Determinacy Testing for Nondeterminate Logic Programming Languages

E. Tick — University of Oregon
M. Korsloot — Delft University of Technology

CIS-TR-93-07
April 1993

Abstract

This paper describes an algorithm for the code generation of determinacy testing for nondeterminate flat concurrent logic programming languages. Languages such as Andorra and Pandora require that procedure invocations suspend if there is more than one candidate clause potentially satisfying the goal. The algorithm described has been developed specifically for a variant of flat Pandora based on FGHC, although the concepts are general. We have extended Kliger and Shapiro's decision-graph construction algorithm to compile "don't know" procedures which must suspend for nondeterminate goal invocation. The determinacy test is compiled into a decision graph quite different from those of committed-choice procedures, but we argue that in most cases, the same low space complexity is retained.

This paper will be published in the *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1993.

Printing history: TR-90-18, November 1990.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

1 Introduction

This article describes compilation techniques for the Andorra/Pandora family of parallel logic programming languages [1, 2, 3, 4, 6]. These languages share a common execution model wherein a goal invocation is suspended if it can potentially be satisfied by two or more clauses in a procedure definition. We call these "nondeterminate" goal invocations. If all goals suspend, causing deadlock, then one nondeterminate goal is selected and forced to execute, creating a choicepoint for potential backtracking.

In this article, we limit ourselves specifically to our own variation of "flat" Pandora based on Flat Guarded Horn Clauses (FGHC) [15]. The language we consider has flat guards, no synchronization operator, and implicit synchronization rules, all as in FGHC. However, the compilation techniques we developed are applicable to the entire Andorra/Pandora family.

The techniques described are based on the decision-tree and decision-graph algorithms developed by S. Kliger and E. Shapiro [10, 11]. The algorithm we introduce is for generating the code for a *determinism tester*: the code that checks whether two or more clauses can potentially satisfy a goal invocation. Our tester is a decision graph that is built in quite a different manner than graphs for committed-choice languages, as for example done by S. Taylor [12].

2 Pandora: Definitions and Example

In this section we introduce flat Pandora to clarify our later compilation techniques. Our flat Pandora programs consist of FGHC procedures categorized as either *don't care* or *don't know* procedures [2]. In a nutshell, a flat Pandora program executes like an FGHC program, i.e., concurrent fine-grain processes communicate to solve a problem. In addition, a don't-know goal can *backtrack* through its clauses in an attempt to satisfy its invocation.

At the time of its execution, a don't-care goal is treated as in any committed-choice language. If any clause head and guard can satisfy a goal invocation, then the goal *commits* to that clause and reduces. If the head and guard of two or more clauses can satisfy a goal invocation, then *any* one of the clauses is chosen for commitment. If no clause head and guard can satisfy the goal invocation, but one or more can *suspend*, then the goal suspends. Otherwise if no clause head and guard can succeed or suspend, then the goal fails. Note that unlike a pure committed-choice language where goal failure implies program failure, in Pandora goal failure causes backtracking.

A don't-know goal is executed differently from a don't-care goal. If *only one* clause head and guard can satisfy the goal invocation, then the goal commits to that clause and reduces. However, if the head and guard of two or more clauses can succeed, then the goal suspends. We call this a *nondeterminate goal invocation*, and the test to determine if one or more clauses can commit is called the *determinism test.* If no clause head and guard can satisfy the goal invocation, but one or more can suspend, then the goal suspends as well. Otherwise if no clause head and guard can succeed or suspend, then the goal fails.

In summary, don't-know goals act similar to goals in a committed-choice language with the exception that if more than one clause can commit then the goal suspends and we call this a "nondeterminate goal invocation." Flat Pandora execution proceeds much like that of FGHC, with nondeterminate don't-know goal invocations suspending. At any point in time, the group of suspended goals consists of don't-care and don't-know goals. Unbound logical variables are "hooked" to these suspended goals, thus enabling resumption as in any committed-choice language implementation (e.g., [9]).

For don't-care goals, binding a hooked variable is the only method of resumption. Resumption causes the goal to be reconsidered for execution, although it does not guarantee reduction: the goal may suspend on another variable. For don't-know goals, binding a hooked variable will also cause resumption, but it is not the only method. Certain suspended don't-know goals may not even have associated hooked variables because their procedure definitions are truly nondeterminate. In that case, the goals remain suspended until no executable goals remain, i.e., deadlock ensues.

To break deadlock, any suspended don't-know goal is forced to reduce. The fact that the goal was suspended implies that two or more clauses can commit to this goal invocation. Forced reduction must choose a clause to commit, after creating a choicepoint for potential backtracking. The choicepoint will direct execution to alternative clauses. Any clause with a non-failing head and guard can be reduced (as in Prolog). There is no distinction between input and output variables at this stage: the goal is *unified* with the clause heads until one is satisfied. Unlike don't-care goal reduction, output bindings may be performed during head and guard unifications.[1] However, certain guards cannot be forced to reduce with unbound inputs, e.g., `X>3` cannot be evaluated if `X` is unbound. These guards must suspend and another clause would be chosen. It could be the case that the don't-know goal chosen to break the deadlock *cannot* reduce because of such tests. If this occurs, another goal must be chosen.

Given an "incorrect" program, all suspended goals may be don't-care in which case deadlock is fatal. It can also be the case that although don't-know goals exist, none of them can be forced to reduce because of guards that require bound variables. This is

[1] As for all bindings, trailing must be performed if the variable to be bound is older than the current choicepoint.

another form of fatal deadlock. Given a "correct" program, forcing reduction of a don't-know goal may issue new don't-care goals and/or create bindings that resume old don't-care goals. However, some don't-know goals may generate only nondeterminate don't-know goal invocations and/or create bindings that do not resume old don't-care goals. In this case, deadlock remains and another goal must be chosen for forced execution. In general, it is hoped and expected that the execution periods of determinate reduction of don't-care goals are much longer than the periods required to break deadlocks.

Choicepoints leading to multiple, independent OR-branches of the execution tree can potentially be searched in parallel [4], but this issue is orthogonal to the article and will not be discussed.

To illustrate the flat Pandora procedural semantics, a small example is presented. Consider the following code:

```
a(1,1,1) :- true : true.
a(2,1,1) :- true : true.
a(2,2,1) :- true : true.
a(2,2,2) :- true : true.

b(1,A) :- true | A=yes.
b(2,A) :- true | A=no.
```

The procedures utilize Kernel Andorra Prolog [7] syntax: the wait operator, ':' signifies don't-know, whereas the standard commit operator, '|' signifies don't-care. Suppose we make the following query:

```
?- a(X,Y,Z), b(Y,A), Z=2.
```

Assuming that the goals are evaluated in their sequential order, the sequence of actions executed is: a/3 is found to be nondeterminate and suspends, b/2 suspends, Z is bound to 2, a/3 resumes and is found to be determinate (clause 4), X and Y are bound, b/2 is resumed, and A is bound to no. In contrast, consider the following query:

```
?- a(X,Y,Z), b(Y,no), Z=1, X=2.
```

One possible sequence of actions executed is: a/3 is found to be nondeterminate and suspends, b/2 suspends, Z is bound, a/3 resumes, found to be nondeterminate and suspends again, X is bound, a/3 resumes, still found to be nondeterminate and suspends a third time. Now deadlock ensues so a/3 is *forced to reduce*, Y is bound to 1, b/2 is resumed and *fails*, backtracking retries the execution of a/3, Y is bound to 2, b/2 is resumed and succeeds. These two examples sufficiently illustrate the execution mechanisms of flat Pandora to understand the rest of this article.

3 Decision Graphs: Background

The don't-care procedures in flat Pandora are compiled in a manner similar to the decision-graph algorithm given by Kliger and Shapiro for FCP [11]. Because we are restricting ourselves to FGHC, our notation can be made simpler than that of the FCP algorithm as we have no tell guards. We review the terminology and algorithm here as a foundation for our method of compiling don't-know procedures.

A guarded Horn clause is of the form:

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n.$$

where m and n are zero or positive integers. H is the clause head, G_i is a guard goal,[2] and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). The first step in the compilation process is to translate source clauses into *canonical form.*

> *Definition*: A *simple term* is either a constant, a variable, or a compound term in which the arguments are pairwise different variables. A *complex term* is a compound term in which there exists either a non-variable argument or two arguments with identical names. □

For example, $f(X,Y)$ is a simple term and $g(h(X))$ is a complex term.

> *Definition*: A complex term is *flattened* into a pair (F,S), where F is a simple term and S is a *constraint set.* To flatten a complex term the following rules are applied with S initially empty. Each of k instances of a shared variable X is replaced by a unique variable $\{Z_{i+j} \mid 1 \leq i \leq k\}$ and $S := S \cup \{Z_{j+1} = Z_{j+2}, Z_{j+1} = Z_{j+3}, \ldots\}$, for C($k$,2) pairs.[3] A constant argument α is replaced by a unique variable Z_j, and $S := \{Z_j = \alpha\} \cup S$. A complex argument is flattened into (F', S') and replaced by a unique variable Z_j, and $S := \{Z_j = F'\} \cup S' \cup S$. □

Note that all pairs of instances of a shared variable are explicitly generated to permit rapid failure if any pair fails to unify. This definition can be simplified for don't-care procedures by generating only $k-1$ constraints for k instances of a shared variable. All these constraints link the new variables to the original.

> *Definition*: A clause $C =$ "$H :- G \mid B.$" has a normalized form $C' =$ "$H' :- G' \mid B.$", where H' and G' are computed as follows: the head H

[2] more precisely, an "ask" guard.

[3] $C(x,y)$ is the combinations function. Index j is chosen to produce unique variables.

is flattened into the pair (H', S) and a goal G_i is flattened into the pair (G'_i, S_i). Then the guard $G' := \ S \cup G_1 \cup ... \cup G_m \cup S_1 \cup ... \cup S_m$, for m guards.[4] □

For example, the clause

$$f([X|W]) :- X = g(a(Y), b(Y), c(Y)) \mid ...$$

is normalized to:

$$f(Z_1) :- Z_1 = [X|W],\ X = g(Z_2, Z_3, Z_4),\ Z_2 = a(Z_5),$$
$$Z_3 = b(Z_6),\ Z_4 = c(Z_7),\ Z_5 = Z_6,\ Z_5 = Z_7,\ Z_6 = Z_7 \ \mid ...$$

Normalization is needed to simplify a clause into a trivial head and an extended guard containing the constituent constraints. This form facilitates code generation of triples or quads. Normalization, however, is not enough: we need to rename variables among the clauses belonging to the same procedure such that variables corresponding to the same depth within the same procedure argument have the same name. This characteristic is necessary for indexing purposes, as is shown later.

Definition: A simple term $H = p(X_1, X_2, ..., X_n)$ is *expanded*, with respect to a set of constraints G, into a tree, $tree(H)$, with n branches labeled $1, \ldots, n$. For a variable X_i and term T that appear in a constraint $g = \{X_i = T\} \in G$ or $g = \{T = X_i\} \in G$, the subtree at root i is T, expanded with respect to $G \setminus \{g\}$. Otherwise the subtree at root i is the leaf X_i. □

Definition: A normalized clause $C =$ "$H\ :-\ G \mid B$." is *renamed into canonical form* by expanding H with respect to G into $tree(H)$. Each variable X in C is replaced by Z_p where p is the *label sequence* from the root of $tree(H)$ to the vertex corresponding to X. □

Using this sequence gives a complete ordering of all variables and retains identical names for corresponding variables among clauses. For instance, $f(g(X, Y))$ has normalized form $f(A) :- A = g(B, C)$ and canonical form $f(Z_1) :- Z_1 = g(Z_{1,1}, Z_{1,2})$.

Definition: A canonical-form procedure consists of canonical-form clauses, each of the form $< i, G >$ for clause number i with guard G. The head is not needed because it has been flattened away. The head arguments are always named $Z_1, Z_2, ..., Z_n$ for an arity-n procedure. The body is not listed because the algorithms we discuss in this article do not deal with body compilation. □

[4] Certain builtin predicates, such as =/2, </2, etc., are flattened in a slightly different manner than described above. For example, $X = a$ is considered already flat.

From this point on in the article, all references to "procedures" and "clauses" implicitly assume canonical forms. Both don't-care and don't-know procedures are syntactically identical and are converted into canonical form. The goals in guard G consist of builtin predicates such as =/2, '=/='/2, and >/2. We call these *constraints* or *tests*. For efficiency, we can represent G as a bit-vector corresponding to inclusion in the union of all guards of a procedure. This makes set operations on guards very fast.

> ***Definition***: Variables within a clause are partially ordered on their label sequences: $Z_{i,j,...,k} \geq_\psi Z_{i,j,...,k,l,m,...}$. In words, a variable is ψ-greater than or equal to another variable if their label sequences share the same prefix and the latter sequence is longer or equal. □

For example, $Z_1 \geq_\psi Z_{1,2}$, whereas Z_2 and $Z_{1,2}$ have no ordering, nor have $Z_{1,1}$ and $Z_{1,2}$.

> ***Definition***: The constraints within a canonical-form clause form a relation with respect to $\geq_\psi$, defined as follows:
>
> $$g(Z_a, ..., Z_b) \geq_\psi g(Z_c, ..., Z_d) \ \ \underline{\textit{iff}} \ \ \{\exists\, i \in \{a, ..., b\}, j \in \{c, ..., d\} \ \ Z_i \geq_\psi Z_j\}$$
>
> □

For example, $(Z_1 = Z_2) \geq_\psi (Z_{1,2} > 3)$, whereas both $(Z_{1,2} = Z_2) \geq_\psi (Z_1 = Z_{2,1})$ and $(Z_1 = Z_{2,1}) \geq_\psi (Z_{1,2} = Z_2)$. This ordering is necessary to characterize related variables (see $\Re_{us}(C, \Gamma)$ in Section 5).

> ***Definition***: $suc(Z_p)$ is defined to be the set of all variables within a procedure with label sequence p extended by one index. □

For example, suppose a procedure has the variables $\{Z_1, Z_2, Z_{1,1}, Z_{1,2}, Z_{2,1}, Z_{2,1,1}\}$. Then $suc(Z_1) = \{Z_{1,1}, Z_{1,2}\}$ and $suc(Z_2) = \{Z_{2,1}\}$.

In the generation of a decision graph, each node corresponds to a test of some constraint in the procedure definition, e.g., $Z_1 > 3$. The branches emanating from the node correspond to test results, e.g., yes ($Z_1 > 3$), no ($Z_1 \leq 3$), or undecided (Z_1 is unbound). The children nodes of these branches contain subgraphs built from a subset of the clauses in the parent node. In the following we define the concept of a *residual* which is a function returning all those clauses satisfying a given set of properties. In general, a child node is built from the clauses specified in the residual of the parent (with respect to the condition on the corresponding branch, or the conditions on all sibling branches).

These residual definitions are similar to those of Kliger, who formulated them in terms of ***pairwise consistency*** of goals. One may think of this residual as a don't-care residual $\Re_{dc}(C,g)$ to be consistent with later terminology. Our different derivation requires that we define two additional residuals, $\Re_{no}$ and $\Re_{neither}$, given at the end of this section. The use of these definitions will become clear later in this section.

> *Definition*: Given that A is the set of guard goals for clause i, then the ***residual*** of a clause $C = <i, A>$ with respect to a constraint g is denoted as $\Re(C,g)$.
>
> $$\Re(C,g) = \begin{cases} \{<i, A \setminus G>\} & G = \{g' \in A \mid \tilde{\forall}.\ g \Longleftrightarrow g'\} \neq \emptyset \\ \emptyset & otherwise \end{cases}$$
>
> □

The strict use of if-and-only-if, $g \Longleftrightarrow g'$, is meant to keep the decision graph space-linear in the number of clauses. In other words, the residual filters all and only those clauses satisfying a decision test made in the execution path. The filter is strong in the sense that any clause not "exactly" satisfying the test is not included in the residual. For example, $(Z_1 > 3) \Longrightarrow (Z_1 > 2)$; however, the opposite does not hold. Therefore, $(Z_1 > 3) \not\Longleftrightarrow (Z_1 > 2)$, and a clause containing the latter constraint will not be contained in the residual with respect to the former constraint. We shall see that although this strict definition keeps the graph space-linear, it may not be optimal in the execution path length.

The following definition is made to filter away all clauses that are inconsistent with the inverses of the decision tests made in sibling branches.

> *Definition*: The ***otherwise-residual*** of a clause $C = <i, A>$ with respect to a constraint set Γ is denoted as $\Re_o(C,\Gamma)$.
>
> $$\Re_o(C,\Gamma) = \begin{cases} \emptyset & \{g' \in A \mid \exists g \in \Gamma.\ \neg\tilde{\forall}.\ g' \Longrightarrow g\} = \emptyset \\ \{<i, A>\} & otherwise \end{cases}$$
>
> □

The otherwise residual is used with Γ containing the decision tests of sibling branches. We chose to define the otherwise residual by specifying which clauses are *not* included. A clause is not included if it is inconsistent with the inverse of Γ (i.e., the inverse of each constraint in Γ). More intuitively, if there exists a constraint $g' \in A$ and $g \in \Gamma$, such that $\neg g$ is inconsistent with g', then A is not included in $\Re_o$. Thus if $\neg(\neg g \wedge g') \equiv (g \vee \neg g') \equiv (g' \Longrightarrow g)$, then A is not included, as is specified in the formal definition above. $\Re_o$ is simply the "remainder" of all clauses not chosen by sibling residuals.

As an example of $\Re_o$, suppose $\Gamma = \{Z_1 > 3, Z_1 \leq 3\}$. A clause containing $Z_1 > 0$ will be included in $\Re_o$ (and by definition, it could not be included in any sibling residual). However, a clause containing $Z_1 > 3$, or $Z_1 > 4$, will not be included.

In addition to the definitions above, we also introduce two definitions, $\Re_{no}$ and $\Re_{neither}$ to optimize the code generated for "ask" tests, as described in the next section.

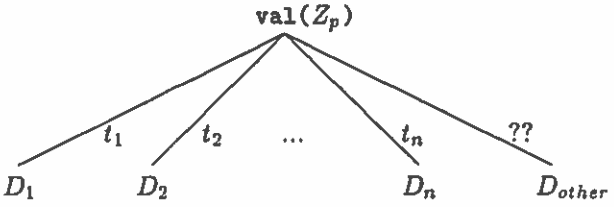
> *Definition*: Given that A is the set of guard goals for clause i, then the ***no-residual*** of a clause $C = <i, A>$ with respect to a constraint g is denoted as $\Re_{no}(C, g)$.
>
> $$\Re_{no}(C,g) = \begin{cases} \{<i, A \setminus G>\} & G = \{g' \in A \mid \tilde{\forall}.\ \neg g \Longrightarrow g'\} \neq \emptyset \\ \emptyset & otherwise \end{cases}$$
>
> □

> *Definition*: The ***neither-residual*** of a clause $C = <i, A>$ with respect to a constraint set Γ is denoted as $\Re_{neither}(C, \Gamma)$.
>
> $$\Re_{neither}(C,\Gamma) = \begin{cases} \emptyset & \{g' \in A \mid \exists g \in \Gamma.\ (\tilde{\forall}.\ g \Longleftrightarrow g')\ \vee \\ & \qquad\qquad\qquad\qquad (\tilde{\forall}.\ \neg g \Longrightarrow g')\} \neq \emptyset \\ \{<i, A>\} & otherwise \end{cases}$$
>
> □

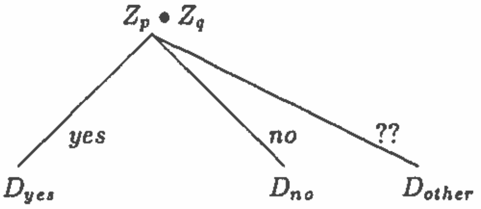
Although we require these extra definitions, they are more intuitive to us (than Kliger's original definition), and have facilitated our implementation. Primarily, these definitions engender "fast failure," wherein the invocation terminates in failure immediately, rather than having control propagate through a series of continuations, as in Kliger's method. Fast failure is more important in Pandora than in committed-choice languages because failure is more frequent in the former than in the latter.

> *Definition*: The residuals of a procedure ($\Re(P, g)$, $\Re_{no}(P, g)$, $\Re_o(P, \Gamma)$, and $\Re_{neither}(P, \Gamma)$) are the union of all residuals corresponding to the clauses in P. More formally:
>
> $$\Re(P, g) = \bigcup_{C \in P} \Re(C, g)$$
>
> □

```
decision-graph($P, C, V$)
        if ($P = \emptyset$) then return(go(C));
        if ($<i, \emptyset> \in P$) then
                — any clause with empty guard can commit
                return(commit(i));
        else
                — no clause(s) with empty guards
                — first select test $\Gamma$ and collect constraints
                $\Gamma :=$ index($P, V$);
                if ($\Gamma = \{Z_p = t_1, ..., Z_p = t_n\}$) then
                        — return the following code tree:
```



```
                        $V' := suc(Z_p) \cup V$;
                        $\{ \forall i \in 1 \ldots n \mid D_i :=$ decision-graph($\Re(P, Z_p = t_i), D_{other}, V'$) $\}$
                        $D_{other} :=$ decision-graph($\Re_o(P, \Gamma)$,C,V);
                else
                        if ($\Gamma = \{Z_p \bullet Z_q, \neg(Z_p \bullet Z_q)\}$) then
                                — return the following code tree:
```



```
                                $D_{yes} :=$ decision-graph($\Re(P, Z_p \bullet Z_q), D_{other}, V$);
                                $D_{no} :=$ decision-graph($\Re_{no}(P, \neg(Z_p \bullet Z_q)), D_{other}, V$);
                                $D_{other} :=$ decision-graph($\Re_{neither}(P, \Gamma), C, V$);
```

Figure 1: Decision-Graph Construction Algorithm (Based on Kliger).

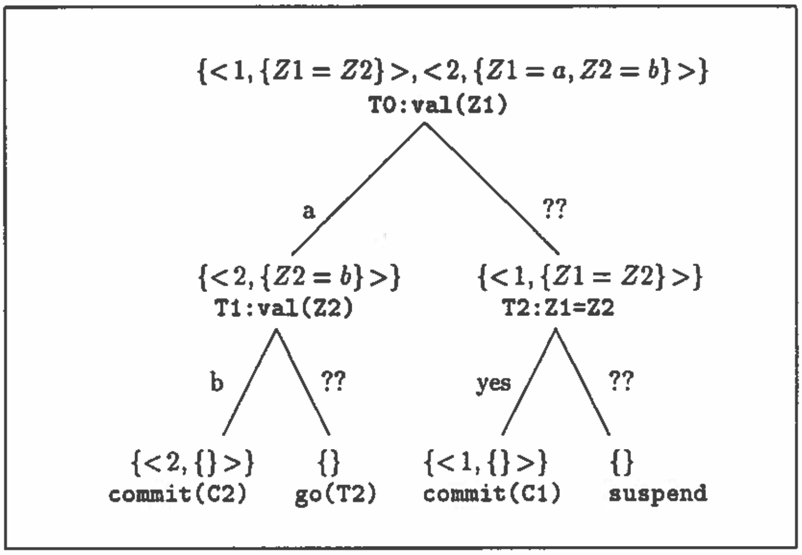


Figure 2: The Decision Graph for `f/2`.

4 Committed-Choice Compilation: Kliger's Method

Kliger's algorithm [11] for compilation of don't-care (committed-choice) procedures is reviewed in Figure 1. The algorithm we discuss is slightly modified for FGHC execution, i.e., no tell guards. The code-generation function *decision-graph(P, C, V)* is passed the initial residual P (representing the procedure to be compiled), the continuation C, and the visible variable set V.[5] The initial continuation is a **suspend** instruction that will be explained later. The function returns an abstract code graph that can easily be flattened into a linear code sequence. To illustrate the algorithm, consider the following don't-care procedure:[6]

```
f(X,X) :- true | C1.
f(a,b) :- true | C2.
```

The final code graph produced is illustrated in Figure 2. Each node is labeled with its residual P, followed by a set of constraints Γ, as returned by the indexer (see Section 5.1). Each leaf is either **suspend**, `fail`, **go**, or a code segment. A branch labeled '??' is an otherwise-residual branch, i.e., the residual at its leaf is computed as $\Re_o(P, \Gamma)$. If Γ represents a switch test $\texttt{val}(Z_p)$ then a branch is labeled with a ground value α and its leaf computed as $\Re(P, Z_p = \alpha)$. If Γ represents some "ask" test $Z_p \bullet Z_q$ then branches are labeled 'yes' and 'no' with leaves computed as $\Re(P, Z_p \bullet Z_q)$ and

[5] When the algorithm is called the first time, the set of visible variables for an arity-n function is $\{Z_1, Z_2, \ldots, Z_n\}$. On later calls, V will be extended to contain those variables that have been exposed due to the partial ordering (as defined by the *suc*() function of the previous section).

[6] As stated before, clause bodies are not relevant to the compilation methods described in this article.

$\Re_{no}(P, \neg(Z_p \bullet Z_q))$ respectively. $\bullet$ represents a built-in guard predicate.

If the residual contains a clause with no remaining constraints, tree generation terminates with code committing to the clause. For an empty residual, tree generation terminates with a control transfer to the current continuation.[7] Control follows the otherwise branches whenever a test fails or cannot be evaluated because of unbound variables. Note that failure will occur at the **suspend** instruction if the suspension stack is empty, as explained in Section 5.3. For instance, if Z_1 and Z_2 are bound to different values, then the test at T2 will fail, taking the otherwise branch to the **suspend**, which will fail because of the empty suspension stack.

The code space required by this example procedure is grossly estimated as three test nodes and four leaves. The actual code generated for each node will of course differ, but in general nodes require more instructions than simple leaves. We assume that **suspend** and `fail` are traps, i.e., their code bodies are not expanded in-line.

5 Determinism Testing

This section introduces an algorithm to generate decision graphs for don't-know procedures in a nondeterminate concurrent logic programming language. Don't-know procedures do not contain guards at the source level, but guards may be subsequently introduced in the normalized canonical form.

Specifically we generate an abstract code graph, for flat Pandora, that can easily be flattened into a linear sequence of triples suitable for code generation. Qualitatively, the algorithm presented has space complexity comparable to Kliger's method of decision *trees* [10], reduced by sharing portions of the tree. The expected path length through the code obviously depends on how procedure arguments are dynamically bound, but it also depends on the sophistication of the indexer at choosing critical constraints near the root, as in Kliger's method (one indexer algorithm is described in Section 5.1). Similarly, the completeness of the algorithm, i.e., the percentage of determinate invocations that commit immediately, is dependent on the strength of the guard inference mechanism. For the simplest inference mechanism over arbitrarily complex unifications, which is the common case, the code generated is complete. Only in the case of complex interacting constraints in the (run-time) query, detection of determinism cannot be guaranteed, and suspension instead of failure may occur. More discussion about completeness over other domains is given in Section 5.2.

Definition: Given that A is the set of guard goals for clause i, then the

[7] In this example, the bottom right subtree is optimized by combining the 'no' branch (terminating in a transfer to the suspend continuation) with the otherwise branch.

don't-know residual of a clause $C = <i, A>$ with respect to a constraint g is denoted as $\Re_{dk}(C, g)$. Let $G = \{g' \in A \mid \tilde{\forall}.\ g \Longrightarrow g'\}$ and $G' = \{g' \in A \mid \tilde{\forall}.\ g \Longrightarrow \neg g'\}$.

$$\Re_{dk}(C, g) = \begin{cases} \emptyset & G' \neq \emptyset \\ \{<i, A \setminus G>\} & \mathit{otherwise} \end{cases}$$

□

This definition means that only if g disproves the clause, will the residual be empty. Otherwise the clause is retained, even if it is not implied by g. The essence of this inclusion is the construction of a full decision tree rather than a decision graph. However, with the code-sharing optimizations described later, effectively a graph is built.

Definition: The *unbound switch residual* of a clause $C = < i, A >$ with respect to a constraint set Γ is denoted as $\Re_{us}(C, \Gamma)$. Let $G = \{g' \in A \mid \exists g \in \Gamma.\ g >_{\psi} g'\}$.

$$\Re_{us}(C, \Gamma) \quad = \quad \{<i, A \setminus G>\}$$

□

This definition means that the original clause C is retained except for those guards g' that test variables dependent on the unbound variables in the constraints in Γ. For example, $\Re_{us}(<1, \{Z_1 = g(Z_{1,1}, Z_{1,2}),\ Z_{1,2} = h,\ Z_2 = 0\}>, \{Z_1 = g/2\}) = <1, \{Z_2 = 0\}>$.

Definition: The *unbound ask residual* of a clause $C = <i, A>$ with respect to a constraint set Γ is denoted as $\Re_{ua}(C, \Gamma)$. Let $G = \{g' \in A \mid \exists g \in \Gamma.\ \tilde{\forall}.\ g >_{\psi} g' \ \wedge\ g \Longleftrightarrow g'\}$.

$$\Re_{ua}(C, \Gamma) \quad = \quad \{<i, A \setminus G>\}$$

□

This definition, although not in its strongest form, maximizes code sharing for an unbound ask branch. This sharing is critical to keep code size low, but may cause superfluous tests of variables which are known to be unbound. For example, $\Re_{ua}(<1, \{Z_1 > 0,\ Z_1 < 10\}>, \{Z_1 > 0,\ Z_1 \leq 0\}) = <1, \{Z_1 < 10\}>$. Here it is clear that Z_1 is unbound, yet a constraint on Z_1 remains in the residual.

Both previous unbound residual definitions used $>_{\psi}$ rather than $\geq_{\psi}$. The former condition is looser and thus encourages more sharing by retaining more constraints in the unbound residual. This condition is sufficiently correct because variables ψ-equal

```
decision-graph($P$, $Table$, $V$)
    if ($P = \emptyset$) then return(fail);                                    (0)
    if ($P = \{<i, G>\}$) then                                                 (1)
        — we can generate code for a single clause
        if ($\exists$ [leaf, $P$, T] $\in$ $Table$) then
            return(go(T));                                                     (2)
        else  $Table$ := [leaf, $P$, Label] $\cup$ $Table$;
            return(Label:execute(i,G));                                        (3)
    else  — multiple clauses
        — first select test $\Gamma$ and collect constraints
        $\Gamma$ := index($P$, $V$);
        if ($\Gamma = \emptyset$) then return(suspend);                        (4)
        if ($\exists$ [$\Gamma'$, $P'$, T] $\in$ $Table$ | $P \supseteq P'$ and $\Gamma \supseteq \Gamma'$) then    (5)
            — matching residual table entry so code sharing possible
            if ($P = P'$) then
                return(go(T));                                                 (6)
            else  — not exact match, so partial code sharing
                $Table$ := [$\Gamma$, $P$, Label] $\cup$ $Table$;
                return(share-node($P$, $P'$, Label, $\Gamma$, $Table$, $V$));  (7)
        else  — miss in residual table, so no code sharing
            $Table$ := [$\Gamma$, $P$, Label] $\cup$ $Table$;
            return(generate-node($P$, $\Gamma$, $Table$, $V$));                (8)
```

Figure 3: Decision-Graph Algorithm for Don't-Know Procedures.

to the unbound variable can always be accessed.[8] Since every variable is ψ-equal to itself, this implies that tests may be shared for which we know, *a priori*, the unbound outcome. This happens occasionally in the benchmarks we examined, but we think the increased cost in execution time is justified by the code space reduction engendered by avoiding strict filtering with $\geq_\psi$.

We now describe the decision-graph construction algorithm for don't-know procedure determinacy testing. A correctness proof for this algorithm is given in Appendix A. The algorithm (outlined in Figures 3, 4, and 5) has three arguments: the initial residual P (representing the procedure to be compiled), a pointer to a global residual table *Table*, which is initially empty, and the visible variables V. Unlike Kliger's algorithm, a continuation is no longer necessary. Code-sharing optimization exploits the depth-first generation of code with the residual table. This table is indexed by either a guard test, such as Z1=Z2, or a special key, leaf, for those entries which have a residual containing a single clause.

When the algorithm is down to a leaf (only a single clause is left in the residual,

[8] Variables $<_\psi$ an unbound variable cannot be accessed because, they would be subterms of that unbound variable.

(1)), the residual table is tested for a matching entry, using `leaf` as the search key. If a match is found (2), we can either generate a `go(T)` instruction or an `execute(i,G)` instruction.[9] For code-size optimization, the go is better because the `execute` instruction includes code for checking the suspension stack and possibly forcing the bindings specified in G. Returning a `go(T)` instruction also maximally shares code. If no match is found, an `execute(i,G)` instruction is returned (3).

If multiple clauses are left in the residual, the indexer is invoked to select a test Γ from P to index on. We first check if the clauses all have empty guards. This situation is detected by the indexer returning $\Gamma = \emptyset$ and a `suspend` instruction is returned (4).

To check if code sharing is possible, the residual table is accessed with Γ, returning a set of residuals corresponding to nodes in the tree that have the same test (5). The residual P is compared with each candidate P' from the table until one is found such that $P \supseteq P'$.[10] If no such entry exists, then code cannot be shared and the function `generate-node()` is invoked (8). If the stronger condition of equality exists, then the entire node can be shared with a simple control transfer and a `go(T)` instruction is returned (6). In the general case, part of the new node must be built with an otherwise-continuation transferring control to the shared node. The new node is then constructed by invoking the function `share-node()` (7).

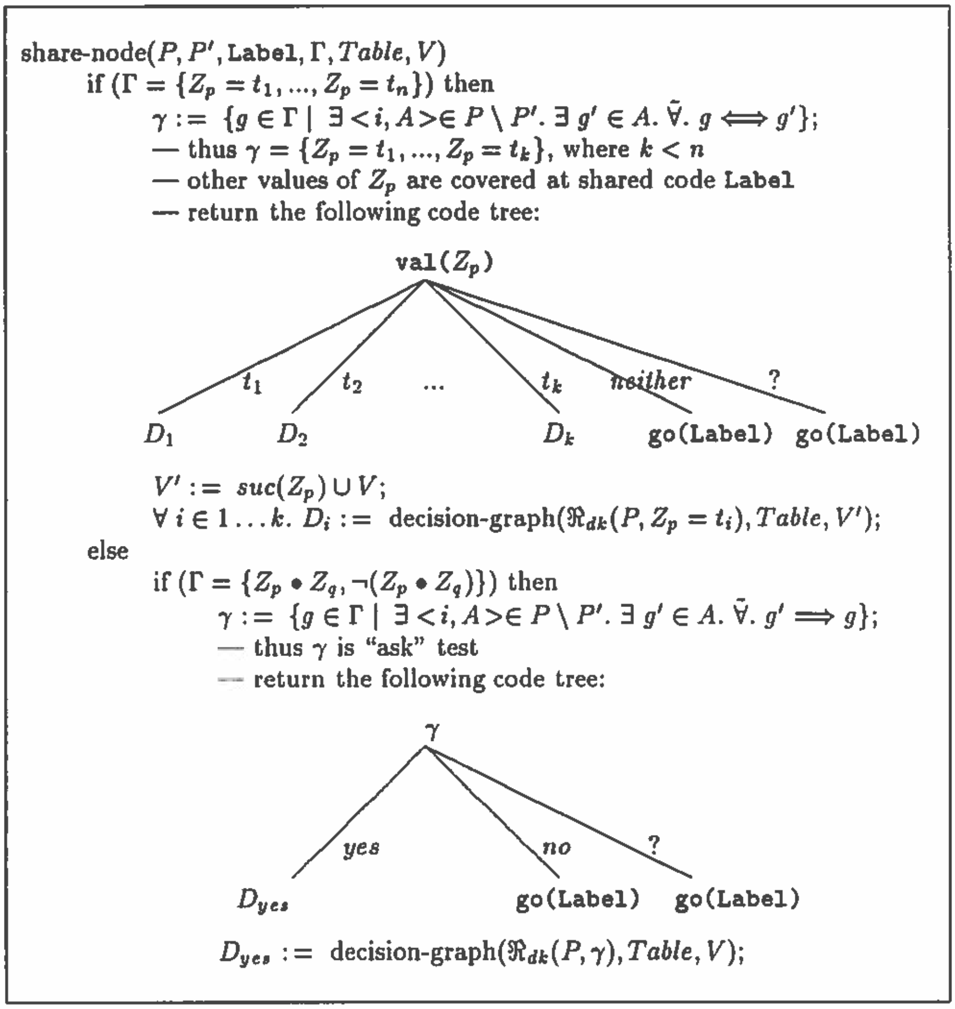
Figures 4 and 5 show the code generation for sharing and non-sharing cases respectively.[11] The generation of a new node, as described in Figure 5, is comparable to Kliger's algorithm, except his otherwise-residual is now split into a neither- and an unbound-residual. In the sharing case (Figure 4), code is generated for $P \setminus P'$, where P' is the clause set for which code has already been generated at `Label`. From the original test Γ, only that part γ is used, which is new, and not yet covered by the code at `Label`.

As an example of this algorithm, consider the previous `f/2` procedure, now declared as don't-know. Its code tree is given in Figure 6 (without collapsing superfluous branches). The residuals are written above each node. Branches labeled '?' are unbound-residual branches and branches labeled with 'neither' are otherwise-residual branches. Control follows the '?' branches whenever a test cannot be evaluated because of unbound variables. Control follows the neither branches whenever a test fails (for example, when a variable is bound, but none of the tests succeed). Control transfers

[9] The exact semantics of `execute(i,G)` is given in Section 5.3. For now it suffices to say that we commit to clause `i`.

[10] Given $P = \{C_{1'}, C_{2'}, \ldots, C_{k'}\}$ and $P' = \{C_1, C_2, \ldots, C_k\}$, $P \supseteq P'$ is defined as $(\forall C_i \in P'.\ C_i \in P)$. Then, $P \setminus P'$ is defined as $\{C_i \in P \mid C_i \notin P'\}$.

[11] A later code-generation phase may choose to do graph rewriting to remove superfluous branches. The condensation techniques are straightforward.



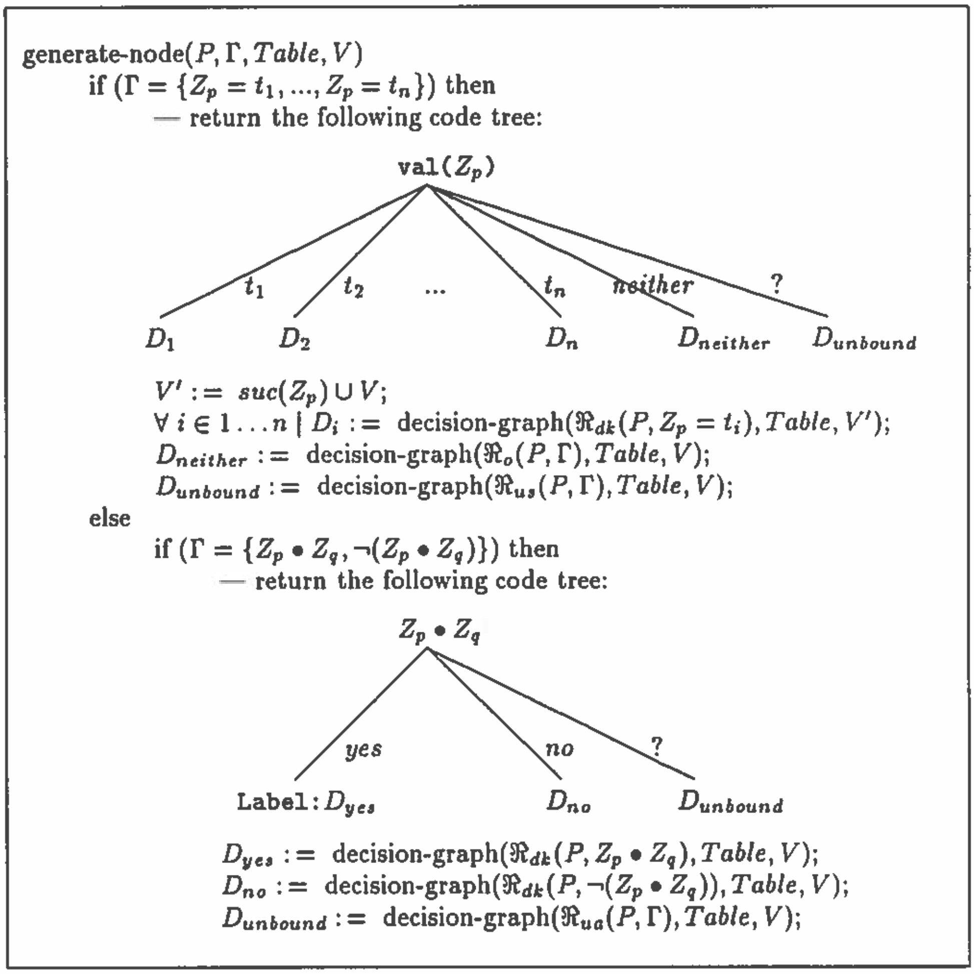
```
share-node($P, P'$, Label, $\Gamma$, $Table$, $V$)
    if ($\Gamma = \{Z_p = t_1, ..., Z_p = t_n\}$) then
        $\gamma := \{g \in \Gamma \mid \exists <i, A> \in P \setminus P'. \exists g' \in A. \tilde{\forall}. g \Longleftrightarrow g'\}$;
        — thus $\gamma = \{Z_p = t_1, ..., Z_p = t_k\}$, where $k < n$
        — other values of $Z_p$ are covered at shared code Label
        — return the following code tree:
        $V' := suc(Z_p) \cup V$;
        $\forall i \in 1 \ldots k.\ D_i :=$ decision-graph($\Re_{dk}(P, Z_p = t_i), Table, V'$);
    else
        if ($\Gamma = \{Z_p \bullet Z_q, \neg(Z_p \bullet Z_q)\}$) then
            $\gamma := \{g \in \Gamma \mid \exists <i, A> \in P \setminus P'. \exists g' \in A. \tilde{\forall}. g' \Longrightarrow g\}$;
            — thus $\gamma$ is "ask" test
            — return the following code tree:
            $D_{yes} :=$ decision-graph($\Re_{dk}(P, \gamma), Table, V$);
```

Figure 4: Shared-Node Generation for Don't-Know Procedure.

with go represent code sharing. In this simple example, entire subtrees can be shared because the corresponding residuals are identical. This code is *complete* in the sense that it is guaranteed to detect determinacy.

5.1 Indexing

The decision-graph generation mechanism, as defined in Kliger's work and extended here, hinges on the indexer selecting a test either in the form of a "switch on value," or builtin predicate, e.g., >/2. Apart from the strength of its inference mechanism, it is important that the indexer is fully determinate (allowing efficient lookup in the residual table, with Γ as the key). This means that given the same set of residuals, the indexer should always return the same Γ, even if multiple, equally well-suited choices



```
generate-node($P, \Gamma, Table, V$)
    if ($\Gamma = \{Z_p = t_1, ..., Z_p = t_n\}$) then
        — return the following code tree:
```

```
        $V' := suc(Z_p) \cup V$;
        $\forall\, i \in 1 \ldots n \mid D_i :=$ decision-graph($\Re_{dk}(P, Z_p = t_i), Table, V'$);
        $D_{neither} :=$ decision-graph($\Re_o(P, \Gamma), Table, V$);
        $D_{unbound} :=$ decision-graph($\Re_{us}(P, \Gamma), Table, V$);
    else
        if ($\Gamma = \{Z_p \bullet Z_q, \neg(Z_p \bullet Z_q)\}$) then
            — return the following code tree:
```

```
            $D_{yes} :=$ decision-graph($\Re_{dk}(P, Z_p \bullet Z_q), Table, V$);
            $D_{no} :=$ decision-graph($\Re_{dk}(P, \neg(Z_p \bullet Z_q)), Table, V$);
            $D_{unbound} :=$ decision-graph($\Re_{ua}(P, \Gamma), Table, V$);
```

Figure 5: New Node Generation for Don't-Know Procedure.

exist. If no choice is possible, e.g., all guard goals are empty, then the indexer returns $\emptyset$.

Our indexing algorithm is based on the *min-variability* indexing function given by Hickey and Mudambi [8] and Kliger and Shapiro [11]. Our procedure index($\Re$,V) takes an input residual and the visible variables, returning the set Γ, which is either a `val`-test or an `ask`-test. The reader is refered to [8, 11] for a description of the general algorithm. As an example, consider

$$\begin{aligned}\Re = \{&<1, \{Z_1 = 1, Z_2 = 1, Z_3 = 1, Z_4 > 0\}> \\ &<2, \{Z_1 = 2, Z_2 = 1, Z_3 = 2, Z_4 > 0\}> \\ &<3, \{Z_1 = 3, Z_2 = 2, Z_3 = 2, Z_4 < 0\}>\}\end{aligned}$$

The following five tests are possible: `val(`Z_1`)`, `val(`Z_2`)`, `val(`Z_3`)`, $Z_4 > 0$, and $Z_4 < 0$.

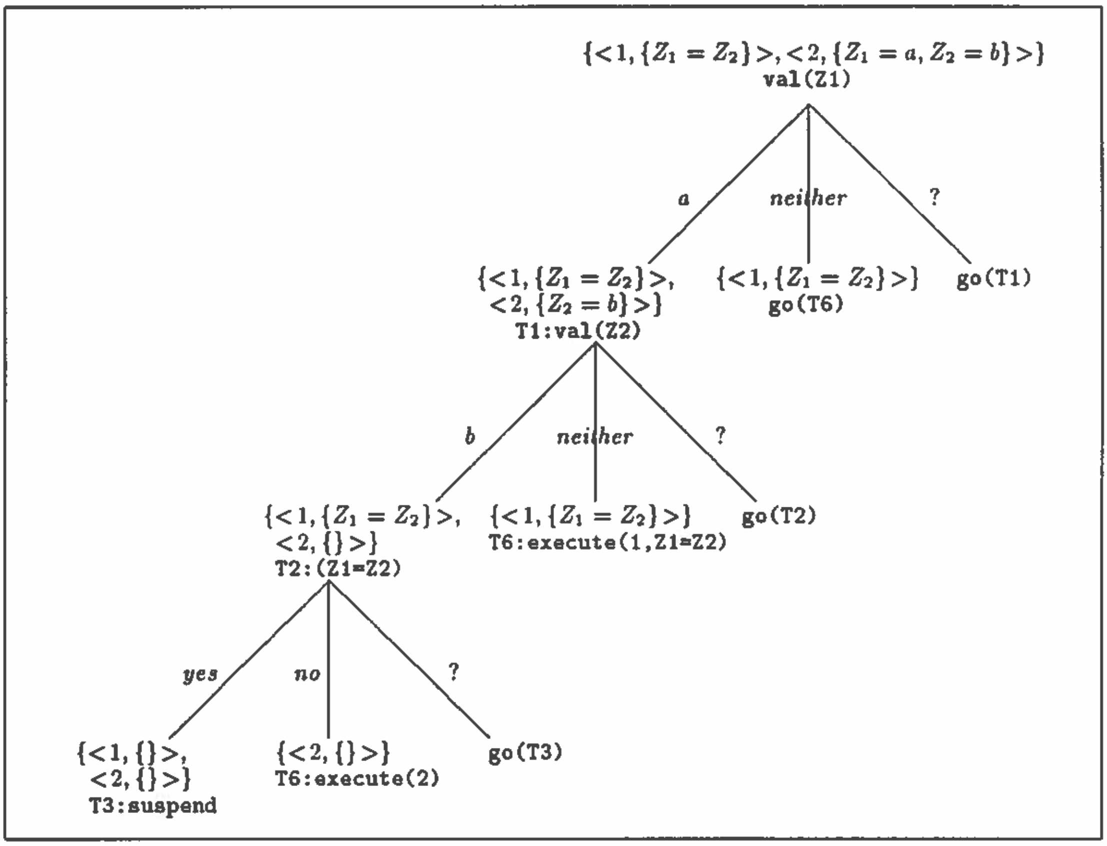


Figure 6: Decision Graph for `f/2` (Compiler's Internal Code Tree).

We find that 3, 3, 3, 2, and 1 clauses "care" [8, 11] for each of these tests, respectively. Maximizing the number of caring clauses thus leaves us with `val(`Z_1`)`, `val(`Z_2`)`, and `val(`Z_3`)`. From these tests, the first one produces three branches ($Z_1 = 1$, $Z_1 = 2$, and $Z_1 = 3$), while the other two only produce two branches, so the first test is eliminated. Finally, we select the test for the variable with the lowest variable number, in this case `val(`Z_2`)`. When we assume that, repeatedly, the selected test is removed from all clauses in the residual, and the indexer is called again, the tests `val(`Z_3`)`, `val(`Z_1`)`, $Z_4 > 0$, and $Z_4 < 0$ are returned, respectively.

5.2 Computing Residuals

Our method is only as complete as is its inference mechanism in determining the implications (e.g., $g' \Longrightarrow g$) in the residual definitions. In this regard, qualitatively we have found that simple implications rules are quite effective. Table 1 shows a subset of the

g	g'	$(g \Rightarrow g')$ iff	g	g'	$(g \Rightarrow g')$ iff
$X > I$	$X > J$	$I \geq J$	$X > I$	$X \geq J$	$I \geq J$
$X < I$	$X < J$	$J \geq I$	$X < I$	$X \leq J$	$J \geq I$
$X \leq I$	$X \leq J$	$J \geq I$	$X \leq I$	$X < J$	$J > I$
$X \geq I$	$X \geq J$	$I \geq J$	$X \geq I$	$X > J$	$I > J$
$X > I$	$X \neq J$[a]	$I \geq J$	$X > I$	$X \backslash= J$	$I \geq J$
$X < I$	$X \neq J$	$J \geq I$	$X < I$	$X \backslash= J$	$J \geq I$
$X \geq I$	$X \neq J$	$I > J$	$X \geq I$	$X \backslash= J$	$I > J$
$X \leq I$	$X \neq J$	$J > I$	$X \leq I$	$X \backslash= J$	$J > I$
$X =:= J$	$X > I$	$J > I$	$X = J$	$X > I$	$J > I$
$X =:= J$	$X \geq I$	$J \geq I$	$X = J$	$X \geq I$	$J \geq I$
$X =:= J$	$X < I$	$I > J$	$X = J$	$X < I$	$I > J$
$X =:= J$	$X \leq I$	$I \geq J$	$X = J$	$X \leq I$	$I \geq J$
$X = J$	$X \neq I$	$I \neq J$	$X =:= J$	$X \neq I$	$I \neq J$
$X =:= J$	$X \backslash= I$	$I \neq J$			

[a] Although not essential for the compiler, note the difference between $\neq$, which expects both its arguments to be arithmetic expressions, and evaluates them, and $\backslash=$, which simply tries to unify its arguments, without evaluation.

Table 1: Simple Rules Determining $g \Longrightarrow g'$, for Integers I and J.

rules (involving integers) in use in Monaco, our current compiler [14].[12]

Shared variables cause problems because they transfer constraints indirectly. For example, $\{\mathtt{X} = \mathtt{Y}, \mathtt{Y} = \mathtt{Z}, \mathtt{Z} = \mathtt{W}\}$ implies that $\mathtt{X} = \mathtt{W}$. Similarly, $\{\mathtt{X} > \mathtt{Y}, \mathtt{Y} > \mathtt{Z}\}$ implies that $\mathtt{X} > \mathtt{Z}$. The first case can easily be handled during conversion to canonical form, as mentioned earlier. The latter case can be handled in a similar manner without significant code expansion, since sharing is not frequent. However, this method is limited, and does not easily operate *across* clauses. Furthermore, constraints involving both equalities and comparisons need a strong inference mechanism. Given the constraint $g = \{\mathtt{Z} = 0\}$, the indexer is able to infer that "$\{\mathtt{Z} > 0\} \Longrightarrow \neg g$," given the rules in Table 1. However, we are confident that complex sets of interacting constraints within the same procedure are rare.

5.3 Commit and Suspension

Because suspension and resumption of goals are costly operations, it is necessary to perform them as efficiently as possible. In this section, we describe an efficient implementation of this suspension mechanism, using a combination of an intelligent compile-time

[12] The Aquarius Prolog compiler [16] simplifies formulae in a similar manner. Both Monaco and Aquarius use about 50 rules of comparable complexity.

code generator and a simple, but efficient, run-time implementation of this scheme.

A **suspend** leaf is meant to suspend execution, enabling either resumption by subsequent binding of a hooked variable, or forced resumption upon deadlock. Ideally, the clause bodies to be considered upon resumption should be the smallest set that satisfy the suspended invocation. At compile time, when the code generator is down to a **suspend** leaf, the residual has the form $\Re = \{<C_i,\emptyset>,<C_j,\emptyset>,\ldots,<C_k,\emptyset>\}$. For this leaf, we can generate the WAM-like code:

```
    suspend_or L
L:  try   Ci_H
    retry Cj_H
    ...
    trust Ck_H
```

where `Ci_H` points to the head of clause `i`. Suspension sets the continuation in the goal record to point to the `try` instruction. This ensures that only the subset of clauses known to be candidates is executed if the goal is forced to resume upon deadlock.[13]

A simpler option (implemented in the Monaco compiler) is to generate a single sequence of `try`, `retry`, and `trust` instructions for *all* clauses, and to give all suspensions the *same* continuation `C1_H`, i.e., the first clause of the procedure. This obviates the need for individual `try` sequences, a great savings in code size, at the potential expense of execution redundancy.

At run time, when a variable in a test is unbound, the variable is pushed onto the suspension stack. A continuation pointing to this suspended test can also be pushed onto the stack. Upon reaching a **suspend** instruction, the stack is popped, and each unbound variable is hooked to the goal. As an optimization, the associated continuations can be attached to each hook, so that resumption continues precisely at the relevant test. With this optimization, it is possible to resume a goal (triggered by binding its variable) at exactly the point where suspension took place, instead of restarting at the root of the graph, thus having to perform previous tests again.[14]

As previously mentioned, residuals with single clauses terminate as code leaves. The idea here is to force execution, even if some clause variable(s) have been pushed onto the suspension stack, c.f., committed-choice (don't-care) execution where suspended variables imply goal suspension. There are two problems here: 1) we need to determine which, if any, pushed variables are present in the leaf clause, and 2) we need to collect

[13] The `try` sequence semantics is more complex than in Prolog: as mentioned in Section 2, guards may suspend when a don't-know Pandora procedure is forced to resume. Also, clauses may be tried out-of-order, perhaps managed by a clause vector in the choicepoint.

[14] Note that code sharing can cause multiple copies of the same variable to be pushed onto the suspension stack. Each variable should be hooked only once! If continuations are to be used, the earliest one (closest to the base of the stack) is required.

the constraints associated with these suspended variables and attempt to execute them. The latter collection proves too cumbersome, so we propose to simply re-execute the clause head (and guard) if suspended variables exist.

In summary, if no code sharing is possible for residual $\Re = \{< C_i, G >\}$, we generate `execute(i,G)`, with the semantics:

```
if (suspension_stack_check == ok)
    then {G; go(Ci_B);}    /* Ci_B pts to clause i body */
    else go(Ci_H);
```

The test `suspension_stack_check` returns `ok` if no suspension stack variables are relevant to clause i. In this case, only the constraints in `G` are forcibly executed. Otherwise, the entire clause is re-executed. Forcible execution implies sophisticated compilation, e.g., for unbound `X`, `X=3` must bind `X`, but `X>3` must suspend.

This scheme can be implemented efficiently with *bit vectors*. At compile time, when expanding each clause into its canonical form, the compiler generates a bit vector for each clause, where for each variable which occurs in the canonical form, the corresponding bit in the vector is set. For example, if the head of clause i is $f(g(X,Y))$, the canonical form would be $f(Z_1) :- Z_1 = g(Z_{1,1}, Z_{1,2})$ and the bit vector would be set to $B_i = [Z_1, Z_{1,1}, Z_{1,2}] = [100]$.[15]

The original residual is extended with a new argument, the bit vector B_i, as defined above. When a new residual is calculated, a new bit vector B_i' is calculated as well, with the following rules: for all branches labeled with '?' or '??', the bit vector is unaffected. For all other branches,[16] the bits in B_i' corresponding to the variables in Γ are zeroed.

When 1) the entire compile-time bit vector B at a leaf is 0, and 2) no shared nodes (targets of `go` instructions) appear on the path from the root to the leaf, then we can replace the `execute` instruction with the simpler code sequence "`G; commit(i).`" The semantics for the `commit` instruction is simply: "`go(Ci_B).`" Condition (2) can be removed by combining bit vectors at shared nodes, although we do not pursue this here.

At run time, a single bit vector SS is used to represent the suspension stack. It contains bits for all unique variables in the procedure, which are initially set to 0. Each time a variable is pushed on the suspension stack, its corresponding bit in SS is set to 1. The `suspension_stack_check` is then reduced to testing the logical AND of two bit vectors. The `execute(i,G)` becomes:

```
if ((SS AND Bi) == 0)
    then G; go(Ci_B);
    else go(Ci_H);
```

[15] The actual test is $Z_1 = g/2$, therefore the bits for $Z_{1,1}$ and $Z_{1,2}$ are not set.

[16] This corresponds to those branches for which Γ can be fully evaluated, implying that all variables in Γ are ground.

```
% don't-care version:
cell(I, J, J, I, I, I, begin, end, begin, end) :- true | body1.
cell(_, _, _, _, _, _, Hc, Hc, Vc, Vc)         :- true | body2.

% don't-know version:
cell(I, J, J, I, I, I, begin, end, begin, end) :- true : body1.
cell(_, _, _, _, _, _, Hc, Hc, Vc, Vc)         :- true : body2.

% don't-know version (hand-translation into FGHC):
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- H\==J |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- V\==I |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- L\==I |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- R\==I |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- V\==L |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- L\==R |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,Left1,Right1,Left2,Right2) :- R\==V |
    Left1=Right1, Left2=Right2, body2.
cell(I,J,H,V,L,R,begin,end,Left,Right) :- true |
    H=J,V=I,L=I,R=I,Left=begin,Right=end, body1.
cell(I,J,H,V,L,R,Left,Right,begin,end) :- true |
    H=J,V=I,L=I,R=I,Left=begin,Right=end, body1.
```

Figure 7: Three `cell/10` Benchmarks.

6 Code Space Evaluation

This section presents empirical measurements of code characteristics and size generated by the previous algorithms. As an example of the benchmarks, Figure 7 shows the `cell/10` procedure, taken from an active-constraints program for solving N-Queens. In the clauses, `body1` and `body2` represent unspecified clause bodies. The first two procedure definitions have similar code, but the semantics are vastly different. The don't-know semantics are simulated by first translating the don't-know version into an equivalent don't-care procedure with mutually exclusive clauses (third program shown), as first indicated by R. Bahgat [1]. Note the additional body goals added to force unification once the procedure has been found to be determinate. Although the number of clauses has increased as a function of the number of constraints, the resulting clauses each have only a small number of constraints. The decision graphs generated for the first two of these procedures are shown in Figures 8 and 9, respectively.

Table 2 gives two alternative code-size measurements of a set of small compiled benchmark procedures. Three program types are given. **Kliger** represents the code size

```
dcproc_entry(cell(Z1,Z2,Z3,Z4,Z5,Z6,Z7,Z8,Z9,Z10),
ask(Z1=Z4,
  yes(
    ask(Z1=Z5,
      yes(
        ask(Z1=Z6,
          yes(
            ask(Z2=Z3,
              yes(
                switch(Z7,
                  [case(begin,
                     switch(Z8,
                       [case(end,
                          switch(Z9,
                            [case(begin,
                               switch(Z10,
                                 [case(end, commit(1,[body1])),
                                  case(tag(??), go(L23))
                                 ])),
                             case(tag(??), L23: go(L22))
                            ])),
                        case(tag(??), L22: go(L21))
                       ])),
                   case(tag(??), L21: go(L20))
                  ])),
              no(go(L20)),
              other(L20: go(L19)))),
          no(go(L19)),
          other(L19: go(L18)))),
      no(go(L18)),
      other(L18: go(L17)))),
 no(go(L17)),
 other(L17:
   ask(Z7=Z8,
     yes(
       ask(Z9=Z10,
         yes(commit(2,[body2])),
         no(go(L26)),
         other(L26: go(L25)))),
     no(go(L25)),
     other(L25: suspend)))))
```

Figure 8: Don't-Care cell/10 Decision Graph.

```
dkproc_entry(cell(Z1,Z2,Z3,Z4,Z5,Z6,Z7,Z8,Z9,Z10),
ask(Z1=Z4,
  yes(
    ask(L39: Z1=Z5,
      yes(
        ask(L41: Z1=Z6,
          yes(
            ask(L43: Z2=Z3,
              yes(
                ask(L45: Z7=Z8,
                  yes(
                    ask(L47: Z9=Z10,
                      yes(
                        ask(L49: Z7=begin,
                          yes(
                            ask(L51: Z8=end,
                              yes(
                                ask(L53: Z9=begin,
                                  yes(
                                    ask(L55: Z10=end,
                                      yes(suspend),
                                      no(L57:commit(2,[body2])),
                                      unbound(suspend))),
                                  no(go(L57)),
                                  unbound(go(L55)))),
                              no(go(L57)),
                              unbound(go(L53)))),
                          no(go(L57)),
                          unbound(go(L51)))),
                      no(L61:
                        execute(1,[Z7=begin,Z8=end,
                                   Z9=begin,Z10=end],[body1])),
                      unbound(go(L49)))),
                  no(go(L61)),
                  unbound(go(L47)))),
              no(L64: execute(2,[Z9=Z10,Z7=Z8],[body2])),
              unbound(go(L45)))),
          no(go(L64)),
          unbound(go(L43)))),
      no(go(L64)),
      unbound(go(L41)))),
  no(go(L64)),
  unbound(go(L39)))
```

Figure 9: Don't-Know `cell/10` Decision Graph.

procedure name	# clauses	Kliger size	Bahgat size	K&T size	K&T/ Baghat
Decision Graph Node Count					
f/2	2	3	6	4	0.66
a/3	4	6	8	8	1.00
cell/5	2	3	7	5	0.71
cell/10	2	10	20	12	0.60
omerge/3	4	6	6	9	1.50
total		28	47	38	0.81
Monaco Instruction Count					
f/2	2	28	39	32	0.82
a/3	4	55	49	52	1.06
cell/5	2	38	46	22	0.48
cell/10	2	105	144	112	0.78
omerge/3	4	33	33	39	1.18
total		259	311	250	0.80

Table 2: Empirical Measurements of Code Size.

if compiled as a don't-care procedure. Such a program does not have the semantics of the next two don't-know versions, and is given simply as a baseline. **Bahgat** represents the code size if first hand-translated into a don't-care procedure with equivalent don't-know semantics, and then compiled [1]. **K&T** represents directly compiling the don't-know procedures with our method.

The upper portion of the table lists the number of complex nodes within the decision graphs, including **ask**, **switch**, **commit**, and **execute** nodes (the latter two are counted only if they are not trivial jumps). The lower portion of the table lists the number of intermediate abstract machine instructions generated by the our compiler for each program.[17] For **K&T**, the compiler normally appends a **suspend_or** code block to each procedure, effectively constituting the Prolog compilation of all clauses to try upon deadlock. We discount these instructions in the **K&T** measurements given in Table 2 to calibrate with **Baghat** which also requires, but does not include, the block.

The final column in Table 2 gives the ratio of code size between our and Baghat's method (comparison to Kliger does not always make sense, since the functionality can be vastly different). Both the graph and Monaco code ratios closely match at about 80%, although these metrics can differ more significantly for individual programs. This evidence supports the claim that we can perform determinism testing at least as

[17] Relevant to this discussion, it should be noted that the Monaco compiler does local dataflow analysis including common subexpression and deadcode elimination, branch-chain shorting and various peephole optimizations. Thus the Monaco instruction count is a fairly optimized metric.

procedure	cls.	ask	switch	execute	sum	don't-	ratio
`flatten_guard/9`	5	0	6	0	6	care	2.50
		5	9	1	15	know	
`lower_ask_vars/4`	8	2	8	0	10	care	2.80
		16	7	5	28	know	
`select_best/3`	4	0	3	0	3	care	2.33
		2	4	1	7	know	
`better/5`	6	11	3	0	14	care	2.57
		27	4	5	36	know	
`lookup/3`	5	2	7	0	9	care	1.33
		7	2	3	12	know	
`match_ask_test/5`	5	9	5	0	14	care	1.93
		20	2	5	27	know	
`equal/3`	9	17	9	0	26	care	1.96
		40	2	9	51	know	
`implies4/3`	46	115	20	0	135	care	0.70
		31	17	46	94	know	

Table 3: Comparison Between Don't-Care and Don't-Know Size by Procedure.

cheaply, in code space, as the alternative method based on decision graphs. This result is somewhat misleading because Baghat's translation itself may have high complexity (in the number of clauses generated) for certain degenerate procedures, so we have not disproved the possibility that our method may also exhibit explosion in certain instances.

In the five examples here, the code sizes of all the methods are equal. However, it is clear that the space complexity of our method is in worst case exponential because it derives from a full decision tree [10]. The effectiveness of heuristic code sharing to avoid exponential behavior can be thwarted by degenerate procedures. In the context of decision trees, Kliger describes the existence of such degenerate procedures in large committed-choice systems. It is an open question if, among don't-know procedures in real Andorra/Pandorra applications, any or a significant number have similar degenerate characteristics.

Table 3 shows the graph size measurements of a selection of complex procedures from the Monaco compiler itself. Each procedure in the system was compiled both as don't-care *and* as don't-know.[18] We count only complex graph nodes, a metric

[18] Don't-know procedures are hard to come by, since Pandora and Andorra are such new languages. Within an entire Pandora program, there are relatively few don't-know procedures. Thus, for the sake of characterizing code metrics, we chose to compile standard FGHC procedures, *as if they were don't-know procedures.*

calibrated to pseudo-code size, as discussed previously.

Although the procedures were not written with don't-know semantics in mind, this exercise allows us to further compare the code sizes of don't-care vs. don't-know. The measurements indicate that compiling into don't-know semantics requires a (up to 2.8 times) larger graph than for don't-care semantics because of the added power of rapidly detecting determinism and forcing execution of unbound constraints. Again, this comparison has limited meaning because the semantics of the two types of graphs are different.

Depending on the procedure, the relative code size varies between our method and hand-translation. The causes why our method sometimes does not achieve the size of the hand-translation are as follows:

- In some cases, hand-translation into don't-care equivalents removes the need for some constraints. The effective action of testing for these constraints is done by body failure in the don't-care equivalent.
- In direct-compilation of don't-know procedures, some code leaves are complex `executes`, whereas in don't-care procedures, all code leaves are trivial `commits`.
- The method outlined generates "neither" branches causing "fast failure," whereas in Kliger's graphs, failure propagates through all otherwise branches until the initial suspend continuation is reached. Note that committed-choice implementations do not need to handle failure quickly, since it is an exception. In any case, our faster failure has a cost in additional nodes.

To our knowledge, no algorithm has yet been found for Bahgat's method of don't-know into don't-care translation [1]. The compilation complexity of this hand-translation increases dramatically with the number of clauses. We feel comfortable that our algorithm has approximately the same space complexity, at low compile-time cost.

Table 4 compares the maximum and average path lengths for all three methods. Assuming that all paths have an equal probability, and normalizing with respect to Kliger's method, Bahgat's method has an average path length 20% longer than Kliger, while our method increases the path length by only 10%.

Composition of the graphs is also of interest, as summarized in Table 5. This table gives the distribution of switch and ask node types for the compiler benchmark. Whether compiled as don't-care or don't-know, the distribution of types is approximately the same. Over half of all nodes are ask tests of a variable unified with a constant. Note that switch nodes with a single child are converted into these ask nodes. Switch nodes are categorized by the number of data types represented in their children. For example a popular double-type switch node is a list-recursive procedure

procedure	Kliger		Bahgat		K&T	
name	max	avg	max	avg	max	avg
f/2	4	2.8	5	3.1	3	2.5
a/3	3	2.6	5	3.1	7	4.3
cell/5	2	1.8	5	3.4	3	2.0
cell/10	14	8.2	12	6.9	13	8.1
omerge/3	6	3.4	4	2.5	4	2.5

Table 4: Maximum and Average Path Lengths.

			don't-care		don't-know	
node	type	children	occur	%	occur	%
	single	2	45	10.0	58	11.7
switch	type	≥ 3	7	1.6	9	1.8
	double	2	55	12.3	60	12.1
	type	3	4	0.9	9	1.8
		≥ 4	1	0.2	2	0.4
	$Z = k$		263	58.7	265	53.6
ask	$Z_1 = Z_2$		41	9.2	51	10.3
	$Z_1 \bullet Z_2$, $Z_1 \bullet k$		12	2.7	19	3.8
	integer, atom, etc.		20	4.5	21	4.3
total			448	100.0	494	100.0

Table 5: Breakdown of Decision Graphs by Non-Leaf Node Type.

with both ./2 and [] values. In the compiler itself, no more than two types are found in any switch, and usually the number of branches (children) is quite small.

Naturally, execution-time measurements are necessary to fully evaluate the utility of the proposed compilation method, for instance with respect to the Andorra-I compiler [5]. Unfortunately, we have not yet implemented support for backtracking within the Monaco runtime system [13] and thus execution measurements remain an area of future work.

7 Conclusions

This article introduced a decision-graph construction algorithm for code generation of determinacy testing in nondeterminate flat concurrent logic programming languages. The code generated is complete over unification, i.e., determinacy is guaranteed to be detected no matter how complex the data structures and shared variables are. Completeness over other domains, such as arithmetic comparison, is the responsibility of the algorithm for computing implications, for which completeness has not yet been proved.

Our algorithm is formulated in the context of committed-choice compilation techniques given by Kliger and Shapiro [10, 11]. For simple procedures, the two are shown to have comparable code size. Thus we believe that with this method, complete determinacy testing over unification need not significantly increase code size for a wide selection of procedures. However, for degenerate procedures the don't-know graph size (theoretically of exponential complexity and reduced only by the heuristic means of code sharing) can be significantly larger than the don't-care graph size (of bounded linear complexity). Comparing our algorithm to an elegant method of hand-translation from don't-know into don't-care code [1], the code size appears to be least as good, for small procedures (how these compare for degenerate procedures is unknown). Our algorithm is, however, more general because it does not require that the clauses be mutually exclusive and can be done fully automatically.

Acknowledgements

M. Korsloot was supported by a grant from the Delft University of Technology. E. Tick was supported by an NSF Presidential Young Investigator Award with matching funds granted by Sequent Computer Inc. Many thanks to Xiaoxiong Zhong for his extensive help with the correctness proofs. We also thank the referees for their insightful comments that led to clarifying and strengthening the article.

References

[1] R. Bahgat. *Pandora: Non-deterministic Parallel Logic Programming.* PhD thesis, Imperial College, Department of Computing, February 1991.

[2] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In *International Conference on Logic Programming*, pages 471–486. Lisbon, MIT Press, June 1989.

[3] V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 83–93. Williamsburg, ACM Press, April 1991.

[4] V. S. Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *International Conference on Logic Programming*, pages 825–839. Paris, MIT Press, June 1991.

[5] V. S. Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *International Conference on Logic Programming*, pages 443–456. Paris, MIT Press, June 1991.

[6] S. Haridi and P. Brand. Andorra Prolog—An Integration of Prolog and Committed Choice Languages. In *International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, November 1988. ICOT.

[7] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *International Conference on Logic Programming*, pages 31–46. Jerusalem, MIT Press, June 1990.

[8] T. Hickey and S. Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, 7(3):193–230, November 1989.

[9] Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society, August 1987.

[10] S. Kliger and E. Y. Shapiro. A Decision Tree Compilation Algorithm for FCP(—,:,?). In *International Conference and Symposium on Logic Programming*, pages 1315–1336. University of Washington, MIT Press, August 1988.

[11] S. Kliger and E. Y. Shapiro. From Decision Trees to Decision Graphs. In *North American Conference on Logic Programming*, pages 97–116. Austin, MIT Press, October 1990.

[12] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[13] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE'93*, Springer Verlag, June 1993.

[14] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming.* Budapest, MIT Press, June 1993.

[15] K. Ueda. Guarded Horn Clauses. In E. Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

[16] P. L. Van Roy and A. M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

A Correctness Proof

We present a correctness proof of the code-generation algorithm presented in Section 5. Refer to the following definitions for clarification. First we prove a weaker theorem that a simpler algorithm, A_0, without code sharing, is correct. Then we prove the stronger correctness theorem by showing that the code-sharing algorithm, A_1, preserves the correctness invariants.

Definition: Given a query Q, i.e., an invocation of procedure P,

$$P_Q = \{c \in P \mid c \text{ satisfies } Q\}$$

for clauses c. We call P_Q the *induced clause set* of P with respect to Q. □

For a clause to satisfy Q, the clause must commit with Q. Furthermore for a clause to satisfy a *branch*, no guard can contradict the decision represented by that branch. For example, in Figure 6, the `val(Z1)` test and its result `Z1=a` comprise a decision. Clauses satisfying the branch cannot have guards such as `Z1=b` and `Z1\=a`. More precisely,

Definition: Consider a switch node with $\Gamma = \{g_1, g_2, ..., g_m\}$. A clause $c = < i, A, B >$ with guard $A = \{g'_1, g'_2, ..., g'_k\}$ satisfies a branch if

$$\begin{cases} \tilde{\forall}.\ g \Rightarrow (g'_1 \wedge g'_2 \wedge ... \wedge g'_k) & \text{for case branch } g \\ \forall\ g \in \Gamma.\ \neg\tilde{\forall}.\ (g \Rightarrow g'_1 \wedge g'_2 \wedge ... \wedge g'_k) & \text{for neither branch} \\ \text{true} & \text{for unbound branch} \end{cases}$$

Similar definitions are needed for ask nodes. □

Definition: Consider a procedure corresponding to a decision tree. A ***candidate clause set***, P_u, is the set of clauses, from P, comprising the subtree rooted at node u. In other words, P_u is the recursive argument in A_0. □

Intuitively, when successfully executing a case branch with test g, some clauses are filtered out because they cannot possibly commit. Any clause remaining in the candidate clause set must be such that each guard goal g' can be implied by g. Inversely, if the neither branch is executed, any clause with guards implied by any case branch test g cannot be included in the candidate clause set (it has been included in the candidate clause set of a sibling case branch). Finally, all clauses are included in the candidate clause set of an unbound branch since during execution the switch variable is unbound and the test is inconclusive. It should be clear that given a query Q, if program execution proceeds down branch b in some procedure with clause c, then (c satisfies Q) $\Rightarrow$ (c satisfies b).

Definition: A decision tree (produced by A_0) is *correct* with respect to a query Q of procedure P iff

$$\forall \text{ nodes } u \in \text{ tree } . \; P_Q \subseteq P_u$$

□

Theorem 1 *For any procedure P, the decision tree, produced by algorithm A_0, is correct with respect to any query Q.* □

The weak theorem is proved in Figure 10 for decision trees produced by algorithm A_0. The proof hinges on a lemma concerning the mechanism by which code is generated, which is given in Figure 11. The lemma is proved only for the switch statement. The proof carries over, in a similar manner, to the ask statement. The crucial points in the proof are (2) and (5) for D_i branches and the $D_{neither}$ branch, respectively. For these formulae, it is useful to consider the real domain with simple equalities for branch tests and inequalities for constraints. Then we can view g and g' as specifying sets of real values. (2) cannot be true unless the value specified by g is contained by g'_i, for all possible bindings. In this case there exists some value satisfying the RHS.

Showing (5) is a bit trickier. We first consider only clauses that have no self-contradictory constraints. If this is true, then $\{g'_1, g'_2, ..., g'_k\}$ must have at least one shared value. Assume that g'_i is a singleton set with the same value as g_j. This is the only situation making the RHS false. Since all g'_i have at least one shared value, this value must be g_j. But this assumption will cause the LHS to be false because the row corresponding to g_j will have all false disjuncts since g_j lies within all g'_i. Thus (5) is true. For programs with self-contradictory clauses, this proof technique fails. In fact, such clauses always travel down $D_{neither}$, but can never commit, so the program remains correct.

To strengthen the weak theorem, we must consider the effect of code sharing in A_1. Code is shared if an entry in the global residual table contains proper subsets of both the candidate set and the index set, of the current node. The strategy is to generate local code for the difference between the index sets. A final '??' branch transfers control to the shared code (previously generated) if all previous tests failed. The stronger theorem is then:

Theorem 2 *For any procedure P, the decision graph, produced by algorithm A_1, is correct with respect to any query Q.* □

The proof of this theorem is given in Figure 12. The proof hinges on the fact that an entry in the residual table that can be used for sharing must meet the criteria stated above. If the entry's index set is a proper subset of the local index set, then the entry's candidate set cannot satisfy the constraints in the difference of the index sets. Induction in this proof is done on the size of the table, corresponding to some creation-ordering of the nodes in the tree.

We now informally prove that these algorithms terminate. Consider A_1. Each step of the algorithm reduces the candidate-clause set, either in the number of clauses, or in the number of constraints within the clauses. Within the algorithm (Figure 3) are three cases that do not recurse:

- If the candidate set is empty (0).
- If the candidate set has a single member (1).
- If the index set is empty (4). This happens when no constraints remain in the residual.

Since the initial candidate set has a finite number of clauses and constraints, it follows that eventually these will be reduced to either no candidates, a singleton candidate, or a set of candidates, all with empty constraints. These three cases do not recurse, thus the algorithm will terminate.

The previous two theorems are concerned only with queries that do not suspend. Additional correctness proofs for suspension are not included here.

PROVE: For any procedure P, for any query Q, algorithm A_0 produces a correct decision tree of P with respect to Q.

BASE CASE: at tree depth 0

By definition, $P_0 = P$, thus $P_Q \subseteq P_0$

INDUCTION STEP: at tree depth $j > 0$

Consider a node u with child node v, connected by branch b.
Assume that $P_Q \subseteq P_u$ (partial correctness assumption).
We must first show that $P_Q \subseteq P_v$.

By the various code generation rules (see Lemma 1):
$P_v \supseteq \{c \in P_u \mid c \text{ satisfies } b\}$ (1)
By the semantics corresponding to executing along branch b, we know that for any clause c,
(c satisfies Q) $\Rightarrow$ (c satisfies b) (2)
Also recall that by definition
$\forall c \in P_Q$. c satisfies Q
For any clause c from P_Q, we know:
$c \in P_u$ (3)
c satisfies Q (4)
By (2) and (4),
c satisfies b (5)
By (1), (3), and (5),
$c \in P_v$
Thus since any c in P_Q is also in P_v,
$P_Q \subseteq P_v$

By induction on the tree depth j,
$\forall$ nodes $u \in$ tree . $P_Q \subseteq P_u$
Therefore the tree of P is correct with respect to Q.

Figure 10: Proof of Weak Theorem For Decision Trees.

PROVE: For a node u, any child node v connected by branch b, it is true that

$P_v \supseteq \{c \in P_u \mid c \text{ satisfies } b\}$

case D_i:

$$P_v = \Re_{dk}(P_u, g) = \bigcup_{c \in P_u} \Re_{dk}(c, g)$$

$$= \bigcup_{c \in P_u} \{c = <i, A \setminus G> \mid \ \{g' \in A \mid \tilde{\forall}.\ g \Rightarrow \neg g'\} = \emptyset\}$$

$$= \{c \in P_u \ \mid \ \{g' \in A \mid \tilde{\forall}.\ g \Rightarrow \neg g'\} = \emptyset\}$$

$$= \{c \in P_u \ \mid \ \forall g' \in A.\ \neg(\tilde{\forall}.\ g \Rightarrow \neg g')\}$$

$$= \{c \in P_u \ \mid \ \forall g' \in A.\ \tilde{\exists}.\ g \wedge g'\} \qquad (1)$$

Consider that for some $g \equiv (X = t)$ and some constraint g'_i

$$(\tilde{\forall}.\ \neg g \vee g'_i) \Rightarrow (\tilde{\exists}.\ g \wedge g'_i) \qquad (2)$$

because for LHS to be true, t must be within set defined by g'_i, thus

$$(\tilde{\forall}.\ g \Rightarrow g'_i) \Rightarrow (\tilde{\exists}.\ g \wedge g'_i)$$

$$(\forall g' \in A.\ \tilde{\forall}.\ g \Rightarrow g') \Rightarrow (\forall g' \in A.\ \tilde{\exists}.\ g \wedge g')$$

$$(\tilde{\forall}.\ g \Rightarrow (g'_1 \wedge \ldots \wedge g'_k)) \Rightarrow (\forall g' \in A.\ \tilde{\exists}.\ g \wedge g')$$

$$(c \text{ satisfies } b) \Rightarrow (\forall g' \in A.\ \tilde{\exists}.\ g \wedge g') \qquad (3)$$

Thus by (1) and (3)

$P_v \supseteq \{c \in P_u \mid c \text{ satisfies } b\}$

case $D_{neither}$:

$$P_v = \Re_o(P_u, \Gamma) = \bigcup_{c \in P_u} \Re_o(c, \Gamma)$$

$$= \bigcup_{c \in P_u} \{c = <i, A \setminus G> \mid \ \{g' \in A \mid \exists g_i \in \Gamma.\ \tilde{\forall}.\ g' \Rightarrow g_i\} = \emptyset\}$$

$$= \{c \in P_u \ \mid \ \{g' \in A \mid \exists g_i \in \Gamma.\ \tilde{\forall}.\ g' \Rightarrow g_i\} = \emptyset\}\}$$

$$= \{c \in P_u \ \mid \ \{g' \in A \mid (\tilde{\forall}.\ g' \Rightarrow g_1) \vee \ldots \vee (\tilde{\forall}.\ g' \Rightarrow g_m)\} = \emptyset\}$$

$$= \{c \in P_u \ \mid \ \forall g' \in A.\ \neg((\tilde{\forall}.\ g' \Rightarrow g_1) \vee \ldots \vee (\tilde{\forall}.\ g' \Rightarrow g_m))\}$$

$$= \{c \in P_u \ \mid \ \forall g' \in A.\ (\tilde{\exists}.\ g' \wedge \neg g_1) \wedge \ldots \wedge (\tilde{\exists}.\ g' \wedge \neg g_m))\}$$

$$\begin{aligned} = \{c \in P_u \ \mid \ & ((\tilde{\exists}.\ g'_1 \wedge \neg g_1) \wedge \ldots \wedge (\tilde{\exists}.\ g'_1 \wedge \neg g_m)) \wedge \\ & \ldots \\ & ((\tilde{\exists}.\ g'_k \wedge \neg g_1) \wedge \ldots \wedge (\tilde{\exists}.\ g'_k \wedge \neg g_m))\} \end{aligned}$$

$$= \{c \in P_u \ \mid \ \tilde{\exists}.\ g'_i \wedge \neg g_j\} \qquad (4)$$

for all i, j. By discussion in the text,

$$\begin{aligned} & [((\tilde{\exists}.\ g_1 \wedge \neg g'_1) \vee \ldots \vee (\tilde{\exists}.\ g_1 \wedge \neg g'_k)) \wedge \\ & \ldots \\ & ((\tilde{\exists}.\ g_m \wedge \neg g'_1) \vee \ldots \vee (\tilde{\exists}.\ g_m \wedge \neg g'_k))] \Rightarrow (\tilde{\exists}.\ g'_i \wedge \neg g_j) \end{aligned} \qquad (5)$$

After manipulating the LHS,

$$\forall g \in \Gamma.\ \neg\tilde{\forall}.\ (g \Rightarrow g'_1 \wedge \ldots \wedge g'_k) \Rightarrow (\tilde{\exists}.\ g'_i \wedge \neg g_j)$$

$$(c \text{ satisfies } b) \Rightarrow (\tilde{\exists}.\ g'_i \wedge \neg g_j) \qquad (6)$$

Thus by (4) and (6)

$P_v \supseteq \{c \in P_u \mid c \text{ satisfies } b\}$

case $D_{unbound}$:

in this case, all clauses (with constraints suitably filtered) are included in the residual so the proof goes through trivially.

Figure 11: Lemma (1) of Decision Tree Construction: Switch Statement.

PROVE: For any procedure P, for any query Q, algorithm A_1
produces a correct decision tree of P with respect to Q.

BASE CASE: *Table* size $j = 0$
By definition, $P_0 = P$, thus $P_Q \subseteq P_0$

INDUCTION STEP: *Table* size $j > 0$
Case 1: No sharing: $\neg(\exists\, [\Gamma_w, P_w] \in \mathit{Table} \,.\, P_u \supseteq P_w, \Gamma_u \supseteq \Gamma_w)$
Consider a node u with child node v, connected by branch g.
By proof in Figure 10,
$P_Q \subseteq P_v$
By induction on *Table* size j,
$\forall$ nodes $u \in$ tree . $P_Q \subseteq P_u\}$
Therefore the tree of P is correct with respect to Q.

Case 2: Sharing: $(\exists\, [\Gamma_w, P_w] \in \mathit{Table} \,.\, P_u \supseteq P_w, \Gamma_u \supseteq \Gamma_w)$
Where current node u has index set Γ and shares candidates from P_w.
We calculate $\gamma \subseteq \Gamma$ (see Figure 4)
Partial correctness assumption: $P_Q \subseteq P_u$ (1)
There are two types of branches to consider:
Case 2a: $\{g \in \gamma\}$ are local, generated by $\Re_{dk}$
By (1) and Case (1), the tree through these branches is correct.

Case 2b: '??' branch b corresponds to 'neither' or '?' branches.
We must show that execution through b, sharing code at P_w, is correct.
By Lemma 2 (Figure 13):

$$\begin{aligned} \Gamma_u &= \Gamma_w \cup \gamma \\ &= (\Gamma_w \setminus \gamma) + \gamma = \alpha + \gamma \qquad (2) \end{aligned}$$

where α are nonlocal branches and γ are local branches.
By the semantics corresponding to executing along branch b,
we know that for any clause c,
(c satisfies Q) $\Rightarrow$ (c satisfies b)
Thus no branches $g_i \in \gamma$ from P_w can satisfy Q.
Thus by (2), only branches $g_i \in \alpha$ from P_w can satisfy Q.
By (1) and Case (1), any tree through these branches is correct.

Disjoint branches γ and α combine to prove that all branches from node u are correct.

Figure 12: Proof of Strong Theorem For Decision Graphs.

PROVE: For a node P_u sharing code with node P_w,
such that $P_u \supseteq P_w$ and $\Gamma_u \supseteq \Gamma_w$,

$$\Gamma_u = \Gamma_w \cup \gamma$$

where $\gamma = \{g \in \Gamma_u \mid \exists <i, A> \in (P_u \setminus P_w).\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\}$ (1)

We can enumerate (tautologically) the sets Γ_u and Γ_w:

$$\Gamma_u = \{g \in \Gamma_u \mid \exists <i, A> \in P_u.\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\}$$

$$\Gamma_w = \{g \in \Gamma_w \mid \exists <i, A> \in P_w.\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\}$$

By definition of the indexer, we can redefine Γ_w:

$$\Gamma_w = \{g \in \Gamma_u \mid \exists <i, A> \in P_w.\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\} \tag{2}$$

Combining (1) and (2):

$$\begin{aligned}\Gamma_w \cup \gamma = &\{g \in \Gamma_u \mid \exists <i, A> \in (P_u \setminus P_w).\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\} \cup \\ &\{g \in \Gamma_u \mid \exists <i, A> \in P_w.\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\} \\ = &\{g \in \Gamma_u \mid \exists <i, A> \in (P_u \setminus P_w + P_w). \\ &\qquad \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\}\end{aligned} \tag{3}$$

Since $P_u \supseteq P_w$ (given):

$$P_u = P_w + P_u \setminus P_w$$

Thus by (3),

$$\Gamma_w \cup \gamma = \{g \in \Gamma_u \mid \exists <i, A> \in P_u.\ \exists\, g' \in A.\ \tilde{\forall}.\ g \Longleftrightarrow g'\} = \Gamma_u$$

Thereby proving the lemma. The situation is illustrated below.
The potential overlap in γ and Γ_w are those tests that are identical in both local and non-local code.

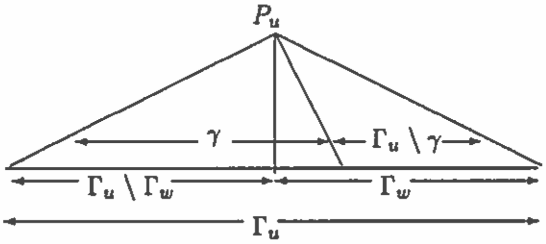


Figure 13: Lemma (2): Local and Nonlocal (Shared) Code Completely Cover Parent.