Propagation techniques in WAM based architectures

The FIDO-III Approach

Hans-Günther Hein

Contents

List of Figures

Abstract

In the paper we develop techniques to implement finite domain constraints into the *W*arren *A*bstract *M*achine (WAM) to solve large combinatorial problems effciently. The WAM is the de facto standard model for compiling PROLOG. The FIDO system ("*FI*nite *DO*main") provides the same functionality as the finite domain part of CHIP.

The extension includes the integration of several new variable types (suspended variables, domain variables and suspended domain variables) into the WAM.

The "firing conditions" are lookahead and forward control schemes known from CHIP. We have developed a constraint model where the constraint is divided into constraint initialization code, constraint testing code and constraint body. Furthermore, we supply a deeply integrated WAM builtin to realize the first fail principle. Besides the summary of the important theoretical results, the specification of the compilation process in the WAM Compilation Scheme is given.

We also present a simple graphical analysis method to estimate the computational burden of lookahead and forward constraints.

Part I

Basic Concepts

Chapter 1

Introduction

In this chapter we will informally introduce the reader to the field of constraint logic programming. To understand the added structures and concepts implemented in the WAM — Warren's Abstract Machine ([War83]) —, we need to look first at constraints and at the language to compile. I have been patient that a reader not familar with WAM and constraint logic programming will be able to read this chapter without difficulty and get the points and problems I have worked on. For the following chapters, a good knowledge of the WAM and constraints is mandatory. The unfamilar reader is referred to [AK90, Hen89].

1.1 Logic Programming and Constraints

1.1.1 Logic Programming

The logic programming paradigm originates from theoretical work done by Kowalski ([Kow74]) and by Colmerauer, the latter used it for natural language analysis. Kowalski discovered that horn clauses — a subset of first order logic — can be given a procedural interpretation. The first interpreter of PROLOG is due to Roussel [Rou75]. The first compilative approach has been presented by D. H. D. Warren ([War77]). It used structure sharing which turned out to be not optimal. In 1983 ([War83]), a WAM with structure copying was presented, which is the basis for *all* efficient implementations of PROLOG today. Since there is no official PROLOG standard (although there is currently an ISO standard draft circulating), many flavors of syntaxes and different semantics (of mainly builtin predicates) can be identified.

Since the language has a simple procedural and declarative semantics — in contrast to *most* other languages — many approaches are done to extend its expressive power preserving the semantic properties. Clear procedural and declarative semantics seem to be a k.o.-criterion for extensions. In many cases, the language is modified to include some sort of constraint processing. According to the CLP scheme (see section 2.2) a structure with some properties is enough to maintain simple semantics.

After the Japanese selected PROLOG as the basic language for their Fifth Generation Project, the logic programming paradigm has received much interest from the computer science community. The latest boost of attention results from database and AI researches realizing the limitations of SQL and the appealing power of the logic paradigm resulting in the "deductive databases" trend.

But not only semantics and "political" reasons make the language attractive: Problems can be stated easily in a simple, natural, elegant, declarative and relational form. The builtin search

strategy frees the programmer from writing an own search procedure. Furthermore, programs are small and compact shortening the development time.

The drawbacks and criticism deal with the unsound negation and the simple control strategy (SLD-resolution[Llo84]).

Concerning combinatorial problems, the programmer is often seduced to write down the program in a generate and test strategy: Having a problem with n variables, one of the possible values is bound to one variable and checked against the constraints. If the choice of a value results in a failure, backtracking occurs and the next possibility is tried.

In the following we will use a standard example to reveal the problems: On a (quadratic) chess board with a $n \times n$ square field n queens must be placed in a way that no queen can attack another queen.

The straightforward approach is to take n variables X_i $(1 \leq i \leq n)$ representing the different columns and bind them successively to values v_j $(1 \leq v_j \leq n)$ denoting on which row v_j the queen in column i has been placed. Taking 8 queens, we have 8^8 (approximately 16 millions) possibilities to generate in an exhaustive search. When using this approach, we can only dream of finding a solution for e.g. 100 queens.

Using backtracking, we immediately invoke those tests which can be checked. Thus when taking a queen in column i we have to check all conditions where instantiated variables are present. Using this strategy, an a posterori search space pruning is achieved. However, when placing queens only regarding the "constraints" on the left hand side, we can make choices for subsequent columns (on the right hand side) impossible without detecting these conditions. The failure *occurs lately* when the conflicting variable is to be instantiated and failures for all "possible" values appear.

Partial solutions appear in the queens example, when some variables have been instantiated and an arbitrary free variable can only get one value. If we are not immediately instantiating this variable, we are going to *rediscover* this partial solution over and over again. If the variable which could be bound deterministically is then conflicted by another subsequent choice, we are having unnecessary choicepoints and a lot of "uninformed" failures. In Fig. 1.1 the next variable[1] to be instantiated is X_4, although in column 6 (X_6) the only possibility to place the queen is deterministic in this configuration and now being conflicted by the choice of X_4. This is noticed when the other variable has been instantiated, rediscovering the partial solution several times by backtracking.

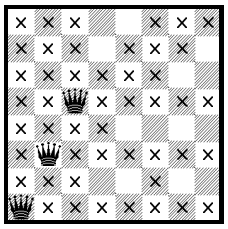


Figure 1.1: Queens example: Partial solutions are not detected

The backtracking version of the program is less declarative than the generate&test program. The programmer has to take care of the tests, especially *when* they must be performed.

By restricting the view to the left hand side we still have a lot of useless nodes in the search tree, which a human would not consider.

[1] The "x" in the figure are places known to be impossible to use.

1.1.2 Constraints

Since [Wal72] constraint satisfaction methods have been applied in many complex combinatorial areas. These problems can be characterized by a set of variables X_i which range over domains D_i. The assignment of the *constraint* variables is in general not a single value, but elements from 2^{D_i}. Constraints are (constraining) relations $R_j(X_1, \ldots, X_n)$ and the set of constraints forms a constraint net. Together with initial values of $X_i \subseteq D_i \neq \emptyset$ this is called a constraint problem. A single solution is an assigmnent $\sigma : V_i \rightarrow D_i$ for all variables X_i such that all constraints R_j are satisfied simultaneously. Finding the set of all solutions is known to be NP-complete. However, we are interested in an approximation $X_i \subseteq D_i$ of the variables removing inconsistent values (values which can not be part of *any* solution). Using consistency algorithms to remove incompatible values is polynomial ([MF85]). In Logic Programming we are interested in local consistency which assures that no incompatible values are in the domain variables of a relation ([Hen89]).

Graphically, these constraints can be visualized as a hypergraph (e.g. see fig. 1.3), where the nodes represent the variables with their domains and the hyperarcs denote the relations. In the above eight-queens example, a human solver would not restrict his attention to the left hand side of the queen placed, but mark the impossible positions as shown in fig. 1.1. Thus we are having a set of 8 variables ranging over the finite domain $\{1,2,3,4,5,6,7,8\}$ and wipe out the impossible values. We actually "remember" the constraints and apply them as soon as *enough* information is present. If we are placing a queen in the some row, we wipe out the "column" in the other queens' domains and then wipe out the diagonal places of those queens the placed queens can attack.

Thus we collect the tests first, and apply these constraints when a variable is instantiated. More or less, we are using constraint propagation: We are having relations between variables ranging over a finite domain. These domains are propagated when enough information is present.

The source code of the FIDO program for the eight queens example can be found below (fig. 1.2). Please note that the `queens` predicate calls the `safe` predicate first, whose action is to build the constraint net before instantiating the variables. The initial constraint graph which is present after the invocation of `safe` is shown in fig. 1.3. The constraint graph after having placed the third queen is shown in fig. 1.5. The dotted lines represent those constraints which became active after the 6^{th} queen was placed deterministically. The hatched lines are "dead" constraints[2]. All steps to the first solution of the eight queens problem using constraints are summarized in appendix B.

1.2 What is the WAM ?

The WAM [War83] is an ingenious machine model for executing logic programs using techniques of conventional languages compilation combined with structures for nondeterministic behaviour. In order not to be restricted to a particular "hardware philosophy", abstract machines are designed for the compilation of a specific language and do not obey real-world restrictions. For example, the X-registers in the WAM are used to hold arguments when calling procedures. A *real machine* may e.g. have 32 of them. In the abstract model an infinite set of X-registers is assumed and nothing is said about the handling of procedures which have

[2] "Dead" constraints are relations which have been invoked ("fired") and a successice invocation of the constraint can not yield any new information.

```
queens(X) :-
   X is
   [X1:{1,2,3,4,5,6,7,8}, X2:{1,2,3,4,5,6,7,8},
    X3:{1,2,3,4,5,6,7,8}, X4:{1,2,3,4,5,6,7,8},
    X5:{1,2,3,4,5,6,7,8}, X6:{1,2,3,4,5,6,7,8},
    X7:{1,2,3,4,5,6,7,8}, X8:{1,2,3,4,5,6,7,8}],
   safe(X),
   labeling(X).

safe([]).
safe([X]).
safe([X, Y|Z]) :-
   noattack(X, [Y|Z]),
   safe([Y|Z]).

noattack(X, Y) :- noattack(X, Y, 1).
noattack(X, [], _).
noattack(X, [H|T], N) :-
   regular(X, H, N),
   N1 is N + 1,
   noattack(X, T, N1).

regular(X, Y, N) :-
   forward(Y =\= X),
   forward(Y =\= X + N),
   forward(Y =\= X - N).
```

Figure 1.2: FIDO source code for the N queens problem

more arguments than X-registers available. In fig. 1.4 the internal structure of the WAM is shown in the darkened box. The registers in hatches boxes are the extensions we need for FIDO. Usually, running out of registers is solved by raising a compilation error, although for *most* programs this "limitation" seems to be no real obstacle.

As in conventional languages local variables are stored in the environment frame of the called function or procedure. This is the same in the WAM architecture, which provides environments referenced by the register `E` and holding so called "Y-variables".

In conventional langauges a heap is present for data structures that are dynamically allocated. The same is true for the WAM: The register `H` points to the top of the heap which contains structures, lists and goal trees.

The real extensions are a trail stack and choicepoint entries in the local stack. Choicepoints on the local stack represent the non-determinism and administrate which clause of a procedure must be invoked next. These choicepoints are referenced by the register `B`.

When instantiating variables in PROLOG, binding of variables must be undone upon backtracking. This information is stored on the trail stack whose top is accessed by `TR`.

When compiling PROLOG, the unification algorithm is basically *specialized* according to the arguments occurring in the atoms. Recent global compilation techniques ([Tay90b, Tay91, Roy90]) with mode analysis, variable type inference and determinism analysis are capable of generating code which is sometimes more efficient than C code.

1.3 Remarks

Of course, we have not found any miracle to solve NP-complete problems. We still have an exponential behaviour not only in the (obvisious) generate & test scheme but also in using backtracking and constraint methods. However, with more sophisticated methods we can solve *larger* problems. Especially when applying constraint methods we can solve surprisingly huge problems. In general, we are in most cases interested in *one* solution satisfying certain constraints. The constraints do active computations restricting variables — or even better — instantiating variables deterministically. By using local information between variables in an active manner, we reduce the search space, perform an informed search and obtain global consistency with choice methods.

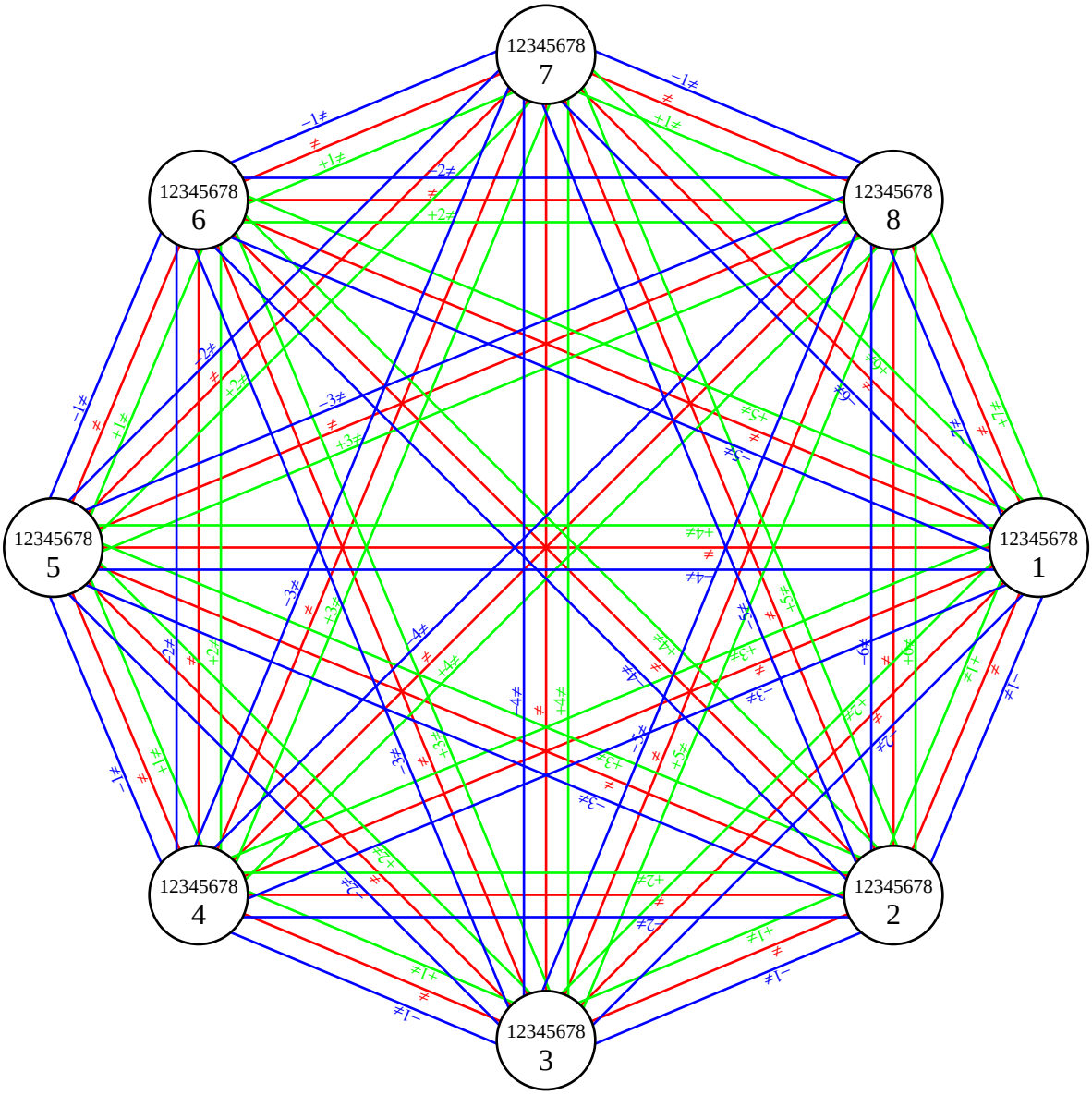


Figure 1.3: Initial constraint graph of the eight queens example

Looking at Kowalski's ([Kow79]) equation "algorithm = logic & control" the standard search method is not adequate for combinatorial search problems. Using backtracking search, the burden of control is put on the programmer loosing portions of its declarative specification. The finite domain extensions of PROLOG mainly alters the *control* aspect of PROLOG for a large class of search problems by a kind of data-driven computation.

1.4 Relations to the ARC-TEC project

Within the ARC-TEC Project (*A*quisition, *R*eprsentation and *C*ompilation of *Tec*hnical Knowledge) ([BBH+91]) the compilative expert system shell COLAB ([BHHM91]) has been developed. Its four main components include forward reasoning, backward reasoning with a functional extension, taxonomical reasoning in a KL-ONE-like manner (restricted to a decidable subset) and hierarchical finite domain constraint satisfaction. These components are loosely

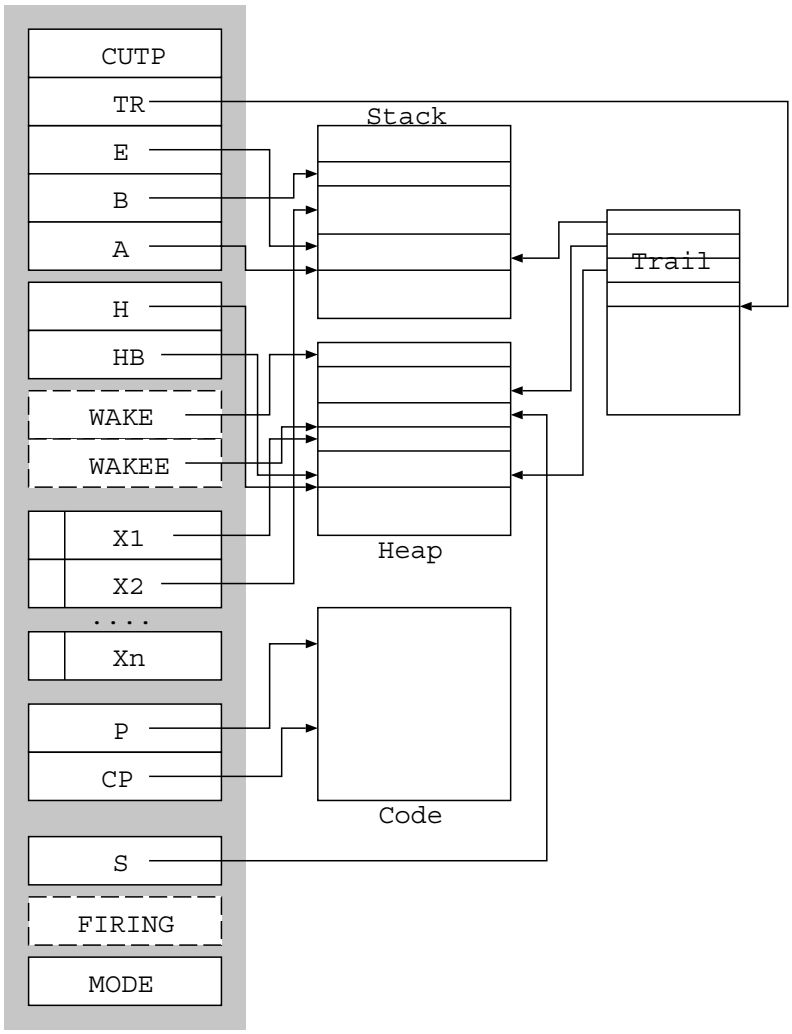


Figure 1.4: The internal structure of the WAM

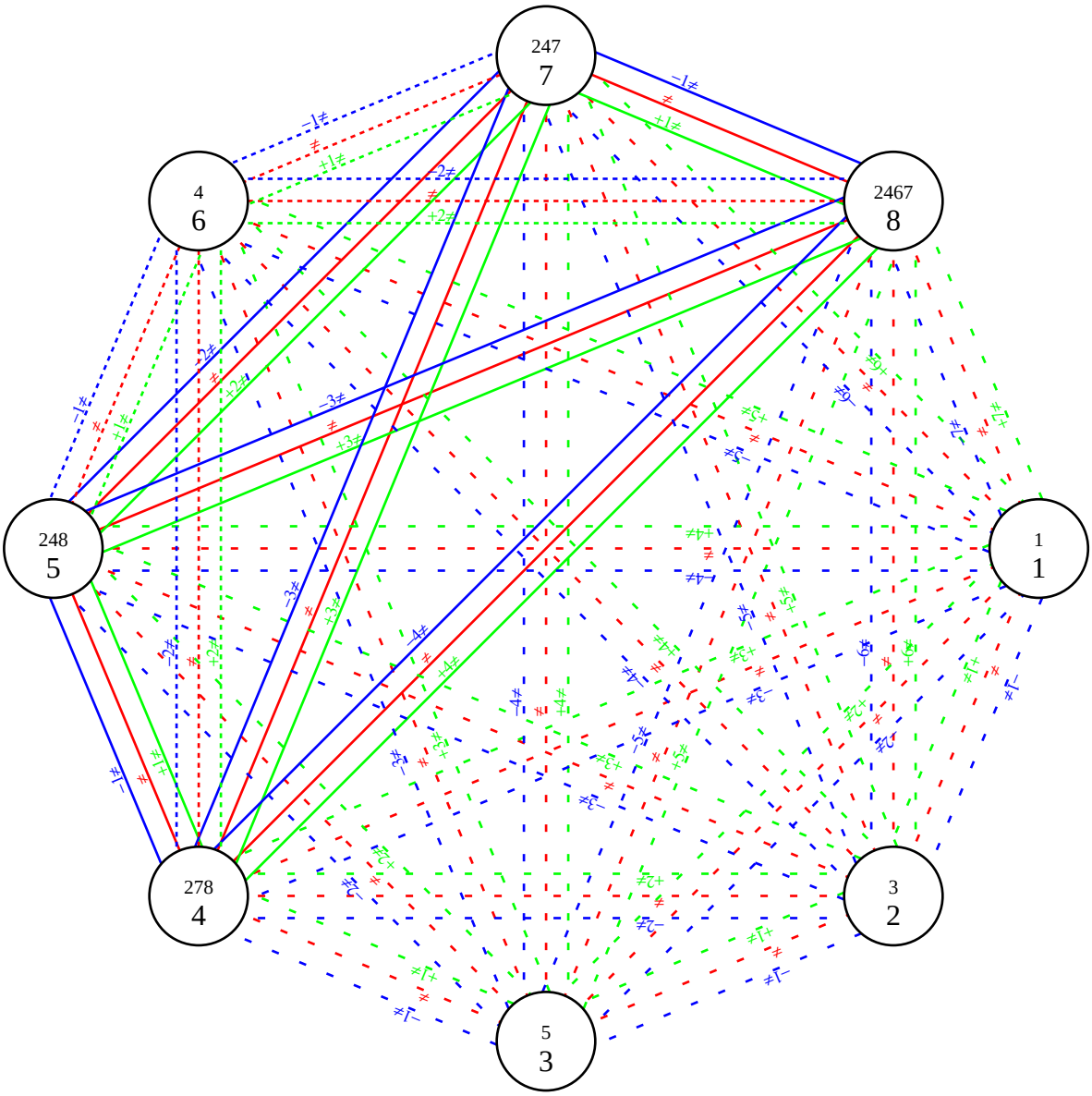


Figure 1.5: Constraint Graph after 3^{rd} placement of a queen propagating the 6^{th} queen

integrated forcing the programmer to state which reasoning component must be invoked and *when* this must be done. The latter points are opposed to the declarative challenge.

In this paper I integrate two formalisms: The backward reasoning part (without the functional extension) and the (to finite domains restricted version) of the constraint part.

1.5 What are the problems ?

Compiling PROLOG programs into the WAM results in *sequential* code. The birds eye's view on the code is specialized unification instructions generated for the head of the clause and instruction for setting up argument registers and *calling* these literals[3] in the body of the clause.

[3]Please remember the procedural interpretation of PROLOG.

The problems are:

- We must break up the sequentiality: When a constraint is applicable, we must interrupt the work determined by the sequential code and handle the woken constraint.
- We have to discuss how to represent the constraint graph (see fig. 1.3 for a sample initial constraint graph) in memory and the administration of it: A constraint is waiting for a condition so it can "fire". After it has fired, it may be "dead" (forward constraints) or it may be woken subsequently, if a domain variable is touched again (lookahead constraints)[4].
- Dead constraints have provided their information when they have "fired" but as soon as backtracking occurs to a point they were alive this "firing information" is lost and the constraint *must be woken again*. This is a major difference to the COLAB type of integration. Consider the queens example again: After having placed the third queen as already seen in the example above (fig. 1.1) the resulting constraint graph can be visualized in fig. 1.5. The dashed lines are dead constraints (which must be rewoken on backtracking) and the dotted lines at node 6 are activated constraints which fire since the 6^{th} queen can only be placed at row 4.
- In the COLAB viewpoint, constraints should be static so the information is present at the start of the computation. Although it is a good programming practise (see the queens example above: we stated the constraints first and then generated solutions), we do not restrict ourselves to static constraints. Constraints may be issued arbitrarily at runtime[5], and must be deleted when backtracking is done beyond the time/place of their creation.
- We must support some builtins to handle the extensions. When possible, the builtins have been written in WAM code to ensure an easy modification or even writing them in FIDO itself. However, some basic builtins are necessary:
  - We must have a user access to the domain of the variables to implement choice (labeling) methods.
  - We have to extend the machine with consistency procedures assuring *local* consistency. In constrast to CONTAX ([Mey92], we are not supporting global consistency. This must be satisfied by the labeling procedure. CHIP has a set of builtin finite domain operators $(+, -, /, *)$ and predicates $(=, \neq, \leq, \geq, <, >)$. We can write down these constraints in LISP by giving a lambda expression holding the constraint to special consistency procedures. Furthermore, we provide extensional constraints.
  - Finally, the first fail principle is implemented as a primitive routine.
- When we come to a final success, there may still be "alive" constraints in store. This is called floundering. CHIP produces a failure when such a situation takes place. We experimentally have chosen the alternative to print out the bindings of the user's variables and the collected "alive" constraints. However, this may lead to unsound answers. The user should notice that the answer is only correct, if the stored constraints can be satisfied.

[4] A lookahead constraint is dead, when there is at most one domain variable.

[5] This is mandatory for disjunctive constraints implemented as constraint as choices.

1.6 Contributions

Besides the Boisumault ([Boi86]) implementation of two special control builtins (`dif` and `freeze`) in an interpreter based environment, the WAM-based modification of control was due to Carlsson ([Car87]). We are building upon his result and extending towards the control primitives we need for lookahead and forward constraints. The language and concepts of CHIP have been intensively described by [Hen89]. The implementation techniques and structures of CHIP have *not* been revealed. The FIDO team in the ARC-TEC project ([MMS91] investigated meta-interpretation [Sch91] and horizontal compilation [Mül91]. Vertical compilation into the WAM and the extended WAM itself will be presented here. The FIDO compiler is currently under development [Ste92].

1.7 Overview

In chapter 2 we give an overview of the state of the art of constraint systems and implementations. Some interesting applications are outlined. Chapter 3 will give the underlying theory and definitions needed to implement the system. In chapter 4 the terminology and basics of the utilized WAM are given. In chapter 5 we will discuss the model of `freeze`, from where we start our extension. We will use this simple model to outline an implementation close to this scheme and discuss its weakness in chapter 6. In chapter 7 we will develop some new concepts and redesign our WAM ommitting a lot of unnessesary work. Compilation is specified in chapter 8. It has been a challenge to make the FIDO extensions fit in our WAM compilation scheme [HM92] with only minor modifications. In chapter 9 we give the internals of the propagation routines for the constraint bodies and review the WAM basic builtins for implementing choice methods and first fail. In chapter 10 we present a simple analysis technique well suited to state dynamic properties of programs. The summary in chapter 11 concludes the paper.

Chapter 2

Logic Programming and Constraints

In chapter 1, we gave an informal introduction to the constraints present in FIDO. However, constraints in PROLOG means much more than finite domains. In this chapter we will present the state of the art in logic programming constraint systems. Where FIDO (and CHIP) are good for solving large combinatorial constraint problems, other constraint systems are tailored for special contexts (e.g. verifying circuits, modelling mathematical connections, integer programming, etc.).

2.1 Constraint Domains

Constraint languages differ in their computational domain and methods to ensure consistency or to solve constraints. The following enumeration will be outlined and those areas related to FIDO will be examined in more detail.

- finite domains and propagation techniques
- PROLOG III constraints and saturation and SL-resolution
- boolean unification or boolean Groebner bases
- intervals and propagation techniques
- algebraic domains using Groebner bases.
- linear (in)equalities — Gauss and Simplex

Systems with boolean unification use implementation strategies extending the unification algorithm yielding theory unification [MN86, BS87, BR89, Bry86]. A generalization of boolean algebras are finite algebras([Büt88, Fil88]) which can be utilized to verify hardware circuits where more than two values are to be considered. The power of this approach has been demonstrated by solving the Lion and Unicorn Puzzle in the constraint system integrated in PROLOG-XT ([SSF89])

2.2 The CLP Scheme

The Constraint Logic Programming Scheme is a framework for a class of logic programming languages replacing unification by constraint solving a over special *strutures A* [JM87, JMSY90, JL87]. They still obtain simple declarative and operational semantics. Constraint solving is not restricted to equations.

Definition 1 *A structure A consists of:*

- *the domain of A: $D(A)$.*
- *a set Σ $(A^n \rightarrow A)$ of n-ary functions.*
- *a set Π $(A^n \rightarrow \{true, false\})$ of n-ary relations with equality.*

Definition 2 *A-Terms are either variables or functions $f(X_1, \ldots, X_n)$ where the functor $f \in \Sigma$, and the arguments X_i are A-terms* [1].

Definition 3 *A-constraints have the following form: $c(X_1, \ldots, X_n)$ where $c \in \Pi$ and X_i are A-terms.*

Definition 4 *An A-atom is of the form $p(X_1, \ldots, X_n)$ where p is a predicate symbol $p \notin \Pi$ and the X_i are A-terms.*

Definition 5 *A solution-compact structure allows to approximate elements in $D(A)$ by a possibly infinite set of A-constraints. Two different elements in D(A) are separable by two disjoint finite sets of A-constraints.*

A logic programming language CLP(A) is obtained by specifying such a solution-compact structure. Finite and inifinite trees are examples of solution-compact structures. PROLOG[2] can be viewed as a an instance of the CLP-Scheme with finite trees.

Definition 6 *A CLP(A) program consists of a finite number of rules of the form:*
$A_0 :- C_1, \ldots, C_n, A_1, \ldots, A_m$ where A_i are A-atoms and C_i are A-constraints.

Definition 7 *A CLP(A) goal is of the form:*
:-$C_1, \ldots, C_n, A_1, \ldots, A_m$
where A_i are A-atoms and C_i are A-constraints.

Definition 8 *Let goal G_1 have the form: :-$C_1, \ldots, C_n, A_1, \ldots, A_m$ (C_i are A-constraints, A_i are A-atoms)*

and a rule in P selected by a search strategy:

$B_0 :- D_1, \ldots, D_k, B_1, \ldots, B_l$ where B_i are A-atoms and D_i are A-constraints.

Let $C_1, \ldots, C_n, D_1, \ldots, D_n$ be solvable. Let A_i be the selected atom by an atom selection rule[3].

Then the derived goal G_2 looks like:

:-$C_1, \ldots, C_n, D_1, \ldots, D_k, A_i = B_0, A1, \ldots, A_{i-1}, A_{i+1}, \ldots A_m, B_1, \ldots, B_l$.

A sequence of derivation steps is called a *derivation sequence*. A goal is an answer constraint, if a finite derivation sequence contains only A-constraints.

[1] Constants are 0-ary functions.
[2] with occur-check.
[3] Typically the first A-atom is selected.

2.2.1 CLP($\mathcal{R}$)

CLP($\mathcal{R}$) is an instance of the CLP scheme based on the structure of reals and trees of reals: real constants and variables are real terms and if t_1, t_2 are real terms, then $(t_1 + t_2), (t_1 - t_2), (t_1 * t_2), (t_1/t_2)$ are also real terms. If t_1, t_2 are real terms, then $t_1 = t_2$, $t_1 > t_2$, $t_1 \geq t_2$, $t_1 < t_2$ and $t_1 \leq t_2$ are real constraints.

It is a system for solving constraints in the domain of uninterpreted functors over real (linear) arithmetic terms. The underlying constraint solver is a "sort of" simplex/gauss algorithm — being modified to allow incrementally adding constraints[4] and deleting constraints upon backtracking.

The CLP($\mathcal{R}$) system is an approximation of the CLP scheme: Since the simplex-based algorithm can not determine the solvability of non-linear constraints, they are delayed until sufficiently instantiated, e.g. other (linear) constraints make variables in the non-linear constraint and thus eventually linear.

2.2.2 Implementation

The structure of the CLP($\mathcal{R}$) system can be seen in fig. 2.1. CLP($\mathcal{R}$) is a compiler-based system translating into extended WAM code. The constraint solver is a seperate module in the design. For the sake of efficiency, simple constraints are handled by the machine or the solver interface directly. The constraints are accumulated in the constraint solver store. The constraint solver is subdivided into a linear constraint solver (using Gaussian elemination) and an inequality solver (using simplex algorithm). Nonlinear constraints are delayed until they become linear.

The solver is (in theory) completely independent of the underlying WAM. A new computational domain is implemented by replacing the solver for reals with the solver for the new computational domain. Although the WAM structures are quite standard, the solver structures (and the interaction) are heavily optimized, since the solver must have the following property: It must be incremental and it should be fast. Considering the interaction between solver and WAM, the runtime behaviour of the system is remarkable.

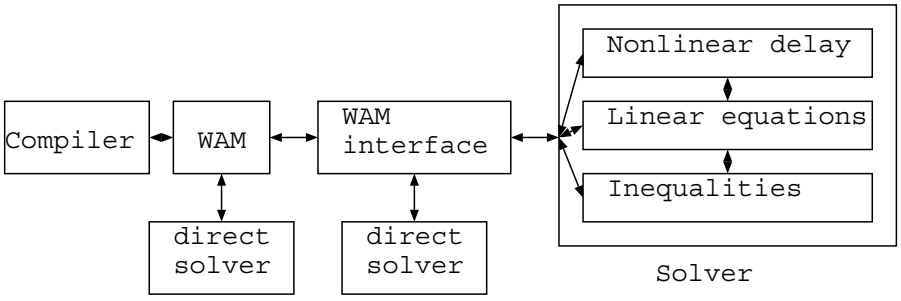


Figure 2.1: The overall design of the CLP($\mathcal{R}$) system

2.2.3 CLP($\mathcal{R}$) applications

The applications done with the CLP($\mathcal{R}$) system are problems with have a "strong mathematical problem structure" enriched with an underlying search problem. An incomplete list of

[4] Incrementality means that upon issueing a new constraint to the constraint store, the test procedure for solvability must *not* reconsider the whole constraint store.

published applications follows:

- electrical engineering [HMS87]
- molecular biology [Yap91]
- option trading analysis system [HL88]

2.3 Metavariable Approach

The only thing a user can not easily extend in PROLOG systems is unification. The idea of metavariables ([Hol90]) is to have a specific structure in the system, which is handled not as an ordinary structure, but in a very different way: Whenever the unification routine touches a metastructure, user defined unification handlers (in PROLOG) are called. Although the metaterm is seen as a variable, it can have attributes bound to it (e.g. a domain). A special built-in predicate allows to re-bind the attributes, so domains can be easily made "smaller".

This concept has been implemented in SEPIA, a PROLOG system devoloped at ECRC[ECR91, MAC+89]. Together with SEPIA's high level predicates for handling woken procedures, a finite domain extension of unification can be easily implemented. However, this skillful method may impose some problems: The unification routine — which should be considered as an important basic routine — should be made as fast as possible. In hardware-based implementations, this routine is a very good candidate for the implemenation in microcode. It should be a difficult problem to interrupt an (atomic) instruction and call a user handler from a microcoded routine and return back to the microcode — if possible at all. However, taking the supposed death of specialized hardware and the trend towards a "truly general purpose processor"[DNT91] into consideration, this is not a strong argument.

But since such a handler can arbitrarily use WAM registers we must save the complete WAM state, because we do not know, which registers are used[5]. The user handler must not be restricted to be deterministic: It may create choicepoints so even non-unitary unification algorithms become possible. The average user might not be able to determine when a choicepoint is (unnecessarily) created in a PROLOG program and thus in a unification handler[6]. All in all, this "unification hook" is a powerful extension for the specialist. In constrast to our work, we use an in depth integration of the domain extensions in the unification routine and we do *not* invoke constraints within the unification routine. Instead, we call constraint processing when we know, how many argument registers are alive. Furthermore, the approach is restricted to depth-first handling of constraints. In our system the constraints are invoked in the order in which they were issued to the system. When switching over to infinite domains like intervals this fairness aspect may become important.

2.4 Temporal PROLOG

Temporal PROLOG ([Hry88]) is a constraint system loosely coupled to a PROLOG system. The constraint domain is the one of Allen's temporal intervals([All83]). In contrast to finite domains and interval domains, the nodes of the constraint graph do not have sets of possible

[5] If the machine has 32 X-registers and e.g. 5 are used, all 32 registers have to be saved. This might even complicate garbage collection.

[6] The best thing should be looking at the WAM code.

values, but the relations (the arcs) have a set of possible relations ([Ric89]). The system is interfaced to a constraint system written in C for efficiency reasons and is seperated from the machine. Allen's temporal logic is a widely accepted knowledge formalism for representing time in areas such as planning, qualitative simulation, natural language or databases.

2.5 Logical Arithmetic — Interval PROLOG

In Logical Arithmetic [Cle87] variables can have intervals as their computational domain. Open, closed and half closed intervals can be represented. This can be considered as a sort of infinite domains. The problem is how to detect when a variable has become a singleton.

Another problem is that intervals have to be split and considered as an alternative thus resulting in the creation of choicepoints. The constraints which can be expressed must not be linear. By using their propagation techniques quite interesting results can be obtained. Although intervals can be represented in CLP($\mathcal{R}$) (e.g. $X > 1.5, X < 1.9$) the Interval PROLOG system can infer much more "information" when dealing with nonlinear constraints. However the system behaviour of solving a set of linear equations is exponential in the number of variables. While CLP($\mathcal{R}$) solves the constraints symbolically ("qualitative resonning"), this system is based on interval numerics doing a sort of "quantitative reasoning". An excerpt from the running system can be seen in fig. 2.2. The problem can also be seen. Although the solution is close to optimal, the floundered goals can not be deleted.

The main difference to FIDO is the choice of open, closed and half-closed and half-open intervals as their computational domain. This system uses a split builtin to halven an interval and generate a choicepoint, which is related to our choice primitives.

```
;; poly:  Find solution to the equation X^3-6X^2-7X-6 = 0
(what (poly ?X)
(add 6 ?A ?X) (mult ?X ?A ?B) (add 7 ?C ?B) (mult ?X ?C 6))

?- ((poly ?X))
Solution: ((poly (< 7.104213591116952 7.104213591116959 >)))
Floundered goals:
  (mult (< 7.104213591116952 7.104213591116959 >) (< 0.8445692015091354 0.8445692015091371 >) 6)
  (mult (< 7.104213591116952 7.104213591116959 >) (< 1.1042135911169542 1.1042135911169568 >)
        (< 7.844569201509134 7.844569201509139 >))
  (add 7 (< 0.8445692015091354 0.8445692015091371 >) (< 7.844569201509134 7.844569201509139 >))
  (add 6 (< 1.1042135911169542 1.1042135911169568 >) (< 7.104213591116952 7.104213591116959 >))
Splitting goals:
More? ;
no more solutions
```

Figure 2.2: Sample program and query of Interval PROLOG

2.6 Echidna

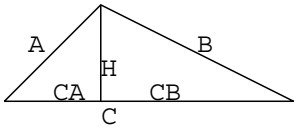
Echidna ([SH91]) is an extension of Logical Arithmetic (see section 2.5). Their computational domain is a finite set of disjoint intervals. In [Cle87] only a single interval could be associated with a variable. A disjoint interval domain variable may e.g. look like: $X \in [0,1] \cup (3,4] \cup (7,10)$. Processing disjoint intervals does not result in a generation of new alternatives (and choicepoints).

2.7 CAL

CAL (Contrainte Avec Logique) ([SA89]) has been developed at ICOT using Gröbner base algorithms to solve algebraic and boolean equations. Their constraint domain is not restricted to linear equations and disequations. The techniques originate from geometric theorem proving. They report applications where CAL performs three dimensional inferences (dynamic & static) in a robot arm application [HFKF91]. Below an example ([SA89]) of the system demonstrating its unique features of solving nonlinear constraints:

```
sur(H,A,S) :- A*H=2*S. /* area of a triange */
right(A,B,C) :- A*A + B*B = C*C. /* Pythagoras */
tri(A,B,C,S) :- C=CA+CB, right(CA,H,A), right(CB,H,B), sur(H,C,S).

?- tri(A,B,C,S).
S^2 =( -A^4 - B^4 - C^4 + 2*B^2*C^2 + 2*C^2*A^2 + 2*A^2*B^2) / 16
/* Heron's formula */
```



2.8 CHIP

CHIP (Constraint Handling in PROLOG) is the most eleborated constraint system and is also commercially successful. It incorperates finite domains, boolean unification algorithms and rational numbers. Although many combinatorial problems can be stated as finite domain constraints the additional usage of rational and boolean constraints can lead to a further increased pruning effect.

Successful application to real-world problems are

- planning and configuration: timetables, crew allocation, management, scheduling, resource allocation, assembly line scheduling
- cutting stock problems
- circuit design and verification
- financial applications
- logistics (warehouse location and transport logistics)
- expert systems

A remarkable production managament application is done by Dassault Aviation [Pra91] to schedule over 250 aircrafts (M2000) to deliver over a period of 5 years in several factories.

2.9 CHARME

CHARME, distributed by BULL, is basically a descendant from CHIP, although it looks quite different. It is a C language library, extending C with domain variables, nondeterministic computation, arithmetic and symbolic constraints and special minimization predicates.

2.10 PROLOG III

PROLOG III [Col87b, Col87a] computational domains are the infinite trees from PROLOG II, linear rational equalities and inequalities, booleans and special constraint operations on lists.

The system has been used for modeling technical systems ([Sku89b]), especially for designing the model based expert system PROMTEX for fault detection in gasoline motors for cars[Sku89a]. Their expert system does not contain rules saying something about the relations of symptoms and faults, but they have a description of the components with their function in an algebraic form and the state of the input and output variables (functional engine models, [KS88], [JS89]).

2.11 Trilogy

Trilogy ([Vod88]) supports linear *integer* arithmetic as their constraint domain. The operators "+" and "−" and integer multiplication factors can be used, the constraint relations are "$\leq$,=" and "$\neq$". Although some problems can be stated as linear integer problems, the main applications seem to be logical puzzles since the solver is rather inefficient.

2.12 FIDO-I & FIDO-II

The FIDO constraint domain is the finite domain subset of CHIP together with efficient consistency techniques such as forward checking, weak lookahead in FIDO-II, and lookahead in FIDO-III. As expected, the meta-interpreter approach ([Sch91]) is a computational slow, prototypical implementation technique. Even the parts, which are "pure" PROLOG are metainterpreted (e.g. the usage of append/3).

The idea of FIDO-II ([Mül91]) is to reduce the interpretational overhead by horizontal (source-to-source) compilation of FIDO programs into SEPIA PROLOG ([MAC+89]) using the delay mechanism of SEPIA as the only extension from "ordinary PROLOG".

For a full discussion of FIDO-II drawbacks, the reader is referred to chapter 8 of [Mül91]. Since logical variables can only be bound once, domain variables have to be simulated by a structure[7] &(X_{id},X_{length},$X_{\#cons}$,X_{value},X_{domain}), where X_{id} is a unique identifier, X_{length} is an open list representing the length of the domain, $X_{\#cons}$ is the number of constraints (e.g. for the first fail principle ([Hen89])). X_{value} is a place holder for a possible singleton and X_{domain} is the actual domain on this level. Since domain variables are bound to either constants or domains smaller[8] than the actual domain, an open list technique is applied. Thus, the handling of

[7] or low level bind operations in PROLOG could be used, but we wanted to restrict ourselves to the usage of `delay` as the only non-standard PROLOG extension.

[8] in terms of set inclusion.

domain variables is done on the PROLOG level. Clearly, the more often a domain variable is bound to a smaller domain, the more time is used to scan through the list to access the domain. In the WAM this very basic operation is called dereferencing. In our WAM-based approch, the operation performed to bind a domain variable to a smaller domain is to create this smaller domain and to bind the pointer to the new domain resulting in an instant access. Another drawback is explicit handling of unification: If we unify a FIDO-II domain variable (which are structure) with a constant, it is obvisious that the builtin unification routine can not be used. It must be done by code in the horizontally compiled FIDO-II program.

A problem not yet addressed is whether the usage of the delay primitive in SEPIA is appropriate for the simulation of the constraints — we did not have the access to the internals of the SEPIA PROLOG system. The statistical runtime package in SEPIA seems to be too course grain to state anything about this behaviour.

Chapter 3

Theoretical Framework

An intuitive idea of constraints has been given in chapter 1. PROLOG has the claim of having elegent and simple procedural and declarative semantics. The question is when we integrate some constraint extension, does the language still have the elegant mathematical properties ?

A way out of this dilemma is the CLP scheme ([JL87]) requiring several properties of a structure to be integrated as the constraint domain and having the desired well defined semantics. Unfortunately, the finite domain extension does not belong to scheme. Therefore, we must justify how to integrate finite domain consistency techniques without great penalty and how SLD-resolution can be modified to yield a sound and complete search procedure [Hen89].

The domain extension is done by modifying the unification procedure, the lookahead and forward constraint techniques are formalized as inference rules having great impact on control and thus on SLD-resolution. We are specially interested in these concepts defining the procedural semantics for our implementation.

In the logical viewpoint the domain variables can be considered as an abreviation for logical variables with a monadic predicate (with finite clauses) specifying the different values. But on the procedural level we can hardly express that a specific clause of such a monadic procedure can not be applied achieved with the lookahead and forward rules to obtain a priori pruning.

Definition 9 *A domain d is a non-empty finite set $\mathcal{R}$ of constants with $d \in 2^{\mathcal{R}} \Rightarrow e \in 2^{\mathcal{R}}$ for $e \subset d, e \neq \emptyset$*

Constraints have the same syntactic form as normal PROLOG procedures, except for a declaration that some procedure is to be treated as a constraint under a certain control regime (forward, lookahead). A constraint should have the following property:

Definition 10 *An n-ary procedure p/n is a* constraint *iff for arbitary ground terms $t_1, \ldots, t_n$ either $p(t_1,\ldots,t_n)$ has a successful refutation, or has only finitely failed derivations.*

This property is important, since for checking consistency after a constraint can be applied, the domain variables of the constraint are successively instanted to the constants the domain variable(s) denote.

3.1 Declarative Semantics

In this section we will give an overview of the modifications done to the standard ([Llo84]) declarative semantics to include the domain variable concept.

1. The alphabet of the first order language must be enriched by a set of domain variables x^d with domain d.
2. Terms, definite programs and goals are defined as usual.
3. The semantics of first-order logic with domain variables must be given for the declarative semantics:
   - Extend the usuual interpretation $\mathcal{I}$ by the interpretation of a domain d as the assignment of subset d' of the universe D.
   - For defining satisfiability a formula in L must be mapped to a truth value $\{true, false\}$ w.r.t. an interpretation $\mathcal{I}$ and a variable assignment $\mathcal{A}$. The standard definition must be extended by $\mathcal{I}(\exists x^d.\mathrm{P}) =$ true iff there exists c $\in$ d' with $\mathcal{I}^{x^d,c}(\mathrm{P}) =$ true.
     $\mathcal{I}(\forall x^d.\mathrm{P}) =$ true iff $\mathcal{I}^{x^d,c}(\mathrm{P}) =$ true for all c $\in$ d'.
   - Define the notions *logical consequence*, *Herbrand interpretations* and *models* as usual.
4. Define *substitution*, *correct substitution* and *correct answer substitution* extended by the notion of a domain variable in the following way: a domain variable x^d with domain d can be substituted by a constant $c \in d$ or by a domain variable y^e with domain e and $e \subset d$.

3.2 Procedural Semantics

Procedural semantics are given by SLD-resolution and the unification algorithm. In the unification algorithm, we must cope with the unification of domain variables adding the following cases:

- If a domain variable x^d with domain d and a constant c have to be unified, x^d is bound to c, if $c \in d$, otherwise the unification fails.
- If the domain variable x^d with domain d and the domain variable y^e with domain e have to be unified, the intersection $l = d \cap e$ is computed. If the intersection is empty, a failure is issued, otherwise the domain variables x^d and y^e are bound to a new domain variable z^l.
- If a domain variable y^d and and ordinary variable x are to be unified, x is bound to y^d.

It should be noted that the unification algorithm can be optimized concerning memory aspects, e.g. in the unification of the domain variables x^d and y^e special handling can be performed for $d \subseteq e$ or $e \subseteq d$. Furthermore the unification algorithm given in fig. 3.1 ([Mül91]) is completely different from the unification algorithm given in chapter 7, where the suspended goals and a bunch of memory optimisations are considered. This is a mere theoretical version.

```
k := 0; θ_k := ϵ;
while Flag = true do
 {
  if singleton(θ_k)
  then
   { return θ_k(E); STOP }
  else
   {
    compute_disagreement_set(θ_k(E), D_k);
    if $(v \in D_k)$\verb and—(t ∈ D_k) and is_simplevariable(v)
                    and not occurs_in(v, t)
    then \* unification of simple variable and term *\
     { θ_{k+1} := θ_k[v←t]; k := k + 1 }
    else
     if v_d ∈ D_k and is_domvar(v_d) and a ∈ D_k)
                      and is_constant(a) and a ∈ d
     then \* unification domain variable - constant *\
      { θ_{k+1} := θ_k[v_d←a]; k := k + 1 }
     else
      if v_{d1} ∈ D_k and is_domvar(v_{d1}) and v_{d2} ∈ D_k
                      and is_domvar(v_{d2}, D_k) and d_2⊂d_1
      then \* unification of two domain variables over the same domain *\

       { θ_{k+1} := θ_k[v_{d1}←v_{d2}]; k := k + 1 }
      else
       if v_{d1} ∈ D_k and is_domvar(v_{d1}) and v_{d2} ∈ D_k
                       and is_domvar(v_{d2}, D_k) and d_1∩d_2 ≠ ∅
       then \* General unification of two domain variables *\
        {e := d_1∩d_2; θ_{k+1} := θ_k[v_{d1}←w_e, v_{d2}←w_e]; k := k + 1 }
       else
         Flag := false;
   }
  RETURN("not unifiable!");
  STOP
}
```

Figure 3.1: A Domain Variable Unification Algorithm

Definition 11 *The* disagreement set *of a finite set of simple expression S is the set of all subexpressions obtained by locating the leftmost symbol position at which not all expressions in S have the same symbol and extract from each expression in S the subexpression at that symbol position.*

SLD resolution with the finite domain unification algorithm is referred to as SLDD resolution. In the following we will summarize the main results of [Hen89], the proof can be found there.

Theorem 1 (Unification theorem) *For the domain variable unification algorithm 3.1 the following holds:*

- *The algorithm always terminates.*

- *Let E be a finite set of expressions. If E is unifiable, the unification algorithm gives the mgu θ for E. If E is not unifiable, the unification algorithm reports "not unifiable"*

As a conclusion, domain variables can be integrated in a logic programming framework preserving the duality of procedural and declarative semantics. We are now defining addtional inference rules realizing the consistency techniques forward checking and lookahead.

3.3 Forward Checking

The idea of forward checking is that a constraint whose argument places are ground except at most one, this argument can be inferred. Some constraints are only useful when such a condition is present: The $\neq$ constraint in the queens example (1.2) can only be reasonably applied when one argument is ground and the other is a domain variable, where the ground value is removed from the domain. We will define now forward checkable and the forward checking inference rule (FCIR):

Definition 12 *Be $p(t_1,\ldots,t_n)$ an atom. We say that $p(t_1,\ldots,t_n)$ is forward-checkable, if*

- *p is a constraint*
- *there exists only one t_i, $1 \leq i \leq n$, that is a domain variable, called the* forward variable, *and all others are ground.*

Definition 13 (The FCIR) *Be P a program, goal G_i has the form $\leftarrow A_1,\ldots,A_k,\ldots,A_m$ and θ_{i+1} a substitution. G_{i+1} is derived by the FCIR from G_i, P, θ_{i+1}, if the following conditions hold:*

1. *A_k is forward-checkable, x_d be the forward variable inside A_k.*
2. *The new domain e is computed by $e = \{a \in d | P \models A_k\{x_d \leftarrow a\}\}$. If $e = \emptyset$ a failure occurs.*
3. *The new substitution θ_{i+1} is defined by*
   $\theta_{i+1} = \{x_d \leftarrow c\}$, if $e = \{c\}$ (singleton case), or $\{x_d \leftarrow y_e\}$, where y_e is a new domain variable, otherwise.
4. *The goal G_{i+1} is constructed by $G_{i+1} = \leftarrow (A_1,\ldots,A_{k-1},A_{k+1},\ldots,A_m)\theta_{i+1}$.*

Since A_k is considered to be a constraint (def. 10), $P \models A_k\{x_d \leftarrow a\}$ in definition 13 is a decidable "test". The inference rule handles the constraints actively:

- when the new domain e is calculated either values are deleted from a domain variable resulting in a domain variable with less elements. When choice methods are applied to the domain variables, these inconsistent values are not considered. This is an a priori pruning.
- when the domain becomes a singleton the variables is instaniated. If the variables can be found in other constraints it reduces the number of nonground arguments.

In the following we summarize some properties of the FCIR.

Theorem 2 (Soundness of the FCIR) *Be P a program, G_i be the goal $\leftarrow A_1, \ldots, A_k, \ldots, A_m$. Let A_k be forward-checkable, x_d be the forward variable and $d = \{a_1, \ldots, a_n, b_1, \ldots, b_k\}$, d, $e \neq \emptyset$. Let the goal G_{i+1} be obtained from G_i by $G_{i+1} = \leftarrow (A_1, \ldots, A_{k-1}, A_{k+1}, \ldots, A_m)\theta_{i+1}$. Then G_i is a logical consequence of P iff G_{i+1} is a logical consequence of P.*

The theorem assures that using the FCIR no wrong results will be achieved. In [Hen89], the completeness result has been proved for a procedure that

- applies the FCIR for forward-checkable predicates whenever possible
- uses normal SLD-derivation, otherwise.

This resolution procedure is called "SLDFC resolution".

Theorem 3 (Completeness of FCIR) *P be a logic program and G be a goal. If an SLDD refutation of $P \cup \{G\}$ exists, then there exists an SLDFC refutation of $P \cup \{G\}$. If θ is the answer substitution from the SLDD-refutation of $P \cup \{G\}$, and ρ is the answer substitution from the SLDFC-refutation of $P \cup \{G\}$, then $\rho \leq \theta$.*

We can say that every provable goal using SLDD-reolution can be also proved using SLDFC-resolution. Ordinary SLDD resolution is the normal proof procedure as long as no forward checkable goals can be applied. An important point is that a forward checkable constraint can only be applied once and is afterwards considered to be *dead*. The effect of a forward checkable goal is a failure (when the bindings are inconsistent) or in a further restriction of the forward-variable.

3.3.1 Notation

In some form we must tell the system that it must use a particular inference rule for some constraint (procedure). We will give the syntactic form how a procedure p is declared to be a forward constraint:

Definition 14 *Given an n-ary predicate p, a* forward declaration *has the following form:* `forward p(`$a_1, \ldots, a_n$`)` *with $a_i \in \{d, g\}$. p is a constraint and the procedure p in the program is said to be submitted to forward declaration.*

Definition 15 *A procedure p submitted to forward declaration is forward-available iff*

1. *all arguments of p declared "*`g`*" are ground.*
2. *p is forward checkable (see Def. 12)*

Thus a constraint must not necessarily only have domain variables or constants but can also have complex terms in the argument places where a "`g`" is declared.

Definition 16 *A computation rule R is* forward consistent *iff an atom submitted to forward declaration is selected by R only when it is forward available or all arguments are ground.*

In our system, of course, we want to have a forward consistent computation rule. Then we can state the (forward)-constraints before the generators and these constraints will be selected when they are forward available or the arguments are ground.

Definition 17 *A proof procedure is* forward consistent *iff*

1. *it uses a forward consistent computation rule R.*
2. *when a forward-available atom p is selected by R, the FCIR is used to solve p.*
3. *when an atom p is selection not submitted to FCIR, normal derivation is used.*

3.4 Looking ahead

We will introduce the looking ahead inference rule (LAIR) in logic programming motivated by the looking ahead scheme in constraint processing. The FCIR is applied when not more than one argument is nonground — which is a very strong condition. In constraint processing the reduction of the domains is possible when more than one variable is left uninstantiated. Consider the example $X : \{1,2,3\} + Y : \{2,3,4,5\} < Z : \{5,6,7,8\}$ where the domains can be reduced to $X : \{1,2\} + Y : \{2,3\} < Z : \{5,6,7,8\}$. This leads to much earlier pruning of the search space although more work ensuring consistency must be done.

The LAIR applicability is defined in the following way:

Definition 18 *An atom $p(t_1, \ldots, t_n)$ is lookahead-checkable if*

- *p is a constraint and*
- *There exists at least one t_i that is a domain-variable. All other t_j are either ground or domain variables.*

The domain variables in $t_1, \ldots, t_n$ are called *lookahead-variables*

Definition 19 (The LAIR) *Let P be a program and G_i have the form $\leftarrow A_1, \ldots, A_k, \ldots, A_m$ and θ_{i+1} a substitution. G_{i+1} is derived by the LAIR from G_i and P using the sustitution θ_{i+1} if the following holds:*

1. *A_k is lookahead-checkable, $x_1, \ldots, x_n$ are the lookahead variables of A_k, which range over $d_1, \ldots, d_n$.*
2. *For each $x_j, e_j = \{v_j \in d_j | \exists v_1 \in d_1, \ldots, \exists v_{j-1} \in d_{j-1}, \exists v_{j+1} \in d_{j+1}, \ldots, \exists v_n \in d_n$ such that $A_k\theta$ is a logical consequence of P with $\theta = \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}\} \neq \emptyset$.*
3. *Let y_j be the constant c if $e_j = \{c\}$ (singleton condition) or a new domain variable which ranges over e_j, otherwise.*
4. *$\theta_{i+1} = \{x_1 \leftarrow y_1, \ldots, x_n \leftarrow y_n\}$.*

5. G_{i+1} *is either* $\leftarrow (A_1, \ldots, A_{k-1}, A_{k+1}, \ldots, A_m)\theta_{i+1}$ *if at most one* y_i *is a domain variable, or* $\leftarrow (A_1, \ldots, A_m)\theta_{i+1}$, *otherwise.*

In the above definition (19) in point 5 it is defined when a lookahead constraint is regarded to be dead: The atom submitted to lookahead is again included in the resolvent when more than one lookahead variable ws present, otherwise the constraint is no longer included in the resolvent and thus being dead. The effect is that a lookahead constraint can be applied several times. However, we must take care in a real implementation that a lookahead constraint is not always called: According to the above definition the LAIR can (always) be applied to the lookahead atom and an infinite loop results. Thus we should extend definition 18 with a condition like "a domain variable has been touched by a binding".

Concerning the lookahead control regime we can prove its soundness, but unfortunately not its completeness (see [Hen89]).

Theorem 4 *Let* P *be a program and goal* G_i *have the form* $\leftarrow A_1, \ldots, A_k, \ldots, A_m$. *Let goal* G_{i+1} *be derived by the LAIR from* G_i *and* P *using sustitution* θ_{i+1}. G_i *is a logical consequence of* P *iff* G_{i+1} *is a logical consequence of* P.

3.4.1 Notation

We will give the syntactic form how a procedure p is declared to be a lookahead constraint:

Definition 20 *Given an n-ary predicate p, a* lookahead declaration *has the following form:* `lookahead p(`$a_1, \ldots, a_n$`)` *with* $a_i \in \{d, g\}$. *p is a constraint and the procedure p in the program is said to be submitted to forward declaration.*

Definition 21 *A procedure p submitted to lookahead declaration is lookahead-available iff*

1. *all arguments of p declared "*`g`*" are ground.*
2. *p is lookahead checkable (see Def. 18)*

Definition 22 *A computation rule R is* lookahead consistent *iff an atom submitted to lookahead declaration is selected by R only when it is lookahead available or all arguments are ground.*

Definition 23 *A proof procedure is* lookahead consistent *iff*

1. *it uses a lookahead consistent computation rule R.*
2. *when a lookahead-available atom p is selected by R, the FCIR is used to solve p.*
3. *when an atom p is selection not submitted to LAIR, normal derivation is used.*

3.5 Conclusion

In this section we have shown how to embed consistency techniques in logic programming by embedding domain variables and two inference rules altering the selection strategies of SLDD-resolution. By incorperating the constraint techniques, the elegant declarative semantics are close to the standard theory, the procedural semantics remain elegant. A priori pruning is performed by consistency techniques, which can be formalized as inference rules.

Part II

FIDO-III: Concepts and Implementation

Chapter 4

The underlying WAM

The WAM emulator written by Sven-Olof Nystrøm[Nys] is a remarkable implementation concerning the abstraction of structures used within the WAM. Special defining functions for registers, instructions, offsets to environment and choicepoint locations are present allowing easy and quick modifications of WAM based structures and the implementation of new instructions. This WAM is a basis of COLAB developed within the ARCTEC-project and forms starting point for a variety of extenions (e.g. [Hei89, Bol90, Hin91]). However, the main goal of this implementation is readability and not efficiency. We will start from this WAM to make our constraint prototype. Therefore, we will shortly describe the underlying structures and notations. A reader unfamiliar with the WAM is referred to [War83, AK90].

4.1 Terminology

The basic entity in the WAM is a *word*. It contains a *tag* describing the type of the word and a *value* containing the address or representation of a *simple type*. Since the word is normally fixed in bit length[1], all *simple types* must fit into a word. A simple type is a number, a reference to a symbol table or an address, including references to other references, references to lists and references to structures.

The WAM has three stacks:

heap The heap is often referred to as *global stack*. All data types not fitting into a word are constructed in the heap. Thus, lists and structures are only found in this data area. Since suspended variables and domain variables are too large to fit into a word, they must be created in the heap. However, simple data structures can also be found on the heap.

stack The stack is also called local stack or runtime stack. It contains *environments* and *choicepoints*. Environments hold the *Y-variables* from a clause. These variables could be stored on the heap — but the environments can be deallocated earlier than the heap thus saving a lot of space, especially when doing deterministic computations. Environments can be compared to "calling frames" in procedural languages, where local variables for a function or subroutine are stored. In our FIDO extension (fig. 5.3,5.4) we will "misuse" environments to save the state of the WAM machine in order to invoke constraint processing. Choicepoints are used to store pointers to possible alternatives of procedures. When

[1] The length of a word is usually 32 bits or 64 bits.

the first clause of a procedure is called and indexing can not reduce the set of possible clauses to a single clause, a choicepoint is created, which also contains the arguments of the clause. Upon backtracking the next clause must be invoked — with the arguments previously stored in the choicepoint.

trail The binding of variables must be undone upon backtracking. In the original WAM bindings were only done to "normal" variables. Since an unbound variable is represented by a pointer to the location where it is located, only these addresses were saved. In our FIDO extension we will also have to save the previous value of the variable, since a domain variable can be bound a number of times — and must be unbound correctly to it previous "value".

In the orginal WAM another stack used is the push-down-list (PDL) for temporal states of the unification routine. The unification routine in this WAM is recursive and it stores its temporal information in local variables inside the unification routine and in the arguments of the unification routine.

The reader should notice that A-registers and X-registers are the same. In literature they differ for presentational reasons.

4.1.1 The data structures

Tags (see 4.1) are a portion of a word. In low level implementations (e.g. [Tay90a]) tags are stored in places of a machine word which are normally not used: In modern RISC machines all basic data types of the machine are aligned at addresses which can be divided by 4 (assuming 32 bit architecture). Thus the least significent two bits are always zero, which is a possibility to store the tag. However, with two bits four tags can be coded — which is not enough. Other bits of a machine word must be used, e.g. the most significant bit is a candidate. If the most most significant bit is utilized, quick access to this bit is possible, since most architectures set a bit when loading a "negative" value into the register. The SPARC architecture used by SUN has special support for tags in their machine implementation. The Motorola M68K makes bit processing (tag processing) easy with a bunch of bit manipulation instructions and memory based bit extraction instructions. Other implementations are more radical: Instead of using one machine word for one WAM word, they use one machine word for the tag and another machine word for the value ([MAC+89]).

The coding of tags is a major problem when implementing low-level machine WAMs: The representation choosen in one architecture can be bad in another. We will not care about these problems in our implementation — we will even not define how to store them. All possible atoms can be used as a tag in the WAM. Some of the tags in the WAM are for making the user's life easier. The empty, code and trail tags are used for readability when debugging WAM programs. However, having the problems above in mind, we shall *use as few new tags as possible.*

4.2 The registers

In the WAM, there are several functions coping with the administration of registers. A register is defined by `(define-register` *register*`)`, resulting in automatically printing the register when debugging mode is switched on. The regsiter is accessed by `(reg` *register*`)` and set by `(set-reg` *register new-value*`)`. The X-registers are predefined and are read by `(argument-reg`

Tag	Value
empty	undefined
ref	a reference to a memory address
struct	a reference to an address with a fun-word
list	a reference to an address with a list-word
const	constant symbol
fun	a list (function-name arity) and beginning of a structure
code	a list (procedure-name . rest-of-instruction-list)
trail	a list of references to bound variables
Added tags:	
dly	a marker for a constraints indicating if the constraint is dead.
domain	a domain
susp	a suspend variable

Figure 4.1: Tags used in the WAM

register) and are written by (`set-argument-reg` *register new-value*). Table 4.1 gives an overview of the WAM registers and the new registers. The overall design can be seen in fig. 1.4. The indexes to the argument register for X- and Y-registers start with 1.

Register	Description	points to
P	program counter	program code
CP	continuation pointer	program code
E	last environment	local stack
B	last choicepoint	local stack
A	top of stack	local stack
TR	trail list	
H	top of heap	heap
HB	heap backtrack point	heap
S	structure pointer	heap
X_i	registers	heap,stack
Added registers:		
WAKE	begin of woken goals	heap
WAKEE	end of woken goals	heap
FIRING	True during constraint processing	

Table 4.1: Registers used in the WAM

4.3 Memory layout

The memory layout is shown in fig. **??**. Address 0 is at the top of figure increasing downwards. The trail, normally found in such a memory layout, is realized as a lisp list since the WAM strictly accesses only the top of the stack.

4.3.0.1 The local stack

The local stack contains environment and choicepoint frames. An environment must be created in a clause (using the `allocate` instruction) as soon as local variables become necessary.

A choicepoint is needed, if there is more than one clause in a procedure. If a recent goal failed, the next clause must be explored with all argument registers appropriately (re-)set and the variables bound later than the invocation of the current clause restored to an unbound state.

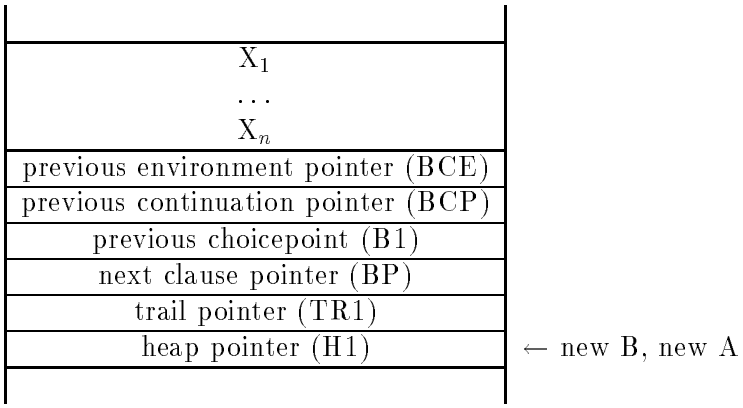


Figure 4.2: The memory layout of a choicepoint (backtrack point)

4.4 The instructions

The instructions are defined by `(definstr` *name arguments body*`)`. Structures are coded by a list `(fun arity)`. Although the names of the instructions are not strictly equal to [War83], there is no problem of identfying equal instructions.

4.4.1 Unification related instructions

Only some of the instructions in table 4.2 invoke the general unification algorithm. However, e.g. a put_constant instruction can also be seen as a unification — the unification with an (unbound) variable residing in an argument register which must not be trailed ([HM92]).

4.4.2 Procedural instructions

The procedural instruction determine the control of the program. They can be seperated in instructions handling the creation, modification and deletion of choicepoints (or-processing) and the handling of environments (and-processing).

The cut implementation is due to [Bee85]. Compared to [Bee85], we slightly modified the semantics of the instructions and decreased the number of instructions. The save_cut_pointer instruction must always be used when an environment has been created and a cut is present in a clause. [Bee85] argues that we must not save the additional B-register in an environment, when we do not use it. However, a word is reserved by the allocate instruction to store the value.

(put_variable_perm Y_{from} X_{to})	(get_variable_perm Y_{to} X_{from})	(unify_variable_perm Y_i)
(put_variable_temp X_{from} X_{to})	(get_variable_temp X_{to} X_{from}i)	(unify_variable_temp X_i)
(put_value_perm Y_{from} X_{to})	(get_value_perm Y_{to} X_{from})	(unify_value_perm Y_i)
(put_value_temp X_{from} X_{to})	(get_value_temp X_{to} X_{from})	(unify_value_temp X_i)
(put_unsafe_value_perm Y_{from} X_{to})		(unify_local_value_perm Y_i)
(put_constant C X_{to})	(get_constant C X_{from})	(unify_constant C)
(put_nil X_{to})	(get_nil X_{from})	(unify_nil)
(put_structure F X_{to})	(get_structure F X_{from})	
(put_list X_{to})	(get_list X_{from})	
		(unify_void n)
		(unify_local_value_temp X_i)

Table 4.2: Unification related instruction

(try L n)	(try_me_else L n)
(retry L n)	(retry_me_else L n)
(trust L n)	(trust_me_else_fail n)

Table 4.3: Choicepoint handling instructions

(allocate n)
(deallocate)
(proceed)
(execute proc/n)
(call proc/n envsize)

Table 4.4: Environment handling instructions

(has-succeeded)
(has-failed)
(save_cut_pointer)
(cut n)
(mcall X_i envsize)
(switch_on_type Lvarunbound Linteger Lsymbol Llist Lstruct Lnil Lother)
(switch_on_constant Len Table Default)
(switch_on_structure Len Table Default)

Table 4.5: Cut instructions, user interaction, switch and metacall instructions

Chapter 5

The implementation of delay

Mats Carlsson ([Car87] extended the WAM by the `delay` builtin, claiming that the method does not incur any overhead in programs not using the extension. This seems to be a good starting point for our implementation. Here, we will briefly discuss the method.

We must mention an assumption inherent in the extension: An interruption of the program is only allowed when we know how much registers are alive. This can only be stated when we invoke a procedure by the `call` or `execute` instructions or when we leave a procedure by `proceed`. This is contrasted to the assumption that we can always save the entire state as required by the concept described in section 2.3.

5.1 `freeze` and `delay`

Let X be a variable and $P(\ldots X \ldots)$ be a term containing X.

The delay primitive `delay(`$X, P(\ldots X \ldots)$`)` delays the calling of $P(\ldots X \ldots)$ until it is bound, even to a variable. The meta-predicate `freeze(`$X, P(\ldots X \ldots)$`)` delays the calling of $P(\ldots X \ldots)$ until X has been instantiated to a non-variable. `Freeze` can be implemented by `delay`:

```
freeze(X,G) :- var(X), delay(X,freeze(X,G)).
freeze(X,G) :- nonvar(X), call(G).
```

Whenever a frozen variable X is bound to another variable, the `freeze` predicate is invoked which immediately delays X again with the freeze predicate when it is a variable, otherwise it has been bound to a nonvariable term and meta-calls the goal.

5.2 Suspended Variables

There are two types of variables: ordinary variables and suspended variables. Suspended variables need to have two memory cells, the first for the value (with the tag `susp`) and the second cell is a tree of suspended goals. We will see, why we do not have a list of suspended goals, although this would be desirable for the user.

The algorithm for delay is[1]:

[1] in a PASCAL like notation.

```
delay(X,G)
       if (nonvar(X))
          then call(G)
          else if (X==$susp(V,G))
                  then bind X to $susp(_,list(G0,G))
                  else bind X to $susp(_,G);
```

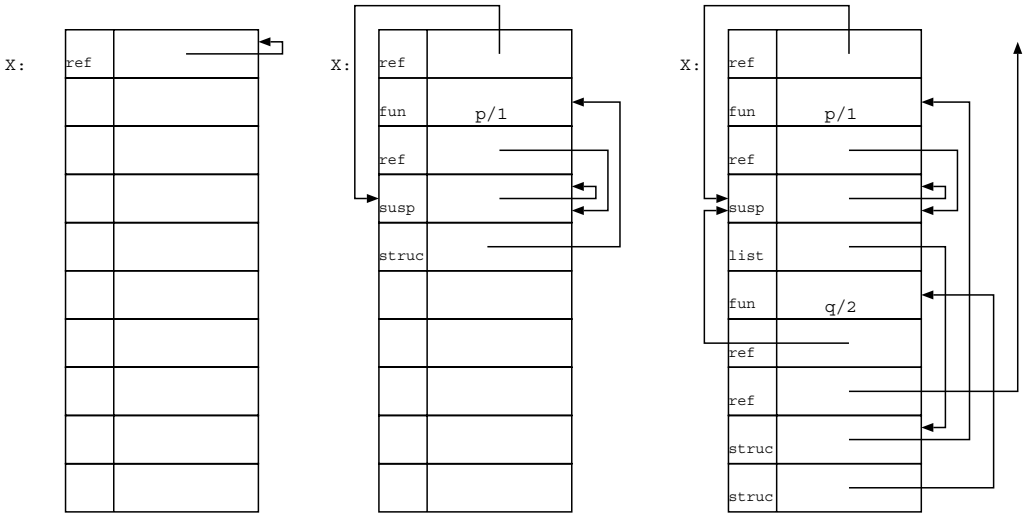


Figure 5.1: Creation of a delayed variable

In fig. 5.1 three heap segments are shown. The left one has an unbound variable X which is consecuatively delayed by `delay(X,p(X))`. The memory structures are shown in the middle segment: The reference points to a suspension, whose first cell points to itself, indicating an unbound state. The second cell has a `struct`-pointer to the structure `p/1` and the argument points to the original variable. A following `delay(X,Q(X,U))` has to cope which the fact that X is already delayed. First, the structure is created on the heap and and the second cell of the suspension is set to a list entry referencing two structure cells[2].

5.3 The overall strategy handling suspended variables

Whenever a suspended variable is bound to another term, a wakeup event is raised. In the framework, the unification of two suspended variables is *not* handled by the unification routine but must be handled by the delay primitive. The advantage is that only minor modifications have to be done to the unification routine, although Carlsson ([Car87] writes that "this can lead to a lot of wasteful wakings and re-suspendings, but has the advantage simplicity". Please note that these re-suspending consume not only time but a lot of memory needed to resuspend the variables (see 5.1 for the implementation of freeze. The overall strategy can be seen in fig. 5.2).

[2]In Carlsson original paper a constructure "," is used to create the suspension tree. In the WAM an optimized representation for list-cells exists, saving one node per entry. We use them instead of this structure proposed. The modification is minor and hence trivial.

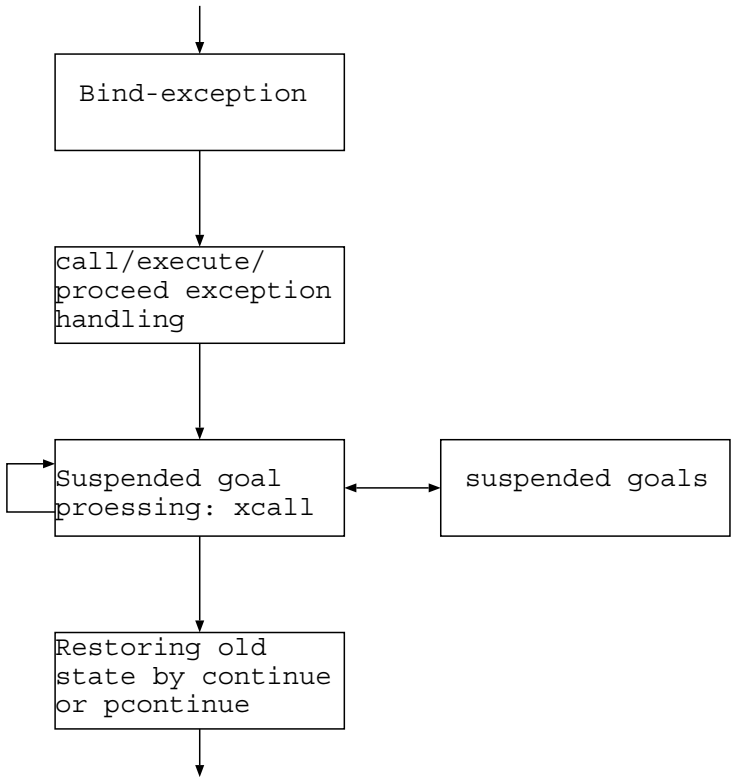


Figure 5.2: Strategy for suspended goal processing

5.3.1 Handling a binding event

After a suspended variable has been touched by the binding primitive — a portion of rudimentary code in the WAM "assigning" terms to variables[3], an exception is raised. Because of the basic character of the bind routine, it must be handled with care: The additional work done in the routine is to test, if a suspended variable is to be bound. If not, the routine does its conventional work. The handling of a suspended variable and the invocation of the suspended goal tree belonging to the variable can not be done immediately. The handling must be preserved until we know how much argument registers are alive and thus must be saved for handling the exception. The normal flow of the WAM program can only be interrupted when we invoke a `call`, `execute` or `proceed`[4] instruction. The routine of waking up the suspensions on the variable is called a *wakeup*. However, several bindings may be done before an instruction occurs, which can be interrupted. Thus, the woken suspensions must be collected. The new register `WAKEUP` contains either the constant `nil` or points to a suspension tree constructed by consing the different suspensions in the following way:

```
wake(Goals):
 if (WAKE = nil)
    then WAKE := Goals
    else WAKE := list(WAKE,Goals);
```

Compared to [Car87] we do not use extra flags to indicate an exception. His additional flag is only useful when other "exceptions" might also occur.

[3] Trailing is only necessary if a forthcoming failure will not automatically deallocate this variable.

[4] In [Car87] the `proceed` instruction is not included — we will present the necessary (simple) extensions.

5.3.2 Handling the exception

In the WAM instructions `call`, `execute` and `proceed` a test must be performed to see, whether `WAKE` is `nil` or not. If `WAKE` is `nil` no suspended goals have been woken and the ordinary semantics of the instructions apply. When a suspended goal tree is present, we have to save the state of the machine so that it can later restore the state as soon as the interrupt is finished. When a `call` or `execute` instruction is invoked, the state is saved in an ordinary WAM environment so that no new flavor of local stack entry is necessary. The number of active arguments is given in the instruction and known at compile time. The environment for `call` and `execute` exception frames can be seen in in fig. 5.3. The instruction `call` saves the program counter `P` in the continuation-field (`OCP`) of the environment, whereas the `execute` instruction saves the continuation register `CP` in this slot.

When a `proceed` instruction is invoked, we have a little oddity in handling the instruction: In general, an environment has been previously thrown away by the `deallocate` instruction. This information of the environment frame must be again saved on the stack (see fig. 5.4), before the suspended goals can be executed.

Register `CP` is set to a procedure named `continue` for `call` and `execute` and to `pcontinue` for `proceed`. These procedures are responsible for restoring the state to continue "normal" processing. The register `WAKE` is copied to the argument register `X1`, `WAKE` is reset to nil and a WAM procedure `xcall/1` traversing the tree in order to meta-call the suspended goals is invoked. The following is an extension to [Car87]: During constraint processing other suspended goals might be woken. Instead of doing a depth first strategy, we apply a fair strategy, meaning that the goals are executed in the order they have been woken. To implement this strategy, we need another register named `FIRING` which is set to `true` by handling the exception. This register indicates that the machine does currently suspended goal processing and no `call`, `execute` or `proceed` is allowed to invoke another woken constraint handling. However, the newly woken suspended goals are still collected by the binding mechanism.

5.3.3 Doing suspended goal processing

A PROLOG notation of `xcall/1` looks like:

```
xcall1(S) :- struct(S), mcall(S).
xcall1([P|L]) :- struct(P), mcall(P), xcall(L).
xcall1([L1,L2]) :- list(L1), xcall(L1), xcall(L2).
```

while the xcall predicate has to take care of newly woken constraints:

```
xcall/1:
 allocate 0
L0:
 call xcall1/1,1,0
 ifagainwakego L0
 deallocate
 proceed
```

The new instruction `ifagainwakego` *label* jumps to *label* if `WAKE` is not `nil`. Since the procedure `xcall1/1` is deterministic when called with a suspended goal tree, it does not leave any

choicepoints on the stack. Regarding the determinism, there might be compilers having trouble to generate indexing code so that choicepoints are created. This is the first reason that the builtin procedure is implemented in WAM code. The other reason is that during the processing of the suspended goals a depth first processing of suspended goals shall be not allowed. Thus, if the register `WAKE` is not `nil` a new suspended goal tree is waiting to be processed. This is indicated in fig. 5.2 by the arrow from the `xcall` box to itself.

5.3.4 Restoring the old state

Restoring the old state takes place after the procedure `xcall/1` has finished. There are two different restoring procedures: When an exception was invoked initiated by `call` or `execute`, a restoring procedure including argument registers must be performed. Restoring after a `proceed` makes the handling of argument registers senseless. Besides the argument registers, the old program counter and, by deallocating the environment frame, the continuation register `CP` and the environment pointer `E` must be reloaded.

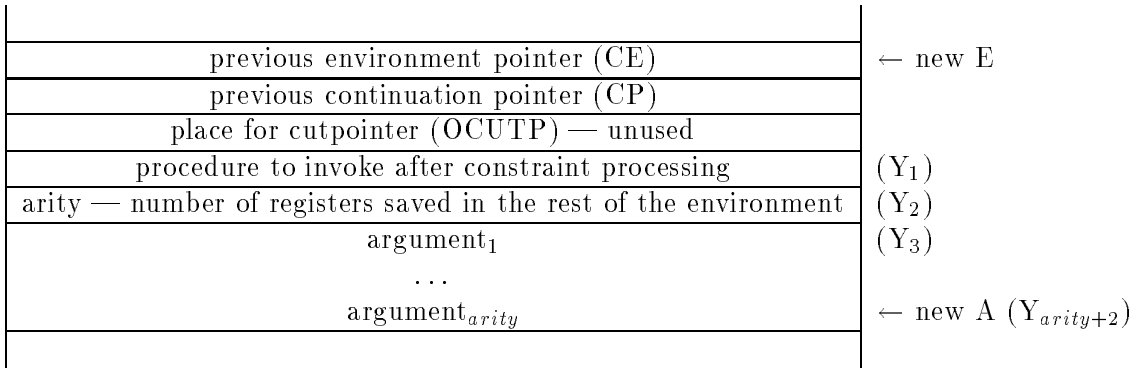


Figure 5.3: The memory layout of an environment frame holding the WAM state for execute/call

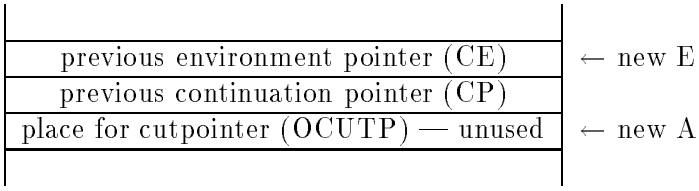


Figure 5.4: The memory layout of an environment frame holding the WAM state for proceed

5.4 Caveats

To preserve soundness, suspended goals which might be still waiting, should be detected. The simplest way (as in PROLOG II) is to put the burden on the programmer not to suspend goals infinitely. Another method is to fail which leads to incompleteness. Our "method" is to collect the delayed goals at a final `proceed` and give the list of suspended goals — which is neither a good choice: some meta-predicates (e.g. `setof,not`) may leave suspended goals and it is questionable how to deal with them.

Another citisism that we have a large memory consumption when re-freezing of variables is performed induced by "higher level" control predicates built upon freeze (see 5.1). In the following design in chapter 6, we will cut down memory consumption to a single constraint delay record.

Another caveat is due to the "high level" integration of the freeze primitive, thus unifying two suspensions with two suspended goal tree is handled de facto on the PROLOG level. A deeper integration into e.g. the unify routine is an appealing idea.

Chapter 6

The Busy Constraint WAM

In this chapter we develop an extension of the WAM closely based on ideas in chapter 5. Many extensions in this chapter are reused in chapter 7 (e.g. the locality of the delay records). Therefore, this chapter contains implementational issues not repeated in chapter 7.

The term "Busy Constraint WAM" is inspired by the busy wake of constraints whenever a suspended variable is bound. This is partly unnecessary work which we analyse at the end of the chapter. If we had omitted this chapter and only presented our final concept in chapter 7, some design features in chapter 7 could hardly be motivated.

Conceptually, we have to specify how control is done according to the lookahead and forward control regimes. The hope is to get an similar simple method as in chapter 5 for the `freeze` builtin.

6.1 Variable representation

In this WAM and in the WAM following in the next chapter, we have four different flavours of variables: Ordinary variables are those which are found in PROLOG. Suspended variables have a tree of constraints (or delayed procedures) in their additional cell. Domain variables are "restricted" to a finite set of values to which they can be bound. Suspended domain variables have a domain and a set of constraints attached to it. The representation is given in fig. 6.1.

6.2 Memory consumption

Let us again look at the code of `freeze` in 5.1. The first clause delays the variable `X` (consecutively) and builts a structure "`freeze(X,G)`" on the heap[1]. We will call such a structure *delay record* although it is not different from the representation of ordinary structures in the WAM. In the delay record the procedure to be called is saved. Unfortunately the `freeze` builtin generates the useless structure over and over again. Even worse, we have to handle each variable in a constraint which results in further useless creations of delay records.

[1] The structure appears in the body of the clause which is built with the `put_structure` instructions, which writes the structure onto the heap and sets the mode to write mode forcing the following unify or build-instruction to write on the heap.

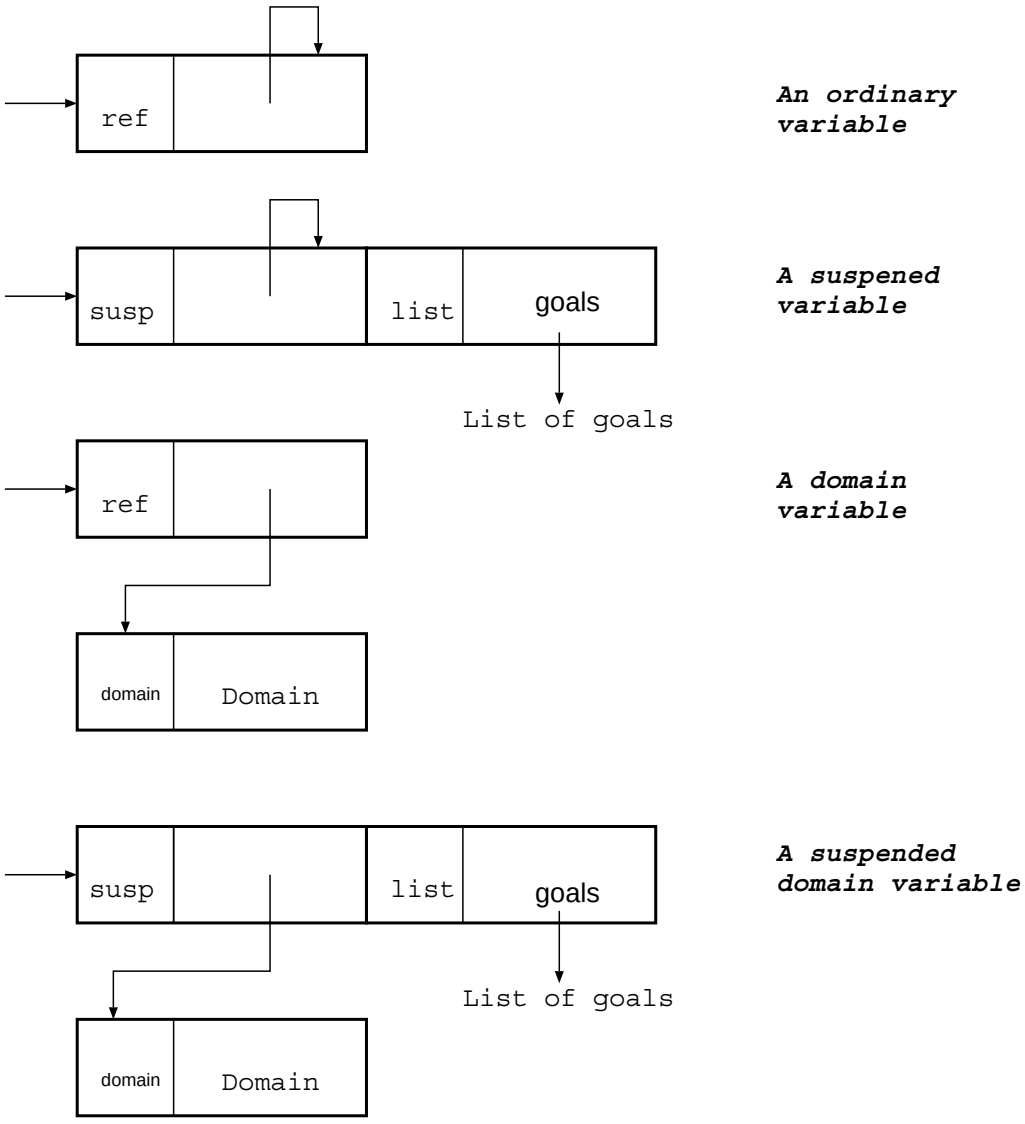


Figure 6.1: Variable representation

6.2.1 What do we have to expect in Finite Domain Programming

Consider that a huge domain variable is many times reduced to a smaller domain. Of course, we can not avoid the creation of new domain variables as described in the unification algorithm in fig. 3.1, but we should try to avoid the creation of any useless delay record. Furthermore, if a constraint is dead and there is only one delay record, we can notify this fact in one place. However, it should be noted that we can not do this without minor modifications to the model discussed in chapter 5.

6.3 Constraint handling model

The actions a constraint has to perform (during its lifetime) models the following steps which can be extracted:

- **constraint initialization routine**: This portion of code in a constraint is called when the constraint is invoked. It duty is to check whether it can directly jump to the constraint body and forget about the constraint (lookahead and forward constraints), or whether it must create a delay record and suspensions for the constraint variables. Lookahead must create the delay record and nevertheless invoke the constraint body. The last alternative is that the constraint body can not be called, so a delay record must be created and the argument variables must be suspended.
- **constraint testing routine**: We have seen in chapter 5 that the control primitive is called whenever a variable is bound. When this happens, the constraint firing conditions have to be checked. If a constraint is applicable and is dead afterwards, it is marked in the delay record — which must be unbound upon backtracking. Finally, either the constraint body is called or the constraint is redelayed *without* generating a new delay record.
- **constraint body** It contains the routine to ensure consistency. For the lookahead procedure the consistency procedure must be modeled according to point 2 in Def. 19. For the forward consistency procedure, we should obey point 2 in Def. 13.

6.3.1 Compilation impacts

Concerning PROLOG to WAM compilation, we have to recall an obstacle inherent in nearly any compilation model: Given a PROLOG clause, compilation is sequentially from left to right, so we can say which instruction belongs to which argument place in a (usual) compilation model. Violating this compilation principle will result in difficulties when extending some compilers with constraints. For example, compiling for example the clause

```
p(a,b,c).
```

yields the following code:

```
p/3:
  get_constant a,1    ; 1st argument
  get_constant b,2    ; 2nd argument
  get_constant c,3    ; 3rd argument
  proceed
```

So, given a lookahead or forward declaration (see def. 14,20), e.g. `forward p(d,g,d)`, we would like to have an initialization code *resembling* the following scheme:

```
p/3:

...

  fdomain  ...,1    ; 1st argument
  fground  ...,2    ; 2nd argument
  fdomain  ...,3    ; 3rd argument

...
```

The specification of the code represented by the dots will be given below.

Another desire is to have the initializing code and the testing code to be *physically the same*. Besides little memory savings the scheme is inspired by similarities in the WAM: The unify instructions work different depending on a mode register (read/write mode). When the above code is called from an ordinary PROLOG program in order to built the constraint net, we are in a normal processing mode. A delay record is not present and we eventually have to built the delay record and suspend the variables. When doing constraint processing, we should be able to tell in the above instructions that we are not in the normal processing mode and have access to the delay record which *must* be present since the constraint is already delayed[2].

To summarize, we request the following:

- *locality* of the delay record.
- we must determine whether we are in normal processing *mode*.
- initializing code and testing code should be the same.
- simple compilation scheme for the lookahead/forward declarations.
- easy detection whether a constraint is dead.

6.4 Solving the problems

6.4.1 Locality and mode problem

The locality problem is to be considered as crucial. If we do not solve it, huge memory consumption must be expected. Let us look at the routine, which is the source of the problem (see the procedure `xcall` in section 5.3.3): When processing the suspended goal tree the routine meta calls the delay record. The called routine does *not* have access to the pointer of the structure representing its invocation. The solution is to have an additional register `DLYRCRD` set by the meta-call instruction pointing to the delay record. If this register is set to nil upon failure and upon the control instructions, we have an indicator whether we are in normal processing mode or whether we do constraint processing.

[2]Please note, that we can not invoke a new constraint when in non-user mode. This is not true for the WAM presented in chapter 7.

6.4.2 Detection of dead constraints

If we are going to modify the metacall instruction, we can also put the burden of detecting dead constraint into this routine.

As already mentioned in section 5.3.3, other constraints may be woken during constraint processing. The WAM routine handling this fact is given below. The instruction `ifagainwakego` *Label* jumps to *Label* if constraints have been woken during constraint processing. The control instructions with the suffix "!" are an optimized version of the control instruction. In section 5.3.2 we described that the control instructions `execute`, `call`, `proceed` may be interrupted when goals are woken and we are not doing exception handling. When being in `xcall/1` we *know* that we are doing exception handling and save the check by the optimized instructions[3]. We are providing the `xcall/1` routine in WAM code, because a formulation in PROLOG is hardly possible. (in contradiction to the `xcall1/1` procedure which is given in PROLOG code)

```
xcall/1:
 allocate 0
L0
 call! xcall1/1,1,0
 ifagainwakego L0
 deallocate
 proceed!
```

The `xcall1` procedure contains a new version of `mcall`, namely `mcallifactive` which previously checks, if the constraint is alive. If the constraint is dead, the effect of the operation is just returning immediately, otherwise it has the effect of a metacall.

```
xcall1(S) :- struct(S),mcallifactive(S).
xcall1([P|L]) :- struct(P), mcallifactive(P), xcall(L).
xcall1([[L1],L2]) :- xcall([L1]), xcall(L2).
```

6.4.3 Simple compilation scheme for the lookahead/forward declarations

Let us first recall which conditions are to be examined if we determine whether a constraint shall be fired. Those argument places where a "g" was specified, must be ground. If there is an argument which is nonground, we can not invoke the constraint body. Let us have a new register named `FIRE` which is set to `true` by the new WAM instruction `fire_true`. This flag is affected by the fground instruction set to `nil` when an argument specified "g" if nonground.

Let us have another register `COUNT` which counts the domain variables. It is initially set to 0 by the WAM instruction `count_0` and incremented by 1 when a fdomain instruction detects a domain variable.

Both `fdomain` and `fground` instructions have to do other tasks. Therefore, we look at their complete syntactic appearance:

`fdomain` *arg label arity*
`flookahead` *arg label arity*

[3]These "optimized" instructions are semantically identical to the corresponding WAM instructions without the constraint extensions.

Consider, that `fdomain` and/or `fground` are invoked with `DLYRCRD` = `nil`. Then the constraint is called for the first time and we have do see the invocation as an initialization step. Both fdomain and fground instructions suspend variables in the terms given by the argument register *arg*. The delay record to build is specified by *label*. The number of arguments to save in the delay record is given by *arity*. The first fdomain or fground instruction forced to build a delay record saves the pointer in the register `DLYRCRD`, so subsequent `fdomain/fground` instructions have access to this record.

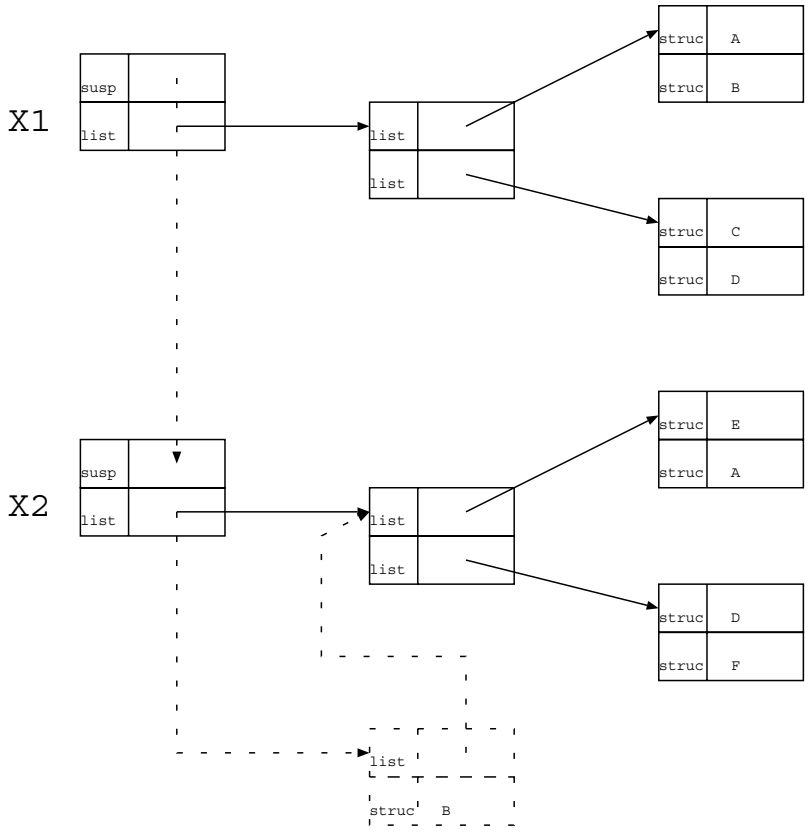


Figure 6.2: Inheritance of constraints in the Busy WAM

Consider, that `fdomain` and/or `fground` are invoked after a constraint variable has been touched. The register `DLYRCRD` points to a delay record which must be inherited to any variable bound to a suspended variable. Thus, a constraint tree is successively rebuilt by the single constraints delayed.

This can be seen in fig. 6.2. The suspended variable called X1 is bound to X2 and the suspended constraints of X1 are woken. The constraints A,B,C,D are successively called each processing their arguments with fdomain/fground. When A is invoked, the fdomain/fground checks verify that A is already in the suspended goal tree. The figure indicates the action performed by invoking the constraint B. The constraint code of B verifies that B is not in the suspended goals of X2 and allocates a new cell with B and adds it to the suspended goal tree.

When we have checked all the arguments, we must decide whether to fire or not. The end of the init/testing code for forward checking constraints is `forward_nofire_proceed`. The algorithm is

```
if (not FIRE) or (COUNT > 1)
  then proceed
```

```
   else bind(DLYRCRD.fired, true); /* continue with next
                                      instruction (constraint body) */
```

The corresponding algorithm for `lookahead_nofire_proceed` is

```
 if (not FIRE)
   then proceed
   else if (COUNT = 1) then bind(DLYRCRD.fired, true);
```

As already mention there is a placeholder in the delay record indicating that the constraint is not fired. In the code, this place is referenced by `DLYRCRD.fired`. It is "set" by a bind operation, since it must be rewoken upon backtracking.

6.5 Unnecessary Work

- a delay record may be allocated without need: Consider we have a predicate p declared `p(d,d)`. The first variable is a domain variable, the other a constant. So the constraint satisfies the forward firing condition. When we test the first argument with `fdomain`, we have to build a delay record since we do not (yet) know that the second argument is ground.
- since the a constraint has to do the inheritance of the suspended goals, it has to search for its entry in the suspended goal tree and if it is not present, the constraint must be added to the suspended goal tree. Thus we have unnecessary memory usage in rebuilding a suspended goal tree which is already present in memory.
- constraints are woken although their firing condition is not satisfied, since constraint checks for firing condition in the code presented.
- Due to the simple wakeup mechanism, constraints are woken several times and the checks are repeated.

In fig. 6.3 the suspension tree is visualized after the first queen is placed in the first row in a 4×4 queens problem. Nine constraint are correctly woken. In the constraint bodies of the 9 woken constraint, the other variables are touched. As soon as they are touched by `bind`, they are woken. The woken constraint tree can be seen in fig. 6.4. It should be noted that the tree is unnecessarily woken. The round nodes are list nodes where the number indicates the address and the leafs are delay record given identified by their address. A careful reader may notice that delay records are present in the tree occuring more than once. In fact, the underlying structure is a directed acyclic graph. Using a DAG representation see fig. 6.5, we can directly see that constraints are woken twice. The size of the DAG is not very harmful, but the same situation in the eight queens example reveals a frightening growth of the useless work. To give an *impression*, the graph to be traversed is given in (fig. 6.6).

Address	Structure	Alive
18	DUNGL(1,24)	
26	DUNGL-(1,24,1)	
35	DUNGL+(1,24,1)	
44	DUNGL(1,50)	
52	DUNGL-(1,50,2)	
61	DUNGL+(1,50,2)	
70	DUNGL(1,76)	
78	DUNGL-(1,76,3)	
87	DUNGL+(1,76,3)	

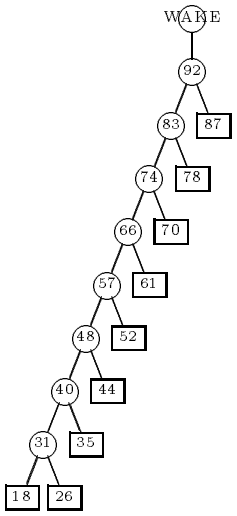


Figure 6.3: Wakup tree immediately after setting the first queen in row 1

Address	Structure	Alive
18	DUNGL(1,186)	+
26	DUNGL-(1,186,1)	+
35	DUNGL+(1,186,1)	+
44	DUNGL(1,194)	+
52	DUNGL-(1,194,2)	+
61	DUNGL+(1,194,2)	+
70	DUNGL(1,202)	+
78	DUNGL-(1,202,3)	+
87	DUNGL+(1,202,3)	+
96	DUNGL(186,194)	
104	DUNGL-(186,194,1)	
113	DUNGL+(186,194,1)	
122	DUNGL(186,202)	
130	DUNGL-(186,202,2)	
139	DUNGL+(186,202,2)	
148	DUNGL(194,202)	
156	DUNGL-(194,202,1)	
165	DUNGL+(194,202,1)	

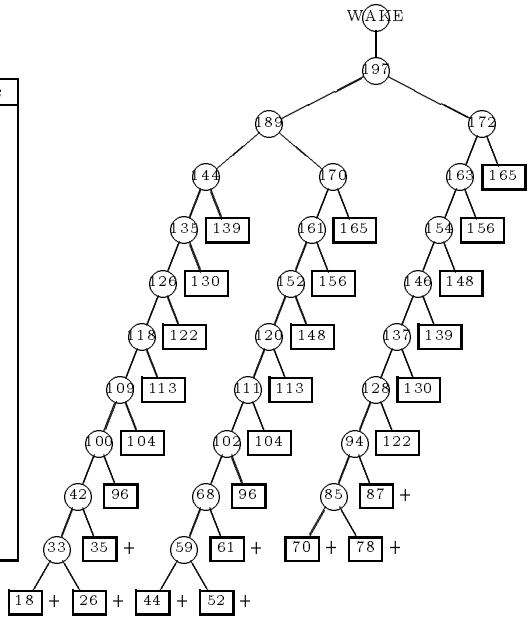


Figure 6.4: Wakup tree after first propagation

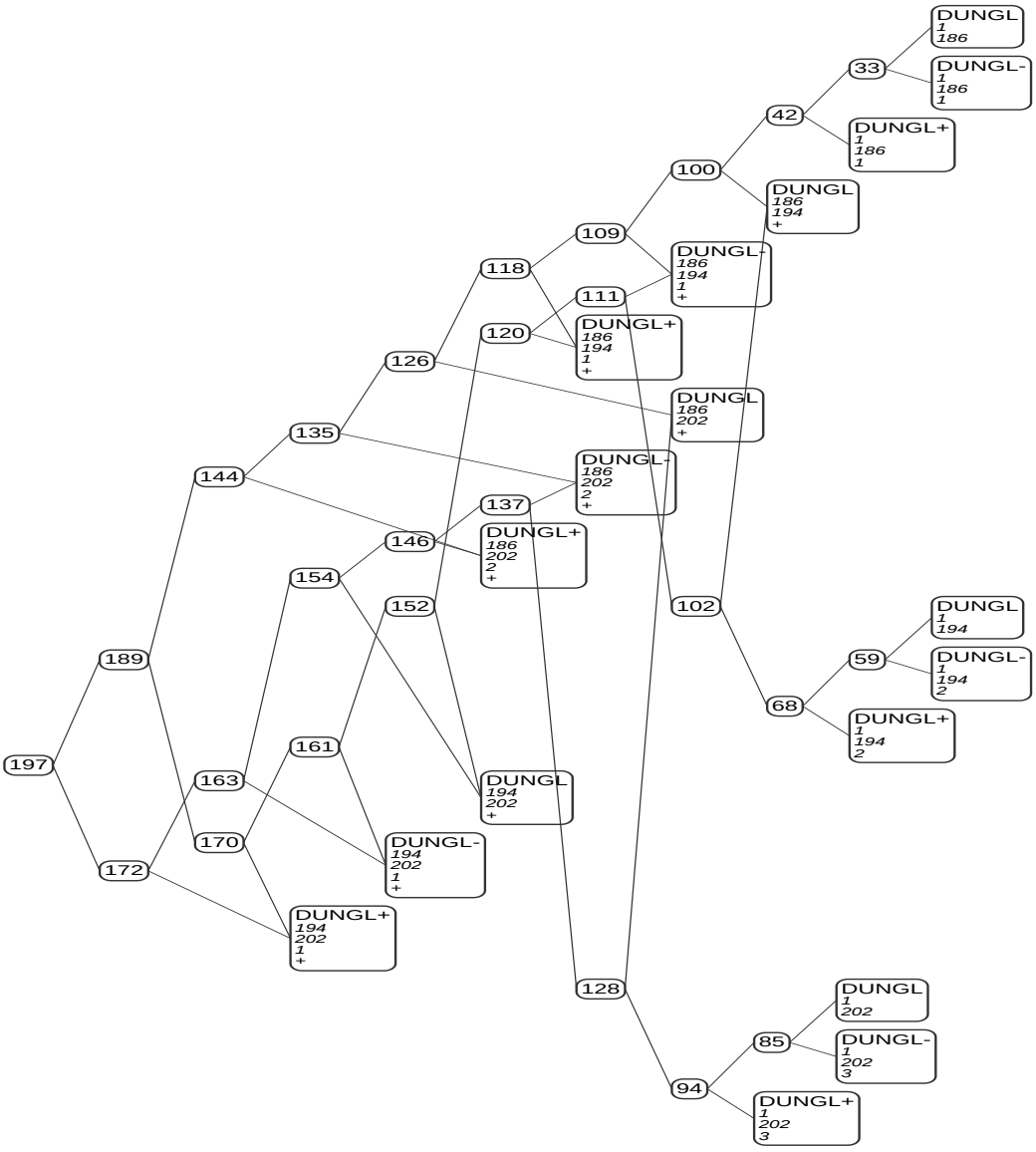


Figure 6.5: The woken goal tree in the four queens example in DAG representation

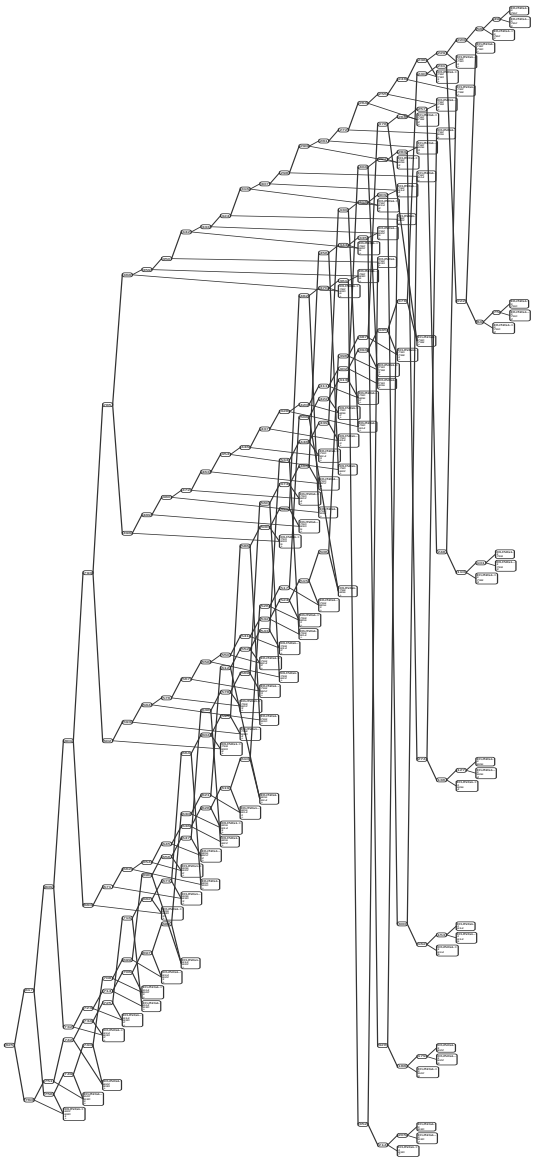


Figure 6.6: The woken goal tree in the eight queens example in DAG representation

Chapter 7

The optimized WAM

In this chapter, we will make a redesign of the WAM proposed in chapter 6. We will shortly outline our approach: The handling of the suspended goals should not be done by control primitives. These had to check, whether the constraints are already delayed in the variables suspended goal list and therefore had to search the suspended goal tree for a specific delay entry. If we implement the handling in the unification routine, we can save the work. The idea of having constraint initialization code and constraint testing code in physically the same code turned out to be inefficient. We will seperate constraint initialization code and constraint testing code. We will even put constraint testing code very deep into the bind mechanism forcing the goals of a delayed variable not to be woken when the variable is touched, but when the wakeup tests say that a constraint can fire. The idea of splitting the constraint initialization code into sperate WAM instructions was motivated to have a set of WAM-alike instructions. We give up this desire and put the initialization code into a single instruction. The idea of having a single delay record in the system turned out to be a good approach.

The compilation of for this WAM is described in chapter 8. You may want to have a look at the compilation of the added control constructs first.

7.1 The lookahead and forward instructions

The lookahead and forward instructions initialize the constraint by

7.1.1 The forward instruction

The task of the forward instruction is to check whether the constraint can already fire. In that case the forward instruction behaves like an execute instruction. The constraint must not be delayed, since forward constraints can only be fired once. If the constraint can not fire, a delay record must be created and the variables must be suspended. In the following instruction `arity` is a number indicating the number of arguments, spec is a list of length `arity` containing "g" and "d" specifications for the arguments. `X_i` accesses the argument register X_i, `spec_i` references the the "g" or "d" specification in the specification list. `Ground, ordinary_variable, suspended_variable, domain_variable, suspended_domain_variable, constant` and `ground` are tests to check if the argument is ground, an ordinary variable, etc. The procedures `inherit-goal` and `new-delayrecord` are described later.

We give the algorithm in a PASCAL like form:

7.1.2 The lookahead instruction

The lookahead instruction is more complex than the forward constraint. First, we have to check, whether the lookahead procedure is lookahead available. If there is only one domain variable left, we can directly jump to the constraint body. If not, we must allocate a delay record, delay the arguments and jumps to the constraint. If the constraint is not lookahead available, we either fail or delay the constraint. The notation is similar to the notation in the forward instruction.

We give the algorithm in a PASCAL like form:

7.2 The creation of a delay record

The procedure `new-delayrecord` (*label*,*arity*,*spec*,*type*) allocates the structure in fig. **??** on the heap. A delay record contains all information to wake a constraint. Furthermore it contains the specification list of the constraint and a marker indicating whether the constraint is dead or alive. The bind mechanism, which accesses the stored information must know which type of constraint is present (forward, lookahead). We are having a special tag `dly` for a delay record for finding unfired constraints at the end.

7.3 Inheriting Goals

When delaying a constraint, the variables in the constraint must get an entry of the constraint's delay record, so when a variable is bound it can be checked whether the constraint can fire or not. With the notation Var.domain, we access the domain of a domain variable and Var.goals references the suspended goal tree of the variable. `Ref.car` (`Ref.cdr`) access the car (cdr, resp.) of a list.

The algorithm for inheriting suspended goal lists is recursive:

```
inherit-goals (gl ref):

if (ordinary_variable(ref)) then
    bind!(ref, new-suspended-variable(gl)) else
 if (domain_variable(ref)) then
    bind!(ref, new-suspended-domain-variable(gl,Var.domain)) else
  if (suspended_variable(ref)) then
     bind!(ref, new-suspended-variable (new-goalnode(ref.goals,gl))) else
   if (suspended_domain_variable(ref)) then
      bind(ref, new-suspended-domain-variable(new-goalnode(ref.goals,gl),ref.domain) else
    if const(ref) then begin end else
     if list(ref) then begin
      inherit-goals(gl,deref(ref.car));
      inherit-goals(gl,deref(ref.cdr)); end else
       if struct(ref) then for i:=1 to ref.arity do
         inherit(gl,deref(ref.args[i]));
```

7.4 Binding mechanism

We have now all the structures to be able to look at the binding algorithm. The task is to bind a reference to a certain value checking whether the location must be trailed. If we are doing

a binding to a delayed variable, we are taking the goal tree of the variable and are recursively checking whether a single constraint can fire. This is done by `wakeup`. The wakeup algorithm is given in the following:

Wakeupstruct is the procedure testing wether a structure can be fired or not. It marks the constraint, if it will be dead after firing and appends it to the woken constraint list. With `struct.fun`, `struct.arity`, `struct.notfired`, `struct.cnsttype` and `struct.arg_i` we access the functor, the number of arguments, the flag telling whether the constraint is dead, the constraint type and the arguments of the structure, resp. The & is the address operator known from C. The aim of the wakeup1 procedure is to append a woken structure to the already woken constraints.

```
wakeup(struct):
if struct.nonfired
 then if (struct.cnsttype = forward)
  then begin
        count:=0; wake:=t; i:=1;
        while (wake and (count <=1) and (i<=arity)) do
         begin deref(X_i);
          case (spec_i) of
           'g': wake := ground(X_i);
           'd': if (ordinary_variable(X_i) or suspended_variable(X_i))
                 then wake:=nil else
                  if (domain_variable(X_i) or suspended_domain_variable(X_i))
                   then count:=count+1 else
                    if (not(constant(X_i))) fail;
          end;
          i:=i+1;
        end;
        if (wake and (count <= 1)) then
         begin wakeup1(ref);
               bind(&struct.notfired,true);
         end;
        end else /* structure is lookahead */
        begin
         count:=0; wake:=t; i:=1;
         while (wake and (i<=arity)) do
          begin deref(X_i);
            case (spec_i) of
            'g': wake := ground(X_i);
            'd': if (ordinary_variable(X_i) or suspended_variable(X_i))
                 then wake:=nil else
                 if (domain_variable(X_i) or suspended_domain_variable(X_i))
                     then count:=count+1 else
                 if (not(constant(X_i))) fail;
             end;
             i:=i+1;
            end;
          if wake then begin if (not (w1member ref)) wakeup1(ref);
                             if (count <= 1) then bind(&struct.notfired,true);
end;
```

7.5 Waking a structure up unconditionally

When we are at the point where we can wake up a constraint unconditionally, it is desirable that we are building a wakeup list and no tree. Furthermore, we want to invoke the constraints in the order they occurred.

We are having two registers `WAKE` and `WAKEE` for the wakeup. `WAKE` points either to `nil` or to the head of the wakeup list, the register `WAKEE` points to the last list element which is modified destructively to ensure the proper order on the wakeup list.

The algorithm to wake up one structure is:

```
wakeup1(ref):

 if (WAKE = nil) then begin WAKE:=new-list-cell;
                            WAKEE := WAKE;
                            new-value(ref);
                            new-value(nil);
                      end
                 else
                      begin WAKEE.cdr := new-list-cell;
                            WAKEE := H;
                            new-value(ref);
                            new-value(nil);
                      end;
```

subsectionThe new XCALL Routine The new xcall routine can be made simpler compared to the routine in section 6.4.2. There we had to cope with woken goal *trees*, here we must handle goal *lists*. to a simpler version, which must not check whether the constraints are already dead. If they are dead, they are not woken by lookahead or forward constraint testing code in the binding mechanism. We will see in section 7.8 that we must handle internal interrupts. Up to now the `xcall` routines could be written as:

```
xcall/1:
 allocate 0
L0:
 call xcall1/1,1,0
 ifagainwakego L0
 deallocate
 proceed
```

and `xcall1/1` reads:

```
xcall1([]).
xcall1([S|L]) :- mcall(S),xcall1(L).
```

7.6 The Unification Procedure

In the following we will examine the different cases, the unification routine has to deal with. Whenever possible, we will give a graphical representation of the unification work. In the figures, the dotted boxes are words altered during unification, the hatched boxes are new allocated memory cells on the heap. In the following we describe what happens in the unification procedure with the different sorts of variables. In the figures we will show the "interesting" cases: E.g. when unifying two domain variable we will show the case where the intersection of the domain is neither empty nor a singleton.

7.7 Binding stategies

Binding a variable to a value involves saving its old value[1] on the trail if the variable is *not* deallocated upon backtracking. If the variable to be bound is a suspended variable or a suspended domain variable, the constraints belonging to the variable have eventually to be appended to the list of woken constraints.[2]. When handling variables, we somtimes already know which type of variable is to be bound. Therefore, we have a `bind` function checking if the variable to be bound is a suspended variable or suspended domain variable. The `bindw` functions is for variables known to be delayed variables. Their goal tree must be woken in every case. The `bind!` function is for bindings where the variable is an ordinary or a domain variable.

Concerning unification, in some cases new data structures have to be created (e.g. when unifying two domain variables, a new domain variable *may* be created). In some cases, we can use some existing memory portions from the structures to be unified, modify them and use trailing to make the modifications reverseable. Since trailing is a costly operation, the above mentioned optimizations must be considered with great care. It should be assumed that a trailing operation costs two words (one for the address and one for the previous value). Trailing must only be done when the variable is not deallocated by the next backtracking step.

7.7.1 Unification of two ordinary variables

The unification of two ordinary variables (fig. 7.1) is more difficult as it first seems to be. This arises from the fact that ordinary variables can be found in the stack and in the heap. To avoid dangling references bindings have to be done from the variable created later to the variable created earlier. Thus, if we are going to deallocate a variable on the stack by environment trimming or last call optimization, we can never obtain a reference from a variable to a variable being deallocated. Another aspect is that references must not point from the stack to the heap, since the variables on the heap are more "persistent" than those on the stack. Heap addresses are lower than stack addresses (see fig. **??**) and older variables have lower addresses than younger variables. So the binding has to be done from the higher address to the lower address. The binding is performed by the `bind!` function.

[1] In normal WAM systems only the address of a variable must be stored — The contents of the memory location is its own address (which is the represenation for an unbound variable).

[2] In the Busy WAM a goallist is always completely woken, in the optimized WAM only the constraints are woken, which can be fired.

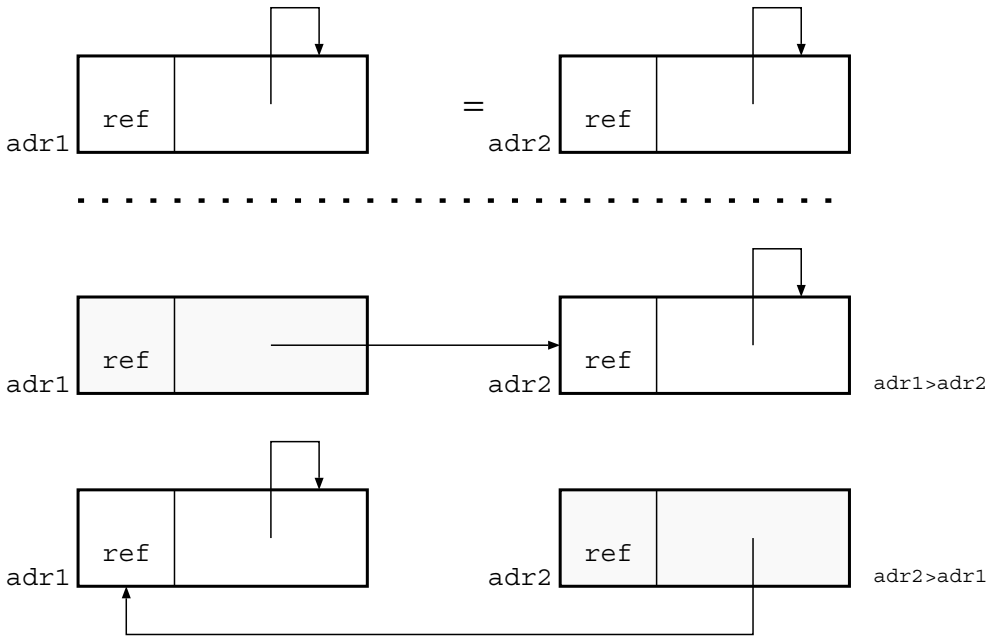


Figure 7.1: Unification of two ordinary variables

7.7.2 Unification of an ordinary variable with a domain variable

A domain variable is located on the heap. Thus, the binding direction is from the ordinary variable to the domain variable. No goals have to be considered, so we call the `bind!` mechanism. The binding can not be done from the first ref-cell (see fig. 7.2) to the domain cell, but must be done to the ref-cell of the variable on the right side. If a successive binding of the domain variable on the right side to e.g. a constant is performed[3], the variable on the left would still point to the domain.

7.7.3 Unification of an ordinary variable with a suspended variable

A suspended variable can only be found on the heap. The binding direction is from the ordinary variable to the suspended variable (see fig. 7.3). We do not have to wake the constraints on the suspended variable, since the binding can not affect any variable in the frozen goal list. In principle, only the ordinary variable has been touched, but not the suspended variable, so we use `bind!` for the binding.

7.7.4 Unification of an ordinary variable with a suspended domain variable

The binding must be done via `bind!` from the ordinary variable to the suspended domain variable (see fig. 7.4). No constraints must be woken since the resulting binding does not affect any constraint firing condition.

7.7.5 Unification of two domain variables

From the theoretical viewpoint, unifying two domain variables means that the variables are bound to a new domain variable where the domain is obtained by intersecting the two domains

[3]This means, the ref-cell is bound to a const-cell with the constant as its value.

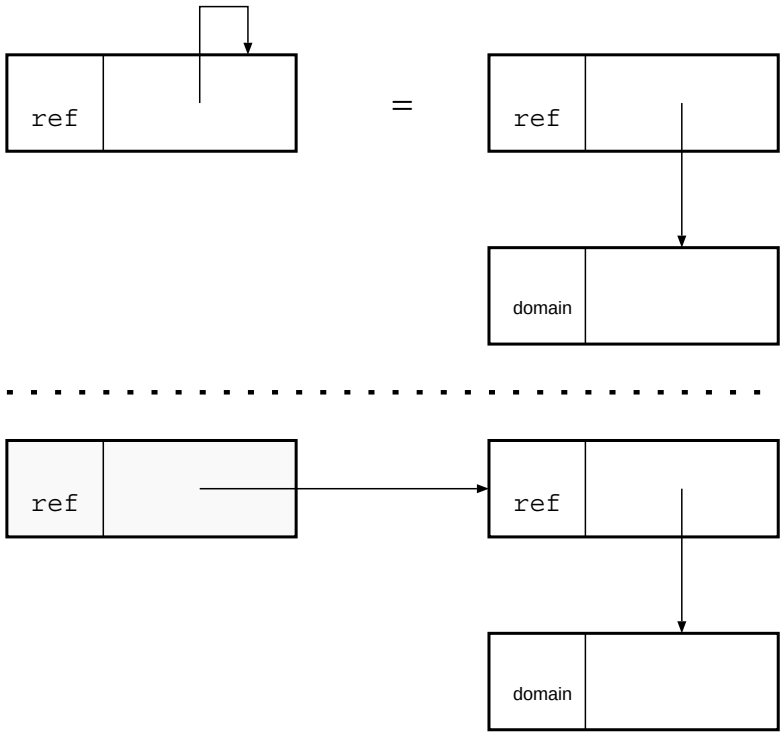


Figure 7.2: Unification of an ordinary variable with a domain variable

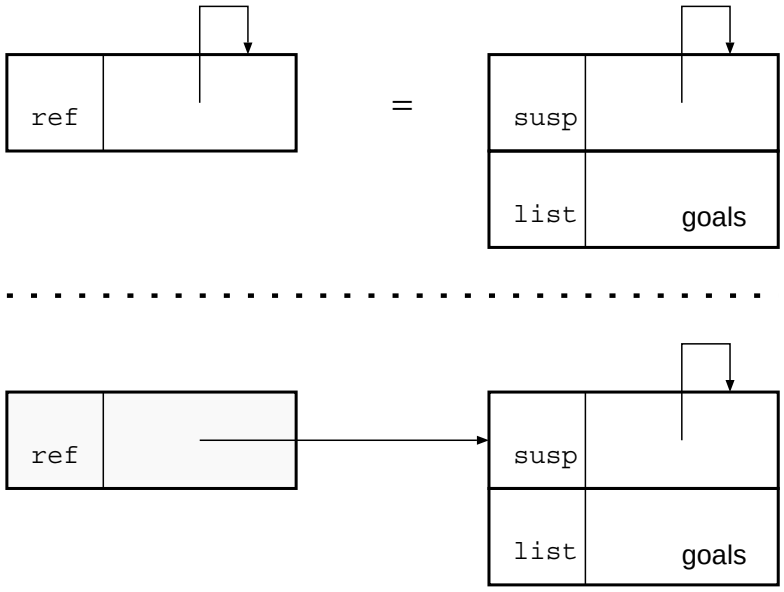


Figure 7.3: Unification of an ordinary variable with a suspended variable

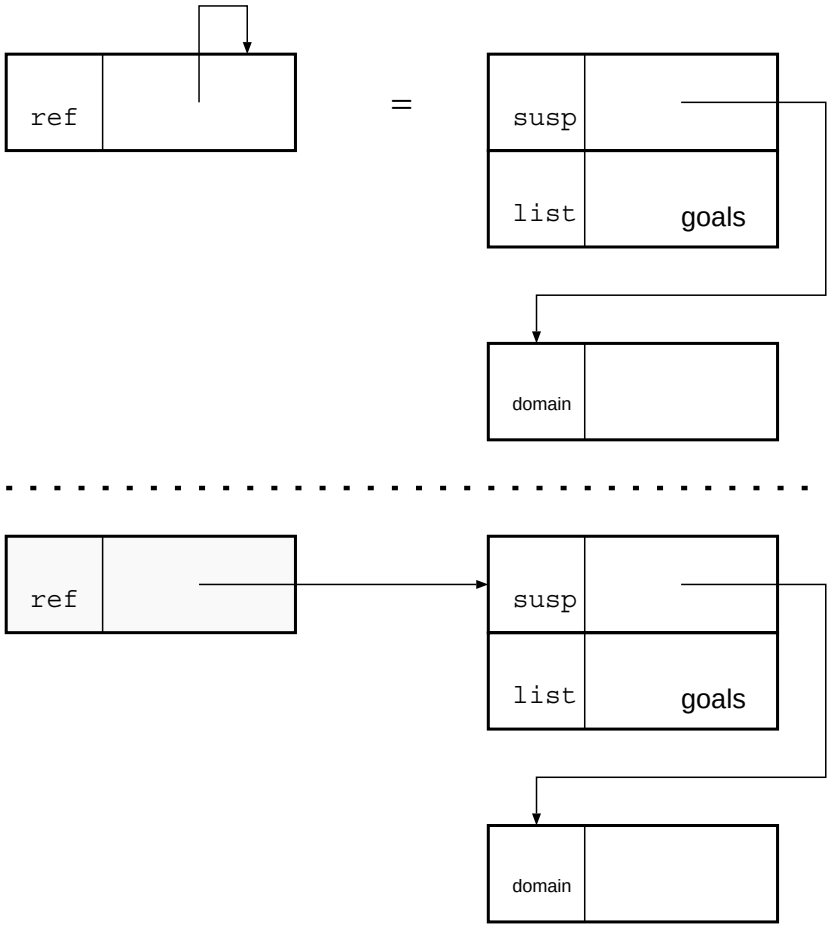


Figure 7.4: Unification of an ordinary variable with a suspended domain variable

(Thus the total memory consumption is 2 new cells on the heap and up to 2 trail entries). A failure is generated when the intersection ist empty. We can save a memory cell by not allocating a full new domain variable: One reference cell on the left side (see fig. 7.5) points to the reference on the right side, assuring that successive bindings affect the two variables. The reference cell on the right side points to the new generated domain. However we can eventually further save the generation of the new domain: If $D1 \subseteq D2$, the right reference cell can be bound to the domain cell $D1$ on the left side[4]. If a singleton is obtained by the intersection of the domains, both variables are bound to the constant. If the intersection is empty, a failure occurs.

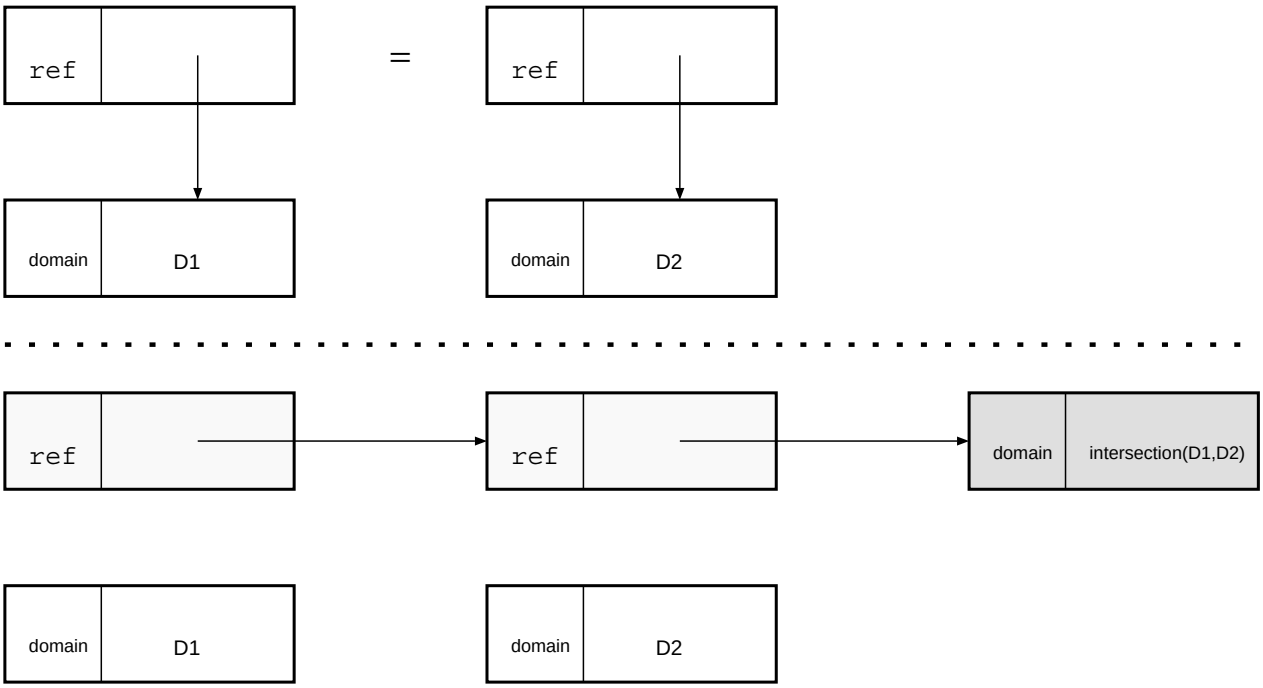


Figure 7.5: Unification of a domain variable with a domain variable

7.7.6 Unification of a domain variable and a suspended variable

Unifying a domain variable and a suspended variable means creating a suspended domain variable. We will not use any new heap memory. (see. fig. 7.6) The reference cell on the left side is bound to the suspension cell while the domain reference is put into the suspension cell. The arising configuration on the left side is a suspended domain variable, while the right side is a reference on the same suspended domain variable.

7.7.7 Unification of a domain variable and a suspended domain variable

Unifying a domain variable and a suspended domain variable results in a new suspended domain variable with the same suspended goal list and an intersected domain (see fig. 7.7). We are only generating a new domain, the domain variable on the left hand side is altered to point to the new domain. The reference on the left hand side is bound to the suspended domain variable. The new domain cell is saved when one domain is a subset of the other. The suspended domain variable (domain variable) is bound by `bindw` (`bind!`, resp.).

[4] similar strategy for $D2 \subseteq D1$.

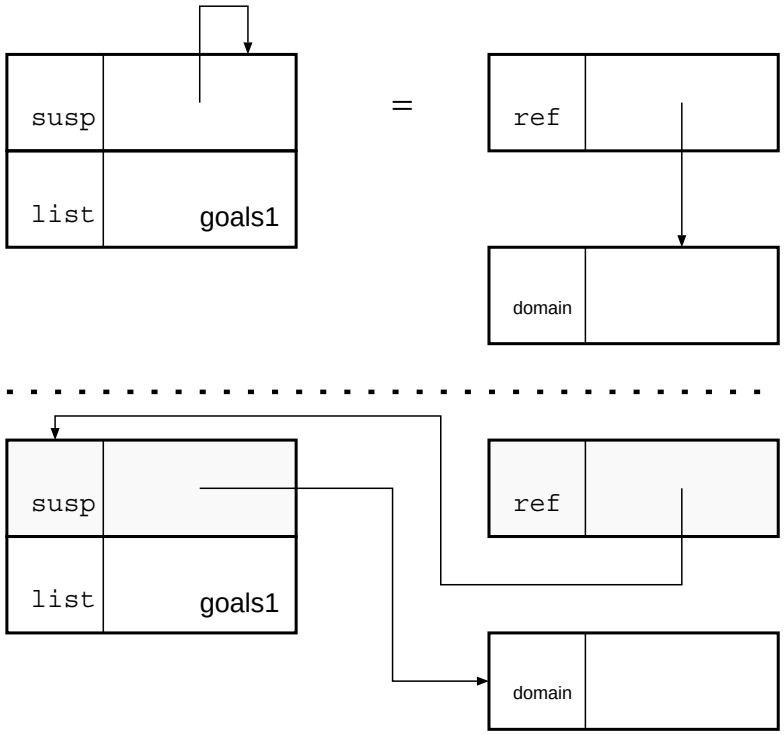


Figure 7.6: Unification of a domain variable with a suspended variable

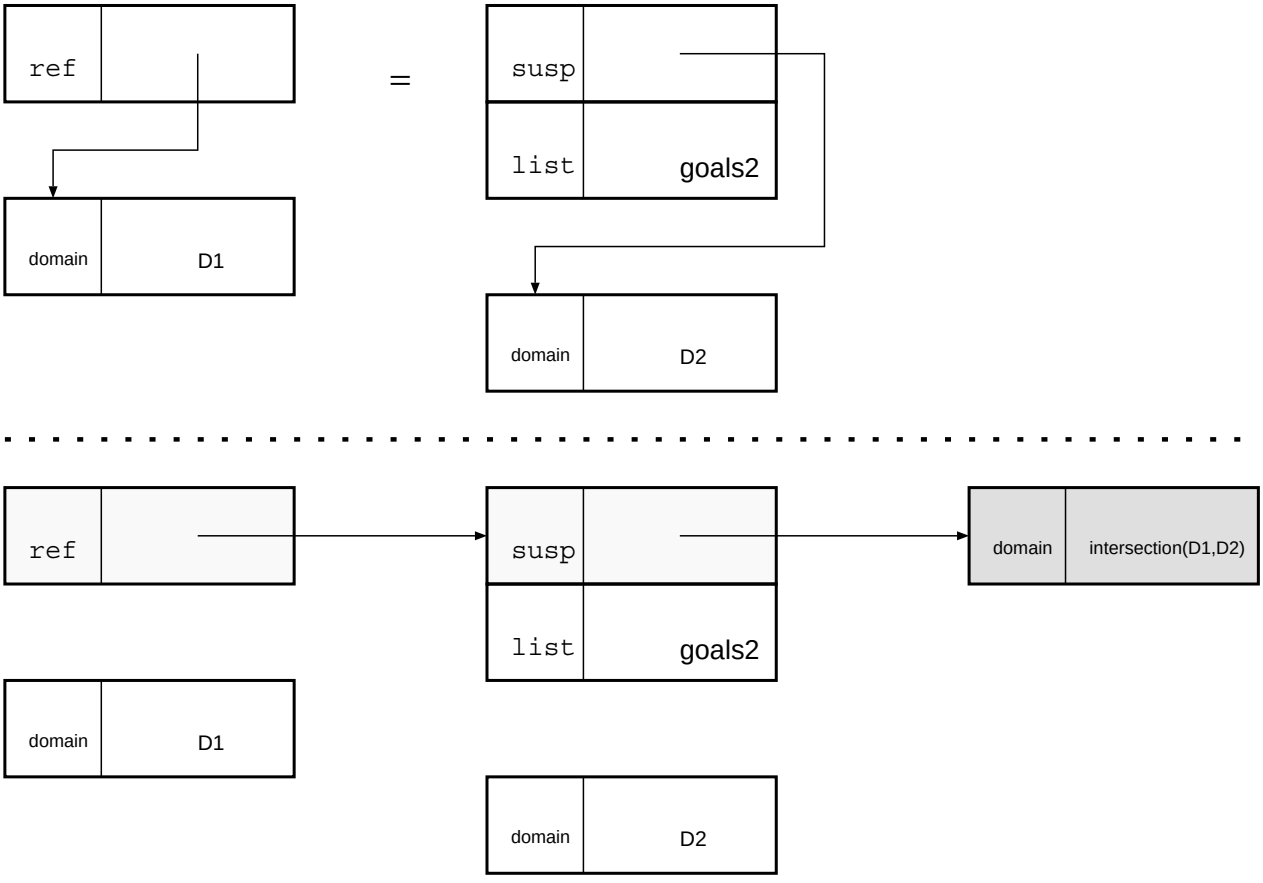


Figure 7.7: Unification of a domain variable with a suspended domain variable

7.7.8 Unification of a suspended variable and a suspended domain variable

Unifying a suspended variable and a suspended domain variable results in a new suspended domain variable whose goals are constructed by consing the two goal trees. When phsically allocating the above new suspended domain variable with the new goal node, we use 8 memory cells: 4 cells for the suspended domain variable with the new goal node and 2 new references from the orginal variables to new the allocated suspended domain variable, resulting in 8 cells (binding needs two cells for each). In the strategy in fig. 7.8 we need 6 memory cells. The goals of the left hand side (goals1) have to be woken, the constraints of the right hand side can not fire.

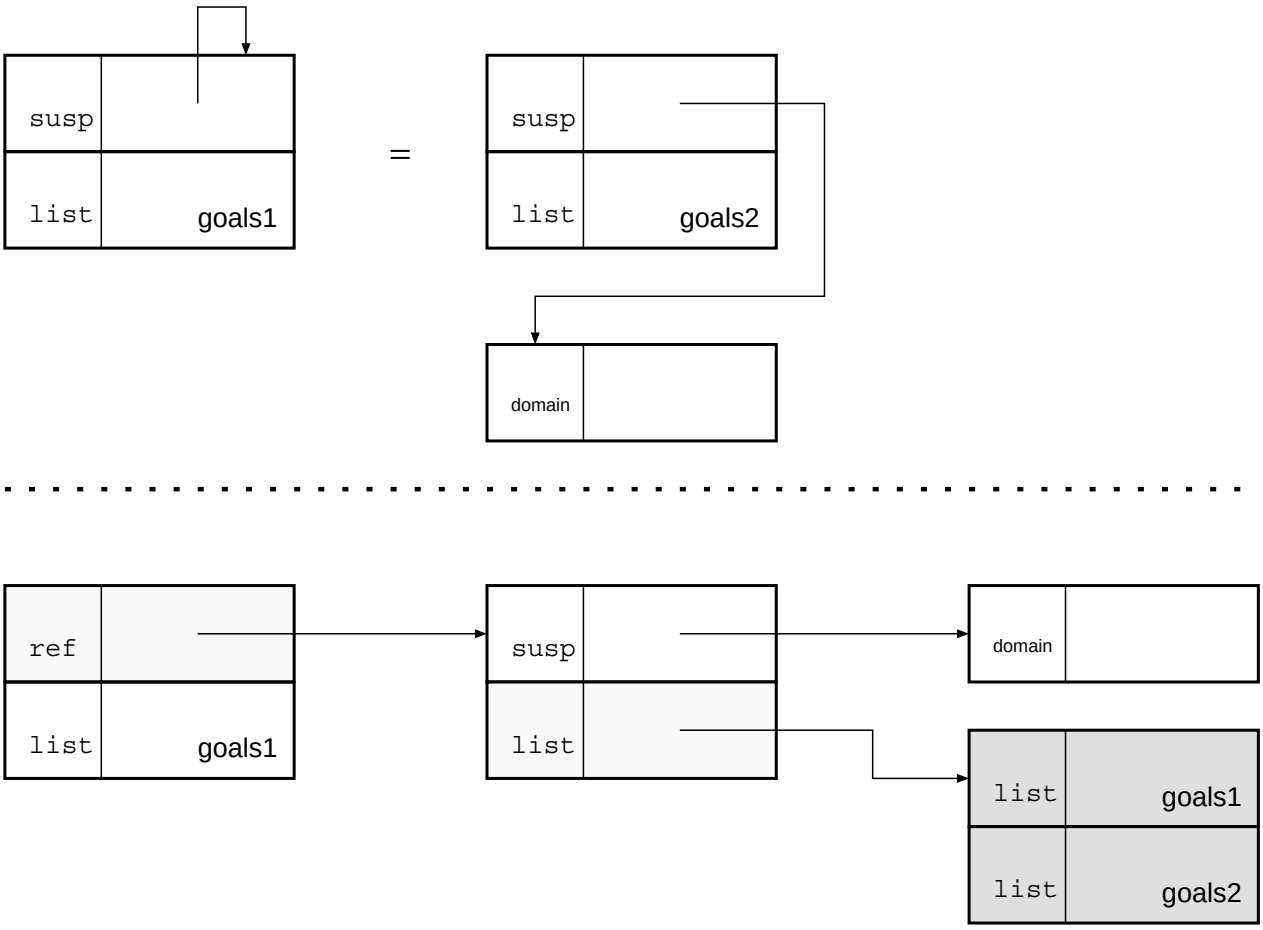


Figure 7.8: Unification of a suspended variable and a suspended domain variable

7.7.9 Unification of a suspended variable and a suspended variable

7.7.10 Unification two suspended domain variables

Unifying two suspended variables is the most complex case concerning the handling of variables. The intersection of the domain variables must be performed, the goal trees of the two suspended variables must be consed together and the goals must be woken. Instead of generating a new suspended domain variable, we are altering the existing structures to fit our needs (see fig **??**) Only a new goal node and a new domain node are needed. If a singleton is obtained by the intersection of the domains, both variables are bound to the constant. If the intersection is empty, a failure occurs.

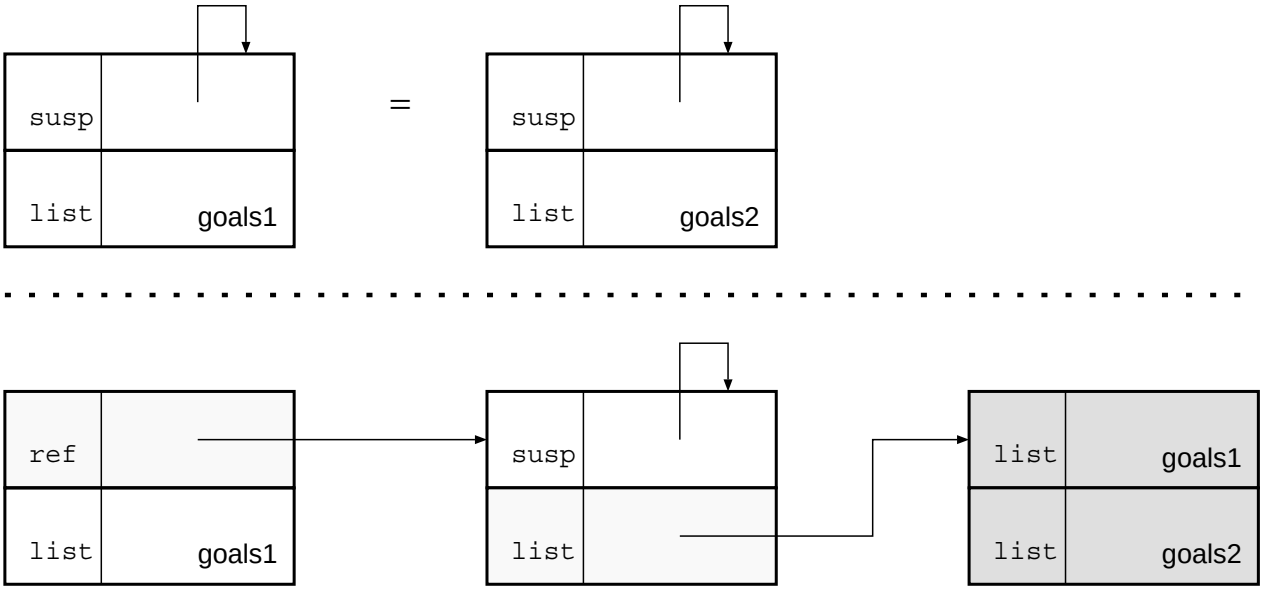


Figure 7.9: Unification of a suspended variable with a suspended variable

7.7.11 The other cases

The rest of the possibilities are similar to the standard WAM unification routine. Since all cases, where two variables are present have been examined above, at most one variable must be considered.

If one of the arguments to unify is an ordinary variable, the ordinary variable is bound by `bind!` to the other argument.

If one argument is a suspended variable, we have to inherit the goals of the suspended variable to the other argument. Inheriting is done by the procedure described in section 7.3. Afterwards, the suspended variable is bound to the other argument by the `bind` procedure.

Another possibility is that one argument is a domain variable. Then the other must be a constant, otherwise a failure occurs. It is bound by the `bind!` function.

The same is true for suspended domain variables, except we are binding via `bindw`, waking the suspended goal tree.

In the following, no variables can occur, since all cases with variables have been handled up to now. Then we check, whether the two arguments have the same tags. If not, a fail is issued.

If the arguments are constants and are equal, all is fine, otherwise a failure is issued.

If the arguments are compound terms, whose functor is equal, the unify routine is called recursively for the elements.

7.8 Problems

In the sections above, we have taken great care that the handling of suspended goal lists is deeply located in the WAM and is not implemented by WAM primitives or any constraint testing code. Unfortunately, we have to face a problem occuring when building new structures

in write mode caused by `get_list` or `get_structure` instructions. Consider a suspended variable[5] to be unified with a compound term (structure or list). Due to the fact that the variable is unbound, the compound term must be constructed on the heap. Assume further that within the structure or list to built is another delayed variable. So we have to construct a union of the two suspended goal lists. Apparently, we are in trouble since during constructing compound term elements, we are not allowed to use more than a cell per element. But we have to construct a new goal nodes which take additional two cells ! We can not allocate these when we are building compoud terms. Since building compound terms is handled by a sequence of instructions, we have no chance to allocate the new goal cell behind the structure to be allocated. This leads to the concept of *internal interrupts* to delay computation which can be performed when the term has been constructed.

7.8.1 Invoking internal interrupts

Internal interrupt entries are appended to the *beginning* of the woken constraint list (compare to section 7.5). They happen when a `get_list` or `get_structure` instruction is called whose argument is a suspended variable. *Before* the structure is built on the heap, an internal interrupt entry is allocated in the heap. The layout of the interrupt record is shown in fig. **??**.

7.8.2 Handling internal interrupts

Internal interrupts must be handled be the `xcall/1` mechanism. In the implementation, we have list cells for clueing the list entries of *structures* and *references* together. Structure pointers are references to delay records, ref-cells point to internal delay records. So, when dealing with internal delay records, we must explicitly handle ref-cells normally invisible to the WAM, since they are handled by the dereferencing mechanism. We must utilize two new WAM instructions: `switch_on_type_noderef` having the same semantics as `switch_on_type` except that no dereferencing is performed and `ifref_inherit_goals` *arg*,*label* which tests if register *arg* is a reference. If the test is negative, the WAM jumps to *label*. If *arg* is a reference, it is a pointer to an internal delay record (see **??**). If takes the compoud term reference (the term which has been built) and inherits the variable's suspended goal list. The procedure `xcall1/1` is now more complex:

```
xcall1/1:
  switch_on_type_noderef 1,
  nil, ; Variable -> failstruct:              3
  nil, ; DomVariable -> fail
  nil, ; number -> fail
  nil, ; symbol -> fail
  list0, ; list
  nil, ; Struct , call it
  proceed1, ; nil -> fail
  nil ; otherwise -> fail
list0:
  allocate 1
  get_list 1
```

[5]Suspended domain variables can not be unified with structures.

```
  unify_variable_temp 2
  unify_variable_perm 1
  ifref_inherit_goals 1,mcallit
  put_value_perm 1,1
  deallocate
  execute! xcall1/1,1
mcallit:
  mcall 2,1
  put_value_perm 1,1
  deallocate
  execute! xcall1/1,1
proceed1:
  proceed
```

7.8.3 Can internal interrupts be avoided at all ?

Internal interrupts are a necessary, but ugly feature of the described WAM. This is caused by the handling of unifying structures in WAM code. Due to the fact that there is another caveat in building compound term (the mode register is subsequently checked), schemes have been developed to make the building of the structures nonlocal [Mei90]. Building upon these results we could integrate a scheme which would avoid internal interrupts. Then, however, we could not claim that our WAM extensions can be easily implemented in existing compilers.

7.9 Termination

At a final proceed the answer is only sound, if there are no more constraints in store. In **??** a delay record is shown. It has a special tag at the beginning with a value indicating whether the constraint is still alive or dead. The only thing to do at termination is to search for those tags on the heap. If a record is still active, we must tell the user that there are still unfired constraints. However, the algorithm is trivial. In [Car87], Carlsson proposes to trail every suspension and search the trail suspensions and follow the pointers on the trail to see whether it is still a suspension. When it is still a suspension the goals of that suspension are still delayed. Our scheme is simpler, but it costs an extra tag to mark the delay records — this may be a rare resource in low level implementations.

7.10 Conclusion

Upon a casual look, one could argue that much the work done by the WAM in the last section is only performed "deeply inside". So let us look at the structural differences:

- In the previous WAM, waking was done as soon as a suspended variable was touched. In this WAM, we wake constraints when they can be fired. Of course, we do not save the testing code, but the time and space the unecessary waking causes and the time and space[6] by `xcall`.

[6] used in the `XCALL` environment.

- Delay records are only created when they are needed since `lookahead` and `forward` instructions have an overview of all arguments involved and not a local view on one argument.
- The union of suspended goal list is performed by the unify routine (and sometimes by internal interrupts). It's not the combined constraint initialization/testing code responsibility to search on the suspended constraint list, whether it is already delayed. (completely saved)
- in the old WAM complete suspended goal lists were woken, resulting in possible repeated wakeups of constraints.
- A forward constraint is immediately marked dead, when it is appended to the woken constraint list. Thus, subsequent tries to wake up the constraint will fail immediately.

Chapter 8

The new compilation scheme

In [HM92] a compilation scheme is presented consisting of *horizontal (source-to-source)* and *vertical (source-to-instruction)* compilation steps. The first step of the source-to-source transformation is to group clauses together. Afterwards a flattening process is performed to remove the nested compound terms. In FIDO, domain variables in compound terms must be treated in a special manner, because domain variables need more than one word. Therefore, they must be flattened. After partially evaluating the resulting flattened code, the $\doteq$ constraints are to be reordered. In the next step, variables are classified so the $\doteq$-constraints can be easily mapped to WAM instructions. In the chapter, we will give the defintions and procedures which must be altered to cope with the finite domain extensions. You will see that there are surprisingly few changes which confirms that our compilation scheme is very flexible. The "fixes" are due to the memory organisation memtioned above. Please note that our definition of "domain variable" in the compilation process is slightly different to the notion of "domain variable" in the WAM. Another surprise is that we must not cope with "delayed" variables (suspended variable, suspended domain variable) during compilation.

8.1 Horizontal compilation

The aim of horizontal compilation is to reduce the syntactical manifold and obtain clauses with simple, homogenious and declarative constructs which can be transformed into WAM code by a relatively small set of transformation rules. In this chapter we will assume clauses and procedures that are syntactically correct, although we will not define any formal grammar and rely on the reader's knowledge of syntactically correct PROLOG *clauses*.

Definition 24 (variable) *An ordinary variable is a variable that is* not *a domain variable.*

In the following we will see that we are going to replace the syntactic appearance of a domain variable by an $\doteq$ construct with an ordinary variables. Whenever the term "variable" without prefix is presented, we refer to "ordinary veriables".

Definition 25 (domain variable) *A domain variable is a variable which can be instantiated to an element of a finite domain. This domain set is given at the first occurence of the domain variable. The syntactical form of a domain variable is X:{domain}. By definition, if there is a variable X without domain at its first occurence but written with a domain in the following occurrence, this latter domain is removed and attached to the first occurence.*

In the WAM flattening process we will get rid of the domain variables of the form *Var*:domain and produce instead $Var \doteq domain$, where *domain* is the domain of *Var*. *Var* is further considered to be an ordinary variable[1].

8.1.1 Grouping together

First we group together clauses by a mere syntactic transformation, resembling the WAM level where only one entry point to a procedure is present.

Definition 26 (Grouping) *Let the clause*

$$\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \leftarrow \mathsf{B}_1, \ldots, \mathsf{B}_n.$$

be part of a program P. In the first step, we replace the constraints in the head by new variables $X_1, \ldots, X_n$, *writing*

$$\mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n) \leftarrow X_1 \doteq t_1, \ldots, X_n \doteq t_n, \mathsf{B}_1, \ldots, \mathsf{B}_n.$$

We obtain a set of clauses with the same head:

$$\begin{aligned} \mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n) &\leftarrow \mathsf{E}_1 \\ &\vdots \\ \mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n) &\leftarrow \mathsf{E}_k \end{aligned}$$

with $E_i \equiv [X_1 \doteq t_1, \ldots]$ *for* $1 \leq i \leq n$. *Now, we group the clauses together and obtain a definite equivalence:*

$$\mathsf{p}(\mathsf{X}_1, \ldots, \mathsf{X}_n) \leftrightarrow \mathsf{E}_1 \vee \ldots \vee \mathsf{E}_n.$$

The flattening of compound terms is performed as a horizontal precompilation step. In order not to cope explicitly with lists at the horizontal compilation level, we rewrite lists as structures of the form `cons(`*L1*`,`*L2*`)`, where *L1* is the head of the list and *L2* is its tail. In the vertical transformation process, we will handle the *cons strutures* by different compilation rules, since during runtime, the representation of lists is optimized in a way so that a memory cell for the functor is saved.

Definition 27 (WAM flattening) *Let* $p(t_1, \ldots, t_n)$ *be an atom in the definite equivalence. If* t_i $(1 \leq i \leq n)$ *is not an ordinary variable, replace* $p(t_1, \ldots, t_n)$ *by*

$$Var \doteq t_i, p(t_1, \ldots, t_{i-1}, Var, t_{i+1}, \ldots, t_n),$$

where Var is a new ordinary variable not yet occurring in the definite equivalence.

Let $s_1 \doteq s_2$ *be a* $\doteq$*-constraint.*

If s_1*:domain is a domain variable and* s_2 *is an ordinary variable replace by* $s_1 \doteq domain$, $s_1 \doteq s_2$.

If s_1 *is an ordinary variable and* s_2 *is a domain replace by* $s_2 \doteq s_1$.

[1] which is opposed to the runtime behaviour.

If s_1:$domain_1$ and s_2:$domain_2$ are domain variables, replace by $s_1 \doteq domain_1 \cap domain_2$, $s_2 \doteq s_1$.

If s_1:domain and s_2 is neiher an ordinary variable nor a domain variable, replace by $s_1 \doteq domain$, $s_1 \doteq s_2$.

If s_2:domain and s1 is neiher an ordinary variable nor a domain variable, replace by $s_2 \doteq s_1$.

The aim of the rules above is to eliminate domain variables getting $\doteq$-constraints of the form $Var \doteq domain$.

If s_2 is a ordinary variable and s_1 is not a variable, replace $s_1 \doteq s_2$ by $s_2 \doteq s_1$.

Let s_1 be a variable and s_2 be a structure $f(t_1, \ldots, t_n)$ of arity n: If t_i $(1 \leq i \leq n)$ is neither an ordinary variable nor a constant, replace $s_1 \doteq f(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n)$ by

$$Var \doteq t_i, s_1 \doteq f(t_1, \ldots, t_{i-1}, Var, t_{i+1}, \ldots, t_n)$$

where Var is a new variable not occurring in the definite equivalence.

If neither s_1 nor s_2 are ordinary variables, they are either constants or structures. If s_1=$const_1$ and s_2=$const_2$ with $const_1 \neq const_2$, replace the constraint by a `fail`*, if $const_1$=$const_2$, delete the equation. If s_1=$f(t_1, \ldots, t_n)$ and s_2=$f(u_1, \ldots, u_n)$ are structures with same arity and functors, replace the $\doteq$-constraint by n $\doteq$-constraints of the form*

$$t_1 \doteq u_1, t_2 \doteq u_2, \ldots, t_n \doteq u_n;$$

otherwise replace the $\doteq$-constraint by a `fail`*.*

WAM flattening continues until no more rules can be applied.

Note that the flattening *order* is subject to change in the constraint reordering process.

8.1.2 A note on memory organisation

In the above flattening scheme, domain variables are flattened when they appear in a structure. This is important since a domain variable uses in general more than one word: When a structure is to be unified with an unbound variable, the write mode is switched on and the structure has to be built in memory. Assume, the functor has already been built and a domain variable must be unified directly assuming a memory usage of two word. We have no means to allocate the domain variable, because we can neither put the domain variable in front of of the structure nor at the end. We *must* have direct access to the n^{th} argument of the functor since it is mandatory WAM design requirement. The same is valid for list cells. A list is assumed to have two consecutive cells. Accessing the "cdr" cell would result in testing the first cell to be a domain variable and calculating the address from the "car" element of the list node. The same is true for suspended variables, but this is a dynamic feature occuring at runtime and the machine has to cope with it.

The resulting definite equivalences have a restricted syntactical form:

- The head is an atom whose arguments are variables.
- The arguments of the literals in the body are variables.
- The $\doteq$-constraints have one of the forms:

  - $Var_1 \doteq Var_2$
  - $Var \doteq constant$
  - $Var \doteq domain$
  - $Var \doteq f(t_1,\ldots,t_n)$, where t_i are either constants or variables $(1 \leq i \leq n)$.
- The or-operator $\vee$ separates the or-branches.
- A special atom `fail` indicates compile time failure.

Herein, the arguments occurring in the procedure head are temporary variables. The finite tree and the finite domain are handled by the code generation for $\doteq$ for both head and tail. Please note that the only addition compared to the WAM Compilation Scheme [HM92] is the addtion of $Var \doteq domain$.

Variable could be allocated without exception on the stack or on the heap. This would result in a very bad runtime bahaviour (and increased memory consumption). From conventional language compilation, we know that speed is due to the intensive use of registers. The WAM model is a register oriented machine with a set of X-registers used for argument passing and temporary variable assignment. In [Deb86] variables are classified *permanent* (Y-variables) or *temporary* (X-variables). Permanent variables (Y-variables) have to be stored in the environment while the temporary ones (X-variables) can be held in a register.

A X-register can *not* contain any free logical variables since a register can not be accessed by an address so it is impossible to have a reference to the register itself — which is the representation for a variable. However, we speak of X-variables since they may contain pointers to variables *in memory*. Domain variables, suspended variables and suspended domain variables belong to a class of variables, which can only be found in the heap. They can however be X-variables (and Y-variables): The feature of being a temporary X-variable says something about its lifetime in the procedure and *not* where it is located.

What we already know from the suspended-, domain-, and suspended domain variables types that the are always located on the heap. Thus the can not be unsafe variables. They can be temporary or permanent variables. When they occur for the first time, they have a domain associated.

The definition of destroying and preserving literals, chunk, pseudo-chunk, variable types (temporary, permanent) do not alter and can be looked up in [HM92].

$$\frac{Var1_{\text{temp,first}} \doteq domain}{(\texttt{put\_dvariable\_temp}\ \ Var1^{\rightarrow \text{X-reg}}\ \ domain)} \tag{8.1}$$

$$\frac{Var1_{\text{temp,nonfirst}} \doteq domain}{(\texttt{get\_dvalue\_temp}\ \ Var1^{\rightarrow \text{X-reg}}\ \ domain)} \tag{8.2}$$

$$\frac{Var1_{\text{perm,first}} \doteq domain}{(\texttt{put\_dvariable\_perm}\ \ Var1^{\rightarrow \text{Y-reg}}\ \ domain)} \tag{8.3}$$

You should be aware that the instructions in the compilation rules given in [HM92] have an extended semantics, e.g. a `get_constant` operation must now deal with domain variables.

8.2 Specifying the extended control primitives

In this section, we describe how to compile control of the constraints. We distinguish between *constraint initialization code* and *constraint body*. The constraint initialization code is responsible for jumping to the constraint if it can fire, building the runtime structures for the bind-mechansim and for delaying the variables.

Concerning the constraint body (the firing code), we have two sorts of constraints in FIDO: Intensional constraints are constraints given by an algebraic expression over the operators $(+, -, /, *,)$ and predicates $(>, <, \leq, ge, \neq)$ [2]. These are to be compiled into a LISP expressions for optimal efficiency. Extensional predicates are a set of clauses and when applying local consistency the new consistent domains are determined by calculating the set of solutions of the predicate taking the union of the single arguments.

8.2.1 Lookahead and forward definitions

Here, we are defining how to compile the constraint initialization. The presentation of lookahead and forward definition to the system has been presented in definition 20 and 14. Taking a constraint definition for a procedure *p/arity*, we apply the initialization at the label *p/arity* delaying the constraint body under the name *cp/arity*. The compilation of forward and lookahead is given in table 8.1. The nonterminal *arity* is the number of arguments. In the compilation rules, you find atoms with a ":". These are labels and mark special entry points in the code and are *no* instructions. The non-terminal *speclist* determines which arguments must be ground or domain variables.

$$\frac{\texttt{forward}\ \textit{procedure}\ (\ \textit{speclist}\ )}{(\texttt{forward}\ \textit{arity}\ \textit{speclist}\ \texttt{\$c\$}\textit{procedure})} \tag{8.4}$$

$$\frac{\texttt{lookahead}\ \textit{procedure}\ (\ \textit{speclist}\ )}{(\texttt{lookahead}\ \textit{arity}\ \textit{speclist}\ \texttt{\$c\$}\textit{procedure})} \tag{8.5}$$

Table 8.1: Compilation of forward and lookahead declarations

The following is a compilation of an initialization code of $\neq$ (`ne`):

```
forward ne(d,d).
```

is compiled to:

```
ne:
        (forward 2 (d d) $c$ne)

$c$ne:
        ....
```

[2] In our implementation we can take every legal LISP expression. For the sake of portability to any other language we should restrict ourselves to these "builtins".

8.2.2 Intensional Constraints

In many applications the constraints of interest are expressions over $(+, -, /, *, >, <, \leq, ge, \neq)$ and domain variables and user variables. The task to perform is to synthesize a LISP expression from the expression in the constraint predicate p and generate the form seen in fig. 8.1. The argument *argumentplaces* specifies these argument registers from where the parameters are passed to the consistency routines.

$$\frac{p(\ldots) \texttt{ :- } \textit{expression expressable in LISP}}{\begin{array}{c}\texttt{\$c\$}p\textit{/arity:}\\ (n\texttt{-consistent } \textit{argumentplaces } \texttt{(lambda (}\textit{Variables}\texttt{) } \textit{expression expressable in LISP}\texttt{))}\\ \texttt{(proceed)}\end{array}} \tag{8.6}$$

Figure 8.1: Compilation of an intensional constraint

So the above not-equal predicate is compiled to:

```
ne:
        (forward 2 (d d) $c$ne)

$c$ne:
        (2-consistent (1 2) (lambda (x y) (<> x y)))
        (proceed)
```

Constraints may be disjunctive. Take for example the `neighbour` predicate in the houses example (see appendix C). They are compiled obeying the rules for normal control:

```
forward neighbour(d,d).
neighbour(X,Y) :- X =:= Y - 1.
neighbour(X,Y) :- X =:= Y + 1.
```

is compiled to

```
neighbour/2:
        (forward 2 (d d) $c$neighbour/2)
$c$neighbour/2:
        (try_me_else L1 2)
        (2-consistent (1 2) (lambda(x y) (u v) (= u (1- v))))
        (proceed)
L1:     (trust_me_else_fail 2)
        (2-consistent (1 2) (lambda(x y) (u v) (= u (1+ v))))
        (proceed)
```

8.2.3 Extensional Constraints

Extensional constraints are written in FIDO. While the termination using intensional constraints is no problem, the programmer may specify a procedure offending def. 10. The burden

$$\frac{p(\ldots)\ \texttt{:-}\ \ldots}{\begin{array}{c}\texttt{\$c\$}p/arity\texttt{:}\\ (n\texttt{p-consistent}\ argumentplaces\ \texttt{\$c\$c\$}p/arity)\\ \texttt{(proceed)}\end{array}} \qquad (8.7)$$

$$\begin{array}{c}\texttt{\$c\$c\$}p/arity\texttt{:}\\ (tryL00)\\ (trustL10)\\ \texttt{L0}:\\ (call!\$c\$c\$c\$p/arityarity0)\\ (constraint-succeeded)\\ \texttt{L1}:\\ (constraint-failed)\end{array}$$

Figure 8.2: Compilation of an extensional constraint

is put on the programmer ! The compilation is more complex, since we must collect the solutions by calling the FIDO machine inside FIDO. The initialization code is the same as for intensional constraints. The code of the body is compiled differently and can be seen in fig. 8.2.

Compiling the predicate `p/2` results in:

```
forward p(d,d).
p(1,2).
p(2,4).
p(2,5).
p(5,7).

p/2:
        (forward 2 (d d) $c$p/2)

$c$p/2:
        (2p-consistent (1 2) $c$c$p/2)
        (proceed)

$c$c$c$p/2:
        (try L0 0)
        (trust L1 0)
L0:
        (call $c$c$c$c$p/2 2 0)
        (constraint-succeeded)
L1:
        (constraint-failed)

$c$c$c$c$p/2:
        (try_me_else LL1 2)
        (get_constant 1 1)
        (get_constant 2 2)
        (proceed)
```

```
LL1:
        (retry_me_else LL2 2)
        (get_constant 2 1)
        (get_constant 4 2)
        (proceed)
LL2
        (retry_me_else LL3 2)
        (get_constant 2 1)
        (get_constant 5 2)
        (proceed)
LL3
        (trust_me_else_fail 2)
        (get_constant 5 1)
        (get_constant 7 2)
        (proceed)
```

8.3 Conclusion

We have extended the WAM compilation scheme by finite domain compilation. The minor changes in the flattening process and the additional instructions show that the scheme verified to be very flexible. A compiler modeled according to the scheme can be easily extended. Compiling control is more difficult: Intensional constraints have a simple form and can be compiled without much difficulty. Extensional constraints must be compiled in a form which seems rather ugly: The reason is that the WAM calls a instruction implemented in LISP which must call the WAM again and after a fail or succeed caused by the constraint body, we must avoid to proceed or fail in the program which called the consistency procedure. So we actually protect the constraint program from continueing the calling code and can jump back to our LISP programm collecting the data for the new domains.

Chapter 9

Builtins — Consistency algorithms & First fail

The body of a constraint procedure must be compiled in a special way. We have two primitives for the compilation of the constraint body, where we can use intensional and extensional constraints. Intensional constraint are compiled into LISP code, while extensional constraints are FIDO predicates where we must apply a surrounding local consistency procedure. Fortunately, we must not use different consistency algorithms for lookahead and forward constraints — it can be done by a single algorithm and thus by a single instruction.

The first fail principle is a choice method. Given a set of variables, the variable is instantiated next, which has the smallest domain length. If two variables have the same domain length the variable is chose, which has the greatest number of active constraints.

9.1 Consistency algorithms

Let us look at the definition of the Forward Checking Inference Rule (FCIR) in 13 and Lookahead Inference Rule 19. In the LAIR, the task is to find for each domain variable x_j with old domain d_j a new domain $e_j = \{v_j \in d_j | \exists v_1 \in d_1, \ldots, \exists v_{j-1} \in d_{j-1}, \exists v_{j+1} \in d_{j+1}, \ldots, \exists v_n \in d_n$ such that $A_k\theta$ is a logical consequence of P with $\theta = \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}\} \neq \emptyset$. More systematically, we instantiate variables $i_1, \ldots, i_n$ with the elements of their respective domains $d_1, \ldots, d_n$ and calculate the union for each argument e_j with a consistent assignment i_j by $e_j = e_j \cup i_j$.

The algorithm to calculate the new domain is:
$e_1 = \emptyset, e_2 = \emptyset, \ldots, e_n = \emptyset$
foreach i_1 in d_1 do
...
foreach i_n in d_n do
if $Ak\{x_1 \leftarrow i_1, \ldots, x_n \leftarrow i_n\}\}$ succeeds
then $e_1 = e_1 \cup i_1, e_2 = e_2 \cup i_2, \ldots, e_n = e_n \cup i_n$

The runtime estimation for the algorithm is $|d_1||d_2| \ldots |d_n|$.

Instead of using only the domain variables as running arguments, use all arguments of the constraint. In a constraint where arguments are defined ground, the running arguments are to be instantiated to a singleton value in the foreach loop. Since the length of those domains is

"1", the runtime estimation is the same as above. This little trick eases the compilation of the constraint body.

The same is valid for forward constraints, were we have only one domain variable, the other arguments are regarded as singleton values. Again, we do no not know at compile time which arguments are going to be instantiated to ground and which argument to be the domain variable.

9.1.1 LISP *n*-consistency algorithms

The *n*-`consistency` algorithm is utilized when we can transform a FIDO constraint into a LISP expression. We call these constraints intensional constraints. See chapter 8 for the exact compilation application. The tasks to be performed before a consistency procedure can be applied are the extraction of the domain variables and the conversion of constants to singleton domains. The consistency procedure is applied yielding new domains. Only if the domains are smaller than the old domains [1] the new domain shall be bound to the domain variables.

9.1.2 FIDO *n*-pconsistency

Calculating extensional constraints is a severe problem. We have to fake the machine by allocating a new choicepoint which controls the constraint handling. The situation can be seen in fig. 9.1. The problem arises from the fact that when a constraint in FIDO finally succeeds or fails it jumps to the code located above the upper fat line. We have to take care of this situation since we want to collect the new domains with the algorithm outlined above. The choicepoint is responsible that after a final succeed or fail control is handed over to LISP collecting and controlling the extensional constraint.

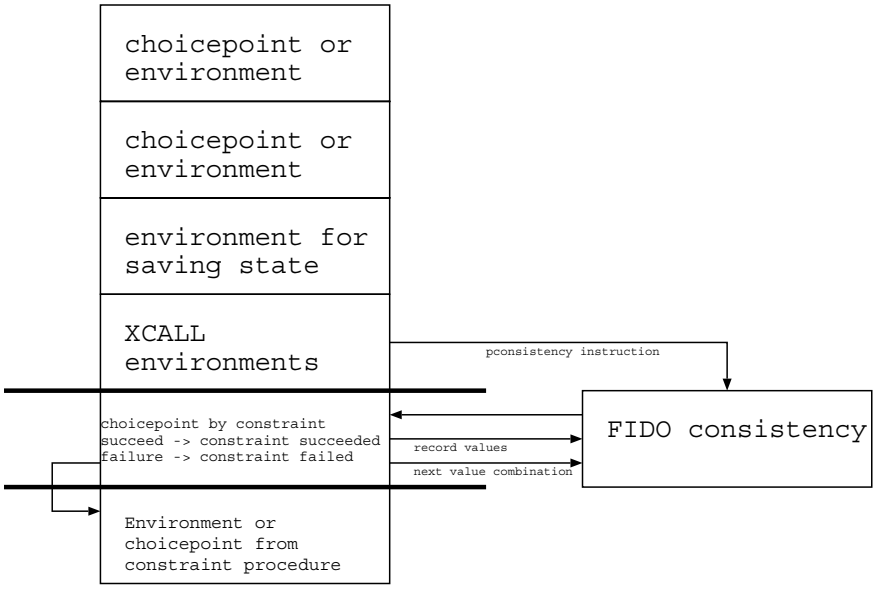


Figure 9.1: Stack structure during FIDO extensional constraints

[1] A binding of an equal domain forces a lookahead to be fired, if all the places specified "g" are ground, possibly comming into an infinite loop.

9.2 First Fail Principle

9.2.1 Conventional labeling

The labeling strategy without any special heuristics takes the first domain variable in a list, gets its domain and tries the values for the variable successively. A FIDO implementation can be written in the following form. `putdomainlist` is a WAM builtin taking the domain variable giving access to the domain represented as a list. While labeling is responsible for working on all variables, the `indomain` predicate extracts the domain list and handles it over to `indomain1` which is responsible for instantiating all domain list elements. The underlying `(putdomainlist` A_i A_j`)` WAM instruction expects a domain variable in A_i, builds the domain list and puts the pointer in A_j.

```
labeling([]).
labeling([X|Y]):- indomain(X), labeling(Y).
indomain([]).
indomain(Dv) :- constant(Dv),!.
indomain(Dv) :- domvar(Dv), L = putdomainlist(Dv), indomain1(Dv,L).

indomain1(Dv,[]) :- !, fail.
indomain1(Dv,[H|T]) :- Dv = H.
indomain1(Dv,[H|T]) :- indomain1(Dv,T).
```

9.3 Implementation of the first-fail principle

The first fail principle is based on the `(deleteffc)` WAM instruction. The builtin instruction `deleteffc` takes a list a of domain variables and constants in the second argument searches for the domain variables with is most constraint by taking the domain variable with the smallest domain and if domain variables have the same cardinality of their domain sets, the one is taken with the gratest number of active constraints. The list without this choosen variable (and without the variables which have already become constants) is constructed and handled over in the third argument. The choosen variable can be found in the first argument. The effect of the predicate is that the choosen variables are taken in a dynamical order.

The altered labeling predicate reads:

```
labeling([]).
labeling(Ls):-
deleteffc(Var,Ls,RestLs),
indomain(Var),
labeling(RestLs).
```

9.4 Conclusion

In this section we presented the necessary WAM level builtins for FIDO. All other builtins can be constructed in FIDO itself. (e.g. `all_different`) The principles of constraint body

propagation have been presented allowing in a general and efficient manner to ensure local consistency. Up to now, all methods and concepts have been shown to compile a FIDO constraint program.

Chapter 10

Analysis

In this chapter, we will develop a simple but useful technique to analyse the behaviour of FIDO programs during runtime. The problem is to estimate the advantage of constraint procedures without actually debugging the whole session and looking at procedures involved — although by using this method, we can extract the most information.

The other possibility is to play around with the `forward` and `lookahead` declarations and try to analyse the runtime figures. However a lookahead constraint may need a lot of time for small examples and may be outperformed by a forward definition of the same constraint. Using more complex input, the exponential growth of time in some parameter may make the lookahead strategy more suited.

Another question arises whether to use disjunctive constraints. They may be either lookahead or forward defined or in a way that "constraints as choices" are made. Pascal Van Hentenryck proposes to use constraint as choice as the best way ([Hen89]).

Sometimes an optimal strategy is known to instantiate variables. If we do not know the best strategy, the first fail predicate `deleteffc` can be applied choosing variables to be instantiated dynamically. It would be fine to estimate the gain of this builtin.

We will now look at our analysis model and finally test it with some selected examples.

10.1 Analysis Model

The overall idea of using constraints is to prune the search space, get less failures and obtain more directed (functional) computation. In a WAM based environment we can easily get some more information by counting the number of fails ($\#f$), expressing the degree of non-determinism, and the number of instructions ($\#i$) expressing the overall work. Directed computation may be estimated by the ratio $\#i/\#f$. However, the numbers are very course grain.

We present a simple two-dimensional plot with the following concept: Start at some base line moving upward when some "functional" computation is done. This is true for call, execute, proceed and propagation (consistency) instructions. When a failure occurs, we fall back onto the base line and go one step to the right side. Furthermore, we color the different steps so that we can estimate what has been done. The color table can be seen in table 10.1. We argue, that the area of the plot is the total work. The length in horizontal direction is proportional to the nondeterminism and the height is proportional to the deterministic computation. Due to the colors, we can see when the constraints are applied and estimate their pruning power.

Action	Color
execute or call	black
proceed	black
forward	green
lookahead	red
exception	blue
has-succeeded	yellow beam

Table 10.1: The colortable assigning WAM actions to colors

The WAM records the internal actions by writing PostScript statistics into a file. By a PostScript convention point (0,0) is located at the lower left corner. For the sake of simplicity, we start our plot at location (0,0) moving upwards when we reach the right edge of the page. Thus a plot containing serveral lines must be read from the bottom to the top.

10.1.1 Five Houses Problem

The Five Houses Problem reads: Five people with five nationalities live in the first five houses of a street. Each of them has a *different* profession, animal and has a favourite drink. Besides, the five houses are painted *differently*.

- It is known that the englishman lives in a red house,
- the spaniard owns a dog,
- the japanese is a painter,
- the italian drinks tea,
- the owner of the green house drinks coffee,
- the sculptor breeds snails,
- the diplomat lives in the yellow house,
- the owner of the middle house drinks milk,
- the violinist drinks fruit juice,
- the norwegian lives in the first house on the left
- and the green house is on the right of the white one.
- The problem is made more complicated by the following disjunctive constraints:
- the norwegian lives next to the blue house,
- the fox is in the house next to the doctor's
- and the horse is in the house next to that of the diplomat.

The question is, now, who owns a zebra and who drinks water [1].

The FIDO source and WAM code can be found in appendix C. In the following, minor modifications (e.g. a redeclaration of a `forward` into a `lookahead`) are explicitly given in the text.

No deleteffc has been used and the predicates neighbour and eq+1 have been forward declared. The number of instruction till the first solution is 3313 with 14 fails, the final fail was obtained after 7092 steps and 52 fails. The following results will be compared to these runtime figures.

This plot originates from the save WAM code as above, but the labeling procedure has been replaced by the first fail procedure. The solution was found after 2747 (83%) steps and 6 fails, the final fail was reached after 3087 (44%) instructions and 14 fails.

Now, we changed the declaration of `eq+1` to a lookahead constraint. We used 2409 (73%) WAM instructions and 1 (!) fail to reach the solution and 3539 (50%) steps and 14 fails to come to an end. Apparently, the deleteffc has a slightly better pruning power (less steps altogether) than the lookahead (less steps to first solution), but the lookahead is more "directed". Please notice in the following two plots the tiny red dots indicating lookahead constraints. Lookahead is not applied foolish often, which has been a criticism of lookahead, but amazingly, one has to search the red color in the plots!

[1] There exists a unary solution for that problem: the spaniard owns the zebra, and he also drinks water.

If we again leave the code untouched and apply the first fail principle, we come to a surprise: We get 2834 (86%) steps and 5 fails for the first solution and 3174 (48%) steps and 13 fails to the "no more solutions prompt". So we have an example that deleteffc must not necessarily get better runtime solutions.

In the following example, we have used lookahead for `eq+1` (!) and constraints as choices for `neighbour`, yielding 3326 instructions(86%) (15 fails) for the reaching the solution and 7044 (99%) steps (52 fails) to the end. The number of fails are remarkable. The early creation of choicepoints does not lead to a quick fail. This example can be considered as a counterexample to Van Hentenrycks argument that constraints as choices are superior (p. 169 in [Hen89]).

Constraints as choice with deleteffc are neither superior. We need 2754 steps with 7 fails to the solution and 3093 steps and 14 fails to the end.

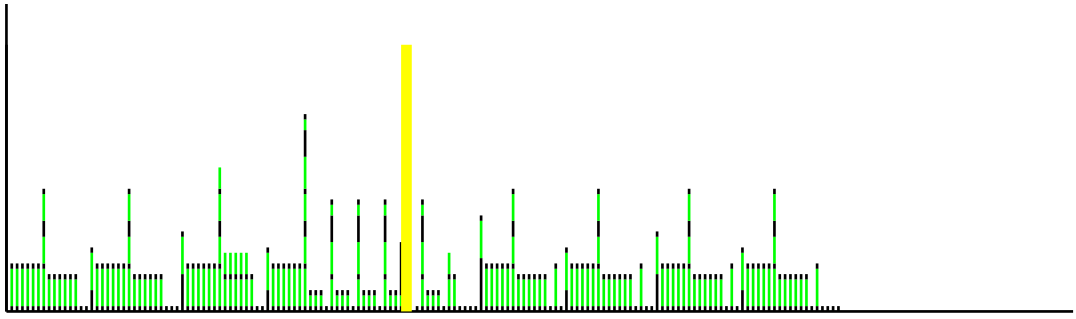
10.1.2 The SEND+MORE=MONEY example

A well-known problem is the SEND+MORE=MONEY puzzle. The letters shall be replaced by digits so that the addition

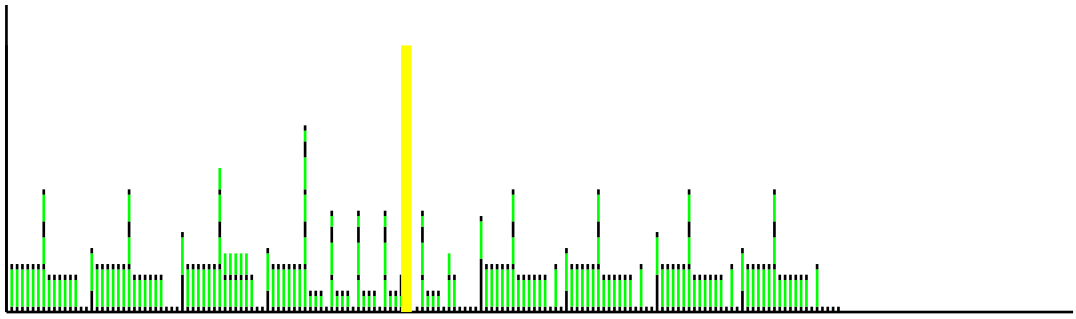
```
    S E N D
+   M O R E
  ----------
=  M O N E Y
```

gives a correct result. The problem can be formulated by defining an adder for each column, returning a carry and a value. The source and WAM code can be found in appendix D.

The first plot resulted from the original source in appendix D with no first fail heuristics. Just for convenience, we repeat the declarations we further want to change, namely the `forward(U + V + W =:= X + 10 * Y)` and the `forward(U + V =:= W + 10 * X)` constraints. The run-time figures are 8565 steps with 74 fails till the first solution and 15707 steps with 155 fails to the end.



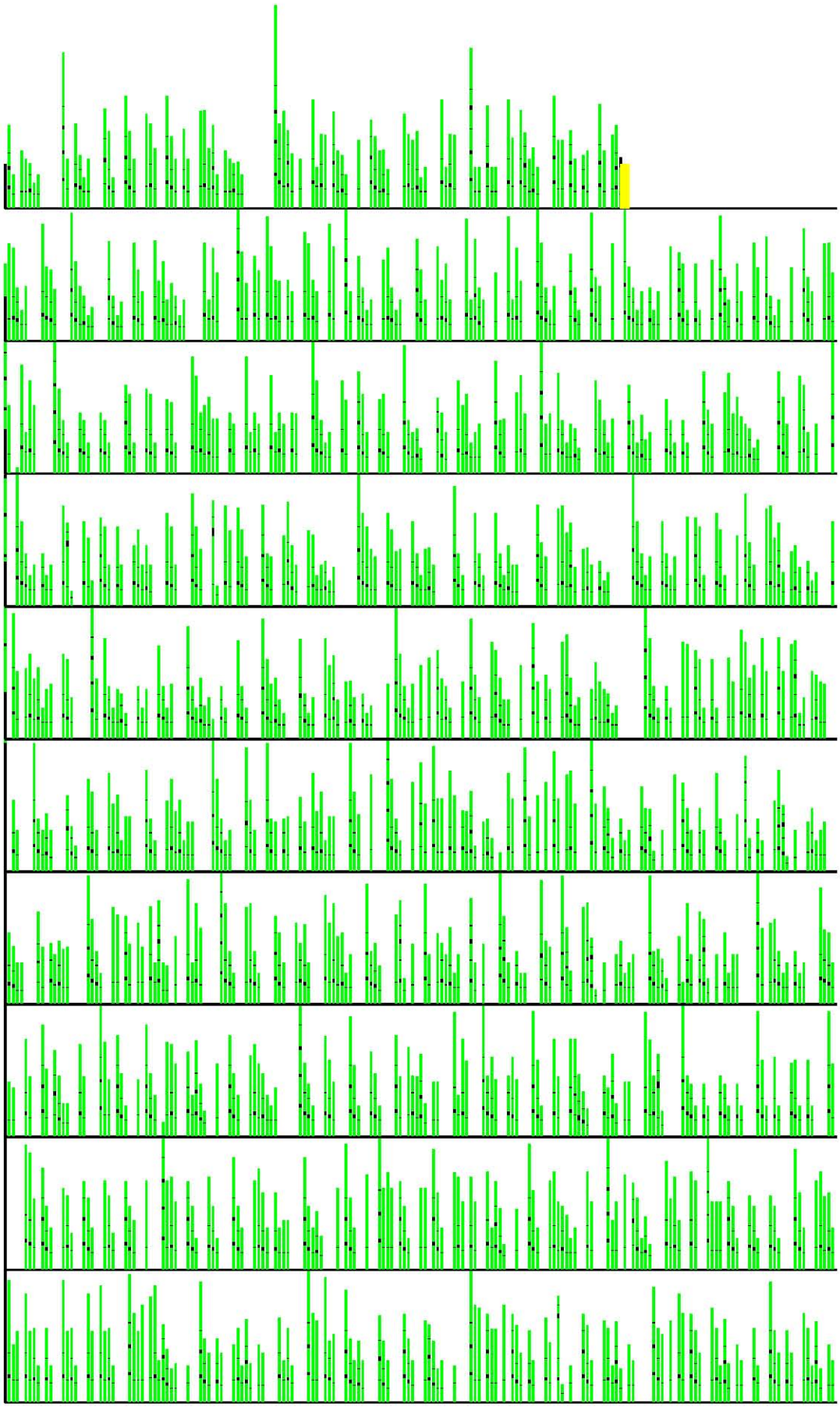
Now we switch over to first fail heuristics. The following plot with 8495 steps with 74 fails to the solutions and 16648 steps with 155 fails show hardly a difference to the plot above. We have to take a ruler to notice the differences. All in all, the number of steps increased.



The next plot demonstrates the pruning power of the lookahead constraints (no deleteffc). The declarations for the forward constraints above have changed to `lookahead(U + V + W =:= X + 10 * Y)` and `lookahead(U + V =:= W + 10 * X)`. The number of instructions was 2062 with 3 failures to the solution and 2460 instructions with 9 fails altogether.

10.1.3 The queens example

To demonstrate that the plot is also useful for larger examples, the queens programs has been started with 20 queens without first fail heuristics yielding 787436 steps with 1939 fails to reach the first solution. The Y direction has been scaled to approx. 1/5. In the plot, there seem to be some gaps. This is caused by the application of scaling. A single action is just 0.2pt which is not seen when few steps are made till the next failure occurs.



The 20 queens example with deleteffc runs in a fraction of the time. To obtain the first solution 17933 steps with 11 fails were necessary.

10.2 Conclusion and possible extensions

A graphical model has been proposed to roughly visualize the runtime behaviour of a FIDO program. The effects of using several declarations for constraints or loading the first fail principle into the system can be clearly seen. An increase in functional computations results in higher lines of the plot, while in increase in nondeterminism yields longer plots in horizontal direction. The goal should be to minimize the "area" of the plot which is proportial to the number of instructions.

The model is *not at all* informative about which effects of already woken constraints still remain present (meaning their effects are not backtracked). This could be achieved by incorporating depth information and putting the points at the height corresponding to the depth. Unfortunately, at the beginning of processing a deterministic computation is usually performed in order to build the constraint net. This may yield tremendous depth so it will not fit on the page.

Using today's color laser technology, many colors can be used. This make the assignment of colors for each constraint in a program possible showing *which* constraints have been fired.

Lookahead constraints have different complexity according to the number of domain variables and their domain legth. Thus a three dimensional plot would be suitable for the exploration of applications with many lookahead constraints.

It would we also very informative to obtain information by some color schme, how useful a constraint has been. This could be measured in the number of domain elements removed.

Chapter 11

Future developments and Extensions

In this chapter, we will review the vertical approach of compiling FIDO programs to the WAM. We will shortly look at the WAM extensions and the basic concepts of compiling constraints into the WAM. Finally, we will mention some extensions resulting from the approach and we will also critically note the caveats inherent in the design.

11.1 What did we achieve ?

In the paper, we presented an extension of the WAM for finite domain constraints. We gave two different flavors of extensions: The first extended the `freeze` concept in chapter 5, the other is a complete redesign more amenable to constraint processing. While the first approach should be considered as a "hack", I suppose the latter approach can be classified as state of the art. This claim is so undistinct since we can not compare our work with the CHIP internals because they are trade secrets.

The main ideas of the in-depth integration of the constraint handling is

- domain unification *with* suspended goal list handling
- constraint initialization is atomic
- constraint testing is done deeply by bind

11.2 Extensions

Sound negation can be implemented, if *no* constraints and *no* delays are allowed: If the procedure invoked by the `not`-predicate delays other procedures, we are in trouble. The only point were we look for floundered goals is when we come to a final proceed. The current behaviour of the machine is the worst case. However, we have a lack of good concepts to deal with the problem. This should be further investigated.

We have seen that forward constraints are cheap to apply, wheras the pruning power of lookahead constraints is remarkable. We can also use disjunctive constraints (as seen in the houses problem in appendix C), which is an important feature conerning allocation and scheduling

problems. When we perform constraint processing, we apply the constraints in the order they appear. We might think of a variant: Handling the forward constraints first, then the lookahead constraints are handled and finally the disjunctive constraints are invoked. In any case the disjunctive constraints should be postponed: They generate a choicepoint, which we want to avoid. So we hope the constraints activated earlier make the disjunctive choice deterministic. In the current version of the WAM, we have a normal processing mode and an exception mode where the constraints are called. I propose a variant were three "exception mode levels" for forward, lookahead and disjunctive constraints are present. This is similar to an interrupt priority scheme of ordinary processors. E.g. an interrupt from a harddisk is of higher priority than the interrupt from a floppy disk which has a higher priority than the terminal input interrupt. There are obvisous examples that postponing disjunctive constraints saves some work. The "differences" by forward and lookahead constraints might be heavily application dependent.

Going hand in hand with the above remarks is the selection of the choice variable in `deleteffc`. When two domains are of equal length, the variable is choosen, which has more constraints. We could also assume that a lookahead constraint is not as good as a forward constraint and develop other heuristics.

The integration of hierarchially structured domains in technical problems is of great use as shown in the CONTAX constraint system ([Mey92]). A possible extension of the FIDO WAM is to integrate these structured domains. Another question is whether we can formalize domains as types over a boolean lattice. Since the hierarchies form a lattice, it may be put into a single framework in the same way domains in chapter 3 are seen as unary predicates. (Types are unary predicates as well). Coding these lattices could be done in a way described in [AKBLN90].

Another completely unsolved problem is the usage of soft constraints in a logic programming framework. Soft constraint should get some sort of importance and they sould be removed from the system if no solution is possible with them.

In chapter 1 we mentioned that the COLAB expert system shell is made of loosely coupled inference systems. The integration of three inference systems (constraints & taxonomical reasoning & backward reasoning) seems to me interesting topic.

11.3 Acknowledgements

I want to thank Manfred Meyer and Prof. Richter for their help, advice, and support.

Joerg Mueller did exceptional work when developing FIDO-II — my work has been greatly simplified by verifying the output results of FIDO-III against FIDO-II and by looking at the runtime behaviour of his systems. Furthermore, my implementation benefitted from his careful analysis of the weak points in FIDO-II.

Appendix A

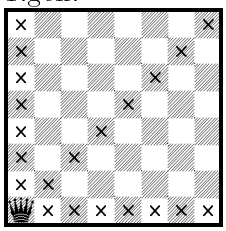
How to obtain the source code

You can get the code by anonymous ftp to `ftp.uni-kl.de` in the directory `/pub/languages/fido`. You will also find the metainterpreter and the version using horizontal compilation.

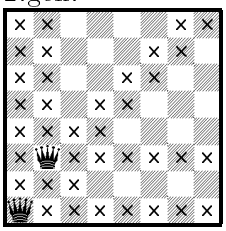
Appendix B

Eight queens (first solution)

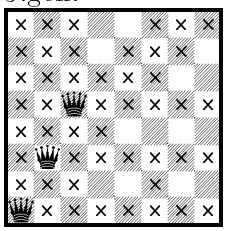
1:gen.



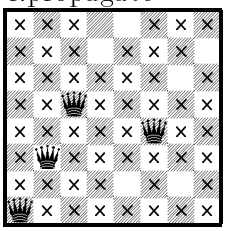
2:gen.



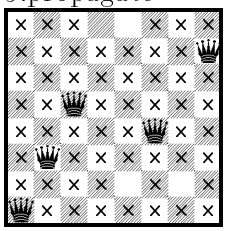
3:gen.



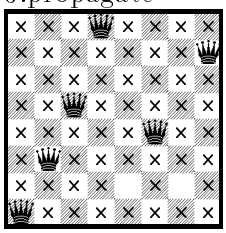
4:propagate



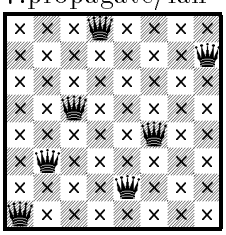
5:propagate



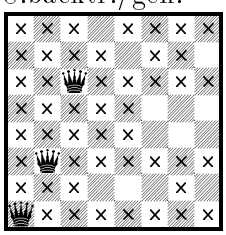
6:propagate



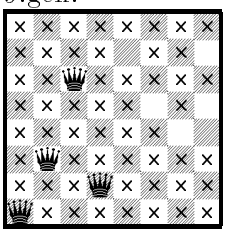
7:propagate/fail



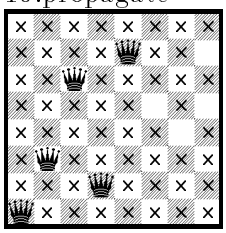
8:backtr./gen.



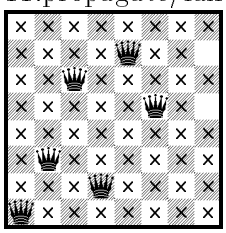
9:gen.



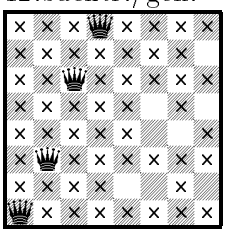
10:propagate



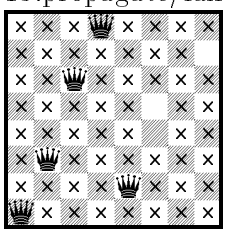
11:propagate/fail



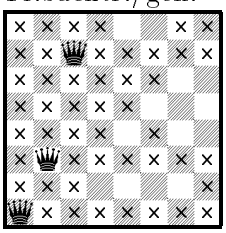
12:backtr./gen.



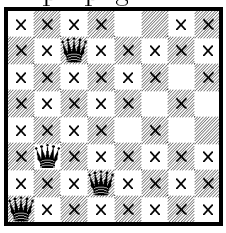
13:propagate/fail



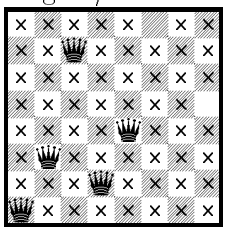
14:backtr./gen.



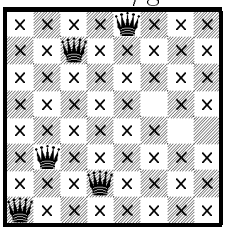
15:propagate



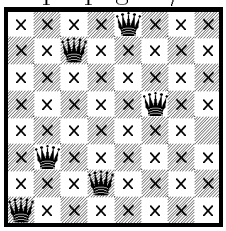
16:gen./fail



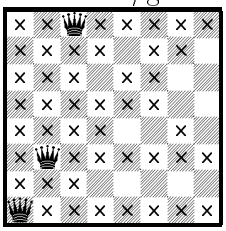
17:backtr./gen.



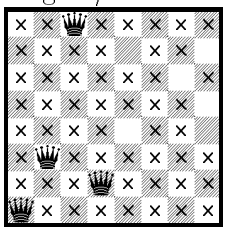
18:propagate/fail



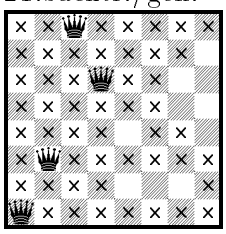
19:backtr./gen.



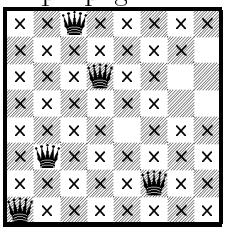
20:gen./fail



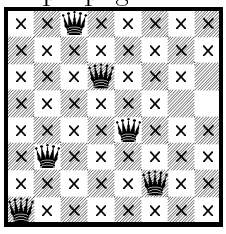
21:backtr./gen.



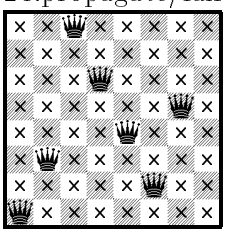
22:propagate



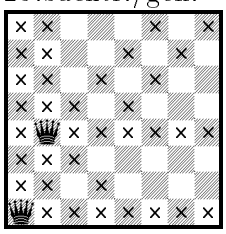
23:propagate



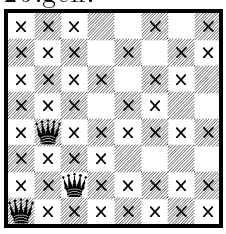
24:propagate/fail



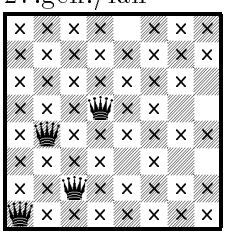
25:backtr./gen.



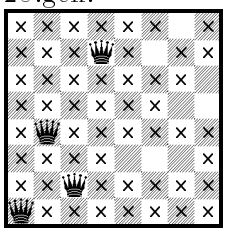
26:gen.



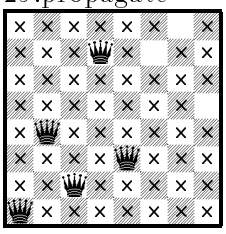
27:gen./fail



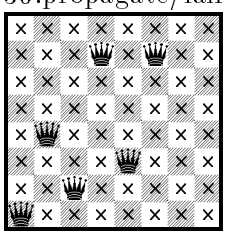
28:gen.



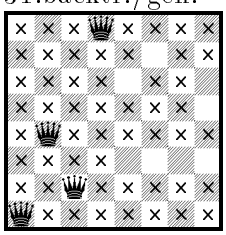
29:propagate



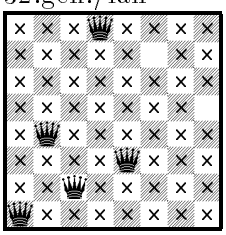
30:propagate/fail



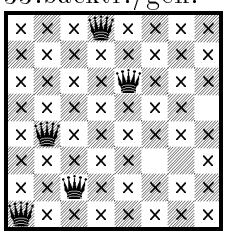
31:backtr./gen.



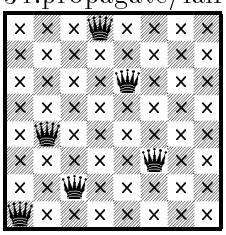
32:gen./fail



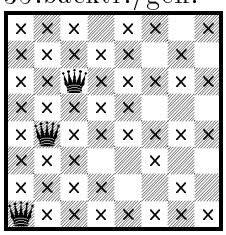
33:backtr./gen.



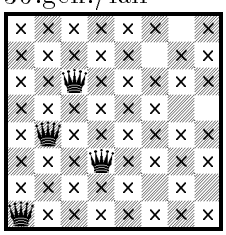
34:propagate/fail



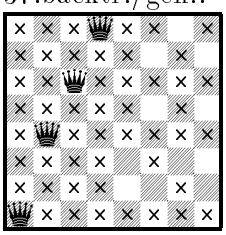
35:backtr./gen.



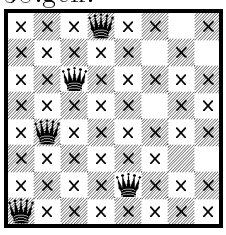
36:gen./fail



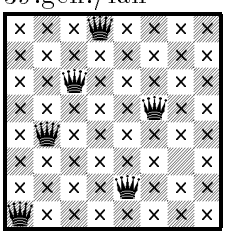
37:backtr./gen.:



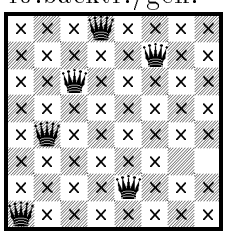
38:gen.



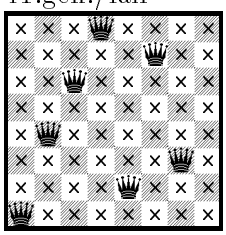
39:gen./fail



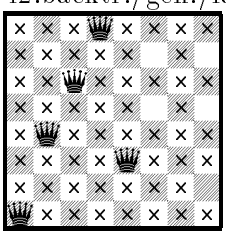
40:backtr./gen.



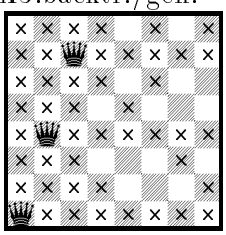
41:gen./fail



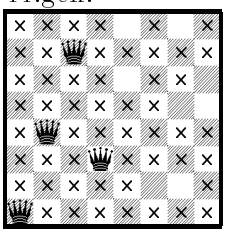
42:backtr./gen./fail



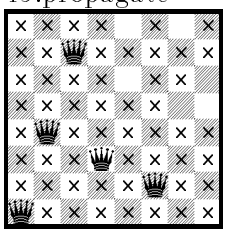
43:backtr./gen.



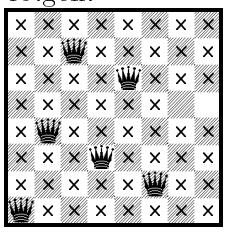
44:gen.



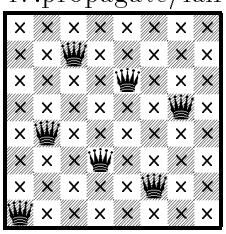
45:propagate



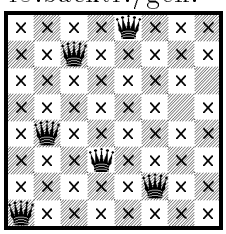
46:gen.



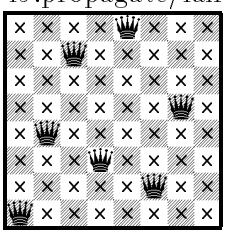
47:propagate/fail



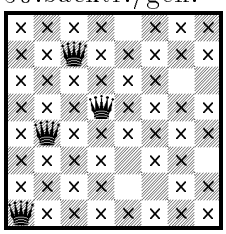
48:backtr./gen.



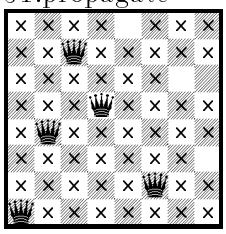
49:propagate/fail



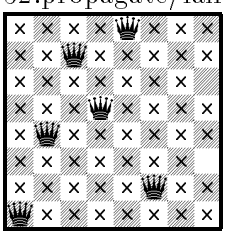
50:backtr./gen.



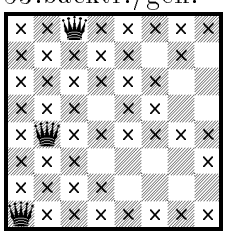
51:propagate



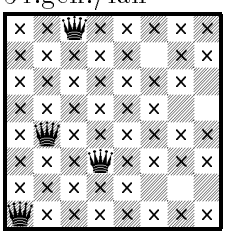
52:propagate/fail



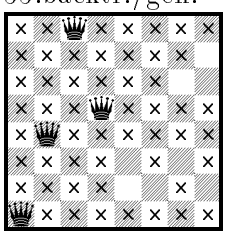
53:backtr./gen.



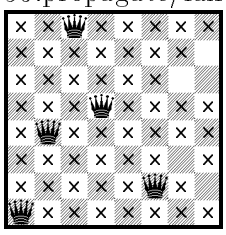
54:gen./fail



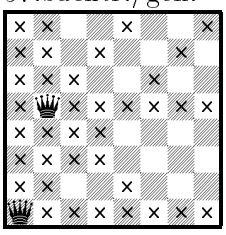
55:backtr./gen.



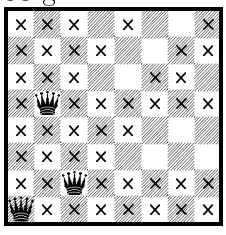
56:propagate/fail



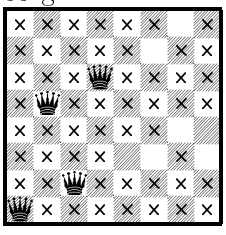
57:backtr./gen.



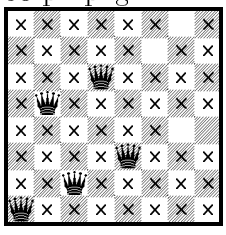
58:gen.



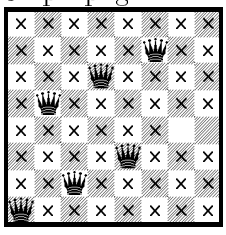
59:gen.



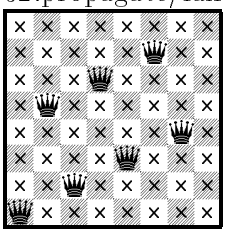
60:propagate



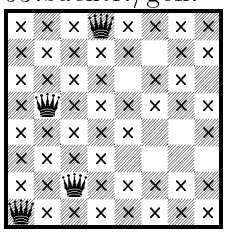
61:propagate



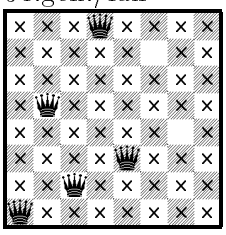
62:propagate/fail



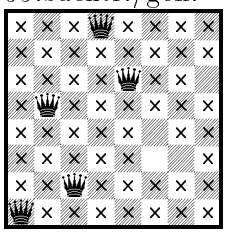
63:backtr./gen.



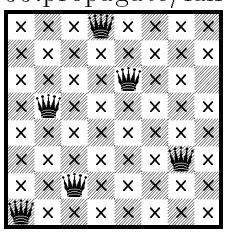
64:gen./fail



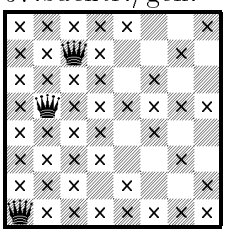
65:backtr./gen.



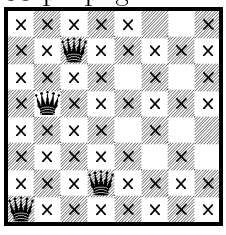
66:propagate/fail



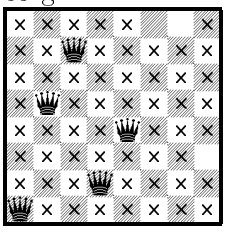
67:backtr./gen.



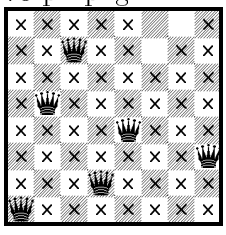
68:propagate



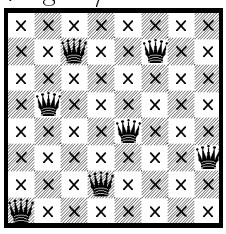
69:gen.



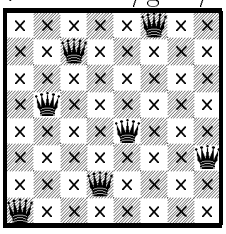
70:propagate



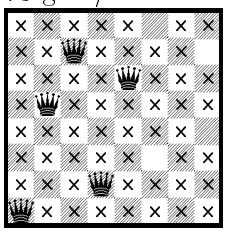
71:gen./fail



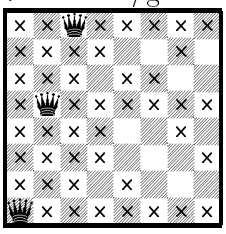
72:backtr./gen./fail



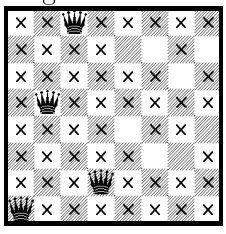
73:gen./fail



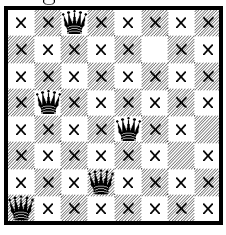
74:backtr./gen.



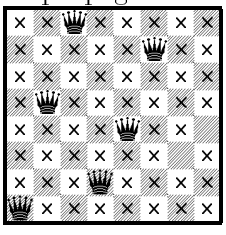
75:gen.



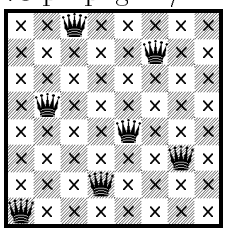
76:gen.



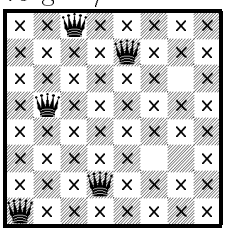
77:propagate



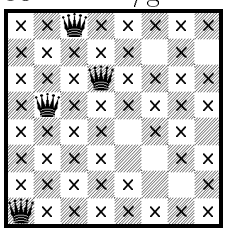
78:propagate/fail



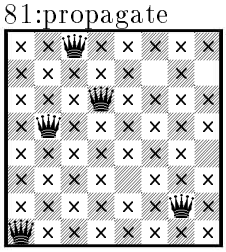
79:gen./fail



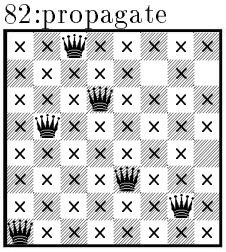
80:backtr./gen.



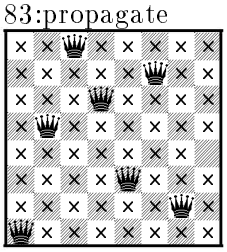
81:propagate



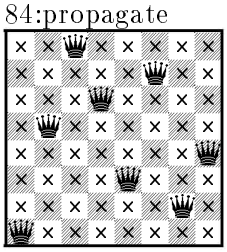
82:propagate



83:propagate



84:propagate



Appendix C

Houses example source code and WAM code

```
;;;   THIS IS THE FIDO-II source:
;;;
;;;   houses( [England, Spain,   Japan,  Italy,   Norway,
;;;              Green,   Red,  Yellow,   Blue,    White,
;;;            Painter, Violinist, Diplomat, Doctor, Sculptor,
;;;                Dog, Zebra,     Fox, Snails,    Horse,
;;;              Juice, Water,     Tea, Coffee,     Milk]) :-
;;;       define_domain(houses, [England, Spain, Japan, Italy, Norway,
;;;                              Green, Red, Yellow, Blue, White,
;;;                              Painter, Diplomat, Violinist, Doctor, Sculptor,
;;;                              Dog, Zebra, Fox, Snails, Horse,
;;;                              Juice, Water, Tea, Coffee, Milk],
;;;                              1..5),
;;;       Norway = 1, Milk = 3,
;;;       neighbour(Norway, Blue),
;;;       neighbour(Fox, Doctor),
;;;       neighbour(Horse, Diplomat),
;;;       forward(Green =:= White + 1),
;;;       England = Red, Spain = Dog,
;;;       Japan = Painter, Italy = Tea,
;;;       Green = Coffee, Sculptor = Snails,
;;;       Diplomat = Yellow, Violinist = Juice,
;;;       all_different([England, Italy, Spain, Norway, Japan]),
;;;       all_different([Green, Red, Yellow, Blue, White]),
;;;       all_different([Painter, Diplomat, Violinist, Doctor, Sculptor]),
;;;       all_different([Dog, Zebra, Fox, Snails, Horse]),
;;;       all_different([Juice, Water, Tea, Coffee, Milk]),
;;;       instantiate([England, Spain,   Japan,  Italy,   Norway,
;;;                    Green,   Red,  Yellow,   Blue, White,
;;;                    Painter, Violinist, Diplomat, Doctor, Sculptor,
;;;                    Dog, Zebra, Fox,    Snails, Horse,
;;;                    Juice, Water,     Tea, Coffee,   Milk]).
;;;
;;;   neighbour(X, Y) :-
```

```
;;;       forward(X =:= Y - 1).
;;;   neighbour(X, Y) :-
;;;       forward(X =:= Y + 1).
;;;

;;; This is the FIDO-III WAM source

(defprocedure houses/1
 (allocate 26)
 (put_dvariable_perm 1 (1 23 4 5))  ; England
 (put_dvariable_perm 2 (1 2 3 4 5))  ; Spain
 (put_dvariable_perm 3 (1 2 3 4 5))  ; Japan
 (put_dvariable_perm 4 (1 2 3 4 5))  ; Italy
 (put_dvariable_perm 5 (1 2 3 4 5))  ; Norway
 (put_dvariable_perm 6 (1 2 3 4 5))  ; Green
 (put_dvariable_perm 7 (1 2 3 4 5))  ; Red
 (put_dvariable_perm 8 (1 2 3 4 5))  ; Yellow
 (put_dvariable_perm 9 (1 2 3 4 5))  ; Blue
 (put_dvariable_perm 10 (1 2 3 4 5))  ; White
 (put_dvariable_perm 11 (1 2 3 4 5))  ; Painter
 (put_dvariable_perm 12 (1 2 3 4 5))  ; Violonist
 (put_dvariable_perm 13 (1 2 3 4 5))  ; Diplomat
 (put_dvariable_perm 14 (1 2 3 4 5))  ; Doctor
 (put_dvariable_perm 15 (1 2 3 4 5))  ; Sculptor
 (put_dvariable_perm 16 (1 2 3 4 5))  ; Dog
 (put_dvariable_perm 17 (1 2 3 4 5))  ; Zebra
 (put_dvariable_perm 18 (1 2 3 4 5))  ; Fox
 (put_dvariable_perm 19 (1 2 3 4 5))  ; Snails
 (put_dvariable_perm 20 (1 2 3 4 5))  ; Horse
 (put_dvariable_perm 21 (1 2 3 4 5))  ; Juice
 (put_dvariable_perm 22 (1 2 3 4 5))  ; Water
 (put_dvariable_perm 23 (1 2 3 4 5))  ; Tea
 (put_dvariable_perm 24 (1 2 3 4 5))  ; Coffee
 (put_dvariable_perm 25 (1 2 3 4 5))  ; Milk
 (get_variable_perm 26 1)
 (get_list 1)
 (unify_value_perm 1)
 (unify_variable_temp 1)
 (get_list 1)
 (unify_value_perm 2)
 (unify_variable_temp 1)
 (get_list 1)
 (unify_value_perm 3)
 (unify_variable_temp 1)
 (get_list 1)
 (unify_value_perm 4)
 (unify_variable_temp 1)
 (get_list 1)
 (unify_value_perm 5)
 (unify_variable_temp 1)
```

```
(get_list 1)
(unify_value_perm 6)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 7)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 8)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 9)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 10)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 11)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 12)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 13)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 14)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 15)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 16)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 17)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 18)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 19)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 20)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 21)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 22)
```

```
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 23)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 24)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 25)
(unify_nil)
(put_value_perm 5 1)
(get_constant 1 1)      ; Norway = 1
(put_value_perm 25 1)
(get_constant 3 1)      ; Milk = 3
(put_value_perm 5 1)
(put_value_perm 9 2)
(call neighbour/2 2 26) ; neighbour(Norway,Blue).
(put_value_perm 18 1)
(put_value_perm 14 2)
(call neighbour/2 2 26) ; neighbour(Fox,Doctor).
(put_value_perm 20 1)
(put_value_perm 13 2)
(call neighbour/2 2 26) ; neighbour(Horse,Diplomat).
(put_value_perm 6 1)
(put_value_perm 10 2)
(call eq+1/2 2 26) ;    ; eq+1(Green,White).
(put_value_perm 1 1)    ; eq(Endland,Red).
(get_value_perm 7 1)
(put_value_perm 2 1)    ; eq(Spain,Dog).
(get_value_perm 16 1)
(put_value_perm 3 1)    ; eq(Japan,Painter).
(get_value_perm 11 1)
(put_value_perm 4 1)    ; eq(Italy,Tead).
(get_value_perm 23 1)
(put_value_perm 6 1)    ; eq(Green,Coffee).
(get_value_perm 24 1)
(put_value_perm 15 1)   ; eq(Sculptor,Snails).
(get_value_perm 19 1)
(put_value_perm 13 1)   ; eq(Diplomat,Yellow).
(get_value_perm 8 1)
(put_value_perm 12 1)   ; eq(Violonist,Juice).
(get_value_perm 21 1)

(put_list 1)
(unify_value_perm 3)    ; Japan
(unify_nil)
(put_list 2)
(unify_value_perm 5)    ; Norway
(unify_value_temp 1)
(put_list 1)
```

```
(unify_value_perm 2)    ; Spain
(unify_value_temp 2)
(put_list 2)
(unify_value_perm 4)    ; Italy
(unify_value_temp 1)
(put_list 1)
(unify_value_perm 1)    ; England
(unify_value_temp 2)
(call all_different/1 1 26)

(put_list 1)
(unify_value_perm 10)   ; White
(unify_nil)
(put_list 2)
(unify_value_perm 9)    ; Blue
(unify_value_temp 1)
(put_list 1)
(unify_value_perm 8)    ; Yellow
(unify_value_temp 2)
(put_list 2)
(unify_value_perm 7)    ; Red
(unify_value_temp 1)
(put_list 1)
(unify_value_perm 6)    ; Green
(unify_value_temp 2)
(call all_different/1 1 26)

(put_list 1)
(unify_value_perm 15)   ; Sculptor
(unify_nil)
(put_list 2)
(unify_value_perm 14)   ; Doctor
(unify_value_temp 1)
(put_list 1)
(unify_value_perm 13)   ; Diplomat
(unify_value_temp 2)
(put_list 2)
(unify_value_perm 12)   ; Violonist
(unify_value_temp 1)
(put_list 1)
(unify_value_perm 11)   ; Painter
(unify_value_temp 2)
(call all_different/1 1 26)

(put_list 1)
(unify_value_perm 20)   ; Horse
(unify_nil)
(put_list 2)
(unify_value_perm 19)   ; Snails
(unify_value_temp 1)
```

```
 (put_list 1)
 (unify_value_perm 18)   ; Fox
 (unify_value_temp 2)
 (put_list 2)
 (unify_value_perm 17)   ; Zebra
 (unify_value_temp 1)
 (put_list 1)
 (unify_value_perm 16)   ; Dog
 (unify_value_temp 2)
 (call all_different/1 1 26)

 (put_list 1)
 (unify_value_perm 25)   ; Milk
 (unify_nil)
 (put_list 2)
 (unify_value_perm 24)   ; Coffee
 (unify_value_temp 1)
 (put_list 1)
 (unify_value_perm 23)   ; Tea
 (unify_value_temp 2)
 (put_list 2)
 (unify_value_perm 22)   ; Water
 (unify_value_temp 1)
 (put_list 1)
 (unify_value_perm 21)   ; Juice
 (unify_value_temp 2)
 (call all_different/1 1 26)

 (put_value_perm 26 1)
 (deallocate)
 (execute labeling/1 1)
)

(defprocedure neighbour/2
  (forward 2 (d d) $c$bneighbour)
)

(defprocedure $c$neighbour/2
  (try_me_else L1 2)
  (2-consistent (1 2) (lambda(x y) (eql x (1- y))))
  (proceed)
 L1
  (trust_me_else_fail 2)
  (2-consistent (1 2) (lambda(x y) (eql x (1+ y))))
  (proceed)
)

(defprocedure eq+1/2
  (forward 2 (d d) $c$eq+1)
)
```

```
(defprocedure $c$eq+1/2
  (2-consistent (1 2) (lambda(x y) (eql x (1+ y))))
  (proceed)
)
```

Appendix D

SEND+MORE=MONEY source code and WAM code

```
;;;   THIS IS THE FIDO-II source:
;;
;;puzzle([S,E,N,D,M,O,R,Y],[C1,C2,C3,C4]) :-
;;    define_domain(digits, [S,E,N,D,M,O,R,Y], 0..9),
;;    define_domain(carry, [C1,C2,C3,C4], 0..1),
;;    forward(S =\= 0),
;;    forward(M =\= 0),
;;    all_different([S,E,N,D,M,O,R,Y]),
;;    forward(C1         =:= M),
;;    forward(C2 + S + M =:= O + 10 * C1),
;;    forward(C3 + E + O =:= N + 10 * C2),
;;    forward(C4 + N + R =:= E + 10 * C3),
;;    forward(     D + E =:= Y + 10 * C4),
;;    permute(1, [C1,C2,C3,C4,M,E,N,O,D,R,Y,S], [Newlist]),
;;    instantiate_dl(Newlist).
;;
;;all_different([]).
;;all_different([H|T]) :-
;;    out_of(H, T),
;;    all_different(T).
;;
;;out_of(_, []).
;;out_of(H1, [H2|T]) :-
;;    forward(H1 =\= H2),
;;    out_of(H1, T).
;;
;;

;;; This is the FIDO-III WAM source
;;puzzle([S,E,N,D,M,O,R,Y],[C1,C2,C3,C4])
(defprocedure puzzle/2
  (allocate 14)
  (put_dvariable_perm 1 (0 1 2 3 4 5 6 7 8 9)) ; Y1 = S
  (put_dvariable_perm 2 (0 1 2 3 4 5 6 7 8 9)) ; Y2 = E
```

```
(put_dvariable_perm 3 (0 1 2 3 4 5 6 7 8 9)) ; Y3 = N
(put_dvariable_perm 4 (0 1 2 3 4 5 6 7 8 9)) ; Y4 = D
(put_dvariable_perm 5 (0 1 2 3 4 5 6 7 8 9)) ; Y5 = M
(put_dvariable_perm 6 (0 1 2 3 4 5 6 7 8 9)) ; Y6 = O
(put_dvariable_perm 7 (0 1 2 3 4 5 6 7 8 9)) ; Y7 = R
(put_dvariable_perm 8 (0 1 2 3 4 5 6 7 8 9)) ; Y8 = Y

(get_variable_perm 13 1)
(get_list 1)
(unify_value_perm 1)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 2)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 3)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 4)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 5)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 6)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 7)
(unify_variable_temp 1)
(get_list 1)
(unify_value_perm 8)
(unify_nil)

(put_dvariable_perm 9  (0 1)) ; Y1 = C1
(put_dvariable_perm 10 (0 1)) ; Y2 = C2
(put_dvariable_perm 11 (0 1)) ; Y3 = C3
(put_dvariable_perm 12 (0 1)) ; Y4 = C4

(get_variable_perm 14 2)
(get_list 2)
(unify_value_perm 9)
(unify_variable_temp 2)
(get_list 2)
(unify_value_perm 10)
(unify_variable_temp 2)
(get_list 2)
(unify_value_perm 11)
(unify_variable_temp 2)
(get_list 2)
(unify_value_perm 12)
```

```
(unify_nil)

(put_value_perm 1 1) ; S
(put_constant 0 2)
(call dungl/2 2 14)  ; dungl(S,0)

(put_value_perm 5 1) ; M
(put_constant 0 2)
(call dungl/2 2 14) ; dungl(M,0)

(put_value_perm 13 1) ; alldifferent([S,E,N,D,M,O,R,Y])
(call all_different/1 1 14)

(put_value_perm 9 1) ; C1
(get_value_perm 5 1) ; M

(put_value_perm 10 1) ; C2
(put_value_perm 1 2)  ; S
(put_value_perm 5 3)  ; M
(put_value_perm 6 4)  ; O
(put_value_perm 9 5)  ; C1
(call num10/5 5 14)

(put_value_perm 11 1) ; C3
(put_value_perm 2 2)  ; E
(put_value_perm 6 3)  ; O
(put_value_perm 3 4)  ; N
(put_value_perm 10 5)  ; C2
(call num10/5 5 14)

(put_value_perm 12 1) ; C4
(put_value_perm 3 2)  ; N
(put_value_perm 7 3)  ; R
(put_value_perm 2 4)  ; E
(put_value_perm 11 5)  ; C3
(call num10/5 5 14)

(put_value_perm 4 1) ; D
(put_value_perm 2 2)  ; E
(put_value_perm 8 3)  ; Y
(put_value_perm 12 4)  ; C4
(call snum10/4 4 14)

(put_list 1)
(unify_value_perm 9)  ; C1
(unify_variable_temp 2)
(get_list 2)
(unify_value_perm 10) ; C2
(unify_variable_temp 2)
(get_list 2)
```

```
  (unify_value_perm 11) ; C3
  (unify_variable_temp 2)
  (get_list 2)
  (unify_value_perm 12) ; C4
  (unify_value_perm 13)

  (deallocate)
  (execute labeling/1 1)
)

;;    forward(U + V + W =:= X + 10 * Y)
(defprocedure num10/5
  (forward 5 (d d d d d) $c$num10)
)

(defprocedure $c$num10/5
  (5-consistent (1 2 3 4 5) (lambda (u v w x y) (= (+ u v w) (+ x (* 10 y)))))
  (proceed)
)

;;    forward( U + V =:= W + 10 * X),
(defprocedure snum10/4
  (forward 4 (d d d d) $c$snum10)
)

(defprocedure $c$snum10/4
  (4-consistent (1 2 3 4) (lambda (u v w x) (= (+ u v) (+ w (* 10 x)))))
  (proceed)
)
```

Bibliography

[AK90] Hassan Aït-Kaci. The WAM: The (Real) Tutorial. Technical Report 5, DEC Paris Research Laboratory, 85, Avenue Victor Hugo, 92563 Rueil Malmaison Cedex, France, January 1990.

[AKBLN90] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1990.

[All83] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[BBH+91] A. Bernardi, H. Boley, K. Hinkelmann, P. Hanschke, C. Klauck, O. Kühn, R. Legleitner, M. Meyer, M.M. Richter, G. Schmidt, F. Schmalhofer, and W. Sommer. ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge. In *Expert Systems and their Applications: Tools, Techniques and Methods*, Avignon, France, 1991. Also available as Research Report RR-91-27, DFKI GmbH.

[Bee85] Joachim Beer. Comments on Compiling Prolog-Programs using Warren's Abstract Instruction Set. Technical report, GMD First, TU Berlin, November 1985.

[BHH+91] H. Boley, P. Hanschke, M. Harm, K. Hinkelmann, T. Labisch, M. Meyer, J. Müller, T. Oltzen, M. Sintek, W. Stein, and F. Steinle. μCAD2NC: A declarative lathe-workplanning model transforming CAD-like geometries into abstract NC programs. DFKI Document D-91-15, DFKI GmbH, November 1991.

[BHHM91] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A Hybrid Knowledge Compilation Laboratory. Presented at 3rd International Workshop on Data, Expert Knowledge and Decisions: Using Knowledge to Transform Data into Information for Decision Support, Reisensburg, Germany, September 1991.

[Boi86] P. Boizumault. A general model to implement DIF and FREEZE. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 585–592. Springer, 1986.

[Bol90] Harold Boley. A Relational/Functional Language and Its Compilation into the WAM. SEKI Report SR-90-05, Universität Kaiserslautern, Fachbereich Informatik, April 1990.

[BR89] Alexander Bockmayr and Hans-Holger Rath. Extended Prolog with Boolean Unification. Technical report, Sonderforschungsbereich 314, Institut für Logik, Kmplexität und Deduktionssysteme, Universität Karlsruhe, Germany, 1989.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS87] Wolfram Büttner and Helmut Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, 1987.

[Büt88] Wolfram Büttner. Unification in Finite Algrebras is Unitary (?). In E. Lusk and R. Overbeek, editors, *Proceedings 9th International Conference on Automated Deduction (CADE 88)*, number 310 in Lecture Notes in Computer Science, pages 368–377. Springer, 1988.

[Car87] M. Carlsson. Freeze, Indexing and Other Implementation Issues in the WAM. In Jean-Louis Jaffar, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 40–58. MIT Press, 1987.

[Cle87] John G. Cleary. Logical Arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[Col87a] Alain Colmerauer. Introduction to Prolog III. In *Annual ESPRIT Conference*, pages 611–629. North Holland, 1987.

[Col87b] Alain Colmerauer. Opening the Prolog III Universe. *Byte*, pages 177–182, August 1987.

[Deb86] Saumya K. Debray. Register Allocation in a Prolog Machine. In *Symposium on Logic Programming*, pages 267–275. IEEE, 9 1986.

[DNT91] Michel Dorochevsky, Jacques Noyé, and Oliver Thibault. Has Dedicated Hardware for Prolog a Future ? In H. Boley and M.M. Richter, editors, *Processing Declarative Knowledge (PDK)*, number 567 in LNCS, pages 17–31, 1991.

[ECR91] ECRC. Collection of new SEPIA 3.0.16 features (Interproc. Comm., Metaterms, Documentation Update). Sepia Distribution, 1991.

[Fil88] Thomas Filkorn. Unifikation in endlichen Algebren und ihre Integration in Prolog. Master's thesis, Technische Universität München, Institut für Informatik, November 1988.

[Hei89] Hans-Günther Hein. Adding WAM-Instructions to support Valued Clauses for the Relational/Functional Language RELFUN. SEKI Working Paper SWP-90-02, Universität Kaiserslautern, Fachbereich Informatik, December 1989.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Ma., 1989.

[HFKF91] Ryuzo Hasegawa, Hiroshi Fujita, Miyuki Koshimura, and Masayuki Fujita. A parallel model-generation theorem prover with ramnified term-indexing. In IEEE Computer Society Press, editor, *Seventh IEEE Conference on AI Applications*, pages 32–38, 1991.

[Hin91] Knut Hinkelmann. Bidirectional reasoning of horn clause programs: Transformation and compilation. DFKI Technical Memo TM-91-02, DFKI GmbH, January 1991.

[HL88] T. Huynh and C. Lassez. A CLR(R) Option Trading Analysis System. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 59–69, 1988.

[HM92] H.-G. Hein and M. Meyer. A WAM Compilation Scheme. In A. Voronkov, editor, *Logic Programming: Proceedings of the* 1^{st} *and* 2^{nd} *Russian Conferences*, volume 592 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 201–214. Springer-Verlag, Berlin, Heidelberg, 1992.

[HMS87] N. C. Heintze, S. Michaylov, and P. J. Stuckey. CLP(r) and Some Electrical Engineering Problems. In Jean-Louis Jaffar, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 675–703. MIT Press, 1987.

[Hol90] Christian Holzbaur. Realization of Forward Checking in Logic Programming through Extended Unification. Technical Report TR-90-11, Austrian Research Institute for Artificial Intelligence, Freyung 6, A-1010 Vienna, Austria, June 1990.

[Hry88] Tomas Hrycey. Temporal Prolog. In *ECAI-88 Proceedings*, pages 296–301, 1988.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of Princlipes of Programming Languages (POPL)*, pages 111–119, 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP system. In Jean-Louis Jaffar, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218. MIT Press, 1987.

[JMSY90] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) Language and System. Technical Report CMU-CS-90-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1990.

[JS89] Thomas Jost and Reinhard Skuppin. Technical¡ Diagnosis Based on Numerical Models Using PROLOG III. In Commission of the European Communities, editor, *ESPRIT'89*, pages 513–527. North Holland, 1989.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In J. Rosenfeld, editor, *Information Processing 74*, pages 556–574. North Holland, Amsterdam, 1974.

[Kow79] R. Kowalski. Algorithm = Logic + Control. *Journal of the ACM*, 22:424–436, 1979.

[KS88] Wolfgang Krautter and Michael Steinert. A Knowledge Representation for Model-Based Reasoning using Prolog-III. In Commission of the European Communities, editor, *ESPRIT'88*, pages 814–825. North Holland, 1988.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[MAC+89] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D.H. de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA – An Extendible Prolog System. In G. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress*, pages 1127–1132, August 1989.

[Mei90] Micha Meier. Compilation of Compound Terms in Prolog. In Saumya Debray and Manuel Hermenegildo, editors, *North American Conference on Logic Programming*, pages 63–79. MIT Press, October 1990.

[Mey92] Manfred Meyer. Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment. In *Fifth International Symposium on Artificial Intelligence (ISAI), Cancun, Mexico*, December 1992.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[MMS91] M. Meyer, J. Müller, and S. Schrödl. FIDO: Exploring Finite Domain Consistency Techniques in Logic Programming. In Harold Boley and Michael M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in Lecture Notes in Artificial Intelligence (LNAI), pages 425–427. Springer-Verlag, Berlin, Heidelberg, 1991.

[MN86] Ursula Martin and Tobias Nipkow. Unification in Boolean Rings. In *Proceedings of the 8th International Conference on Automated Deduction*, pages 506–513, July 1986.

[Mül91] J. Müller. Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining. Diploma thesis, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, 1991.

[Nys] Sven Olof Nystrøm. Nywam - a WAM emulator written in LISP.

[Pra91] Claudine Pradelles. Planning and scheduling using chip, June 1991. Slides from "Tutorial on Industrial Applications of Constraints" given at ILCP'91.

[Ric89] Michael M. Richter. *Künstliche Intelligenz*, chapter 16, pages 243–259. 1989.

[Rou75] P. Roussel. *PROLOG Manual de Reference et d'Utilisation*, 1975.

[Roy90] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming*. PhD thesis, Computer Science Division, University of California, Berkeley, December 1990.

[SA89] Ko Sakai and Akira Aiba. CAL: A Theoretical Background of Contraint Logic Programming and Its Applications. *J. Symbolic Computation*, 8:589–603, 1989.

[Sch91] S. Schrödl. FIDO: Ein Constraint-Logic-Programming-System mit Finite Domains. ARC-TEC Discussion Paper 91-05, DFKI GmbH, Postfach 2080, D-6750 Kaiserslautern, June 1991.

[SH91] Greg Sidebottom and William S. Havens. Hierarchical arc consistency applied to numerical processing in constraint logic programming. Technical Report CSS-IS TR 91-06, Simon Fraser University, Burnaby, British Columbia, V5A 1S6, Canada, 1991.

[Sku89a] Reinhard Skuppin. Das Diagnosesystem PROMOTEX I, 1989. Infosheet.

[Sku89b] Reinhard Skuppin. Modellierung technischer Systeme in PROLOG III, 1989.

[SSF89] Richard Schmid, Hans-Albert Schneider, and Thomas Filkorn. Using an Extended PROLOG to Solve the Lion and Unicorn Puzzle. *J. Automated Reasoning*, 5:403–408, 1989.

[Ste92] W. Stein. Design and Implementation of a Constraint Compiler for FIDO. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, Forthcoming 1992.

[Tay90a] Andrew Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185, Cambridge, Massachusetts London, England, 1990. MIT Press.

[Tay90b] Andrew Taylor. Quicksort Speed in Prolog. USENET electronic article in comp.lang.prolog, July 1990.

[Tay91] A. Taylor. *High Performance PROLOG Implementation.* PhD thesis, Basser Dpt. of Computer Science, University of Sydney, AU, 1991.

[Vod88] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical Report, Complete Logic Systems, North Vancouver, British Columbia, Canada, 1988.

[Wal72] D. L. Waltz. Generating semantics descriptions from drawings of scenes with shadows. Technical report AI-TR-271, Masssachusetts Institute of Technology, Cambridge, 1972.

[War77] David H. D. Warren. IMPLEMENTING PROLOG – compiling predicate logic programs. D.A.I. Research Report 39, dont know, July 1977.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, October 1983.

[Yap91] Roland H. C. Yap. Restriction Site Mapping in CLP($\mathcal{R}$). In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 521–534, Cambridge, Massachusetts London, England, 1991. MIT Press.