Chapter 4

Kernel transformations

1. Introduction

Four optimizing transformations are done on the kernel Prolog representation of programs: formula manipulation, factoring, global dataflow analysis, and determinism extraction. The goal of the transformations is to reduce a single metric: The total execution time of all unifications in the program. This metric is approximated by the number of unifications and by the size of the terms being unified. The chapter first describes the representation of types as logical formulas in the compiler. This is followed by a description of each of the four transformations:

(1) **Formula manipulation**. The compiler implements a set of primitive transformations to replace Prolog code and types (both are represented as logical formulas) with simpler versions that have identical semantics. The simplicity of a formula is defined as the number of goals in the formula. These transformations are done whenever there is a possibility that the code is too complex, i.e. upon reading in a program and after other transformations such as the determinism transformation (see below).

(2) **Factoring**. This transformation groups sets of clauses in a predicate together if they have head unifications in common. This reduces the number of head unifications and shallow backtracking steps.

(3) **Global dataflow analysis**. This stage analyzes the program, annotates it with types, and restructures it. The analyzer uses abstract interpretation to determine the types of predicate arguments.

(4) **Determinism transformation**. This stage rewrites the program to make its determinism explicit, i.e. it replaces shallow backtracking by conditional branching. Many of the other transformations in this chapter are chosen to make this transformation possible more often. The transformation converts the predicate into a series of nested case statements. Sometimes this is only partially successful; certain branches of the case statements may still retain disjunctions (OR choices) that could not be converted into deterministic code.

To improve readability, the examples in this chapter are given in standard Prolog notation. It is understood that they are represented internally in kernel Prolog.

2. Types as logical formulas

Throughout the compiler, type information about variables is represented with logical formulas. During compilation, any information learned is added to the formula, and deduction based on the formula simplifies the generated code. It is a simple and powerful approach to avoid doing redundant operations at run-time. For example, if a variable is dereferenced once, then it should never be dereferenced again. Types in the compiler are defined as follows:

> **Definition T**: Given a predicate f/n with main functor f and arity n, a *type* of f/n is a term $(f(A_1, A_2, \cdots, A_n)$:- `Formula`) where the $A_1, A_2, \cdots, A_n$ are n distinct variables and `Formula` is a logical formula (i.e. a Prolog term).

For example, the type `(range(A,B,C):-integer(A),var(B),integer(C))` says that the first and third arguments of `range/3` are integers and the second argument is an unbound variable. The compiler recognizes all Prolog type-checking predicates in the type formula. Appendix A gives a table of the types recognized by the compiler. In addition to these types, several other types are recognized that do not correspond to Prolog predicates. These types introduce distinctions between objects that depend on the implementation and are indistinguishable in the language, for example, the difference between an integer and a dereferenced integer, and the difference between an unbound variable that is not aliased to any other and an unbound variable that may be aliased. The following types are recognized that do not exist as Prolog predicates:

Internal Type	Description
`uninit(X)`	X is an uninitialized memory argument.
`uninit_mem(X)`	X is an uninitialized memory argument.
`uninit_reg(X)`	X is an uninitialized register argument.
`unbound(X)`	X is of one of the types `uninit_mem(X)`, `uninit_reg(X)`, or `var(X)`.
`deref(X)`	X is dereferenced, i.e. it is accessible without following any pointers.
`rderef(X)`	X is recursively dereferenced, i.e. it is dereferenced, and if it is compound then all its arguments are recursively dereferenced.

These types should not be given by the programmer since incorrect code or a significant loss of efficiency may result if they are used incorrectly. For example, declaring an argument of a predicate to be of uninitialized register type, i.e. the argument is an output that is passed in a register, may cause a large increase in stack space used by the program if that predicate is the last goal in a clause, because last call optimization is not possible in that instance. The safe approach is to leave the use of these types up to the compiler.

The use of logical formulas to hold information during compilation can be contrasted with the use of a symbol table in a compiler for an imperative language.† Representing types as logical formulas has two advantages over a symbol table: (1) They are more flexible during compiler development. The kind of information stored in a symbol table must be known when the compiler is designed. Formulas can contain kinds of information that are not known during the compiler's design. (2) They lend themselves to powerful symbolic manipulation such as deduction. Improving the deductive abilities leads to better code without having to change any other part of the compiler. The disadvantage of this representation is that its manipulation is slow. Future versions of the compiler could use a representation that is faster in the common cases.

Type formulas are used in the following ways in the compiler:

(1) Representing type information known about a set of variables. For example, the formula `(var(X),atom(Y))` means that X is an unbound variable and Y is an atom. The user manual (Appendix A) lists the types recognized by the compiler.

(2) Using a primitive form of deduction to simplify the generated code. For example, assume the formula is `(list(X),var(Y),deref(Z),...)`. To compile a run-time check that X is a nonvariable, the compiler first checks whether this formula implies `nonvar(X)`. This is true because `list(X)` implies `nonvar(X)`, so no run-time check is necessary.

(3) Updating the type formula when new information is learned. After compiling a goal, the formula is updated to represent the new knowledge that is gained. For example, after executing the arithmetic

† Of course, both the assembler and the run-time system use standard symbol tables.

expression `X is A+B` it is known that X is an integer, so the formula is extended with `integer(X)`.

In most cases, logical formulas are immutable, e.g. when a variable X is known to be a list (represented as `list(X)`), that fact remains true forever. This is not true for all types. The types used to denote unbound variables (e.g. `var(X)` and `uninit(X)`) become false as a result of an instantiation. This is also true of the standard order comparisons (e.g. `X@<Y`, `X@>Y`, and so forth) and the types `deref(X)` and `rderef(X)`. The compiler is careful to take this into account when updating the type formula.

Table 4.1 – Primitives to manipulate logical formulas and Prolog formulas

Primitive	Description
F_1 implies F_2	Implication: Succeeds if it can determine that there does not exist an assignment to variables in F_1 and F_2 that causes both F_1 and not(F_2) to succeed.
F_2 := simplify(F_1)	F_2 is a simplification of F_1.
F_2 := subsume(F, F_1)	F_2 is a simplification of F_1, given that F is true.
F_2 := update_formula(F, F_1)	F_2 is the result of removing information contradicted by F from F_1 and adding F to F_1.

3. Formula manipulation

The compiler implements a set of primitive transformations to manipulate formulas. They are summarized in Table 4.1, where F, F_1, and F_2 are logical formulas. Each of these primitives has two versions: a pure logical and a Prolog version. The logical version is used to manipulate types (see previous section). It assumes the formula has a purely logical meaning, i.e. that the operational concepts of execution order of goals, number of solutions, and backtracking behavior are not important. The Prolog version is used to manipulate kernel Prolog code. It assumes the formula must keep Prolog's operational semantics.

Implication is implemented to work well with most combinations of Prolog predicates that are used in type declarations. The following examples all return with success:

Table 4.2 – Examples of simplification

Formula	Simplified formula: logical	Simplified formula: Prolog	Comments
`(true ; true)`	`true`	`(true ; true)`	The Prolog version is unchanged unless the compiler option `same_number_solutions` is disabled.
`(p,fail)`	`fail`	`(p,fail)`	The Prolog version is unchanged unless the compiler can deduce that `p` has no side effects (read / write or assert / retract).
`(!,p ; q)`	`(p ; q)`	`(!,p)`	Cut is logically identical to `true`, but it must be retained since it modifies backtrack behavior in the entire clause containing it.

`atom(X)` implies `nonvar(X)`
`X<Y` implies `integer(X)`
`X<5` implies `X<10`
`uninit(X)` implies `deref(X)`
`functor(X, _, 0)` implies `atomic(X)`
`(X==a; X==b)` implies `atom(X)`

Simplification is done on standard Prolog, on kernel Prolog, and on type formulas. Table 4.2 gives some examples to illustrate the difference between logical and Prolog semantics. A single function simplify(F) handles both logical and Prolog semantics (Figure 4.1). For conciseness, the definition of simplify(F) uses the compound terms `(A,B)`, `(A;B)`, `(A->B)`, and `(\+(A))` both as selectors (to choose the branch of the case statement) and constructors (in the calls to simp_step(F)). Tables 4.3 and 4.4 define part of the definition of simp_step(F), the primitive simplification step. The complete definition contains about 50 rules. The functions subsume(F, F_1) and update_formula(F, F_1) are implemented in a similar way.

```
function simplify(F : formula) : formula;
begin
    case  /* decompose the formula */
    F = (A,B)   : return simp_step( (simplify(A),simplify(B)) );  /* and */
    F = (A;B)   : return simp_step( (simplify(A);simplify(B)) );  /* or */
    F = (A->B)  : return simp_step( (simplify(A)->simplify(B)) );  /* implies */
    F = \+(A)   : return simp_step( \+(simplify(A)) );  /* negation */
    otherwise   : return simp_step(F);
    end
end;
```

Figure 4.1 – Simplification of a formula

Table 4.3 – Simplification rules (part of simp_step's definition)		
Rule		Condition to apply this rule
Input formula	Output formula	
`(true,A)`	`A`	*(none)*
`(A,true)`	`A`	*(none)*
`(true;A)`	`true`	semantics(prolog) $\wedge$ no_side_effects(A) $\wedge$ diff_sol $\wedge$ no_bind(A)
`(true;A)`	`true`	semantics(logical)
`(A,fail)`	`fail`	semantics(prolog) $\wedge$ no_side_effects(A)
`(A,fail)`	`fail`	semantics(logical)
`(fail,A)`	`fail`	*(none)*
`(fail;A)`	`A`	*(none)*
`(A->true;B)`	`A`	semantics(prolog) $\wedge$ succeeds(A) $\wedge$ deterministic(A)
`(A->true;B)`	`A`	semantics(logical) $\wedge$ succeeds(A)
`A`	`fail`	semantics(prolog) $\wedge$ fails(A) $\wedge$ no_side_effects(A)
`A`	`fail`	semantics(logical) $\wedge$ fails(A)

Table 4.4 – The conditions for applying simplification rules	
Condition	Description
semantics(S)	Simplify according to semantics S where S $\in$ {prolog,logical}.
no_side_effects(A)	Formula A does not have side effects when executed.
deterministic(A)	Formula A gives only one solution when executed.
no_bind(A)	Formula A does not bind any variables.
diff_sol	Relax semantics of Prolog to allow a different number of solutions.
succeeds(A)	Formula A always succeeds when executed.
fails(A)	Formula A always fails when executed.

4. Factoring

Factoring is based on the operation of finding the most-specific-generalization, or MSG, of two terms. Factoring collects groups of clauses whose heads can be combined in nontrivial fashion using the MSG operation. The advantage of factoring is that it reduces the number of unifications performed during execution. Figure 4.2 defines the MSG in terms of unification. Given two terms T_1 and T_2, consider the set M of all terms that unify with both of them. The MSG of T_1 and T_2 is the unique element T_m of M which unified with any other element U of M gives T_m. Intuitively, this says that T_m contains the maximal common information of T_1 and T_2.

The MSG (also called anti-unification) is the dual operation to unification. Given two terms, unification finds a term that is a more instantiated case of each of the two, i.e. the most general common instance of the two. The MSG is a term of which each of the two is a more instantiated case. For example, consider the two compound terms `s(A,x,C)` and `s(A,B,y)`. Unifying these two terms results in `s(A,x,y)`. The MSG of the two terms is `s(A,B,C)`. Unification may fail, i.e. the most general unifier

```
function msg(T1, T2 : term) : term;
var
    M : set of term;
    Tm, U : term;
begin
    M := { T | T unifies with T1 and T unifies with T2 };
    Find Tm ∈ M such that ∀ U ∈ M : unify(U, Tm) = Tm;
    return Tm
end;
```

Figure 4.2 – The most specific generalization

is the empty set. Finding the MSG never fails. In the worst case, the generalization of the two terms is an unbound variable, which represents the set of all terms. For example, consider the two atomic terms `x` and `y`. Unifying these two results in failure, whereas the MSG is an unbound variable.

Another way of viewing the MSG operation is as an approximation to the union of two sets. Every term corresponds to a set by instantiating the variables in the term to all possible ground values. In general, the union of two of these sets does not correspond to any term. The MSG finds the smallest superset of the union that is represented by a term. A similar property holds of unification: it finds the largest subset of the intersection that is represented by a term.

For all arguments of the predicate, the factoring transformation finds the largest contiguous set of clauses whose MSG is a compound term. This set is used to define a dummy predicate and the definition of the original predicate is modified to call the dummy predicate. The algorithm is given in Figure 4.3. As an example of factoring, consider the predicate:

```
h([x|_]).
h([y|_]).
h([]).
```

The lists in the heads of the first two clauses are combined: the MSG of `[x|_]` and `[y|_]` is `[_|_]`. The result after factoring is:

procedure factoring;
var
M : term;
C_i , C'_i : clause;
π , P_π : list of clause;
a , i , p , q : integer;
begin
for each predicate P in the program **do begin**
/* At this point P = [C_1 , C_2 , ... , C_n] (list of n clauses) */
/* and C_i = (H_i :- B_i) (Each clause has head H_i and body B_i) */
for a := 1 **to** arity(P) **do begin**
Partition P such that each contiguous group π = [C_p , C_{p+1} , ... , C_q] $(1 \le p \le q \le n)$ satisfies exactly one of the two properties:
1. Either $p = q$ (π contains only one clause), or
2. π is the largest group for which $M = \mathrm{MSG}_{i=p}^{q}$ (argument a of H_i) is compound.
for each contiguous group π **do if** $p < q$ **then begin**
/* Create the dummy predicate P_π */
for i := p **to** q **do begin**
C'_i := C_i ;
Remove M from H'_i ;
Add all variables in M as arguments to H'_i
end;
P_π := [C'_p , ..., C'_q];
/* Create the call to the dummy predicate */
H := (new head with same functor and arity as P and M in argument a);
H_π := (new head with same functor and arity as P_π);
for i := 1 **to** arity(P) **do if** $i \ne a$ **then begin**
Make argument i of H and H_π identical
end;
Replace π in P by the single clause C_π = (H :- H_π)
end
end
end
end;

Figure 4.3 – The factoring transformation

```
h([A|B]) :- h'(B, A).
h([]).

h'(B, x).
h'(B, y).
```

Factoring reduces the number of unifications done at run-time in two ways: (1) compound terms are only created once during predicate execution, instead of being repeated for each clause (e.g. the list `[A|B]` in the example), and (2) the arguments of compound terms become predicate arguments, which more often allows the determinism transformation to convert shallow backtracking into deterministic selection (e.g. the value of the second argument of the predicate `h’` determines the correct clause directly without any superfluous unifications). The following heuristic is used:

> **Factoring Heuristic:** For each argument in a predicate, factor the largest set of contiguous clauses whose MSG is a compound term. Repeat this operation until no more factoring is possible.

This heuristic needs refinement in some cases to avoid superfluous choice point creation which may slow down execution. The savings of multiple structure creation (how many fewer unifications are done) should be weighed against how much deterministic selection is possible in the dummy predicates.

If the compiler option `same_order_solutions` is enabled (the default) then the operational semantics is that of standard Prolog, i.e. the order of solutions returned on backtracking is identical to that of standard Prolog. Disabling the option relaxes the semantics of standard Prolog by also factoring non-contiguous clauses whose MSG is a compound term. This may change the ordering of solutions on backtracking. This option allows experimentation with variations of standard Prolog semantics.

To illustrate how factoring can reduce the amount of shallow backtracking, consider the following predicate, which is part of a definition of quicksort:

```
partition([Y|L],X,[Y|L1],L2) :- Y=<X, partition(L,X,L1,L2).
partition([Y|L],X,L1,[Y|L2]) :- Y>X,  partition(L,X,L1,L2).
partition([],_,[],[]).
```

The first argument of the first two clauses can be factored, resulting in:

```
partition([Y|L],X,L1,L2) :- partition’(L,X,L1,L2,Y).
partition([],_,[],[]).

partition’(L,X,[Y|L1],L2,Y) :- Y=<X, partition(L,X,L1,L2).
partition’(L,X,L1,[Y|L2],Y) :- Y>X,  partition(L,X,L1,L2).
```

(In the compound term `[Y|L]` the rightmost variable L is kept in the same argument position and the other variable Y is put at the end of the goal.) The transformation results in only a single unification of `[Y|L]` instead of two in the original definition. In the dummy predicate the comparisons `Y=<X` and `Y>X` use arguments of the predicate, not arguments of a compound term. This makes it possible to compile `partition/4` with a conditional branch instead of with shallow backtracking.

5. Global dataflow analysis

It is difficult to obtain information about a program by executing it in its original form, since the range of possible behaviors is potentially infinite, and even simple properties of programs may be undecidable. To get around this problem, the idea of *abstract interpretation* is to transform the program into a simpler form which allows practical analysis. After the analysis the inverse transformation gives information about the original program. The fundamentals of a general method based on this approach and its mathematical underpinning are explained by Kildall [37] and Cousot & Cousot [23]. Marriott and Sondergaard [47] give a lucid explanation of the basic ideas. This method has been studied extensively and developed into a practical tool for Prolog [18, 21, 24, 25, 49, 50, 53, 66, 67, 76, 84].

The four sections that follow summarize the relevant parts of the theory of abstract interpretation, present my application of it to Prolog, describe the analysis algorithm in detail, and discuss the integration of the algorithm into the body of the compiler. In Chapter 7 an evaluation is done of the effectiveness of the algorithm.

5.1. The theory of abstract interpretation

The transformed program should mimic the original faithfully. This is made rigorous by introducing the concept of *descriptions* of data objects. Let E be the powerset, i.e. the set of all subsets, of a set of data objects, and D be a partially ordered set of descriptions. Then an *abstract interpretation* is defined by the following conditions:

(1) $E_P : E \rightarrow E, \quad D_P : D \rightarrow D$

(2) $\alpha : E \rightarrow D, \gamma : D \rightarrow E$

(3) α and γ are monotonic.

(4) $\forall d \in D : d = \alpha(\gamma(d))$

(5) $\forall e \in E : e \leq \gamma(\alpha(e))$

(6) $\forall d \in D : E_P(\gamma(d)) \leq \gamma(D_P(d))$

The operator E_P in the first condition describes a single step of the execution of the program P as a state transformation. Symbolic execution of the transformed program is described by the operator D_P. Except for the conditions given above, the choice of E_P and D_P is completely free. The choice is guided by several trade-offs, for example: (1) speed versus precision of the analysis, (2) complexity versus confidence in the correctness of the analysis.

As an example of E_P (from Cousot & Cousot [23]), consider a program in an imperative language represented as a graph where each node is a simple statement such as an assignment or a conditional. Let an *environment* be defined as a correspondence between each variable in the program and a possible value. Then for each edge of the graph a set of possible environments (called a *context*) is given. Initially they are all unknown. An application of E_P transforms all contexts to their new values reached after one execution step.

For Prolog, a natural choice is to identify E_P with the standard operator $T_P : 2^{B_P} \rightarrow 2^{B_P}$ which describes its procedural semantics. In this case E is 2^{B_P}, where B_P is the Herbrand universe of the program P, i.e. the set of all ground goals† that can be constructed using predicates, functors, and constants of the program. T_P does a single ‘‘forward chaining’’ step to find the conclusions that can be inferred from a given set of ground goals. Formally, T_P maps any $I \subseteq \mathrm{B_P}$ into $T_P(I) = \{ A \in \mathrm{B_P} : \mathrm{A} \;\; {:}{-} \;\; \mathrm{A}_1, \cdots, \mathrm{A_n}$ is a ground instance of a clause in P and $\{ A_1, \cdots, A_n \} \subseteq \mathrm{I} \}$. In other words, an application of T_P transforms a subset of B_P into a new subset containing the new goals inferred from the program's clauses given the old goals. The *meaning* of a program P is defined as $\mathrm{lfp}(T_P)$ (where lfp = the least fixpoint operator). This is the set of all ground goals that can be derived from the program clauses. For example, consider the following program:

```
nat(0).
nat(s(X)) :- nat(X).
```

which states that `nat(X)` is true if X is zero or X is the successor of a natural number. The program's meaning is:

† These are called ‘‘atoms’’ in mathematical logic. To avoid confusion with the atom data type in Prolog, this dissertation uses the Prolog terminology.

```
{ nat(0), nat(s(0)), nat(s(s(0))), nat(s(s(s(0)))), ... }
```

which represents the set of natural numbers.

The second and third conditions introduce the operators α (the abstraction function) and γ (the concretization function). The operator $\alpha : E \to D$ determines the description corresponding to a particular set of data objects. The operator $\gamma : D \to E$ determines the set of data objects corresponding to a particular description.

The fourth and fifth conditions ensure that α and γ behave correctly with respect to each other. Condition four means that in going from descriptions to data objects and back no information is lost. Condition five means that in going from a data object to a description and back that the resulting set of data objects includes the original data object. The sixth condition is known as the *safeness criterion*. It is necessary to ensure that the symbolic execution (through D_P) mimics the execution of P accurately (through E_P). In other words, the abstract interpretation gives descriptions that include all the data objects that the execution of the original program gives.

To illustrate what the conditions mean consider the abstract domain of signs of real numbers. The data objects are real numbers. Let there be three possible signs for numbers: + (positive), − (negative), and 0 (zero). The set of descriptions D describes the possible states of a set of reals, so it contains all combinations of the three signs:

$$D = \{ \{\}, \{0\}, \{+\}, \{-\}, \{+,-\}, \{-,0\}, \{+,0\}, \{+,-,0\} \}$$

According to the second condition α maps a set of reals onto its signs, and γ maps a set of signs onto a set of reals. For example:

$$\alpha(\{-5\}) = \{-\}$$

$$\alpha(\{-3, 5\}) = \{+, -\}$$

$$\gamma(\{+\}) = \{ r \in \mathbf{R}, r > 0 \}$$

The fourth condition says that going from a sign to a set of reals and back will give the same sign. The fifth condition says that going from a set of reals to a sign and back will give a set of reals that includes the original set. So for example:

$$\{+\} = \alpha(\gamma(\{+\}))$$

since $\gamma(\{+\}) =$ the set of positive reals, whose sign is {+}, and:

$$\{5\} \subset \gamma(\alpha(\{5\}))$$

since $\alpha(\{5\}) = \{+\}$, and $\gamma(\{+\})$ is the set of positive reals, which contains 5. In order to explain condition six, consider the equation 27×37. Here E_P is multiplication of reals, and D_P is the corresponding operation in the abstract domain of signs. The multiplication corresponds to $\{+\} \times \{+\}$ in the abstract domain. The result of the abstract multiplication should be {+}, since $27 \times 37 = 999$, which is positive. Condition six is a formalization of this requirement.

Dataflow analysis is done by transforming the original program over the domain E described by E_P to a new version over the domain D described by D_P. Then $\gamma(\mathrm{lfp}(D_P))$ (lfp = the least fixpoint operator) gives a conservative estimate of the required information. Much work has been done in discovering useful domains D for particular applications and efficient algorithms for finding fixpoints of D_P [10, 53].

5.2. A practical application of abstract interpretation to Prolog

The implementation of abstract interpretation presented in this dissertation uses a very different E_P from the one suggested in the previous section by the formal definition of Prolog's procedural semantics. The choice of E_P used in the Aquarius compiler closely follows execution on a machine. Consider a program with n predicates P_i. The *data objects* are the n-tuples $(T_1, T_2, \cdots, T_n)$ where each T_i is a functor of same name and arity as P_i and the arguments of T_i are terms constructed using data functors and atomic terms in the program and possibly containing unbound variables. E is the powerset of these data objects. The *descriptions* are the n-tuples $(L_1, L_2, \cdots, L_n)$ where each L_i is a functor of same name and arity as P_i and the arguments of L_i are constrained to be on a given finite lattice. D is the set of these descriptions. A *lattice* is a partially ordered set in which every nonempty subset has a least upper bound (denoted as the lub) and a greatest lower bound (denoted as the glb). Each of the elements of the lattice corresponds to a set of possible values in the original program. This lattice is called an *argument lattice*, since it is used to represent the possible values of a predicate argument. A *predicate lattice* (such as L_i) is the Cartesian product of the lattices of all the predicate's arguments.

The operator E_P that mirrors execution of the program corresponds to a single resolution step. It is a transformation of a set of data objects and an execution state to another set of data objects and a new execution state, following Prolog's depth-first execution semantics, that is, its left-to-right execution of goals in a clause, and its top-to-bottom selection of clauses in a predicate. The operator D_P that mirrors execution of the program over the descriptions is similar, except that the arguments are lattice values.

If the conditions of abstract interpretation hold, then the least fixpoint of the symbolic execution over the lattice is a conservative approximation to the global information, in other words the set of values that a variable can have during execution is a subset of what is derived in the analysis.

The three sections that follow describe the lattice used by the analysis algorithm. The first section introduces and defines the lattice elements and the types with which they correspond. The next section gives an example to show how to derive the types. The last section summarizes the properties of the types that are used by the algorithm.

5.2.1. The program lattice

Dataflow analysis for Prolog differs from that of statically typed languages because it does not check types, but it infers them. The most important information that can be deduced about an argument is whether it is used as an input or an output argument of a predicate, i.e. the *mode* of the argument. After the mode is determined, it is useful to find its *type*, i.e. the set of values that it can have. The remainder of this chapter refers only to the type of an argument, in the assumption that this implies the mode as well. I have experimented with four lattices of varying complexity in the analyzer, and the lattice that is currently implemented has been chosen to give the most information while keeping analysis fast.

During the analysis the algorithm maintains two lattices for each predicate in the program. These lattices correspond to the *entry* and *exit* types of the predicate, i.e. the value of the variable valid upon entering the predicate and upon successful exit from the predicate. The lattice describing the entire program is the Cartesian product of the predicate lattices.

Figure 4.4 – The argument lattice

The argument lattice of the entry and exit types in the current analyzer is shown in Figure 4.4. In this lattice, `any` (the top element) denotes the set of all values, `impossible` (the bottom element) denotes the empty set (i.e. this predicate is unreachable during execution), `uninit` denotes the set of uninitialized variables (unbound variables that are not aliased; see Chapter 2), `ground` denotes the set of values that are ground (i.e. the term contains no unbound variables), `nonvar` denotes the set of nonvariables, `rderef` denotes the set of values that are recursively dereferenced (i.e. the term is dereferenced, which means that it is accessible without any pointer chasing, and if it is compound then all its arguments are recursively dereferenced), and `ground+rderef` denotes the set of values that are both ground and recursively dereferenced.

5.2.2. An example of generating an uninitialized variable type

This section gives a simple example of the generation of uninitialized variable types to give an idea of what abstract interpretation does and to illustrate the argument lattice. Uninitialized variables are generated whenever the analyzer deduces that an unbound variable cannot be aliased to another. For example, consider the following program fragment:

```
pred(...) :- ..., goal(Z), ...
goal(X) :- X=s(Y), goal(Y).
```

If Z is the first occurrence of that variable in the `pred(...)` clause then it is considered a candidate uninitialized variable. This is possible because it is certainly not aliased to any other variable. In the definition of `goal(X)`, if X is uninitialized then the argument Y of the structure `s(Y)` may be considered uninitialized as well. This Y is passed on as an argument to `goal(Y)`. Therefore both calls of `goal(X)` are with an uninitialized argument, so it is consistent to give the argument X an uninitialized variable type.

It may happen that elsewhere in the program there is a call of `goal(X)` where X is not uninitialized (for example it may be a compound term, or it may be aliased). In that case, the assumption that X is uninitialized is invalidated. This may invalidate assumptions about other arguments of other predicates, so it is necessary to propagate this information. For correctness, it is necessary to iterate until the least fixpoint is reached. At that point symbolic execution of the program does not change any of the derived types.

5.2.3. Properties of the lattice elements

The example given above already gives an inkling of the relevant properties of ground, uninitialized, and recursively dereferenced variables that simplify the analysis. Here is a more complete list of these properties:

- The property of being ground, uninitialized, or recursively dereferenced propagates through explicit unifications. The propagation is bidirectional:

  (1) If X is ground, uninitialized, or recursively dereferenced, then after executing an explicit unification with a compound term (e.g. `X=s(A,B)`), all of its variables (e.g. A and B) are ground, all of the new variables (e.g. A and B) are uninitialized, or all of the new variables are recursively dereferenced.

  (2) In the other direction, if all the variables in the compound term are ground, then X is ground. If all the variables are recursively dereferenced, then X is recursively dereferenced if it was

previously uninitialized.

- The property of being ground is independent of aliasing. For example, if X is ground, then it remains ground after executing the unification `X=Y`. This is not true of recursively dereferenced or uninitialized variables.

- An uninitialized variable is not aliased to any other variable. Lattice calculations for uninitialized variables do not affect each other.

5.3. Implementation of the analysis algorithm

Previous sections have introduced the ideas underlying the algorithm, the program lattice used by the algorithm, an example of how types are derived, and the properties of the lattice elements. This section gives a more complete explanation of the algorithm. The presentation starts with an overview of the data representation. It then describes the algorithm, and finally it gives a detailed example of analysis.

Table 4.5 – The components of the variable set VS

Name	Description
S	The set of variables encountered so far in the clause. This set is important because any variable encountered in a goal that is not in this set is known not to be aliased to any other, i.e. it is a new variable, and therefore it is both uninitialized and dereferenced.
G	The set of variables that are ground. These variables are bound to terms that contain no unbound variables.
N	The set of variables bound to a nonvariable term. This set is a superset of G.
U	The set of variables that are uninitialized. A variable becomes uninitialized if it is unbound and known not to be aliased to any other variable. The symbolic execution enforces this constraint. This set is disjoint with N.
D	The set of variables that are recursively dereferenced. A variable is recursively dereferenced if it is bound to a term that is dereferenced, i.e. it is accessible without any pointer chasing, and if it is compound then all its arguments are recursively dereferenced. This set is a superset of U.

5.3.1. Data representation

During analysis the types are represented in two ways:

(1) **As lattice elements**. For each predicate, there are two structures containing a lattice element in each

argument. These structures represent the entry and exit types of the predicate. For example, the predicate `concat/3` has two structures which could have the values:

```
entry: concat(any,ground,uninit)
 exit: concat(ground,ground,any)
```

This says that upon entering `concat/3` the second argument is ground and the third argument is uninitialized. When the predicate is exited the first two arguments are ground.

(2) **As sets of variables**. Type information can also be stored as a set for each type that contains the variables of that type.

These two different representations each have their advantages. The lattice representation makes it easy to calculate the lub (least upper bound). The variable set representation makes it easy to symbolically execute a clause, i.e. to propagate and update information about variables' types through the clause. Functions are provided to convert between the two representations (Figure 4.7). For the lattice in Figure 4.4, there are five sets of variables which are updated during the symbolic execution of a predicate. Conceptually they are part of a 5-tuple VS = (S, G, N, U, D) that holds the current type information (Table 4.5).

5.3.2. Evolution of the analyzer

The current analyzer was preceded by three simpler versions. The lattice of the first analyzer represented only entry types and had three elements: `impossible`, `uninit`, and `any`. The second analyzer added the `ground` type in the entry lattice and an exit lattice of the same structure. The third analyzer added the `rderef` type to these lattices. The current (fourth) analyzer added the `nonvar` type. Despite not using a representation for variable aliasing, the third and fourth analyzers are able to derive many nontrivial `rderef` and `nonvar` types. The added types are independent, i.e. each version of the analyzer does no better than previous versions on types that previous versions also derive.

The choice of what lattice types to add was done by inspecting the compiled code of programs and by deciding what types were easy to derive in the context of the structure of the existing analyzers. Types were added that are present in many programs. Measurements show that having an exit lattice and doing back propagation (see below) are essential features to derive good `ground`, `rderef`, and `nonvar` types. A numerical evaluation of the efficiency of the analysis (the percentage of arguments for which types are derived) and the effect of analysis on execution time and code size is given in Chapter 7.

For the next version of the analyzer the added types `rlist` (recursive list, i.e. the term is either nil or a cons cell whose tail is a recursive list), `integer`, and `((nonvar+deref) or uninit)` (the term is either a dereferenced nonvariable or uninitialized) are contemplated.

---

type varset = (set, set, set, set, set); /* 5-tuple */

var *Program* : set of predicate;
L_{entry} : mapping predicate $\rightarrow$ lattice;
L_{exit} : mapping predicate $\rightarrow$ lattice;
P : predicate;

procedure analysis;
var E : set of predicate;
VS : varset;
begin
$E := \{ P \mid \text{arity}(P) = 0 \} \cup$ (declared entry points);
Initialize L_{entry} with the types of the declared entry points;
Initialize L_{exit} to `impossible` for all predicate arguments;
while $E \neq \emptyset$ **do begin**
for each predicate $P \in E$ **do begin**
VS := lattice_to_varset($L_{entry}[P]$, P);
VS := update_exit(VS, predicate_analyze(P, VS), P)
end;
$E := \{ P \mid L_{entry}[P]$ has changed or $\exists\, G \in P : L_{exit}[G]$ has changed$\}$
end
end;

Figure 4.5 – The analysis algorithm: top level

5.3.3. The analysis algorithm

The analysis algorithm is presented at three levels of detail. An English-language description is given of the basic ideas. A detailed pseudocode definition (Figures 4.5 through 4.7) describes the complete algorithm at a high level of abstraction. Appendix G gives the implementation in the compiler.

The algorithm maintains entry and exit lattice elements for each predicate argument in the program. Analysis proceeds by traversing the call graph starting from a set of entry points that have known types. The entry points include all predicates of arity 0 and any entry declarations given by the programmer (Appendix A). The traversal is repeated until there are no more changes in the lattice values, that is, until a

```
function predicate_analyze(P : predicate; VS : varset) : varset;
var   F : formula;
      VS_i : array [1 .. n] of varset;
      G_e : goal;
      i , j : integer;
begin
      /* At this point P = [ C_1 , ... , C_n ] (list of n clauses) */
      for each non-active clause C_i ∈ P do begin /* Symbolic execution of clause C_i */
            /* At this point C_i = (G_i1 , ... , G_in_i) (conjunction of n_i goals) */
            VS_i := VS;
            for j := 1 to n_i do begin /* Symbolic execution of goal G_ij */
                  if (G_ij is a unification) then begin
                        VS_i := symbolic_unify(VS_i , G_ij) /* Figure 4.8 */
                  end else if G_ij ∈ Program then begin /* G_ij is defined in the program */
                        L_entry [G_ij] := lub(L_entry [G_ij], varset_to_lattice(VS_i , G_ij));
                        if non-exponentiality constraint then begin
                              VS_i := update_exit(VS_i , predicate_analyze(G_ij , VS_i), G_ij)
                        end
                  end else begin /* G_ij is not defined in the program */
                        F := varset_to_type(VS_i , G_ij);
                        G_e := entry_specialize(G_ij , F);
                        VS_i := update_exit(VS_i , exit_varset(G_e), G_ij)
                  end
            end;
            VS_i := back_propagate(VS_i , C_i) /* To obtain more precision */
      end;
      return ∩_{i=1}^{n} VS_i /* Merge the exit values of all VS_i */
end;
```

Figure 4.6 – The analysis algorithm: analyzing a predicate

fixpoint is reached. With suitable conditions (i.e. all type updating is *monotonic* and types are propagated correctly) this fixpoint is the least fixpoint and the resulting types give accurate information about the original program. When a goal is encountered during a traversal three things are done: (1) the goal's entry lattice type is updated using the current value of VS, (2) if the goal's definition is part of the program then the definition is entered, and (3) upon return, the new value of VS is used to update the goal's exit lattice type. A correct value of VS is maintained at all times during the traversal of a goal's definition.

The definition of the algorithm in Figures 4.5 through 4.7 leaves out some details but is a faithful description of the analysis. The two conditions *non-active* and *non-exponentiality* are explained in the next section. The following sections describe what happens in symbolic execution of a predicate (including back propagation) and symbolic execution of a goal.

```
function update_exit(VS1 , VS2 : varset; G : goal) : varset;
var    VS : varset;
begin
       /* Calculate new VS from old VS1 and exit VS2 */
       VS.nonvar := VS1.nonvar ∪ VS2.nonvar;
       VS.ground := VS1.ground ∪ VS2.ground;
       VS.rderef := (VS1.rderef ∩ VS1.ground) ∪ VS2.rderef;
       VS.sofar := VS1.sofar ∪ vars(G);
       VS.uninit := VS1.uninit − vars(G);
       /* Calculate new exit lattice */
       Lexit[G] := lub(Lexit[G], varset_to_lattice(VS, G));
       return VS
end;

function lub(L1 , L2 : lattice) : lattice;
return (least upper bound of L1 and L2);

function lattice_to_varset(L : lattice; G : goal) : varset;
return (varset corresponding to L using variables of G);

function varset_to_lattice(VS : varset; G : goal) : lattice;
return (lattice corresponding to VS using variables of G);

function back_propagate(VS : varset; C : clause) : varset;
return (improved exit varset from VS using unification goals of C);

function varset_to_type(VS : varset; G : goal) : formula;
return (type formula corresponding to VS using variables of G);

function entry_specialize(G : goal; F : formula) : goal;
return (specialized entry point of G when called with type F);

function exit_varset(G : goal) : varset;
return (exit varset stored for the known goal G);
```

Figure 4.7 – Utility functions needed in the analysis algorithm

5.3.4. Execution time of analysis

This section shows that the average analysis time for programs that contain only linearly recursive predicates (i.e. no clauses contain more than one recursive call) and that have bounded arity is proportional to the size of the program. The analysis time $T_{analysis}$ is proportional to the time of each iteration T_{iter} and the number of iterations N_{iter} needed to reach the least fixpoint:

$$T_{analysis} = O(T_{iter} \cdot N_{iter})$$

For programs that contain only linearly recursive predicates, the time of each iteration is:

$$T_{iter} = O(S \cdot A)$$

where S is the total number of goals in the program and A is the maximum number of times a predicate is traversed. (Programs with non-linearly recursive predicates are discussed below.) This is true because the algorithm traverses each clause at most once in an iteration. It assumes that the symbolic execution of a goal whose definition is not traversed is a constant time operation. A predicate is traversed only if the current entry type is worse than the previous worst entry type. The number of times this situation can occur is bounded by the depth of the entry lattice of the predicate, which is proportional to the maximal arity in the program. Therefore:

$$S = \sum_{i=1}^{n} \sum_{j=1}^{n_i} length(C_{ij})$$

$$A = O(\max_{i=1}^{n} arity(P_i))$$

where the program contains n predicates, and each predicate P_i contains n_i clauses C_{ij}. The arity of a predicate is denoted by $arity(P_i)$ and the number of goals in a clause is denoted by $length(C_{ij})$. The number of iterations is trivially bounded by the depth of the program lattice:

$$N_{iter} = O(D_{total})$$

where D_{total} is given by:

$$D_{total} = \sum_{i=1}^{n} 2 \cdot 4 \cdot arity(P_i)$$

In this equation, 2 counts the entry and exit lattices, $arity(P_i)$ is the number of arguments in the predicate lattice, and 4 is the depth of each argument lattice. This bound on N_{iter} is wildly pessimistic. For most real programs N_{iter} is bounded by a small constant. All the benchmark programs satisfy $N_{iter} \leq 7$ (Chapter 7). However, there exist pathological predicates P_x for which $N_{iter} = \theta(arity(P_x))$. For example, consider the program:

```
main :- a(9,a,_,_,_,_,_,_,_,_,_).

a(0,_,_,_,_,_,_,_,_,_,_).
a(N,A,A,C,D,E,F,G,H,I,J) :- N1 is N-1, a(N1,A,C,D,E,F,G,H,I,J,A).
```

The analyzer requires 10 passes to determine that all arguments of `a/11` are ground and dereferenced upon exit.

To summarize these results, the worst-case and average case total execution times of analysis for programs without non-linearly recursive predicates are:

$$T_{analysis,worst} = O(A \cdot S \cdot D_{total})$$

$$T_{analysis,ave} = O(A \cdot S)$$

If the arity is bounded, then the average execution time of analysis is proportional to the program's size.

For programs that contain non-linearly recursive predicates this result needs to be amended. There is a trade-off between precision and execution time of the analysis. If not enough predicates are traversed then analysis information is lost. If too many predicates are traversed then analysis time becomes too long. Two constraints are used to prune the traversal of the call graph:

(1) The *non-active* constraint. A clause that is in the process of being traversed is called an *active* clause. During recursive calls of predicate_analyze, the algorithm maintains a set of the active clauses and will not traverse an active clause twice.

(2) The *non-exponentiality* constraint. Traverse a predicate (i.e. call predicate_analyze) only if one of two conditions hold: (a) The entry type has changed since the last traversal of the predicate, or (b) At least one of the predicate's clauses is active.

Condition (a) is understandable: it is needed to ensure that an updated type is propagated correctly. The rationale for condition (b) is more subtle. If it did not hold, then the exit types derived by the analysis would be significantly worse because the base case of a recursive predicate may not be reached during the traversal. Running the analyzer both with and without this condition shows this to be true for most programs.

The problem with condition (b) is that it leads to an analysis time that is exponential in the number of non-linearly recursive clauses in a predicate. For many programs this is not serious. However, it occurs often enough that it should be solved. One of the benchmark programs, the nand benchmark, has this problem. A better condition is needed to replace condition (b). It must (1) ensure that the base case of all recursive predicates is reached (for good exit types), and (2) not result in time exponential in the number of non-linearly recursive predicates.

5.3.5. Symbolic execution of a predicate

The heart of the dataflow analysis algorithm is the symbolic execution of a predicate (Figure 4.6). Each clause of the predicate is traversed from left to right. During the traversal the type information is kept in the variable set VS. Symbolic execution of the predicate consists of four steps:

(1) For each clause of the predicate, translate the lattice entry type of the predicate into the variable set VS, and start traversing the clause.

(2) Symbolically execute each goal in the clause and update VS.

(3) At the end of each clause, *back propagation* improves VS by deducing information that only becomes available at the end of the clause. For example, consider the clause:

```
a(X) :- X=[Y|L], b(Y, L).
```

If both Y and L are in the ground set G of VS at the end of the clause then this is also true of X because of the unification `X=[Y|L]`. Back propagation is used to improve the exit types for ground, recursive dereference, and nonvariable types. Measurements show that it is a necessary step to get good exit types.

(4) At the end of the predicate, combine the variable sets of all clauses by intersecting their corresponding components. Convert the result back to the lattice representation and update the exit type for the predicate.

5.3.6. Symbolic execution of a goal

Symbolic execution of a goal is done in three ways, depending on whether the goal is a unification, the goal is defined in the program, or the goal is not defined in the program.

5.3.6.1. Unification goals

Symbolic execution of unification is defined by the function symbolic_unify(VS, X=T) in Figure 4.8, which converts VS = (S, G, N, U, D) into VS′ = (S′, G′, N′, U′, D′). These equations use the utility functions of Table 4.6. For each component of VS, any equation in Figure 4.8 with a true condition can be

Table 4.6 – Utility functions of a term T	
Notation	Definition
$vars(T)$	The set of variables in the term T.
$dups(T)$	The set of variables that occur at least twice in the term T.
$new(T) = vars(T) - S$	The set of all variables in T that have not occurred before.
$old(T) = vars(T) \cap S$	The set of all variables in T that have occurred before.
$deref(T) = vars(T) - (S - U)$	The set of all variables in T that are candidates to be recursively dereferenced. This is the same as $new(T) \cup (vars(T) \cap U)$, i.e. $new(T)$ supplemented with the variables in T that are uninitialized.

$$S' = S \cup vars(X{=}T)$$

$$G' = \begin{cases} G \cup vars(T) & \text{if } X \in G \\ G & \text{otherwise} \end{cases}$$

$$N' = \begin{cases} N \cup G' \cup \{X\} & \text{if nonvar(T) or (var(T) and } T \in N) \\ N \cup G' & \text{otherwise} \end{cases}$$

$$U' = \begin{cases} U \cup new(T) - old(T) - \{X\} - dups(T) & \text{if } (X \notin S \text{ or } X \in U) \\ U - vars(X{=}T) & \text{otherwise} \end{cases}$$

$$D' = \begin{cases} D \cup deref(T) \cup \{X\} & \text{if } (X \notin S \text{ or } X \in U) \text{ and } old(T) \subset (D \cup U) \\ D \cup deref(T) & \text{if } (X \notin S \text{ or } X \in U) \text{ or } X \in (D \cap G) \\ D \cup deref(T) - \{X\} & \text{if } dups(T) = \emptyset \text{ and } X \in D \text{ and } old(T) \subset U \\ D \cap G & \text{otherwise} \end{cases}$$

Figure 4.8 – Symbolic unification VS′ := symbolic_unify(VS, X=T)

used. In practice, if more than one condition is satisfied, an equation giving more information (i.e. the resulting set is larger) is used first. These equations are listed first. For example, the first equation of D′ gives a larger set, so it is preferred over the others. If both X and T are variables, then the algorithm switches X and T is to see if one of the more desirable equations is satisfied before attempting one of the lesser equations.

Table 4.7 – Conditions for the lattice entry type	
Name	Condition
C_{ground}	$vars(X) \subset G$
C_{var}	var(X)
C_{old}	$X \in (dups(P) \cup S - U)$
C_{rderef}	$(vars(X) \cap S) \subset D$
C_{nonvar}	$(X \in N)$

Table 4.8 – Calculation of the lattice entry type					
C_{ground}	C_{var}	C_{old}	C_{rderef}	C_{nonvar}	Lattice value
yes	-	-	yes	-	`ground+rderef`
yes	-	-	no	-	`ground`
no	no	-	yes	-	`nonvar+rderef`
no	no	-	no	-	`nonvar`
no	yes	no	-	-	`uninit`
no	yes	yes	yes	yes	`nonvar+rderef`
no	yes	yes	yes	no	`rderef`
no	yes	yes	no	yes	`nonvar`
no	yes	yes	no	no	`any`

5.3.6.2. Goals defined in the program

Symbolic execution of a goal with a definition is done by symbolically executing the definition. Information is kept about the part of the call graph that has already been traversed, so that analysis will not go into an infinite loop. The function varset_to_lattice(VS, P) is defined by Tables 4.7 and 4.8. For each argument X of P, first determine the values of the five conditions in Table 4.7. Then use these conditions to look up the lattice value for the argument in Table 4.8.

5.3.6.3. Goals not defined in the program

Examples of goals that are not defined in the program being analyzed are built-ins and library predicates. Symbolic execution of these goals is done in two parts. First, entry specialization replaces the goal by a faster entry (section 5.4.1). Second, the type declarations that the programmer has given for the entry are used to continue the analysis. If there are none, then worst-case assumptions are made.

5.3.7. An example of analysis

The following program is interesting because it is mutually recursive:

Table 4.9 – Analysis of an example program

	incl_2(A,B,C)			incl_3(A,B,C,D)			
	A	B	C	A	B	C	D
Start							
entry	impossible	impossible	impossible	impossible	impossible	impossible	impossible
exit	impossible	impossible	impossible	impossible	impossible	impossible	impossible
After pass 1							
entry	*rderef*	*uninit*	*uninit*	*uninit*	*rderef*	*rderef*	*uninit*
exit	*nonvar*	*rderef*	*nonvar*	*rderef*	*any*	*ground*	*nonvar*
After pass 2							
entry	rderef	uninit	uninit	uninit	rderef	rderef	uninit
exit	nonvar	rderef	nonvar	rderef	any	*nonvar*	nonvar
After pass 3							
entry	rderef	uninit	uninit	uninit	rderef	rderef	uninit
exit	nonvar	rderef	nonvar	rderef	any	nonvar	nonvar

```
main :- incl_2([A,B], C, D).

incl_2([], C, [C]).
incl_2([A|E], C, D) :- incl_3(C, A, E, D).

incl_3(C, A, E, [A|D]) :- incl_2(E, C, D).
```

The predicates `incl_2/3` and `incl_3/4` are extracted from a definition of set inclusion. Three analysis passes are necessary to reach the fixpoint (Table 4.9). The entries that have changed with respect to the previous pass are in *italics*. The final types are given in Table 4.10. Most of the correct types are determined after the first pass. A single exit type of `incl_3/4` is corrected in the second pass. This is necessary because the third argument of `incl_3/4` is the same as the first argument of `incl_2/3`.

Table 4.10 – Final results of analysis

incl_2(A,B,C)	
entry type:	rderef(A),uninit(B),uninit(C)
exit type:	nonvar(A),rderef(B),nonvar(C)
incl_3(A,B,C,D)	
entry type:	uninit(A),rderef(B),rderef(C),uninit(D)
exit type:	rderef(A),nonvar(C),nonvar(D)

5.4. Integrating analysis into the compiler

Deriving type information is only the beginning. The analyzer must be integrated into the compiler to take advantage of the type information. The dataflow analysis module itself does four source transformations (Figure 4.9) before passing the result to the next stage, which does determinism extraction. The

Figure 4.9 – Integrating analysis into the compiler

following source transformations are done in the dataflow analysis module:

(1) **Entry specialization**. Determine a fast entry point for each occurrence of a call whose definition is not in the program being analyzed and continue analysis with this entry point.

(2) **Uninitialized register conversion**. Convert uninitialized memory types to uninitialized register types when it results in a speedup. It is done when an argument can be returned in a register without giving up last call optimization.

(3) **Head unraveling**. Unravel the heads of all clauses again in the light of the derived type information. For example, the head `a(A,A,A)` can be unraveled in three different ways, namely

`(a(A,B,C):-A=B,C=B)` or `(a(A,B,C):-A=C,B=C)` or `(a(A,B,C):-B=A,C=A)`. If both A and B are nonvariables and C is unbound, then the first or third possibilities allow the compiler to do argument selection. Unraveling is already done during the conversion to kernel Prolog, but it must be done again after dataflow analysis since the new types may allow it to be done better.

(4) **Type updating**. Supplement the type declarations given by the programmer (if any) by the derived types. All inconsistencies are reported and compilation continues with the corrected types.

The first three of these transformations are discussed in more detail in the following sections.

5.4.1. Entry specialization

During analysis, a fast entry point is determined for each call whose definition is not in the program being analyzed (i.e. each *dangling* call). For example, the call `sort(A,B)` is replaced by the entry point `'$sort *2'(A,B)` if B is uninitialized. Analysis continues with the types of the fast entry point. The program is unchanged until the end of analysis, so the determination of the fast entry point is repeated in each analysis iteration whenever a dangling call is encountered. This mechanism is intended to speed up execution of built-in predicates and library routines, but it is also available to the programmer.

The fast entry point is determined by calculating the type formula corresponding to the variable set VS with the function varset_to_type(VS, G) (Figures 4.6 and 4.7). This type formula is used to traverse the modal entry tree for the goal. The modal entry tree is a data structure that contains a set of entry points and the types that each requires (Appendix A). Entry specialization is also done in the clause compiler, and a detailed example of the use of the modal entry tree is given in Chapter 5 (section 3.4).

5.4.2. Uninitialized register conversion

Often an uninitialized memory type can be converted to an uninitialized register type. The compiler uses four conditions to guide the conversion process. Define a *survive* goal as one that does not alter any temporary registers (except for arguments with uninitialized register type, which are outputs). A goal that potentially alters temporary registers is a *non-survive* goal. The compiler maintains a table of survive goals. With these definitions the four conditions for a predicate P are:

(1) All arguments of P with uninitialized memory type are *candidates* to be converted.

(2) A candidate argument of P must occur at most once in the body of each clause of P. In each clause where it occurs, the argument must be in the last non-survive goal or any survive goal beyond it.

(3) For each clause of P, if the last goal G is a non-survive goal, then the candidate argument of P must be in the same argument position in G as in the head of P. This is necessary to avoid losing the opportunity for last call optimization (LCO): if the argument positions are different then a move instruction is needed between the last call and the return. If the last goal is a survive goal then the condition is unnecessary because it is not as important to retain LCO: a survive goal can never be mutually recursive with the predicate it is part of.

(4) Often the last goal G has candidate arguments that are not candidate arguments of P, so they have to be initialized when returning from G. This has two disadvantages: P loses LCO and P must allocate an environment (which may not exist otherwise). The solution to this problem involves a trade-off: is it better to have LCO in P and fewer uninitialized register arguments in G, or to have no LCO in P and more uninitialized register arguments in G? The compiler recognizes a class of predicates G for which the first is true: Define a *fast* predicate as one whose definition contains only built-ins and survive goals. If G is fast then reduce the set of G's candidate arguments to include only those that are candidate arguments of P.

A transitive closure is done until all four conditions are satisfied. These conditions can be relaxed slightly in several ways. However, even with the existing conditions it is possible to convert about one third of all uninitialized types into uninitialized register types (Chapter 7). The third and fourth conditions are not needed for correctness, but only for execution speed. The third condition ensures that LCO is not lost. The fourth condition speeds up the chat_parser benchmark by 1% and was added after code inspection discovered cases where the use of uninitialized registers slows down execution.

5.4.3. Head unraveling

This transformation repeats the head unraveling transformation (Chapter 3) with the information gained from dataflow analysis. This increases the opportunities for determinism extraction. For example, before analysis the clause:

```
a(X,X,X).
```

is transformed to the following kernel Prolog by making the head unifications explicit (i.e. ‘‘unraveling’’ the head unifications):

```
a(X,Y,Z) :- X=Y, X=Z.
```

If analysis derives that X is unbound and both Y and Z are nonvariable, then the above expansion hides the determinism by twice unifying an unbound variable with a nonvariable. Unraveling the head unifications again after analysis results in:

```
a(X,Y,Z) :- Y=Z, X=Y.
```

In this version, the nonvariables Y and Z are unified together, better exposing the deterministic check that is done, and the unbound variable X is only unified once.

6. Determinism transformation

This section groups four transformations that expose the determinism inherent in a predicate. The purpose of the first three transformations is to make the determinism in the predicate easily visible, so that the fourth transformation, determinism extraction, is as successful as possible in generating case statements. The following transformations are done in order:

(1) **Head-body segmentation**. By separating the heads of clauses from the clause bodies, this reduces the code expansion caused by type enrichment and determinism extraction.

(2) **Type enrichment**. This adds types to predicates for which global analysis is not able to determine the type. The compiler creates different versions of the predicate assuming different input types. This increases code size, but improves performance since often a predicate is deterministic at run-time even though this could not be detected at compile-time.

(3) **Goal reordering**. This reorders goals in a clause to expose more determinism. Tests (such as arithmetic relations) are moved to the left and predicates guaranteed to succeed (such as unifications with uninitialized variables) are moved to the right.

(4) **Determinism extraction with test sets**. This transformation converts the predicate into a nested case statement that makes its determinism explicit, so that a straightforward compilation to BAM code is possible.

6.1. Head-body segmentation

This transformation reduces the code expansion resulting from enrichment and determinism extraction. A predicate is split into a new predicate and a set of clause bodies. The new predicate contains only the goals of the original predicate that are useful for determinism extraction, i.e. all explicit unifications and tests (including type checking and arithmetic comparisons, see Table 4.11) in each clause starting from the head up to the first goal that is not in this category. The rest of the clause bodies are separated from the predicate. This is done to avoid code duplication in determinism extraction, since the same clause may occur in several leaves of the decision tree.

For example, the predicate:

```
p(A,B) :-
        ( var(A), p(A), q(A,C), t(C,D), u(D,B)
        ; A=b, r(A), s(A)
        ).
```

is transformed into:

```
p(A,B) :-
        ( var(A), '$d1'(A,B)
        ; A=b, '$d2'(A)
        ).

'$d1'(A,B) :- p(A), q(A,C), t(C,D), u(D,B).

'$d2'(A) :- r(A), s(A).
```

The new predicate consists only of those parts of the original predicate that are useful for extracting determinism. The determinism extraction is free to create a decision tree from the new predicate without worrying about duplicating the clause bodies at the leaves of the tree. The separated clause bodies are compiled once only, and the BAM transformation stage (Chapter 6) merges them with the decision tree, thus creating a decision graph.

The decision exactly where to split the clause bodies depends on several factors. All goals in the body are classified into two kinds: goals that are useful for extracting determinism (called ‘‘tests’’), and other goals. Then the split follows these rules: (1) Only those tests all of whose variables are in the head become part of the new predicate. (2) If the length of the clause body is less than a given threshold, then all of it becomes part of the new predicate.

Head-body segmentation interacts with type propagation. It often occurs that a clause body is called from several leaves in the decision tree with different types. In that case, it is compiled with a type that is the intersection of the types of the entry points. A complication arises when one of the leaves considers a variable to be uninitialized, and another leaf does not. In that case, the first leaf jumps to a piece of code to initialize the variable, and only afterwards jumps to the clause body.

6.2. Type enrichment

By looking at the type or the value of one or more arguments it is possible to reduce the set of clauses that have to be tried. Often the dataflow analysis is able to derive sufficiently strong types so that a good selection can be done, i.e. a deterministic predicate can be compiled efficiently. However, if the types given for the predicate are weak then a source transformation is done to enrich them. The enrichment consists of adding a test to check at run-time whether an argument is a variable or a nonvariable, and to branch to different copies of the predicate in each case.

The number of arguments that are enriched is given by the argument S of the compiler option `select_limit(S)`. Define a *good* predicate argument as one that is an argument of a unification not known to succeed always, i.e. in the unification neither argument is known to be unbound. An argument is *known* to be of a given type if the type is implied by the type formula. Whether or not enrichment is done is based on the following heuristic:

> **Enrichment Heuristic 1:** If the number of good arguments known to be nonvariable is less than the selection limit S, then choose the lowest numbered good argument that is not known to be nonvariable. Otherwise choose only the first argument, if it is a good argument and it is not known to be nonvariable.

This heuristic is applied recursively on enriched predicates. The default selection limit is always S=1. This default is justified given that (1) a selection limit S=1 already generalizes the first argument selection of the WAM, and (2) compilation time and object code size increase rapidly with the selection limit. Even with S=1, the source transformation occasionally results in some duplicate code being generated. This is removed by the BAM transformation stage. When S=1 the heuristic is simpler:

> **Enrichment Heuristic 2:** If there exist no good arguments known to be nonvariable, then choose the lowest numbered good argument that is not known to be nonvariable. Otherwise choose the first argument, if it is a good argument and it is not known to be nonvariable.

This heuristic generalizes the first-argument selection of the WAM, i.e. it always does at least a first argument selection, but depending on the types that the predicate has (often derived from dataflow analysis) and the predicate itself (what kinds of head unifications it does), the amount of selection can be vastly greater. The heuristic may seem complex, but it is a natural way to make a predicate deterministic.

To show how enrichment works, consider the following predicate without type declarations:

```
a(a).
a(b).
```

It is transformed into:

```
a(A) :-    var(A), a_v(A).  % If A is unbound.
a(A) :- nonvar(A), a_n(A).  % If A is nonvariable.

a_v(a).      a_n(a).
a_v(b).      a_n(b).
```

The predicate `a/1` has been enriched with an unbound type (in `a_v/1`) and with a nonvariable type (in `a_n/1`). As another example, consider the definition without any type declarations:

```
member(X, [X|_]).
member(X, [_|L]).
```

In this case the heuristic picks the second argument, since the first one does no useful unifications. After enrichment, the predicate becomes:

```
member(X, L) :-    var(L), member_v(X, L).
member(X, L) :- nonvar(L), member_n(X, L).

member_v(...) :- (same as original definition)

member_n(...) :- (same as original definition)
```

The two tests `var(L)` and `nonvar(L)` determine which of the two dummy predicates to execute, `member_v/2` or `member_n/2`, and are compiled into a single conditional branch. This is a consequence of the fact that the two tests are mutually exclusive, i.e. if one succeeds then the other fails and vice versa. Both `member_v/2` and `member_n/2` have the same definition as the original predicate, but they have different types for the second argument. The predicate `member_v/2` is compiled assuming the second argument is a variable. The predicate `member_n/2` is compiled assuming the second argument is a nonvariable. Both `member_v/2` and `member_n/2` are also targets of the factoring transformation (section 4).

Type enrichment can introduce a significant increase in code size if it is not handled carefully. In practice, the code size is kept small because: (1) the added types result in significantly smaller code for clause selection in each of the two dummy predicates, (2) before doing enrichment, head-body segmentation separates clause heads from the bodies, so that long clause bodies are not duplicated, and (3) the BAM transformation stage (Chapter 6) removes any remaining duplicate code. In a sense, the definitions are first ‘‘loosened up’’ by head-body segmentation and type enrichment to allow more optimization, and then later ‘‘tightened up.’’

6.3. Goal reordering

This transformation reorders goals in a clause to increase determinism and to reduce the number of superfluous unifications that are done. Goals that are useful in determinism extraction are put as early as possible, and goals that are certain to succeed (such as unifications with uninitialized variables) are put later.

The goals in a clause are classified in four categories: tests (Table 4.11), unifications with unbound variables, unifications with uninitialized variables, and other goals. The goals are reordered so that tests are first (for deterministic selection), followed by unifications with unbound variables (may be affected by aliasing), unifications with uninitialized variables (unaffected by aliasing, so they can safely be put last), and the other goals. The reordering takes into account the fact that unification is commutative, i.e. that unification goals can be permuted in any way without changing the semantics. Some reorderings are better than others because aliasing can worsen the type formula, e.g. if X is unbound (`var(X)`) then after performing the unification `Y=Z` it may not be unbound any more, if it is aliased to Y or Z. The reordering is constrained so that aliasing does not change the operational semantics.

For example, consider a predicate that has an uninitialized argument:

```
:- mode((a(A,B,C) :- uninit(C))).
a(X, Y, Z) :- Z=[X|L], X<Y, ...
```

The transformation knows that the unification `Z=[X|L]` does not instantiate X or L because Z is unbound and unaliased. Therefore the unification is moved back:

```
a(X, Y, Z) :- X<Y, Z=[X|L], ..
```

This has two advantages: (1) the test `X<Y` is brought forward so that it can be used by determinism extraction, and (2) the unification `Z=[X|L]` is not done if the test `X<Y` fails.

This transformation compensates for the popular programming style which puts all unifications in the head and all tests in the body, e.g. people prefer to write:

```
a([X|L], [X|M]) :- var(X), ...
```

instead of:

```
a([X|L], Z) :- var(X), Z=[X|M], ...
```

The first version does not imply anything about the instantiation pattern of the arguments, whereas the transformed version does.

6.4. Determinism extraction with test sets

The majority of predicates written by human programmers are intended to be executed in a deterministic way. These predicates are in effect case statements, yet they are too often compiled in an inefficient manner, by means of shallow backtracking (i.e. saving the machine state, unification with the clause heads, and repeated failure and state restoration). This section describes the general technique used in the compiler to convert shallow backtracking into conditional branching.

Figure 4.10 – Some examples of test sets

6.4.1. Definitions

Predicates are compiled into code which is as deterministic as possible through the concept of the *test set*. Two definitions are useful:

Definition ST: A goal G is a *simple test* with respect to the kernel Prolog predicate P and the type formula F if it satisfies the following conditions:

- G uses only variables that occur in the head of P.
- The implementation of G does not change any state in the execution model, i.e. G does not cause side-effects (I/O or database operations), G does not create choice points, and G does not bind any variables.
- G does not always succeed.

Definition TS: A set of goals is a *test set* with respect to the kernel Prolog predicate P and the type formula F if it satisfies the following conditions:

- Each goal in the set is a simple test according to definition ST.
- With a given set of variable values, at most one goal in the set can succeed.
- A multi-way branch in which each destination corresponds to the success of one of the goals in the set can be implemented in the target architecture.

The tests in the set need not actually be present in the definition of P. Whether or not a given set of goals is a test set depends on the architecture and the predicate P.

6.4.2. Some examples

Most conditional branches in an architecture correspond to a test set. For example, a branch-if-less-than instruction corresponds to the test set `{A<B, A≥B}`. More complex conditions such as an n-way branch implemented by hashing can also be represented as test sets. Figure 4.10 shows some examples of test sets. The second and third examples correspond to WAM instructions.

To illustrate the use of test sets, consider the predicate:

```
max(A, B, C) :- A≤B, C=B.
max(A, B, C) :- A≥B, C=A.
```

which is one way to calculate the maximum of `A` and `B`. It is compiled as:

```
max(A, B, C) :- if A>B then C=A
                else if A<B then C=B
                else (C=B or C=A)
```

(The Prolog notation is simplified for readability.) The predicate is executed completely deterministically if `A>B` or `A<B`; a choice point is created only when `A=B`. The choice point maintains the operational semantics: since both clauses of the original predicate succeed when `A=B`, there are two identical solutions.

---

type testset = testset(testset_name, testset_ident, set of goal);

function determinism(D : disjunction; H : goal; F : formula; *Previous* : set of testset) : disjunction;
var TS : testset;
TS_{set} : set of testset;
begin
  if length(D) $\leq$ 1 **then return** D;
  TS_{set} := find_testsets(D, H, F, *Previous*);
  if $TS_{set} = \emptyset$ **then return** D;
  TS := pick_testset(TS_{set});
  return code_testset(TS, D, H, F, *Previous*)
end;

Figure 4.11 – The determinism extraction algorithm

```
function find_testsets(D : disjunction, H : goal; F : formula; Previous : set of testset) : set of testset;
var   TS : testset;
      TS_set : set of testset;
      i , j : integer;
begin
      /* At this point D = (C_1 ; ... ; C_n) where D has n choices */
      TS_set := ∅;
      for i := 1 to n do begin
            /* C_i = (G_i1 , ... , G_in_i) where C_i has n_i goals */
            for j := 1 to n_i do
                  if G_ij = "!" then exit inner loop
                  else for all testsets TS from table do begin
                        /* TS = testset(Name , Ident , Tests) from Table 4.11 */
                        if TS ∉ Previous and vars(G_ij) ⊂ vars(H) and bindset(G_ij , F) = ∅ then
                              if ∃ T ∈ Tests : (G_ij implies T and not(F implies T)) then
                                    TS_set := TS_set ∪ {TS}
                  end
      end;
      return TS_set
end;
```

Figure 4.12 – Finding all test sets in a predicate

```
function pick_testset(TS_set : set of testset) : testset;
var   TS : testset;
begin
      pick TS ∈ TS_set such that
            ∀ U ∈ TS_set : goodness(TS) ≥ goodness(U);  /* From Equation (G) */
      return TS
end;
```

Figure 4.13 – Picking the best test set

6.4.3. The algorithm

Given a predicate, the compiler proceeds by first finding all test sets that contain tests that are implied by goals in the predicate. This depends on the type formula that is known for the predicate; for example, the unification `X=a` is only a test if X is nonvariable, i.e. if the type formula implies `nonvar(X)`. Then a ‘‘goodness’’ measure is calculated for each test set, and the test set with the largest goodness is used first. The goodness measure is calculated heuristically; in the current implementation each test set is weighted by an architecture-dependent goodness (which depends on how efficiently it is

```
function code_testset(TS : testset; D : disjunction; H : goal; F : formula; Previous : set of testset) : disjunction;
var    T : goal;
       Choices : disjunction;
begin
       Choices := [ ];
       /* At this point TS = testset(Name, Ident, Tests) */
       for all T ∈ Tests do begin
              D_test := subsume(T, D);
              D_test := determinism(D_test, H, update_formula(T, F), Previous ∪ {TS});
              append '$test'(T, D_test) to Choices;
              D := subsume(not(T), D)
       end;
       D := determinism(D, H, F, Previous ∪ {TS});
       append '$else'(D) to Choices;
       return '$case'(Name, Ident, Choices)
end;
```

Figure 4.14 – Converting a disjunction into a case statement

implemented in the architecture) and by the number of possible outcomes (e.g. hashing with a large number of cases is considered better than a two-way branch). The predicate is converted into a case statement using the best test set. The algorithm is called recursively for each arm of the case statement to build a decision tree. This tree is collapsed into a graph by the BAM transformation stage.

Figures 4.11 through 4.14 give a pseudocode definition of this algorithm. The figures define the function determinism(D, H, F, $Previous$) that performs the determinism extraction. Given a predicate written as a head H and a disjunction D, along with the type formula F that is true for that predicate, the function finds as many test sets as possible in the disjunction and converts them into case statements. It returns a new disjunction that contains these case statements. The parameter $Previous$ is used to avoid infinite recursion. It contains all test sets that have already been used to make sure each test set is only used once.

The function find_testsets(D, H, F, $Previous$) returns a list of all test sets in the disjunction (Figure 4.12). It picks a test set if there is a goal in the predicate which implies a test in the test set. It limits the goals to those that do not bind any variables (bindset(G_{ij}, F) = $\emptyset$) and those that use only variables that occur in the head ($vars(G_{ij}) \subset vars(H)$). The function pick_testset(TS_{set}) returns the test set with the greatest measure of goodness, as given by Equation (G) (Figure 4.13). The function code_testset(TS, D,

H, F, $Previous$) converts the disjunction D into a case statement when given a test set TS (Figure 4.14). It uses the functions subsume(F, F_1) and update_formula(F, F_1), which are defined in section 3.

Table 4.11 – Test sets

Name	Example Test	Example BAM translation
equal	`X==Y` (X or Y is simple at run-time)	`equal(X,Y,Lbl)`
equal(atomic,A) (A is an atom)	`X==A`	`equal(X,A,Lbl)`
equal(structure,F/N) (F/N is name/arity)	`'$name_arity'(X,F,N)`	`equal([X],F/N,Lbl)`
hash(atomic)	`X=A` (A is atomic)	`hash(tatm,X,N,Lbl)`
hash(structure)	`X=S` (S is a structure)	`hash(tstr,X,N,Lbl)`
comparison(Class,Kind) (Class $\in$ {eq, lts, gts}) (Kind $\in$ {arith, unify, stand})	`X<Y`	`jump(lts,X,Y,Lbl)`
Type (Type $\in$ AllTypes)	`var(X)`	`test(eq,tvar,X,Lbl)`
switch(Type) (Type $\in$ TagTypes $-$ {var})	`atom(X)`	`switch(tatm,X,L1,L2,L3)`

Table 4.11 lists the test sets currently recognized by the compiler. This includes unification goals, all type checking predicates, and all arithmetic comparisons. For each test set it gives the name, a representative test in the test set (only one is given, although usually there are several others), and the translation of that test into a conditional branch of the BAM instruction set. For the test sets hash(atomic) and hash(structure) the BAM code includes a hash table (not shown) in addition to the hash instruction. The following definitions simplify the table:

TagTypes = {var, atom, structure, cons, negative, nonnegative, float}, i.e. all types that correspond to one tag in the VLSI-BAM architecture.

AllTypes = TagTypes $\cup$ {atomic, integer, simple, compound}, i.e. it includes types that correspond to more than one tag.

The goodness measure for a test set in a predicate is calculated using the following rule:

$$Goodness = 1000 \cdot D + G \tag{G}$$

where D is the number of directions of the test set that occur in the predicate and G is the raw goodness measure of the test set. This rule ensures that the number of useful directions in the testset is most important. The raw goodness is used only when the number of directions is the same. Table 4.12 gives the raw goodness of all test sets in the VLSI-BAM architecture [34], with a brief justification of the ranking. The

Table 4.12 – Raw goodness measure of test sets in the VLSI-BAM

Test set	Rank	Comments
switch(cons) switch(structure)	131	Switch is best because it is fast and it is a three-way branch, so it gives the most information. Switch of compound terms is better than other switches because it makes traversing a recursive term (like a list or a tree) fast.
switch(negative) switch(nonnegative) switch(atom)	130	Switch of atomic terms is worse because it penalizes the case of traversing a recursive term.
switch(integer)	129	Switch of integer is worse because the VLSI-BAM has separate negative and nonnegative (`tpos` and `tneg`) tags, requiring two branches.
var atom cons structure negative nonnegative	120	These test sets are types that correspond directly to tags, and there exist fast two-way branches on tags.
equal	85	This test set requires two instructions—a compare and branch, and also possibly loading its arguments into registers.
equal(atomic,_) comparison(_,_)	80	These test sets each require two instructions—a compare and branch.
integer atomic compound	79	These test sets are types that each correspond to two tags, so they need two tag checks.
equal(structure,_)	60	Equality comparison of a structure's functor & arity needs a memory reference.
simple	50	This test set corresponds to a type that needs five tag checks (four without floating point).
hash(atomic)	41	Hashing is the slowest because it needs to calculate the hash address.
hash(structure)	40	Hashing on a structure is slightly slower than hashing on an atomic term because a memory load is needed to access the main functor of the structure, whereas the atomic term is directly available in the register.

value of the rank is not important; only the relative order is important. Architectures rank the test sets according to how efficiently they are implemented in the architecture. To compile for a different architecture, only the ranking is changed in the compiler. The ranking is modified for other processors by a compiler option. For example, for the MIPS processor, the option `mips` changes the ranking to make the test set `equal(atomic,[])` best, i.e. a comparison with the atom `[]` (nil), because it can be implemented with a single-cycle conditional branch instruction. The MIPS does not have separate tags for negative and nonnegative integers, so the test sets negative and nonnegative are not implemented as efficiently as on the VLSI-BAM. These two test sets have lower ranks.