterling. ProMiX: A Prolog partial evaluation system. In L. Sterling, editor, *The Practice of Prolog*, chapter 5, pages 137–179. The MIT Press, 1991.

[LS91b] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[Mah88] M.J. Maher. Equivalences of logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, 1988.

[Mal87] John Malpas. *Prolog: a relational language and its applications*. Prentice Hall, New Jersey, 1987.

[Mal91] Jan Maluszyński. Attribute grammars and logic programs: a comparison of concepts. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 330–357. Springer-Verlag, June 1991.

[Mar88] André Mariën. An optimal intermediate code for structure creation in a WAM-based Prolog implementation. In *Proceedings of the International Computer Science Conference '88*, pages 229–236, December 1988.

[MD91] André Mariën and Bart Demoen. A new scheme for unification in WAM. In Saraswat and Ueda [SU91], pages 257–271.

[Mei91] Micha Meier. Recursion vs. iteration in Prolog. In Furukawa [Fur91], pages 157–169.

[Mey85] Bertand Meyer. On formalisms in specifications. *IEEE Software*, 3(1):6–25, January 1985.

[Mey90] Bertand Meyer. *Introduction to the Theory of Programming Languages*. International Series in Computer Science. Prentice Hall, 1990.

[Mey92] Bertand Meyer. *Eiffel the Language*. Prentice Hall, New York, 1992.

[Min70] Marvin Minsky. 1970 a.c.m. turing lecture, form and content in computer science. *Journal of the ACM*, 17:197–215, 1970.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MNL88] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In Kowalski and Bowen [KB88], pages 909–923.

[MW91] Jan Małuszyński and Martin Wirsing, editors. *3rd International Symposium, PLILP 91*, volume 528 of *Lecture Notes in Computer Science*, Passau, Germany, August 1991. Springer-Verlag.

[Nai92] Lee Naish. Types and the intended meaning of logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 189–216. The MIT Press, 1992.

[Neu86] Gustaf Neumann. Meta-interpreter directed compilation of logic programs into Prolog. Research Report RC 12113 (No. 54357), IBM, Yorktown Heights, New York, 1986.

[Neu88] Gustaf Neumann. *Meta-Programmierung und Prolog.* Addison-Wesley, Bonn, 1988. (in German).

[Neu90a] Gustaf Neumann. Transforming interpreters into compilers by goal classification. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium*, pages 205–217. Department of Computer Science, KU Leuven, Belgium, 1990.

[Neu90b] Ulrich Neumerkel. Extensible unification by metastructures. In *Proceedings of META-90*, pages 352–364, Leuven, Belgium, April 1990.

[Neu92] Ulrich Neumerkel. Pruning infinite failure branches in programs with occur-check. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 172–177, St. Petersburg, 1992. Springer-Verlag.

[Nil84] J. Fischer Nilsson. Formal vienna-definition-method models of Prolog. In J.A. Campbell, editor, *Implementations of Prolog*, pages 281–308. Ellis Horwood, 1984.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λ PROLOG. In Kowalski and Bowen [KB88], pages 810–827.

[O'K90] R. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[Paa91] Jukka Paakki. PROFIT: A system integrating logic programming and attribute grammars. In Małuszyński and Wirsing [MW91], pages 243–254.

[Plü90] Lutz Plümer. *Termination Proofs of Logic Programs*, volume 446 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[Plü91] Lutz Plümer. Automatic termination proofs for Prolog programs operating on nonground terms. In Saraswat and Ueda [SU91], pages 503–517.

[PP89] A. Pettorossi and M. Proietti. Decidability results and characterization of strategies for the development of logic programs. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 539–553, Lisbon, 1989. The MIT Press.

[PP91] M. Proietti and A. Pettorossi. Unfolding-definition-folding in this order, for avoiding unnecessary variables in logic programs. In Małuszyński and Wirsing [MW91], pages 347–358.

[PW80] F.C. Pereira and D.H.D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *AI*, 13:231–278, 1980.

[Rie91] Günter Riedewald. Prototyping by using an attribute grammar as a logic program. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 401–437. Springer-Verlag, June 1991.

[RL88] Günter Riedewald and Uwe Lämmel. Using an attribute grammar as a logic program. In Deransart et al. [DLM88], pages 161–179.

[Rob71] J. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[Rob92] J.A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, 1992.

[Sah90] Dan Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 377–398. The MIT Press, 1990.

[Sah91] Dan Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, March 1991. Report TRITA-TCS-9101, 170 pages.

[SH90] Donald A. Smith and Timothy J. Hickey. Partial evaluation of a CLP language. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 119–153, Austin, 1990. ALP, MIT Press.

[Sha86] Ehud Shapiro, editor. *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, London, 1986. Springer-Verlag.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[Smi91] D.A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 62–71. ACM, 1991.

[SS86] L.S. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[ST89] T. Sato and H. Tamaki. Existential continuation. *New Generation Computing*, 6(4):421–438, 1989.

[Sto77] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* Cambridge, MA. The MIT Press, 1977.

[SU91] Vijay Saraswat and Kazunori Ueda, editors. *Logic Programming, Proceedings of the 1991 International Symposium*, San Diego, USA, 1991. The MIT Press.

[SZ88] P. Sestoft and A.V. Zamulin. Annotated bibliography on partial evaluation and mixed computation. In Bjørner et al. [BEJ88], pages 589–622.

[Tar92a] Paul Tarau. Low-level issues in implementing a high-performance continuation passing Prolog engine. Technical Report 92-02, Université de Moncton, Canada, March 1992.

[Tar92b] Paul Tarau. Program transformations and WAM-support for the compilation of definite metaprograms. In Andrei Voronkov, editor, *Second Russian Conference of Logic Programming 1991*, volume 592 of *Lecture Notes in Computer Science*, St. Petersburg, 1992. Springer-Verlag.

[TB90] Paul Tarau and Michel Boyer. Elementary logic programs. In Deransart and Małuszyński [DM90], pages 159–173.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Second International Logic Programming Conference*, pages 127–138, Uppsala, 1984.

[TS86] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In Shapiro [Sha86], pages 84–98.

[Tur86] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Ued86] Kazunori Ueda. Making exhaustive search programs deterministic. In Shapiro [Sha86], pages 270–282.

[Ued87] K. Ueda. Making exhaustive search programs deterministic : Part II. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 356–375, Melbourne, 1987. The MIT Press.

[Ung87] David Ungar. Self: The power of simplicity. *SIGPLAN Notices*, 22(12):227–242, December 1987.

[Ven84] R. Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation. In T. O'Shea, editor, *ECAI-84, Advances in Artificial Intelligence, Pisa, Italy*, pages 91–100. North-Holland, 1984.

[vWMP+75] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language algol 68. *Acta Informatica*, 5(1–3):1–236, 1975.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming (ESOP '88)*, volume 300 of *Lecture Notes in Computer Science*, pages 344–360, Nancy,France, March 1988. Springer-Verlag.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the 19'th Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[Wan80] M. Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, 1980.

[War77] David H.D. Warren. Implementing Prolog – compiling predicate logic programs, vol. 1 & 2. Technical Report 39-40, D.A.I., May 1977.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.

[WG91] D.A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T.P. Clement and K.-K Lau, editors, *Logic Program Synthesis and Transformation, Proceedings of the International Workshop LOPSTR91*, pages 205–220, University of Manchester, July 1991. Springer-Verlag.

[WS90] B. Wang and R.K. Shyamasundar. Towards a charcterization of termination of logic programs. In Deransart and Małuszyński [DM90], pages 204–221.

Acknowledgements

First of all I would like to thank eva Kühn who introduced me to Prolog in 1986 and who was my direct advisor for a year. Gustaf Neumann is to be mentioned for those endless discussions to further program transformation and many other things. Andi Krall who created the Vienna Abstract Machine in 1985 helped me a lot in understanding interpretive techniques. The derivation of the Vienna Abstract Machine by program transformation was the starting point of my investigations. Both, the notion of partially static goals and the technique for removing unnecessary variables in binary definite clauses were originally developed to 'understand' and derive this machine. The head of our department Professor Manfred Brockhaus always encouraged our work. I thank Franz Puntigam for many comments.

There is more to thank to many friends and colleagues for interesting discussions. Bart Demoen, especially for helping me to find inaccessible literature. Thanks go also to Dan Sahlin for patiently explaining me features of the Mixtus-system.

Curriculum Vitæ

Ich wurde am 25. Juli 1965 in Wien IX als Sohn des Walther Neumerkel und der Renate Neumerkel geboren. Von September 1970 bis Juli 1975 besuchte ich die Volksschule der Marianisten, Marianum in Wien XVIII Scheidlstraße 2. Von September 1975 bis Juni 1983 war ich Schüler der Albertus Magnus-Schule der Marianisten, realistisches Gymnasium, Wien XVIII Semperstraße 45. Am 10. Juni 1983 legte ich die Matura dortselbst ab. Dem österreichischen Bundesheer war ich vom September 1983 bis März 1984 als Präsenzdiener verpflichtet.

Mein Diplomstudium der Informatik an der Technischen Universität Wien, absolvierte ich von September 1984 bis 7. Juni 1989. Währenddessen absolvierte ich im Sommer 1986 ein Ferialpraktikum bei Phillips, Wien und von Juli bis September 1987 ein Praktikum bei IMAG (Institut d'Informatique et Mathematique Appliquè de Grenoble) am Laboratoire de Génie Informatique am Projekt GUIDE, Teil des ESPRIT Projekts 834 COMANDOS unter der Leitung von Professor Sacha Krakowiak.

Von Oktober 1987 bis August 1988 war ich Studienassistent innerhalb des Jubiläumsfondsprojektes Nr. 2791 unter Leitung von Dr. eva Kühn am Institut für Praktische Informatik, Abteilung für Programmiersprachen und Übersetzerbau. Von September 1988 bis Juni 1989 Studienassistent ebenda unter der Leitung von Professor Manfred Brockhaus. Von Juli bis September 1989 Vertragssassistent am selben Institut. Seit Oktober 1989 bin ich Universitätsassistent am Institut für Computersprachen, Abteilung für Programmiersprachen und Übersetzerbau.

```
is_term(var(Nr)) ←
      integer(Nr).
is_term(struct(F/N,Args)) ←
      atom(F),
      integer(N),
      is_term_list(Args).

is_term_list([]).
is_term_list([T|Ts]) ←
      is_term(T),
      is_term_list(Ts).

is_ground_term(struct(F,Args)) ←
      atom(F),
      is_ground_term_list(Args).

is_ground_term_list([]).
is_ground_term_list([T|Ts]) ←
      is_ground_term(T),
      is_ground_term_list(Ts).

is_substitution([]).
is_substitution([var(_) = T|S]) ←
      is_term(T),
      is_substitution(S).
```

Figure 7.10: Ground term representation

[ST89,Ued86,Ued87]

Converting CPS back to direct style. Danvy presents in [Dan92] a transformation back to programs without explicit use of continuations. Direct style may be more desirable as a readable version of a residual. The mechanism of continuation passing enforces a sequential left-to-right execution of the (AND-)continuations. Note that OR-continuations are still open to parallelization. The transformation of a continuation passing program back to direct style may simplify AND-parallel implementations.

[**?**] If (interesting) self-applicability is desired, we probably have to represent variables directly via object variables since one of the criteria for self applicability is that L mixt [sint,p] = p. So any program treated by the meta-interpreter should be mapped to the same program. The partial evaluator has to 'understand' the structure of the meta-interpreter.

In [WG91] Waal and Gallagher present a specialization of a general unification algorithm that works on a ground representation. We use this example to illustrate the expressiveness and simplicity of our approach. While the mentioned paper needs to resort to the techniques of abstract interpretation, we are able to remain in a framework based on fold/unfold techniques only.

[LS87]

They do not preserve the procedural semantics of [LS91b].

We will use the same representation for terms (Fig. 7.10) as given by them.

The example where [WG91] justify the application of abstract interpretation will be taken in order to show that static partial goals can accomplish the same result in a much simpler framework.

The goal to be specialized is formulated as ←Unify(x,y,s,s1), Ground(x), Ground(y) The goals Ground(x), Ground(y) are viewed *as constraints in the partial evaluation of* Unify/4. | It is not evident what constraints are ◁?? in this context. The extended algorithm presented in [WG91] is capable of removing infinite failures. As they say in the conclusion: *Further work to give more precise results [about strong equivalence], with proofs, is to be done.* We prefer to formulate the predicate to be specialized 'the other way round'. Our specialization only applies to such goals where there are already proof trees for the goals Ground(x), Ground(y). Strong equivalence is therefore not guaranteed for the original predicate. But in the following case strong equivalence can be guaranteed. I.e., unify_if_ground/4 will be always equivalent to unify_if_ground_specialized/4.

```
unify(TA,TB,S) ←
    unify(TA,TB,[],S).

unify(TA,TB,S0,S) ←
    deref(TA,NTA,S0),
    deref(TB,NTB,S0),
    unify_derefed(NTA,NTB,S0,S).

unify_derefed(var(V),var(V),S,S).
unify_derefed(var(VA),var(VB),S,[var(VA) = var(VB)|S]) ←
    dif(VA,VB).
unify_derefed(struct(F,ArgsA),struct(F,ArgsB),S0,S) ←
    unify_list(ArgsA,ArgsB,S0,S).
unify_derefed(TA,TB,S0,S) ←
    unify_and_bind(TA,TB,S0,S).
unify_derefed(TA,TB,S0,S) ←
    unify_and_bind(TB,TA,S0,S).

unify_list([],[],S,S).
unify_list([TA|TAs],[TB|TBs],S0,S) ←
    unify(TA,TB,S0,S1),
    unify_list(TAs,TBs,S1,S).
```

Figure 7.11: Unification of ground terms, Part I

```
unify_and_bind(var(VA),struct(FB,ArgsB),S,[var(VA) = struct(FB,ArgsB)|S]) ←
    ⊬ occurs_in_list(var(VA),ArgsB,S).

occurs_in_list(var(V),Ts,S) ←
    member(T,Ts),
    deref(T,NT),
    occurs_in_term(var(V),NT,S).

occurs_in_term(var(VA),var(VA),_).
occurs_in_term(var(VA),struct(_,Args),S) ←
    occurs_in_list(var(VA),Args,S).

deref(struct(F,Args),struct(F,Args),_).
deref(var(V),NT,S) ←
    dif(var(V),T),
    member(var(V) = T,S),
    deref(T,NT,S).
deref(var(V),var(V),S) ←
    ⊬ (   dif(var(V),T),
          member(var(V) = T,S)
      ).
```

Figure 7.12: Unification of ground terms, Part II

```
unify_if_ground(A,B,S0,S) ←
    is_ground(A),
    is_ground(B),
    unify(A,B,S0,S).

unify_if_ground_specialized(A,B,S0,S) ←
    is_ground(A),
    is_ground(B),
    unify_specialized(A,B,S0,S).
```

The scope of applicability of our specialization is therefore restricted to these and similar cases. If we are able to prove by fold/unfold that the unification predicate unify/4 will always terminate and if we could prove as well that

is_ground(T) terminates always (which is trivial if !is_term(T) is known), we could be even able to obtain the same result for a predicate with goals exchanged.

```
unify_then_ground(A,B,S0,S) ←
    unify(A,B,S0,S),
    is_ground(A),
    is_ground(B).
```

We are now giving the definition for our specialization:

```
unify_ground(A,B,S0,S) ←
    !is_ground(A),
    !is_ground(B),
    unify(A,B,S0,S).
```

The resulting predicate is definitely simpler than the original algorithm.

Note that we were happily able to match essentially the same patterns of recursion: Both predicates unify/4 and is_ground/1 contain a linear recursion over the argument lists, and another going down the subterms. There are no evil dependencies between these recursions. Except for the binding environment, which was in our case of no interest, since the transformation was driven by the simpler static goal !is_ground/1 and not by the more complex predicate unify/4 which uses an accumulator represented by S0,S. If that pattern would not have been the same, the simple fold/unfold-techniques we applied would not have been applicable!

Type analysis versus the propagation of static partial goals.

During execution of Prolog programs clause-indexing is an optimization to reduce the number of selected alternative clauses, when a goal is first selected. Many, especially tail recursive programs can be executed with clause-indexing similar to a functional program. Usually clause indexing is restricted to a limited number of arguments, in order to limit the size of the unification code.

Gallagher and Bruynooghe [GB90] have proposed to apply clause indexing statically. | ◁eher

Still there are cases not covered by [Plü91]. The following example contains insufficient information to give a termination proof. In our framework termination is immediately evident.

The predicate is working on a possibly open difference list. Still, termination is always guaranteed, because a double occurrence of is_vam/2 can be reduced via fold/unfold to a single one.

In most practical programs the terms propagated are rigid terms. Therefore the techniques of Plümer are applicable. But ensuring that all terms of interest are rigid, requires a costly global analysis of the program. Using static goals we are able to show termination with fewer assumptions.

The domain to be propagated needs to contain the complete information.

Let us assume that the static goal !is_list(L) has been propagated to the point where a dynamic goal is_list(L) occurs.

```
list_fail(L) ←
    !is_list(L),
    is_list(L),
    fail.
```

Using the fold/unfold-techniques introduced so far, will yield a new version.

```
list_fail([_|L]) ←
    !is_list(L),
    list_fail(L).
```

Now it is evident that list_fail/1 will either fail or loop infinitely.

Another way to prove failure is by defining a new predicate that contains the conjunction of the static and dynamic goals of interest. In the new definition, the static goals are treated as dynamic goals. In our case this would be:

```
list_list(L) ←
    is_list(L),
    is_list(L).
```

It is obvious that this definition can be reduced to the original definition of is_list/1. In any further derivation we are now able to remove the dynamic goal in a conjunction

```
    ...
    !is_list(L),
    is_list(L),
    ...
```

```
sumlist([],N,N).
sumlist([E|Es],N0,N) ←
      N1 is N0 + E,
      sumlist(Es,N1,N).

tree_list(nil,Es,Es).
tree_list(tree(E,A,B),[E|Es0],Es) ←
      tree_list(A,Es0,Es1),
      tree_list(B,Es1,Es).

sum_tree(T,N) ←
      tree_list(T,Ns,[]),
      sumlist(Ns,0,N).

sumlistnew(Es,Es,N,N).
sumlistnew([E|Es0],Es,N0,N) ←
      N1 is N0 + E,
      sumlistnew(Es0,Es,N1,N).

sumlistnew(Es,Es,N,N).
sumlistnew([E|Es0],Es,N0,N) ←
      N1 is N0 + E,
      sumlistnew(Es0,Es1,N1,N2),
      sumlistnew(Es1,Es,N2,N).

sum_tree_new(Tree,Es0,Es,N0,N) ←
      tree_list(Tree,Es0,Es),
      sumlistnew(Es0,Es,N0,N).

eval(number(N),N).
eval(NA + NB, N) ←
      N is NA + NB.
```

By folding multiple goals it is possible to merge loops in a program. The following program is taken from [PP91].

```
sum_size(T,S,N) ←
      sumtree(T,0,S),
      size(T,0,N).

sumtree(tip(E),S0,S) ←
      plus(E,S0,S).
sumtree(tree(LT,E,RT),S0,S) ←
      plus(E,S0,S1),
      sumtree(LT,S1,S2),
      sumtree(RT,S2,S).

size(tip(_),N,s(N)).
size(tree(LT,_,RT),N0,N) ←
      size(LT,N0,N1),
      size(RT,N1,N).

plus(A,0,A).
plus(A,s(B),s(C)) ←
      plus(A,B,C).

sum_size(tip(E),E,s(0)).
sum_size(tree(LT,E,RT),S,N) ←
      new(LT,E,S0,s(0),N0),
      new(RT,S0,S,N0,N).

new(tip(E),S0,S,N,s(N)) ←
      plus(S0,E,S).
new(tree(LT,E,RT),S0,S,N0,N) ←
      plus(S0,E,S1),
      new(LT,S1,S2,s(N0),N1),
      new(RT,S2,S,N1,N).
```

As an alternative we may use generic predicates that operate | ◁!!!

We are considering queried logic programs. A unique initial goal clause

goal trace OLDT-trees

OLD resolution with tabulation [TS86]. The principle is to prevent the interpreter from achieving the evaluation of a goal that was considered previously. In the OLD tree there are two kinds of different nodes: The active nodes that behaves as in ordinary resolution and the passive nodes (look up nodes) that looks for the solutions of a more general active node.

Specialization of an interpreter with respect to a concrete program

Specialization of an interpreter without a concrete program

Programs are still working essentially on the same structure

Given a program in binary form, the first step consists in splitting the principal functor of the predicate and all goals in the continuations into several nested functors. The functor $f(a_1 \dots a_n, C)$ is e.g., split into f_{extern}(... All variables have to appear again in the new atom. Note that we may duplicate variables. The same names for external and internal functors of different predicates may be used. This indeed will help us in the following steps.

Since the head and the goals are transformed with the same transformation we may exploit this redundancy by generalizing the continuation.

This allows us to apply stronger transformation techniques.

We should now be able to find in the continuation some functors in the continuation that share the same set of variables.

Formally we are defining a new equality relation,since now an individual (in our case the continuation functors) have more than one name: the original and the folded one.

Fortunately in our case of continuations we do not need to rewrite all predicates.

In usual programs the continuation is only read and written on a single level. No deeper unifications do occur in usual predicates. Especially predicates that are the result of a preceding binarization are of this simplified form.

Goals that are not involved in our transformation will stop our register allocation. In this case, we have several choices left. We can a) stop the complete allocation procedure, b) propagate the registers further although they are not needed at the risk of generating poor code, c) save the registers into the continuation.

Propagation of registers (b) can only be applied, if the goal is not a recursive call. As an example, consider that we want to parse another string within a DCG:

```
a ⟶
    [list(L)],
    phrase(nt,L).
```

Propagation of registers can only be applied if nt//0 does not depend on a//0.
The choices of a) and c) are always savely applicable and depend solely on efficiency considerations.
It is easy to give extreme examples for any of these cases.

```
cps_dcg([],Is,Is,Es,Es).
cps_dcg([L|Gs],Is0,Is,Es0,Es) ←
    list_difflist(L,Is0,Is1),
    cps_dcg(Gs,Is1,Is,Es0,Es).
cps_dcg([Goal|Gs],Is0,Is,Es0,Es) ←
    call(Goal),
    cps_dcg(Gs,Is0,Is,Es0,Es).
cps_dcg([error(E)|Gs],Is0,Is,[E-Is0|Es0],Es) ←
    cps_dcg(Gs,Is0,Is,Es0,Es).
cps_dcg([g(G)|Gs],Is0,Is,Es0,Es) ←
    cps_dcg_clause(G,Gs0,Gs),
    cps_dcg(Gs0,Is0,Is,Es0,Es).

cps_dcg_clause(G,Gs0,Gs) ←
    decorated_dcg_clause(G,BodyList),
    list_difflist(BodyList,Gs0,Gs).

list_difflist([],Xs,Xs).
list_difflist([_E|Es],[_X|Xs0],Xs) ←
    list_difflist(Es,Xs0,Xs).

decorated_dcg_clause(
        expr(tok(T)),
            [[tok(T)]|Gs],Gs).
decorated_dcg_clause(
        expr(tree(A,B)),
            [[op],g(expr(A)),g(expr(B))|Gs],Gs).


dmi_dcg21([], []).
dmi_dcg21([g(expr(tok(B)))|A], [tok(B)|C]) ←
    dmi_dcg21(A, C).
dmi_dcg21([g(expr(tree(B,C)))|A], [op|D]) ←
    dmi_dcg21([g(expr(B)),g(expr(C))|A], D).


H ←
    p(n_t,n_v),
    p(n_t,n_w),
    A_1,
    ...,
    A_n.
```

The introduction of metamorphosis grammars by Colmerauer in 1975 was the first step in making Prolog a higher level grammar description tool. Definite clause grammars (DCGs) [PW80] introduced in DEC-10 Prolog are a special case of MGs. The rules are restricted to a single nonterminal symbol on the left hand side. Since the beginning [Col78] a DCG is directly translated directly into a Prolog program. Every DCG-rule is expanded to a Prolog rule by associating to each nonterminal symbol the current state with a difference list. Terminals are treated like nonterminals by introducing a nonterminal e.g., 'TERM'/3 defined as 'TERM'(T,[T|L],L).

```
a ⟶
    b,
    c,
    d.
```

```
a(L0,L) ←
    b(L0,L1),
    c(L1,L2),
    d(L2,L).
```

Another more recent approach (suggested in e.g. [SS86,Mal87] to the implementation of DCGs consists of/in writing a meta-interpreter.

and specializing this meta-interpreter with respect to a concrete grammar.

Specialization is either driven directly by the meta-interpreter and user annotations yielding a compiler that is able to translate grammars directly [Neu90a] or by partial evaluation [LS91a]. The results are —in all publications— identical to the traditional translation presented.

The resulting Prolog clauses contain all unnecessary variables.

These unnecessary variables cause severe problems for both execution efficiency and program transformation.

The efficiency problems can be seen by comparing the parsing process implemented by the rules above to an analogous recursive descent parser in an imperative programming language:

While in a procedural language there will be one global state to represent the current input stream (or list) to be parsed, Prolog has to pass the arguments from one goal to the next.

A lot of work on improvements in Prolog implementations is therefore devoted to overcome these disadvantages.

Joachim Beer [Bee88] has proposed an extended (and more complex) WAM that supports the propagation of uninitialized variables by a new internal tag.

Source to source optimization of invariant arguments by binarization

We describe a source to source transformation that

The speedups obtained range from 20% to 50% on current WAM-based machines. On a binary Prolog machine our transformation will always be faster and will yield smaller heap consumption.

Our transformation results in fewer variable bindings, less trail consumption and (if implemented) fewer occur-checks.

Our transformation results in a new translation scheme for definite clause grammars.

Consider the case of a DCG-parser. The result of our transformation is quite similar to this classical implementation technique: The shared input/output-argument pairs that are used to pass data from one goal to the next are mapped into a single pair which is kept in argument registers for the whole computation.

Passing arguments through the goals in a rule is usually implemented in an inefficient manner compared to procedural programs.

Following the idea of avoiding unnecessary variables as much as possible How is it possible to avoid unnecessary variables as much as possible?

The transformation

performed by a user annotated syntactic transformation that generates

Note that our transformation would have been much more complicated if the classical nonlinear representation of Prolog bodies would have been used. If the conjuncts are nested it is more difficult to stop the process of evaluation.

Another way to understand the compilation of DCGs into a more efficient binary version is to apply the transformation directly at the DCG level. The resulting Binary Definite Clause Grammar (BDCG) program can then be mapped directly into a binary program.

While the predicate '$demo'/1 is used by [Tar92b] when transforming definite clauses to binary definite clauses we will use the nonterminal '$phrase'/1.

@ Horizontal compilation.

To increase efficiency, procedures are written with assumptions about how they are called.

!!! Burstall-Darlington-example

[Hui90]; [WS90] Termination;

[MNL88] Most specific programs;

The equivalence between programs is one of the most important relations between programs. Reasoning about programs and in particular specialization (or optimization) of programs relies on the definition of equivalence.

Partial evaluation transforms, reflection evaluates

Application for production systems [FFS89],

Type systems rely on the interpreter and mostly not the concrete program.

Binding time analysis finds a division δ given some initial division for the input parameters that divides every function call or constructor into those that can be executed statically and those to be executed dynamically.

Also [Dem92] mentions as an example the occurrence of unnecessary variable in the body of a clause. He suggests to specialize every goal (from left to right) in the body with respect to their continuation. As we have shown in ... this method cannot remove such variables in general. Furthermore, in the case of difference lists, DCGs and other techniques simulating a global state the code size will explode, without too big advantages gained.

To summarize our experience with optimizing DCG-programs we can expect to gain a realistic speedup of 10% to 20%. There is however a big potential for more advanced grammar formalisms that can profit from our optimization.

A similar observation is made in the area of partial evaluation: Existing hand-crafted programs yield very small speedups, while more generic programs which are already written with partial evaluation in mind result in residuals comparable to the specialized hand-crafted versions. We are therefore proposing some extensions to DCGs that cause no overheads for the regular case compared to simple DCGs provided that our optimization is applied.

Venken [Ven84] describes one of the first implementations [ST89,Ued86,Ued87]