J. Symbolic Computation (1989) **8**, 605–627

First Order Compiler : A Deterministic Logic Program Synthesis Algorithm

Taisuke SATO
Electrotechnical Laboratory
1-1-4, Umezono, Tsukuba
Ibaraki 305, Japan

Hisao TAMAKI
Ibaraki University
4-12-1, Nakanarusawa,Hitachi
Ibaraki 316, Japan

(Received 25 March 1988)

We present the *first order compiler*, a deterministic algorithm that can compile logic programs augmented with universally quantified implicational goals into definite clause programs. It is based on the deductive unfold/fold transformation applied to *universal-continuation forms*, realization of continuation concept in logic programming. The compiled program is assured to be partially correct wrt the source program.

0747–7171/89/120605+23 $03.00/0

1 Introduction

This paper[1] purports to present the *first order compiler*, a completely deterministic algorithm for logic program synthesis. It has been developed in an attempt to reinforce logic programming by adding as goals negations, or more generally universally quantified implications in a way that is logically sound and practically meaningful. Using this expressive power made available by the compiler, one can write programs at higher level, closer to his/her intention.

An input to the first order compiler is a *first order program*, i.e. a finite set of clauses whose body may have universally quantified implicational goals as well as atomic goals. If the compilation successfully terminates, the output will be a definite clause program runnable on Prolog [7], which is guaranteed to be partially correct wrt the input program. Unfortunately, it can happen that the compilation terminates with failure due to the lack of logical power. In any case however, we can see the result of compilation in finite amount of time.

Thus, one can write a goal such as

$$\forall Y(p(X,Y) \rightarrow q(Y,Z))$$

and run it when the compilation is successful. It roughly means; for all Y such that $p(X,Y)$, do $q(Y,Z)$. So it works as a sort of for-all construct. However, it should be emphasized that the goal is not a mere for-all construct but a *declarative for-all construct*, implemented with logical rigor, offering flexible programming on the basis of logical variables. For example, in the above goal, the values of X and Z need not be determined prior to execution. In other words, we can get answer substitutions [18] for X and Z making the goal true.

As compilation goes, the first order compiler scans each clause in a source program. When it detects a universally quantified implicational goal, it does not immediately tackle it. Instead, it chooses a more general pattern, a formula called a *universal continuation form* [22,23], and then tries to synthesize a definite clause program for the latter by *unfold/fold deduction*[2]. The required program is obtained by specialization.

Since the compiler is designed, for the sake of efficiency and termination, to only perform very limited type of logical deductions such as unfold/fold deduction and the introduction of new predicates defined by universal continuation forms, it immediately aborts the compilation when a formula appears which fits none of its deduction patterns. Despite the possibility of compilation failure however, because those universally quantified implicational goals that seem meaningful as

[1] A Japanese version of this paper has also been submitted for publication to Japan Society for Software Science and Technology.

[2] Unfold/fold deduction means unfold/fold transformation by means of logical deduction [3,5,8,10,11,30]. Unfolding means one step symbolic execution, the replacement of a procedure call (atom) by a procedure body (complex formula). Folding is the opposite operation.

programming constructs are actually compilable, according to our experiences, it may make sense to allow them as new programming constructs and to regard the augmented language as a logic programming language built on top of Prolog such that programs are compiled by the first order compiler.

In the rest of this section, we would like to mention related works, though not necessarily exhaustively.

Unfold/fold transformation [3] in logic programming seems to have first appeared in a deductive framework in [5] in which Clark and Sickel showed that it is possible to eliminate quantifiers by folding to obtain definite clauses from first order formulae. Later on, the idea has been intensively studied by many researchers [8,10,11,30] and is now one of the standard methods for logic program synthesis.

The technique of unfold/fold transformation was transferred to logic programming, especially with interest focused on the least model semantics [18], and formulated as meaning preserving transformation systems for logic programs [14,27,28]. Sato and Tamaki showed that they are usable for logic program synthesis when combined with the Negation Technique [2,20]. In [13] however, Kanamori and Horiuchi showed a more direct method. They proposed a meaning preserving unfold/fold transformation system designed for logic program synthesis from (restricted type of) first order formulae.

All the approaches mentioned so far are nondeterministic, or involving large search spaces, and hence it is hard to adopt them as "compilation techniques" of first order formulae. Efforts also have been paid for discovering deterministic methods for logic program synthesis. The simplest one would be the method proposed by Lloyd and Topor [19] which is based on the repetitive use of the *negation-as-failure* inference rule [6,18]. The rule allows one to infer $\neg p$ when the inference of p failed. Unfortunately, for the final result to be logically sound, $\neg p$ must be *ground*, i.e. including no free variables. Therefore, as long as we pursue logical correctness, no variable bindings can be obtained from negative goals by the negation-as-failure approach.

The first order compiler [21,22,23] we shall present in this paper has many in common with those methods mentioned above. It, however, fundamentally differs from them in that the synthesis process is completely deterministic and automated. It also differs from the negation-as-failure approach in that it enables one to obtain correct answer substitutions [18] from negative goals. In addition, since the result of compilation is faithful to the procedural reading (which is compatible with Prolog execution order, see Section 2), it is not hard to anticipate how the compiled program runs. These features of our compiler make first order programs usable for actual programming. Partial correctness, namely, that the computed goal is a logical consequence of the completion [6] of the source program, is guaranteed [22] (see Theorem 5.1).

A similar method based on unfold/fold deduction is recently proposed by Dayantis [8], in which mechanization is achieved by choosing a specific class of

programs. It corresponds, in our framework, to the synthesis of a definite clause programs for a goal $\forall X_1, \ldots, X_n(A \rightarrow B)$ such that the A's predicate is defined by a certain class of (almost) definite clause programs and B is any relation.

In Section 2, we formally define first order programs and give a procedural interpretation to them. In Section 3, a detailed compilation example is presented. The synthesis algorithm is described in Section 4 and its correctness in Section 5. Section 6 is the conclusion. The reader is assumed to be familiar with logic programming and unfold/fold transformation [3,7,11,18,26,27].

2 Preliminaries

We first list several conventions valid throughout this paper. Variables are strings with an upper case letter at their heads. Other strings represent predicate, functional, and logical constants. **f** is a propositional constant denoting falsity and always fails as a goal. Likewise **t** denotes truth, always succeeding as a goal. We assume that "=" represents syntactic identity. Unless otherwise stated, E, F stand for syntactic variables for first order formulae, p, q for predicate symbols and x, y for distinct variable sequences. So when $x = X_1, \ldots, X_n (0 \leq n)$, $\forall x$ denotes the quantification $\forall X_1, \ldots, X_n$. It is also our convention that A, B stand for atoms, u for a term and s, t for term sequences (all possibly suffixed). We stipulate that $F \leftarrow E$ and $E \rightarrow F$ with E empty denote F.

Sequences and multi-sets are often deliberately confused when the distinction does not matter. Thus, $s = u_1, \ldots, u_n$ denotes a multi-set $\{u_1, \ldots, u_n\}$ as well. The length (cardinality) of s is denoted by $|s|$. We use (s, t), or s, t for the concatenation of s and t, and occasionally A, B for the conjunction $A \wedge B$. When $s = s_1, \ldots, s_n$, $t = t_1, \ldots, t_n$ and each $s_i, t_j (1 \leq i, j \leq n)$ is an individual term, $s = t$ abbreviates $(s_1 = t_1) \wedge \ldots \wedge (s_n = t_n)$ and $s \neq t$ abbreviates $(s = t \rightarrow \mathbf{f})$. In case of $|s| = |t| = 0$, $s = t$ denotes **t**, $s \neq t$ **f** respectively.

Let E be an arbitrary expression (term, formula, sequence whatsoever). Then $Fvar(E)$ denotes the set of free variables occurring in E. Hence, $x \subseteq Fvar(E)$ expresses that every variable in x is free in E. $Fvar(E_1, E_2)$ denotes the set $Fvar(E_1) \cup Fvar(E_2)$. Finally $S_1 \setminus S_2$ stands for set subtraction as usual.

2.1 First order program

A *first order program* is a finite set of *first order clauses*. A first order clause is a first order formula of the form

$$A \leftarrow F$$

where A is an atom (atomic formula) and F is empty or a formula in which any universally quantified subformula is of the form $\forall y(F_1 \rightarrow F_2)$ such that $y \subseteq Fvar(F_1)$. In addition, for simplicity, we stipulate that neither "=" nor **t** occurs in the clause and further that **f** is only allowed to occur in the body. Note that

$\forall x(F_1 \rightarrow F_2)$ includes implication $F_1 \rightarrow F_2$ and negation $\neg F_1 = (F_1 \rightarrow \mathbf{f})$ as special cases. If the head A contains a predicate p, the clause is said to be *a clause about p.*

A *basic program* is a finite set of definite clauses and *extended clauses*. An extended clause is a first order clause whose body has the form $\forall y(B_1 \rightarrow B_2)$ where both B_1 and B_2 are atoms and $y \subseteq Fvar(B_1)$ holds.

From here on, we confine our attention to basic programs and the compilation algorithm will be presented for this class. This does not mean any loss of generality. For, by introducing as many predicates as necessary (definitional extension), we can always transform any first order program to a basic one while preserving the logical meaning of the original program as axioms [19].

The output of the first order compiler is a definite clause program that may include goals of the form $\forall y(s \neq t)$. At first sight, they seem to pose serious problems with their execution, but for the reasons explained below, problems are avoidable.

Firstly, suppose as usual that our domain of discourse is the *Herbrand universe* (the set of all ground terms generated from the function symbols occurring in a program). $\forall y(s \neq t)$ then represents a recursive relation (unification failure) wrt its free variables over the universe and hence it becomes possible to compute the same relation by some definite clause program. Thus, as long as we choose Herbrand model semantics (term model semantics), we can obtain a completely positive program.

Secondly, there is a sound and efficient implementation for $\forall y(s \neq t)$ using negation-as-failure; we can, in Prolog [7], implement it as $\backslash+(s = t)$[3] since its success, i.e. the unification failure of s and t is equivalent to $\forall v(s \neq t)$ where $v = Fvar(s,t)$ and the latter implies the truth of $\forall y(s \neq t)$ regardless of whatever instantiation is made by the subsequent computation to variables in $v \setminus y$.

Moreover, this implementation can be complete. That is, if $\backslash+(s = t)$ fails and if an extra condition $Fvar(s,t) \subseteq y$ is satisfied also, we can conclude that $\exists y(s = t)$, i.e. $\neg\forall y(s \neq t)$ is a true sentence, and hence remains true through the rest of computation.

Thus, though the negation-as-failure inference is not logically sound in general, $\backslash+(s = t)$ is an exception. It is a logically sound implementation for $\forall y(s \neq t)$ that works *even when* $\backslash+(s = t)$ *contains variables at the time of execution.* It is also complete if $Fvar(s,t) \subseteq y$ is guaranteed to hold at the time of execution.

In summary, the compiled program can be run on Prolog and we have no need, as far as successful computation is concerned, to check whether $\backslash+(s = t)$ is ground or not at run time in order to guarantee the correctness of the final answer to a top level query (see Theorem 5.1).

Of course, it is possible to have another implementation for $\forall y(s \neq t)$ based

[3] $\backslash+p$ is the Prolog notation for negation-as-failure. $\backslash+p$ succeeds if p fails and fails if p succeeds.

on more elaborated mechanisms such as "diff" and "freeze" [9]. But we do not go into the detail of such implementation matter.

2.2 Procedural interpretation

So we allow a goal of the form, say,

$$\forall Y(p(X,Y) \rightarrow q(Y,Z))$$

in our first order programs. Next task is to make it clear how the (compiled) goal runs. Let us try to explain it in terms of the Prolog interpreter. For simplicity, we assume that $p(X,Y)$ and $q(Y,Z)$ are defined by some definite clause program. We denote the execution of a goal p by ?-p.

First let us note that the above formula is equivalent to $\neg\exists Y(p(X,Y)\wedge\neg q(Y,Z))$ and is partially implemented in Prolog by $\backslash+(p(X,Y), \backslash+q(Y,Z))$ using negation-as-failure. Though this implementation works well when both X and Z are instantiated to ground terms at the time of execution, it won't help at all if we want to get those variables instantiated through the execution of this goal. The control flow in this implementation, however, forms a basis for the following procedural interpretation, where instantiations to the free variables X and Z are taken into account.

[step 1] Get the next value t of Y by ?-$p(X,Y)$. Then check t and abort the whole computation unless every variable in t occurs in the current value of X. Else,
[step 2] Do ?-$q(t,Z)$
[step 3] Go-to **[step 1]** by backtracking with the values of X and Z preserved.

The check at **[step 1]** is inserted to assure the logical soundness of our interpretation[4]. Note that we preserve at **[step 3]** the variable bindings for X and Z created so far despite backtracking upon the completion of ?-$q(t,Z)$. Even if ?-$p(X,Y)$ fails somewhere at **[step 1]**, the value of X will be preserved. Thus, X and Z will be monotonously instantiated every time the loop is tried. By contrast, we unbind Y upon each backtracking at **[step 3]**[5].

As this example suggests, the computation (supposed to be realized by the compiled program) is understandable in terms of the Prolog interpreter, hence, if one is familiar with Prolog, (s)he might not find it difficult to use the above goal, or more generally universally quantified implications as programming constructs.

[4]Consider the case where p and q have respectively $p(X,Y)$ and $q(0,Z)$ as their defining clauses. Without the check, our interpretation would let the goal $\forall Y(p(X,Y) \rightarrow q(Y,Z))$ succeed, which is equivalent to $\forall Y(Y=0)$, a sentence that is false in general.

[5]The difference is due to the fact that X and Z are free whereas Y is universally quantified in the goal.

The procedural interpretation of the general case should be understood by regarding the execution of the antecedent and the consequent of the implication as recursive calls to the interpreter being defined. We omit the details.

2.3 Universal continuation form

Here, we would like to briefly comment on *universal continuation.* It is an adaptation of the continuation concept elementary in functional programming to the context of logic programming and forms the background of our compilation method.

Suppose that we have a functional program realizing function $f(X)$. Then it is always possible to mechanically convert the program to one realizing function f' such that

$$f'(X, C) = C(f(X)).$$

C is a λ term representing *the rest of computation when f(X) has been computed.* Programming style based on the use of continuation is called *continuation passing style computation.* It greatly facilitates implementation of complicated control structures such as coroutine.

An *existential continuation form* [24,29] shown below corresponds to the above equation and characterizes the direct counter part in logic programming of the continuation concept.

$$p'(X, C) \leftrightarrow \exists Y(p(X, Y) \wedge cont_p(Y, C)).$$

Here the correspondence between $p(X, Y)$ and $f(X) = Y$ is assumed. C is a usual first order term representing the rest of computation and $cont_p(Y, C)$ is a substitute for functional application. It represents continuation passing style computation taking nondeterminacy into account. The use of existential continuation in logic programming was initiated by Ueda when he considered "all-solution" programs [29]. Applications are described in [24,29].

On the other hand, in logic programming, we have yet another type of continuation called *universal continuation.* It is characterized by a *universal continuation form* [21,22,23]:

$$p'(X, C) \leftrightarrow \forall Y(p(X, Y) \rightarrow cont_p(Y, C))$$

This form says that the continuation C is to be computed, unlike the existential continuation form, *for every successful computation* of $p(X, Y)$. In other words, we are regarding the SLD tree [18] for $p(X, Y)$ in the universal continuation form as an AND-tree, contrary to the ordinary view.

3 Compilation example

We explain, using a tiny example, how the first order compiler works.

3.1 An example of compiled program

Suppose that one wants to say that every member of list L is either 1 or 2. The following program seems a reasonable answer.

$$one\_two(L) \leftarrow \forall Y(mem(Y, L) \rightarrow Y = 1 \vee Y = 2) \tag{3.1}$$

$$mem(Y, [Y|Z]) \tag{3.2}$$
$$mem(Y, [U|V]) \leftarrow mem(Y, V) \tag{3.3}$$

Our procedural interpretation predicts that in the compiled program, L will be decomposed recursively by clause (3.2) and (3.3) while the variables in L are instantiated by $Y = 1 \vee Y = 2$. By compilation, the following clauses are generated to compute $one\_two(L)$.

$$one\_two(L) \leftarrow mem'(L, f_0) \tag{3.4}$$

$$mem'(L, C) \leftarrow \forall Y, Z(L \neq [Y|Z]) \tag{3.5}$$
$$mem'([Y|Z], C) \leftarrow cont_{mem}(Y, C) \wedge mem'(Z, f_1(C)) \tag{3.6}$$

$$cont_{mem}(Y, f_0) \leftarrow Y = 1 \vee Y = 2 \tag{3.7}$$
$$cont_{mem}(Y, f_1(C)) \leftarrow cont_{mem}(Y, C) \tag{3.8}$$

Since definite clauses remain intact by compilation, the compiled program S_c consists of {(3.2), (3.3), (3.4), (3.5), (3.6), (3.7), (3.8) }. Some remarks are in order.

By inspection, we can see not only is S_c as it is usable for ground L for checking whether it is a list containing only 1 and 2, but also is usable for non-ground L. For example, if a query ?-$one\_two([A, B])$ is given to the Prolog interpreter, it will return $[A, B] = [1, 1], [1, 2], [2, 1], [2, 2]$ in this order. Secondly, note that if L is assured to be or declared as a list, $\forall Y, Z(L \neq [Y|Z])$ equals $L = []$. In this case we obtain a completely positive program that works even for ?-$one\_two(L)$.

The compiled program S_c includes new predicate symbols mem', $cont_{mem}$ and new function symbols f_0, f_1, not existent in the source program. They are all introduced by the first order compiler. We call mem' a *closure predicate* and $cont_{mem}$ a *continuation predicate* according to their roles. f_0 and f_1 are examples of *continuation functions*. They are introduced corresponding to textual positions in the source program. f_0 corresponds to the $mem(Y, L)$, the antecedent of the body of clause (3.1) and f_1 to $mem(Y, V)$, the body of clause (3.3). They work as return addresses.

The variable C conveys continuation, hence called a *continuation variable*. It is always bound to a *continuation term*, i.e. one whose functor is a continuation function symbol. In this example, the term will be of the form $f_1(\ldots f_1(f_0)\ldots)$, representing a stack such that f_1 is pushed down when the mem' clause (3.6) is

called and f_0 or f_1 is popped up when the $cont_{mem}$ clause (3.7) or clause (3.8) is called respectively.

3.2 Compilation as deduction

The compiled program S_c is a logical consequence of the following formulae (axioms) together with certain equality axioms characterizing the Herbrand universe of S.

$$one\_two(L) \leftrightarrow \forall Y(mem(Y,L) \rightarrow Y = 1 \vee Y = 2) \tag{3.9}$$

$$mem(Y,L) \leftrightarrow \exists Z(L = [Y|Z]) \vee \exists U,V(L = [U|V] \wedge mem(Y,V)) \tag{3.10}$$

$$cont_{mem}(Y,f_0) \leftrightarrow Y = 1 \vee Y = 2 \tag{3.11}$$
$$cont_{mem}(Y,f_1(C)) \leftrightarrow cont_{mem}(Y,C) \tag{3.12}$$

$$mem'(L,C) \leftrightarrow \forall Y(mem(Y,L) \rightarrow cont_{mem}(Y,C)) \tag{3.13}$$

Clause (3.9) and (3.10) are respectively the *iff definitions* [6,18] of *one_two* and *mem* clauses in the source program S. Clause (3.11), (3.12) are auxiliary clauses to compute continuation. Clause (3.13) is an example of universal continuation forms. Formally, it is a universal continuation form for *mem* under *mode pattern* $mem(-,+)$ (see Section 4). These formulae are automatically generated by the compiler from the source program.

We are going to demonstrate, in detail, how to derive $\{(3.4),\ldots,(3.8)\}$ from $\{(3.9),\ldots,(3.13)\}$ by unfold/fold deduction. We first compile the extended clause (3.9). By folding $(Y = 1 \vee Y = 2)$ of clause (3.9) into $cont_{mem}(Y,f_0)$ using clause (3.11), we get

$$one\_two(L) \leftrightarrow \forall Y(mem(Y,L) \rightarrow cont_{mem}(Y,f_0)).$$

Then using clause (3.13), this is further folded into

$$one\_two(L) \leftrightarrow mem'(L,f_0)$$

from which clause (3.4) results. We next move to the compilation of clause (3.13).

After unfolding it at $mem(Y,L)$ using clause (3.10), and using a valid propositional schema (a formula pattern whose instantiation is always true): $(A \vee B \rightarrow C) \leftrightarrow ((A \rightarrow C) \wedge (B \rightarrow C))$, we have

$$\begin{aligned} mem'(L,C) \leftrightarrow \quad & \forall Y(\exists Z(L = [Y|Z]) \rightarrow cont_{mem}(Y,C)) \wedge \\ & \forall Y(\exists U,V(L = [U|V] \wedge mem(Y,V)) \rightarrow cont_{mem}(Y,C)). \end{aligned}$$

Then, by appealing to the schemata: $(A \wedge B \rightarrow C) \leftrightarrow (A \rightarrow (B \rightarrow C))$ and $\forall Y(\exists X F_1 \rightarrow F_2) \leftrightarrow \forall Y,X(F_1 \rightarrow F_2)$ and by folding $cont_{mem}(Y,C)$ into $cont_{mem}(Y,f_1(C))$ using clause (3.12), this formula is transformed to

$$mem'(L,C) \leftrightarrow \begin{array}{l} \forall Y,Z(L=[Y|Z] \rightarrow cont_{mem}(Y,C)) \wedge \\ \forall U,V(L=[U|V] \rightarrow \underline{\forall Y(mem(Y,V) \rightarrow cont_{mem}(Y,f_1(C)))}) \end{array}$$

and then to

$$mem'(L,C) \leftrightarrow \begin{array}{l} \forall Y,Z(L=[Y|Z] \rightarrow cont_{mem}(Y,C)) \wedge \\ \forall U,V(L=[U|V] \rightarrow mem'(V,f_1(C))) \end{array}$$

by folding the underlined formula (with a suitable matching) using clause (3.13).

Finally, recalling that for any terms s,t and formula F,

$$\forall X_1,\ldots,X_m(s=t \rightarrow F) \leftrightarrow \forall X_1,\ldots,X_m(s \neq t) \vee \exists X_1,\ldots,X_m(s=t \wedge F)$$

holds over the Herbrand universe provided that $\{X_1,\ldots,X_m\} \subseteq Fvar(t)$ and $\{X_1,\ldots,X_m\} \cap Fvar(s) = \phi$, we reach

$$mem'(L,C) \leftrightarrow \begin{array}{l} \{\forall Y,Z(L \neq [Y|Z]) \vee \exists Y,Z(L=[Y|Z] \wedge cont_{mem}(Y,C))\} \wedge \\ \{\forall U,V(L \neq [U|V]) \vee \exists U,V(L=[U|V] \wedge mem'(V,f_1(C)))\}. \end{array}$$

By distributing $\wedge$ and by cleaning up with variable renaming, it is not hard to see that clause (3.5) and (3.6) are derivable from this formula. Other compiled clauses, (3.7) and (3.8), are apparently derivable from clause (3.11) and (3.12). The first order compiler automatically carries out all such deductions.

4 Compilation algorithm

In this section, we describe the compilation algorithm of the first order compiler. Before proceeding to the description of the algorithm, we add some notations and terminology.

4.1 Mode pattern

An atom of the form $p(e_1,\ldots,e_k)$ where e_i $(1 \leq i \leq k)$ is either $+$ or $-$ is called a *mode pattern* for p [23]. Let $\pi = p(e_1,\ldots,e_k)$ be a mode pattern. Consider an atom $p(u_1,\ldots,u_k)$. We call u_i $(1 \leq i \leq k)$ an *input argument* if e_i is $+$, or otherwise an *output argument*, of $p(u_1,\ldots,u_k)$ under π. Let $(s_1,\ldots,s_m)$ (resp. $(t_1,\ldots,t_n)$) ($m+n=k$) be an order preserved subsequence of $(u_1,\ldots,u_k)$, consisting of the input (resp. output) arguments under π. The sequence $(s_1,\ldots,s_m)$ (resp. $(t_1,\ldots,t_n)$) is called the *input* (resp. *output*) *sequence* of $p(u_1,\ldots,u_k)$ *under* π. In what follows, we adopt labeled atoms such as

$$\begin{array}{c} p(e_1,\ldots,e_k) \\ p(s,,t) \end{array}$$

to express that there is some atom, for instance $p(u_1,\ldots,u_k)$, whose input (resp. output) sequence under $\pi = p(e_1,\ldots,e_k)$ is s (resp. t). We call it the I/O *form*

of $p(u_1, \ldots, u_k)$ under π. An I/O form can substitute the original atom. But since it is quite cumbersome to always write such labeled atoms, we shall omit mode patterns when π is understood from the context.

An atom $p(Z_1, \ldots, Z_k)$ is called a *most general atom for* p if $Z_1, \ldots, Z_k$ are mutually distinct variables. Let $p(x,,y)$ be the I/O form under some mode pattern π of a most general atom for p. It is called a *most general* I/O *form for* p *under* π.

4.2 Algorithm

The first order compiler is composed of three procedures, a main procedure **PROGRAM-COMPILE** and two sub-procedures, **GET-MODE-PATTERN** and **GOAL-COMPILE**. **GOAL-COMPILE** is the only procedure that may fail. So we stipulate that **PROGRAM-COMPILE** fails in the compilation if the failure of **GOAL-COMPILE** occurs. Recall that a basic program is the union of definite clauses and extended clauses.

PROGRAM-COMPILE

Input: a basic program S
Output: a definite clause program S_c or a report of "failure"

[STEP 1] Initialization

Split S into the set of definite clauses Σ_d and the set of extended clauses Σ_e. Put $\Pi = \phi$ (Π stores mode patterns). Set up an empty queue Γ (Γ stores intermediate results of compilation). Further put $\Sigma_c = \Sigma_d$ (Σ_c stores compiled definite clauses).

[STEP 2] Until Σ_e becomes empty, repeat the following.

Remove an extended clause from Σ_e. Let it be $A \leftarrow \forall y_1(p(u_1, \ldots, u_n) \rightarrow B)$.

Generating mode pattern:

First by applying procedure **GET-MODE-PATTERN** to $\forall y_1(p(u_1, \ldots, u_n) \rightarrow B)$, get the mode pattern π for p in $\forall y_1(p(u_1, \ldots, u_n) \rightarrow B)$. Add π to Π.

Generating closure clause and continuation clause:

Next, let $p(x,,y)$ be a most general I/O form for p under π. x, y are sequences of new variables. Prepare, uniquely to π, new predicate symbols p' (of arity $|x| + 1$) and $cont_p$ (of arity $|y| + 1$). p' is called the *closure predicate* and $cont_p$ the *continuation predicate* corresponding to π, respectively. We call the following clause

$$p'(x, C) \leftarrow \forall y(\ p(x,,y)^{\pi} \rightarrow cont_p(y, C))$$

a *universal continuation form* for p under π. C is a new variable called a *continuation variable.* Construct a universal continuation form for p under π and enqueue it into Γ.

In the following, for convenience, we call a clause a *closure clause* (resp. *continuation clause*) if the head contains a closure predicate (resp. continuation predicate).

Let $p(s,,t)$ be the I/O form of $p(u_1,\ldots,u_n)$ under π. $y_1 \subseteq Fvar(t)$ holds from the definition of procedure **GET-MODE-PATTERN**. Construct a continuation clause

$$cont_p(y, f(w)) \leftarrow \forall y_1(y = t \rightarrow B).$$

where $w = (Fvar(t) \cup Fvar(B)) \setminus y_1$ and f is a new function symbol of arity $|w|$ called a *continuation function symbol.* We assume w is alphabetically sorted. Enqueue it into Γ.

Generating definite clause:

Finally construct the following definite clause

$$A \leftarrow p'(s, f(w))$$

and add it to Σ_c.

[STEP 3] Until Γ becomes empty, repeat the following.

Dequeue a formula from Γ. Let it be E.

(Case 1) E is a closure clause (universal continuation form)

$$p'(x, C) \leftarrow \forall y(\ p(x,,y)\ \overset{\pi}{\rightarrow}\ cont_p(y, C)).$$

Let $p(s_1,,t_1) \leftarrow E_1, \ldots, p(s_n,,t_n) \leftarrow E_n$ be an enumeration of clauses about p in the source program S ($0 \leq n$, each head is in an I/O form under π). If $n = 0$, add a unit clause $p'(x, C)$ to Σ_c. Else suppose $n > 0$. First for each $i\,(1 \leq i \leq n)$, put

$$\begin{aligned} z_i &= Fvar(p(s_i,,t_i) \leftarrow E_i) \\ v_i &= Fvar(s_i) \\ w_i &= z_i \setminus v_i. \end{aligned}$$

Construct the formula (E_i may be empty) below

$$\forall w_i(E_i \rightarrow cont_p(t_i, C))$$

and apply sub-procedure **GOAL-COMPILE** to it. Let the result be $G_i (1 \leq i \leq n)$. Now construct the following formula

$$\begin{aligned} p'(x, C) \leftarrow \ & \{\forall v_1(x \neq s_1) \vee \exists v_1(x = s_1 \wedge G_1)\} \wedge \\ & \{\forall v_2(x \neq s_2) \vee \exists v_2(x = s_2 \wedge G_2)\} \wedge \\ & \cdots \\ & \{\forall v_n(x \neq s_n) \vee \exists v_n(x = s_n \wedge G_n)\} \end{aligned}$$

and convert the right hand side into a disjunctive form. Let it be

$$p'(x, C) \leftarrow F_1 \vee, \ldots, \vee F_m.$$

Add to Σ_c each $p'(x, C) \leftarrow F_j$ $(1 \leq j \leq m)$.

(Case 2) E is a continuation clause

$$cont_p(y, f(w)) \leftarrow \forall y_1(y = t \rightarrow F).$$

$y_1 \subseteq Fvar(t)$ holds. F is either an atom or a universally quantified implication. If F is an atom, put $G = F$. Else, apply sub-procedure **GOAL-COMPILE** to F to get the result G. Generate two clauses

$$\begin{aligned} & cont_p(y, f(w)) \leftarrow \forall y_1(y \neq t) \ and \\ & cont_p(y, f(w)) \leftarrow y = t \wedge G. \end{aligned}$$

Add them to Σ_c.

[STEP 4] Remove $\exists$ symbol from Σ_c (with variable renaming if necessary). Apply, optionally, to Σ_c simplifications such as

- the execution of goals of the form $s = t$,
- the reduction of goals of the form $\forall y(s \neq t)$ to **t** in case of s and t being not unifiable or to a simpler form using sort information[6],
- logical simplification concerning **t** and **f**,
- removing clauses containing a failed goal such as **f**

that can improve computational behavior. As for the detailed treatment of equality and inequality, see [15,16]. Return Σ_c as the compiled program S_c.

GET-MODE-PATTERN

Input: a formula $\forall y(A \rightarrow E)$ where A is an atom.
Output: a mode pattern π for the predicate contained in A

[6] An example is the replacement of $\forall Y, Z(L \neq [Y|Z])$ with $L = []$ in case L is known to be a list.

Let the input be $\forall y(p(u_1,\ldots,u_n) \rightarrow E)$. Make a mode pattern $\pi = p(e_1,\ldots,e_n)$ for p by putting $e_i = -$ if u_i includes a variable quantified by $\forall y$, else put $e_i = +$ for each $i\,(1 \leq i \leq n)$. Return π.

GOAL-COMPILE

Input: a formula $\forall y(E \rightarrow F)$
Output: an atom or a formula $\exists z(A \wedge B) \vee C$ where A, B and C are atoms.

(Case 1) E is empty.

In this case F is an atom. If $|y| = 0$, then return F, else fail.

(Case 2) E is an atom.
(Case 2-1) E is **f**. Return **t**.
(Case 2-2) E is an atom other than **f**.

In this case F is either an atom or a universally quantified implication. Write E as $q(u_1,\ldots,u_m)$. Get the mode pattern π for q by applying procedure **GET-MODE-PATTERN** to $\forall y(q(u_1,\ldots,u_m) \rightarrow F)$. Add π to Π and enqueue into Γ a universal continuation form for q under π *only if* π *was not stored in* Π before.

Let q', $cont_q$ be respectively the closure predicate and the continuation predicate corresponding to π. Also let $q(s,,t)$ be the I/O form of $q(u_1,\ldots,u_m)$ under π. $y \subseteq Fvar(t)$ holds. Construct a continuation clause

$$cont_q(y_1, g(w)) \leftarrow \forall y(y_1 = t \rightarrow F)$$

where y_1 is a sequence of new distinct variables whose length is $|t|$, $w = (Fvar(t) \cup Fvar(F)) \setminus y$ and and g is a new continuation function symbol of arity $|w|$. We assume that w is alphabetically sorted. Enqueue the clause into Γ. Return $q'(s, g(w))$.

(Case 3) E is a conjunction.

In this case F is an atom. Write the input as $\forall y((B \wedge E') \rightarrow F)$ where B is an atom and transform it into $\forall y_1(B \rightarrow \forall y_2(E' \rightarrow F))$ where $y_1 = y \cap Fvar(B)$ and $y_2 = y \setminus y_1$. Apply procedure **GOAL-COMPILE** to the transformed goal and return the result.

(Case 4) E is none of the above.

The input formula is written as $\forall y(\forall z(B \rightarrow C) \rightarrow F)$ where B, C and F are all atoms. If $|y| > 0$, fail. If $|y| = 0$, transform it into

$$\exists z(B \wedge (C \rightarrow \mathbf{f})) \vee F$$

and put $G = (C \rightarrow \mathbf{f})$. Apply procedure **GOAL-COMPILE** to G to get the result C'. Return $\exists z(B \wedge C') \vee F$.

4.3 Compilation failure

Compilation failure occurs[7] when the compiler is required to deal with a formula of the form $\forall y A$ with $|y| > 0$ for A atom (see **(Case 1)** of **GOAL-COMPILE**) or $\forall y(A \vee B)$ with $|y| > 0$ (see **(Case 4)** of **GOAL-COMPILE**).

The following is a typical example in the former case[8]

$$r(X,Z) \leftarrow \forall Y(p(X,Y) \rightarrow q(Y,Z))$$
$$p(X,Y)$$
$$q(0,Z)$$

in which, procedurally speaking, even after the success of $p(X,Y)$ in $\forall Y(p(X,Y) \rightarrow q(Y,Z))$, the output argument Y still remains unbound to the input argument X so that the consequent part, the goal to be proved next, becomes $\forall Y q(Y,Z)$.

In fact, the compilation of this program proceeds as follows. First, in **PROGRAM-COMPILE**, $p'(X,C) \leftarrow \forall Y(p(X,Y) \rightarrow cont_p(Y,C))$ is introduced in **[STEP 2]** as the universal continuation form for p under $p(+,-)$. Then at **(Case 1)** in **[STEP 3]**, a call to **GOAL-COMPILE** with $\forall Y cont_p(Y,C)$ occurs, ending with the immediate failure at **(Case 1)** of **GOAL-COMPILE**.

Though it is possible to formally characterize the class of compilable programs, such a characterization would have essentially the same complexity and structure as the compilation algorithm itself, and thus would be of little interest.

5 Correctness

5.1 Termination

First of all, we would like to show that the compilation algorithm always terminates. Note that **GET-MODE-PATTERN** and **GOAL-COMPILE** always terminate. So the only possibility of non-termination lurks in an infinite processing of queue Γ in **[STEP 3]** in **PROGRAM-COMPILE**. As easily seen from the algorithm, Γ must be dequeued infinitely many times for non-termination to occur.

However, firstly because duplication is checked with Π at **(Case 2-2)** in **GOAL-COMPILE** by examining mode patterns so that the universal continuation forms enqueued into Γ are all different (modulo closure/continuation predicates symbols and variable names), and secondly because there are only finitely many universal continuation forms (ditto) (a program has only finitely many predicates), it is impossible to dequeue Γ infinitely.

[7] Universal formulas can define relations not in the class of recursively enumerable relations [26] whereas the compiled program can only define recursively enumerable ones. Hence, it is not unnatural that the compiler fails for some first order programs.

[8] This program was once referred to implicitly when we talked about procedural interpretation.

5.2 Partial correctness

We state in this section a logical relation between an input program S and the compiled program S_c. Before describing it, we prepare notations. Let S be a set of first order clauses and p a predicate appearing in S. Also let $p(s_i) \leftarrow E_i\,(1 \leq i \leq m)$ be an enumeration of the clauses about p in S and $v_i\,(1 \leq i \leq m)$ be an enumeration of free variables in the i-th clause. We use $p(x)$ for a most general atom for p.

Define p^* as follows. If $m = 0$, put $p^* = p(x) \leftrightarrow \mathbf{f}$. Else put $p^* = p(x) \leftrightarrow \exists v_1(x = s_1 \wedge E_1) \vee \ldots \vee \exists v_m(x = s_m \wedge E_m)$. p^* is called the *iff definition* of p in S [6,18]. All free variables in p^* are implicitly universally quantified in front of p^*. Put $S^* = \{p^* | \, p$ is a predicate appearing in $S.\}$. S^* is the set of all iff definitions of predicates in S.

Let K be a set of function symbols. Define $E_w(K)$ and $E_q(K)$ as

$$E_w(K) = \{f(x) = f(y) \rightarrow x = y | f \in K\} \cup \{f(x) \neq g(y) | f \neq g, f, g \in K\}$$

$$E_q(K) = E_w(K) \cup \{X \neq u | \; u \text{ is a term made up of } K,\; X \in Fvar(u)\}$$

All variables in $E_w(K)$ and $E_q(K)$ are implicitly universally quantified. Finally, put $\mathrm{comp}(S) = S^* \cup E_q(K)$ where K is the set of function symbols appearing in S.

By S_c?-A, we mean that a ground query A is given to a top-down interpreter such as Prolog. We consider the goal $\forall y(s \neq t)$ as one that succeeds if s and t are not unifiable and fails otherwise. It is important to note that neither s nor t is required to be ground at the time of execution. This treatment is sound (and complete if $Fvar(s,t) \subseteq y$ holds) as pointed out in Section 2.

When S_c?-A fails, we say that the failure is *safe* if every failed goal of the form $\forall y(s \neq t)$ satisfies $Fvar(s,t) \subseteq y$ at the time of execution.

Theorem 5.1 *Let S be a basic program containing at least one non-constant function symbol, S_c the result of the successful compilation, A a ground atom in the language of S. We have*

$\mathrm{comp}(S) \vdash A$	if	S_c?-A succeeds.
$\mathrm{comp}(S) \vdash \neg A$	if	S_c?-A fails and the failure is safe.

This theorem [22] guarantees the partial correctness of the compiled program S_c wrt the source program S[9]. Apparently, if $\mathrm{comp}(S)$ is consistent [25] and S_c?-A terminates either with success or with safe failure for any ground atomic goal A, we can replace "if" in Theorem 5.1 with "if and only if", which means the total correctness of the compiled program.

[9] It is also easy to see when an answer substitution θ is returned for a non-ground query A, we have S_c ?-B for any query B which is a ground instance of $A\theta$.

5.3 Proof of partial correctness

Since the proof of the above theorem is rather long [22], we only outline it. Suppose that a basic program S is successfully compiled into S_c and U_f, the set of function symbols appearing in S, includes at least one non-constant function symbol. Let C_f, $Clsr$, and $Cont$ respectively be the set of continuation function symbols, the set of closure clauses (= universal continuation forms) and the set of continuation clauses introduced during the compilation.

The proof has three steps **(I)**, **(II)** and **(III)** shown below (A is an arbitrary ground atom in the language of S).

(I) $S_c^* \cup E_w(C_f) \cup E_q(U_f) \vdash A$ if S_c?-A succeeds.
$S_c^* \cup E_w(C_f) \cup E_q(U_f) \vdash \neg A$ if S_c?-A fails safely.

(II) $\mathrm{comp}(S) \cup Clsr^* \cup Cont^* \cup E_w(C_f) \vdash S_c^* \cup E_w(C_f) \cup E_q(U_f)$

(III) $\mathrm{comp}(S) \vdash A$ iff $\mathrm{comp}(S) \cup Clsr^* \cup Cont^* \cup E_w(C_f) \vdash A$

(I) is proved similarly to [6]. **(II)** is equivalent to what the first order compiler does in a successful compilation. **(III)** is the point. For **(III)**, we prove that $\mathrm{comp}(S) \cup Clsr^* \cup Cont^* \cup E_w(C_f)$ is a conservative extension [26] of $\mathrm{comp}(S)$. To do so, it suffices to show that we can always extend a model M of $\mathrm{comp}(S)$ over a domain D to that of $\mathrm{comp}(S) \cup Clsr^* \cup Cont^* \cup E_w(C_f)$ over the same domain by adding to M an interpretation over D of function symbols in C_f satisfying $E_w(C_f)$ and that of continuation predicates and closure predicates satisfying $Cont^*$ and $Clsr^*$ respectively.

5.4 Conservative extension

Let M be a model of $\mathrm{comp}(S)$ and D its domain. By assumption, U_f includes at least one non-constant function symbol, say g. So $\mathrm{comp}(S)$ includes infinitely many equations of the form $X \neq g(u_1, \ldots, u_n)$ where $X \in Fvar(g(u_1, \ldots, u_n))$. As a result, D must be infinite. On the other hand, since $E_w(C_f)$ only requires that continuation functions are one-to-one and their ranges are disjoint, we can always find the required functions by dividing D into appropriately many disjoint subsets each of which has the same cardinality as D. Add to M such an interpretation for C_f satisfying $E_w(C_f)$ and let M' be the extended interpretation. M' satisfies $\mathrm{comp}(S)$ and $E_w(C_f)$.

As for finding an interpretation satisfying $Cont^*$, we do as follows. For the sake of simplicity, we assume that there is the only one continuation predicate in $Cont^*$ (generalization is easy). Then the only formula included in $Cont^*$ has the following form

$$cont_p(x) \leftrightarrow \Phi[\ldots cont_p \ldots](x).$$

Φ is a formula made up of continuation predicates (= $cont_p$ in this case), continuation functions in C_f, functions in U_f, predicates in S and no others (except "=").

Since every symbol in Φ is already interpreted by M' except $cont_p$, we can regard the above formula as an equation for $cont_p$. However, since $cont_p$ occurs *only positively* [13] in Φ by construction, it is easy to see that there exists an interpretation for $cont_p$ over D satisfying $cont_p(x) \leftrightarrow \Phi[\ldots cont_p \ldots](x)$. By adding that interpretation to M', we can extend M' to the interpretation M'' satisfying comp(S), $E_w(C_f)$ and $Cont^*$.

Finally consider $Clsr^*$. It is now just a set of definitions of closure predicates over D because all symbols other than closure predicates are already interpreted by M''. Therefore, by adding to M'' those interpretation for closure predicates defined by $Clsr^*$, we extend M'' to the interpretation M''' over D. Clearly, we have

$$M''' \models \text{comp}(S) \cup Clsr^* \cup Cont^* \cup E_w(C_f).$$

So we are done.

When all function symbols in S are constants, we need an additional condition on S for Theorem 5.1 to hold. A goal is said to be *primitive* if it is an atom or a formula of the form $\forall y(B_1 \rightarrow B_2)$ where B_1 and B_2 are atoms. If the body of a clause is a conjunction of primitive goals and if every free variable in the clause occurs somewhere in an atomic goal in the body, it is said to be *allowed* [4]. An allowed program is one such that every clause is allowed[10].

Let S be an allowed program. Suppose that by introducing as many predicate as necessary, we have converted S to a basic program S' (definitional extension [26]) and S' is compiled into S_c. Then Theorem 5.1 holds for S and S_c as well [22].

6 Conclusion

The first order compiler enables one to write declarative and succinct logic programs by allowing (restricted) first order formulae as goals. Although it is just one of many approaches to the problem of logic program synthesis from first order formulae, it is logically sound and completely automatic, but yet appears to be able to generate usable programs.

We have experimentally implemented the first order compiler using DEC-10 Prolog [7]. It accepts any first order programs and can compile (some of) built-in predicates of DEC-10 Prolog such as `=`, `is`, `length` etc according to their semantics. The size of the source program is about 1500 lines. To help the reader understand first order programs, we show a couple of sample programs[11].

[10] An allowed program is similar to a basic program but differs in that the clause body can include atomic goals and implicational goals at the same time.

[11] In our implementation, a universally quantified formula $\forall X \ldots \forall Z(E)$ is written as `all([X,...,Z],E)`. Similarly an existentially quantified formula $\exists X \ldots \exists Z(E)$ is written as `exist([X,...,Z],E)`. "$\leftarrow$" and "$\rightarrow$" are respectively represented by ":- " and "->". $\forall y(s \neq t)$

Example 1:

```
fib(L):- L=[0,1|_],
         all([F0,F1,F2,A,B],
              (append(A,[F0,F1,F2|B],L) -> F2 is F0+F1)).

append([],Y,Y).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

This program defines fibonacci series. For example, given a query `?- fib([F0,F1,F2,F3_`
the program no doubt terminates with the answer `F0=0`, `F1=1`, `F2=1` and `F3=2`. This behavior, however, is apparently dependent on the order of the **append** clauses: with the order reversed, the program would try to determine the values of the elements in `L` from the tail, causing an error in the **is** primitive. Thus the programmer is required of some knowledge of the procedural interpretation just as in the case of Prolog. The next example is a little more complicated.

Example 2:

```
split(Atom,S,D):-
        name(Atom,L),
        append([D|L],[D],L2),
        all([X,Y,Z,Wd],
            (append(X,[D|Y],L2),
             append(Wd,[D|Z],Y),
             \+Wd=[],
             all([U],(mem(U,Wd)->\+U=D))
             -> exist([N],(name(N,Wd),mem(N,S))) )),
        list(S).
```

This program splits `Atom`, an atom such as `well_foundedness`, into a list `S` of words like `[well,foundedness]` by deleting the specified delimiter `D` (=95, the ASCII code for the under score "_") from the atom. **append** and **mem** are defined previously. **name** is a built-in predicate and **list** defines list terms. However, in practice, it would be better to finish **split** clause with a cut (!) in order to suppress extraneous answers such as `S = [well,foundedness,_]` obtainable by backtracking.

The last example concerns negation. The problem of negation in logic programming [1,2,6,12,17,20] has long been discussed mostly in relation to SLDNF resolution, SLD resolution combined with the negation-as-failure inference rule [1,4,6,12,17]. For the sake of the logical soundness of SLDNF resolution, it is usual to add such condition (*ground condition*) that negative goals must be ground

is implemented as $\backslash+(s = t)$. For other notations, we follow DEC-10 Prolog.