論文 / 著書情報
Article / Book Information

題目(和文)	宣言的論理プログラミング
Title(English)	Declarative logic programming
著者(和文)	佐藤泰介
Author(English)	TAISUKE SATO
出典(和文)	学位:工学博士, 学位授与機関:東京工業大学, 報告番号:乙第1709号, 授与年月日:1987年7月31日, 学位の種別:論文博士, 審査員:
Citation(English)	Degree:Doctor of Engineering, Conferring organization: Tokyo Institute of Technology, Report number:乙第1709号, Conferred date:1987/7/31, Degree Type:Thesis doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Declarative Logic Programming

Taisuke SATO

Electrotechnical Laboratory
1-1-4 Umezono, Sakura-mura, Niihari-gun
Ibaraki 305 Japan

Table of contents

1. Introduction
2. First Order Programming
2.1. What are we going to do?
2.1.1. What are we going to do?
2.1.2. A new goal and its interpretation
2.1.3. Continuation in logic programming
2.1.4. A compilation example
2.1.5. Related works
2.2. Preliminaries
2.2.1. General notations and conventions
2.2.2. Proof theoretic notions
2.2.3. First order programs
2.3. Conversion to basic programs
2.4. Outline of compilation process
2.5. Mode labelling
2.6. Term labelling
2.7. Closure and continuation definitions
2.8. Translation
2.9. Executing compiled programs
2.9.1. An interpreter for compiled programs
2.9.2. Soundness of successful computation
2.9.3. Soundness of failed computation
2.10. Correctness of compilation
2.10.1. Preliminaries
2.10.2. Proving proof theoretic relation
2.10.3. Proving model theoretic relation
2.10.4. First soundness theorem
2.10.5. Second soundness theorem
2.11. System predicates
2.12. More compilation examples
2.12.1. A tiny data base
2.12.2. Fibonacci series
2.12.3. Intuitionistic truth predicate
2.13. Optimization
2.13.1. False goal
2.13.2. Tail recursive optimization
2.13.3. Transformation
3. Transformational Logic Programming
3.1. Equivalence preserving transformation system
3.1.1. What is transformation like?
3.1.2. Transformation System
3.1.3. An optimization example
3.2. Negation technique
3.2.1. Beginning negation technique
3.2.3. Formalization
3.2.3. Double negation technique
3.3. Getting usable programs
4. Conclusion

(Acknowledgements)
(References)

1. Preface

As everyone knows, writing a program is not an automated straightforward process but a labour-intensive and error prone one whatever language we may use. The need for rewriting part of or the whole of a program frequently arises for many reasons, such as: bugs, too poor performance, the need for better presentation of the algorithm or the revealing of any misunderstandings of the specification it must satisfy. The latter sometimes brings us back to the very beginning where we started programming.

To cope with this inherently zig-zag way of program development, we have several alternatives, not necessarily mutually exclusive. The most radical but least feasible one would be fully automated programming which, given a specification, produces a correct program all at once. At the other end of the spectrum, there have been efforts to produce a better programming environment by offering visual interfaces with a pointing device, an elaborated operating system, electronic mail, improved computation power and so on. This approach makes programming smoother and more pleasant, thus enhancing the programmer's productivity.

A third, and perhaps the most practical approach, is to adopt a higher level language. A higher level language is more abstract, closer to the specification level. Abstractness enables us to tackle the essential part of a problem while neglecting the detail and hence allows for more expressive power. In addition, it is said that the number of codes a programmer can write, say, in a year, does not vary much with the programming language used. It is claimed therefore, that a Fortran user is more productive than an Assembler user. Whether this claim is exaggerated a bit or not, we know that a program written in a high level programming language is more understandable than the one written in a low level language. We can also observe, in the history of computer languages, the evolution of higher level languages from machine languages through assembly languages, to Fortran, Algol, Pascal, Ada ..., each stage with more supporting facilities for structured programming.

With reservation necessary to making a general statement, the programming style fostered by these main-stream languages can be summarised as follows: write an efficient program first, then check its correctness. Concerning efficiency, these "imperative" languages (imperative because a program consists of a sequence of commands) seem successful to various degrees. The problem is to what extent they are successful in ensuring correctness.

We assume a program has three components: logic, control and data. A programmer writes a program to express

his/her idea (logic) by means of building blocks (data) and gets the output from an input (control). The logic part is usually invisible but reflected in a program as various declarative properties. A typical example is input-output relation which comprizes, by definition, all pair of an input and the corresponding outputs. It is considered as one of the most distinctive declarative characterizations of programs so that a specification is often stated in the form of input-output relation requirement. Therefore, it is natural to stipulate that a program is correct if the actual input-output relation is the same as the intended one (it is convenient to decompose the notion of correcness into two things as usual. One is "termination" meaning that a program must terminate for the intended inputs. The other is "partial correcness" meaning that every output computed from an input must be an intended one).

In terms of the trichotomy we mentioned above, imperative languages apparently put emphasis most on control, less on data and none on logic. The logic must be operationaly expressed through commands using their side effects. It means that what was clear and visible at the beginning necessarily becomes something implicit, thereby hard to understand in the end of programming. It further means that when it comes to program verificaiotn or program understanding, we have to follow the same process but in the opposite order. Namely, starting from scratch, as nothing is given other than commands, we have to infer a piece of logic from each piece of program text, then have to consitently unite such pieces into a logical description of the whole program.

This process is laborious. It demands much intelligence, preciseness and patience, and worse, is liable to get frustrated even if a good deal of comments are available in the program. It is also clearly true that the difficulty is multiplied by the semantic complexity of imperative features whose semantics are only understandable through the very low level notion of address and content (think x=x+1 in Fortran). Summing up, a program can not be a logical document in an imperative language. The enormous semantic gap between logic which is declarative and a sequence of commands which is operational inevitabley makes the verification of correctness something formidable, if not impossible. Higher level languages seem to have essential difficulties in ensuring correctness as long as they are imperative, regardless of their "height."

To get around this difficulty, not a few remedies have been proposed. Take "abstract data type" for example. By encapuslating data and algebraic operations into single entity and by only allowing access to the entity through its axiomatic description, we can make interfaces between program modules more abstract and more declarative. It is interpreted as mechanism for partially visualizing the logic part and

enahncing declarative nature of a program. The idea naturally lends itself to building a system free from bugs caused by interface mismatches. We also note "functional programming" has been attracting considerable interest because of its declarative nature. In this approach, a program comprizes function definitions and execution is considered as term rewriting (in a wide sense). The formulation of function varies with theories. It may be lambda calculus, equation or combinator. However, since there is no side effect in computing functions and since a variable, if any, has referencial transparency in any formulation, it is easy to treat a program as a mathematical object, thereby easy to achieve rigorous verification. We can also count "programming in a constructive mathematics" as another declarative programming paradigm. It is aimed at the complete elimination of the need for program verification after programming. Correctness is guaranteed by programming process itself. In fact, programming in this system is nothing other than proving a specification in a constructive set theory.

We can see that although there are differences in the motivations and degrees, they are common in that they have brought declarative nature into programming and put more emphasis than imperative languages on the logic part of a program. Then an idea occurs. If computer languages are to evolve to heigher and declarative ones and they did so actually, why not go to extreme? In other words, why not get programming started from the logical part? Programming by logical formulae, or logic programming in short, stemed from this idea. According to this thesis, a program is just a logical formula. Execution must be logical deduction applied to the program. Control means deduction strategy. A prominent example is programming by definite clauses where a program is a finite set of definite clauses (a definite clause is a first order formula of the form A0 <-A1&..&An (n>=0) where Ai (0=<i=<n) is an atomic formula). Program execution is backward application of modus ponens. The point is that we can interpret a definite clause as a procedure with head A0 and body A1&..&An as well as a logical formula (35). When A0 is called, the body A1&..&An is executed by calling each goal serially or concurrently, which corresponds to the backward application of modus ponens. In case of n=0, the empty body, we consider A0 as asserting a fact. Therefore, writing a set of definite clauses is equivalent to writing an executable program as well as its logic component. The first interpreter was called Prolog (10, 25, 35).

A Prolog interpreter in its pure form is an SL resolution theorem prover (6) but specialized to definite clauses. So it is essentially a sequential top-down theorem prover operating on ordered definite clauses with a left-to-right goal selection rule. It starts backtracking to the previous point where an alternative clause remains if any, as soon as the

current goal fails. Consequently, in Prolog, the control is expressed by the textual order of goals and clauses, which obviously has nothing to do with the logical content represented by the clauses themselves ("&" is commutative).

Logic programming has three implications. Firstly, it allows us to have the partial correctness for free. This is because the computed result is a logical consequence of the program. Secondly, it means that we write a program closer to the level of specification. For logic (first order logic, a higher order one, or whatever it may be) originated from the efforts to formalize natural language and it still keeps a closer relation to natural language than other formal languages created artificially for specific purposes. So, a program in logic programming is reasonably expected to be more self-documented and more understandable. Lastly, we have to note that logic programming separates logic from control, thereby facilitates transformation of one component without disturbing the other. More to the point is that it is possible to refine a logic program by giving a better logical expression "and" by giving alternative control suitable for the expression. This observation should not be missed from the viewpoint of "transformational programming," which many people consider as a potential breakthrough for software crisis.

The story of transformational programming goes like this: we first concentrate on writing a correct and clear program using a high level language. In this phase, we do not care about efficiency. Next we apply (correctness preserving) program transformation, manually or mechanically, to opimize it. The final program then would be a correct and efficient one. It is common to take "transformational programming" merely as a way of thinking to satisfy contradictory requirements such as correctness, understandability and efficiency. But it is not hard to see that once we have succeeded in complete automatization of transformation process, and if transformation rules are powerful enough, the effect could be comparable with the advent of the first Fortran compiler.

For this reason, a notion of transformational programming has been attracting considerable attention. It nevertheless remains as a rather theoretical possibility, due largely to the lack of languages suitable for transformation. But now we have a new language which seems very amenable to transformation as mentioned above. We believe logic programming can open a new vista on transformational programming. An experimental implementation of logic program transformation system already has shown that it is possible to completely automate transformation process if the domain of programming is properly chosen (61, 63).

It therefore comes as no surprise that researchers have been working to exploit the possibilities of logic programming

revealed by Prolog. Together with those poeple, we pursue in this thesis two objectives. One is to extend the current definite clause programming language to a highly declarative one, namely first order predicate calculus. The other is to develop a transformation system for definite clause programs. Two topics are interrelated: a program in the first language is "compiled" into a (naive) definite clause program which is assured to be (partially) correct with respect to the source program. Then if the compiled program has turned out to be inefficient, it can be optimized by a transformation system, preserving its meaning. In addition, we must say that our transformation system combined with "negation technique" provides another path from programs written in a first order language to definite clause programs. Both are attempts to extablish a declarative programming style which not only is free from diseases traditional imperative languages have been suffering from but is comparable in performance.

In chapter 2, we introduce "first order programs" and show how to mechanically compile them into pure Prolog programs. First order programming is an extension of pure Prolog allowing implication and universal quantification under certain restiriction. In chapter 3, we describe a transformation system based on unfold/fold transformation together with "negation technique" and show how they are applied to optimizing defininte clause prgrams and to compiling first order programs.

2. First Order Programming

2.1. Introduction 2.1.1. What are we going to do?

Here is a brief view: As a step toward to full first order logic programming, we augment definite clause programs with universally quantified goals and show that they can be mechanically compiled into definite clause programs. A compilation process comprises three phases: mode-labelling, term-labelling and translation. Each phase is deterministic and always halts. A compiled program is runnable directly in Prolog. What is more, we can prove that every result computed by the compiled program is a logical consequence of the "completion" (9) of a source program. It may sound simple, but the proof is not as simple as it looks.

Technically speaking, our compilation is just an application of the well known "unfold/fold" transformation. The difficulty lies in the identification of formulae with which we start the transformation process. We have found that if we start not only with the completion of a source program but also with the "universal continuation" forms determined by mode analysis at the first phase, we can deterministically obtain a definite clause program by the unfold/fold method. The notion of universal continuation is crucial. It is a generalization of functional continuation adapted to logic programming and the basis of our compilation method and its soundness proof.

Another problem in the proof is that we have to ensure the existence of continuation functions and continuation predicates satisfying certain conditions. They are not included in the source program but do exist in the compiled program in order to compute continuations. However, the problem is solved by model theoretical means without much difficulty.

The proposed method enlarges the class of runnable specifications, and hence seems to blur the distinction between program synthesis and program transformation. Some examples are presented to show the intended use of the new programming feature. Also, optimization techniques including tail recursive optimization will be discussed. Now we begin.

Proposing a new programming feature always gives rise to the questions of what it is for and how much it is effective. Introduction of a new first order feature to pure Prolog too, can not be an exception. As the task of answering them in a few words is beyond the author, we would like to explain our motivation in a leisurely manner, hoping that it might help the reader convince him/herself. First we would like to note that that enhancing logical power has been one of the motive forces of logic programming research, and that researchers are making efforts for extending (pure) Prolog to a more expressive programming language, for instance by incorpotationg equality or by allowing more general formulae instead of definite clauses. The extension to first order programming, used here as generic term for the programming that admits of first order formulae, has long been pursued as one of such (4, 22), and our proposal is exactly that kind. We add as goals universally quantified implicaitons of the form all(x, A->B) where A and B are atoms to definite clause programming. Note that this construct includes A->B and ~A (= A->false) as a special case. Also note that we often meet this type of formula as a specification. For example, subset relation is expressed by all(x, member(x, s1) -> member(x, s2)), being given member(x, s) predicate which means that x is a member of list s.

One may suspect that adding universally quantified implicational goals to pure Prolog is too small a step to claim the title of an extension to first order logic programming. But we have to point out that the opposite is true, at least theoretically, since universal quantified implication includes negation and it has been proved that once negative atoms are allowed as goals in pure Prolog, we can write any first order formula at the cost of introducing new predicates (39).

Then a question naturally occurs. If negation is enough for first order programming, what is the point in adding such construct when negation is already a built-in function in every Prolog implementation? The answer lies in the gap between theory and implementation, which we explain now.

In Prolog, negation is computed by the "negation-by-failure" rule (9). It states that a negative goal ~A succeeds iff (if and only if) proving A fails. This rule is easily implementable and memory-effective when combined with backtracking. Nevertheless, there are following drawbacks due to backtracking caused by failure.

Firstly, the result is not necessarily correct. To assure its correctness, we have to check at run time that every negative goal is ground (includes no variables) and this is very time-consuming.

Secondly, as long as failure causes backtracking, it is impossible to get variable bindings because backtracking undoes every variable binding obtained so far. This becomes a serious problem when we try to extend the realm of logic programming. Let us take the following "specification":

```
one-two(L) <- all(Y, mem(Y, L) -> Y=1\/Y=2)

mem(Y, [Y|Z])
mem(Y, [U|V]) <-mem(Y, V)
```

which is intended to express that L is a flat list including only 1 or 2 (from here on, we adopt DEC-10 Prolog [69] notation for variables unless otherwise stated. A variable is a string starting with an upper case letter. Other strings are used as function symbols or predicate symbols. "\/" stands for "or." [A|B] is the "cons" of A and B).

This specification is simple but the intention is clear. Suppose we regard it as a program and use it both to check if L is really a flat list comprizing 1 and 2 and to generate such lists. If we compute it by negation-by-failure, it would look like:

```
one-two(L) <- not(mem(Y, L)& not(Y=1\/Y=2)).
```

Here we assume not(A) succeeds iff (if and only if) A fails. This works for checking but, regrettably, will never work for generation, as long as "not" is identified with failure and failure means backtracking.

Putting these difficulties aside, we also have to pay attention to the role of negation in the context of program transformation. We already have an elegant transformation system [65] for definite clause programs which can transform programs preserving correctness. Therefore, we hope if possible, that it is somehow applicable to a program including negation for the optimizing purpose. Look at the following example:

```
max(M, L) <- mem(M, L) & all(Y, mem(Y, L) -> Y=<M ).
```

This is a specification for the relation "max" stating that M is the maximum element of a flat list L. Using "not", it is compiled into:

```
max(M, L) <- mem(M, L) & not(mem(Y, L) & not(Y=<L)).
```

Runnable as it is when L is a flat list of integers, the time complexity of this program is quadradic in |L| (the length of L) in the worst case. So one naturally feels that there might be a way via transformation to a linear max program. Unfortunately, our transformation system is not capable of

handling "not" as it is. We must, therefore, look for a more sophisticated one which can cope with this situation (31). But suppose this form could be transformed mechanically to a definite clause program to which our transformation system is applicable. Then we could expect to get to the linear max program, through transformation, with more ease.

The first order programming realized by current Prolog implementation is a very limited one for the reaons alone we have seen. By contrast, geneuine first order programming, if it exists, must first guarantee that every computed result is always a logical consequence of the program. Second it must allow us to obtain variable bindings through a goal even if it is not a positive atom. And the last but not the least, it must make practical sense.

In search of better first order programming, we have augmented definite clause programs with a new type of goal, universally quantified implication, then have shown that it is compilable into a definite clause program deterministically, and lastly have proved that the compilation always produces a partially correct program when successful.

It can not be claimed that our extension is genuinely full first order logic programming. It lacks for instance, equalities and induction mechanisms. Logically speaking, it is just an attempt to introduce universal quantification to logic programming so that all logical connectives can be available. However, we can point out the following characteristics in our approach.

(1) Compilation is deterministic and always halts, though not always with success.

(2) A compiled program is runnable in Prolog. A run-time check for negative goals is not required as far as successful computation is concerned.

(3) Every goal computed by the compiled program is a logical consequence of the "completion" of a source program.

(4) To get variable bindings from a negative goal (in the form of implication) through compilation becomes possible.

(5) Optimization of a compiled program by a (-n equivalence preserving) transformation system (64, 65) is possible.

Theoretically, our compilation is based on the notion of "universal continuation." Technically, it is a deductive unfold/fold method (7, 24, 55). What we have discovered is that it is always possible to apply unfold/fold method deterministically to a certain class of first order

formulae to obtain a definite clause program.

Our compilation algorithm itself is simple. But its justification is rather long. We hope therefore that the reader happens to be patient enough and tolerant with mistakes undiscovered.

In the following section, we describe a procedural reading of a new goal and how to execute it by an interpreter. We then analyze the difficulties in the interpreter approach and suggest that the introduction of continuation could solve the problem. In subsection 2.1.4., we apply our idea to the first example and illustrate the compilation process. In the last subsection, we look at our method from a standpoint of program synthesis and compare it with other works.

Section 2.2. is devoted to the definition of basic terminologies. Then, in section 2.3., we introduce the class of first order programs, i.e. the objects to be compiled, and prove that a first order program is always convertible to a basic program from which compilation starts. In section 2.4., we give the outline of the compilation process. The description of the compilation algorithm actually starts in section 2.5. and ends in section 2.8.. A compiled program becomes an executable program. It is a program which may include goals of the form all(X, s=/=t) where s and t are terms as well as ordinary atoms. Such goals are computed by negation-by-failure.

We would like the reader to note that the negation-by-failure rule computes all(X, s=/=t) correctly as long as we adopt "free interpretation" of function symbols in which terms denote themselves. S/he also should note that all(X, s=/=t) is a recursive relation over a Herbrand domain generated from finite symbols so that it can be replaced by some suitable definite clause program. Using section 2.9. and 10, we prove the correctness of our method. Namely we prove that every answer computed by a compiled program is a logical consequence of the completion of the source program.

In a more realistic situation, we have to take account of built-in predicates. These are dealt with in section 2.11.. In section 2.12., we look at other examples and show that the operational semantics (goal order and clause order in the initial program) is well preserved through compilation. Section 2.13. mentions optimization techniques, especially focusing on tail-recursive optimization.

The reader is assumed to be familiar with the basic notions and terminologies of mathematical logic and logic programming (6, 10, 32, 51, 69).

2.1.2. A new goal and its interpretation

We have to apologize in advance that throughout the rest of this section, for simplicity, some terminologies are used without exact definitions. Suppose we have decided to allow a universally quantified implicational formula of the form:

all(Y, p(X, Y)->q(Y, Z))

to appear in the body of a clause. The declarative meaning is clear and we do not mention it. However, the procedural reading is not similarly clear. We simply read it as:

Find such X and Z that all(Y, p(X, Y)->q(Y, Z))

and compute it by:

> Generate the value of Y one by one and every time the value of Y is generated, compute q(Y, Z) with that value and repeat this until no new value of Y is found.

The goal as a whole succeeds if q(Y, Z) succeeds for all the values of Y generated by p(X, Y). It fails if there is some Y such that p(X, Y) succeeds but q(Y, Z) fails. In other words, it fails if we have found Y such that p(X, Y)& not(q(Y, Z)).

Note that during the computation of p(X, Y) and q(Y, Z), X and Z can be instantiated. Suppose for instance, we are computing p(X, Y), with X having the value s0, to obtain values of Y, X can become s1 after the computation. Then the second value of Y, if any, is computed by goal p(X, Y) with X having the value s1.

Our reading is different from one that first generates all answers of Y by p(X, Y) and then tests each of the generated values by q(Y, Z). It is also different from the meta-reading of p(X, Y)->q(Y, Z) in which "->" is interpreted as the provability relation "|-." It reads "p(X, Y)->q(Y, Z)" as q(Y, Z) is provable by adding p(X, Y) to the existing hypothesis. As the reader might notice, this reading is a generalisation of the negation-by-(finite)failure computation rule. To see this, assume q(Y, Z)=false. This means that by assuming p(X, Y) as an additional hypothesis, "false" is deduced, and hence by the law of contraposition, we have ~p(X, Y) from the existing hypotheses. By contrast, in our case, negation is introduced as a special case of a universally quantified implication:

all(Y, p(X, Y)->false).

Therefore, our negation works at object level. It is a classical first order negation unlike the one representing non-provability.

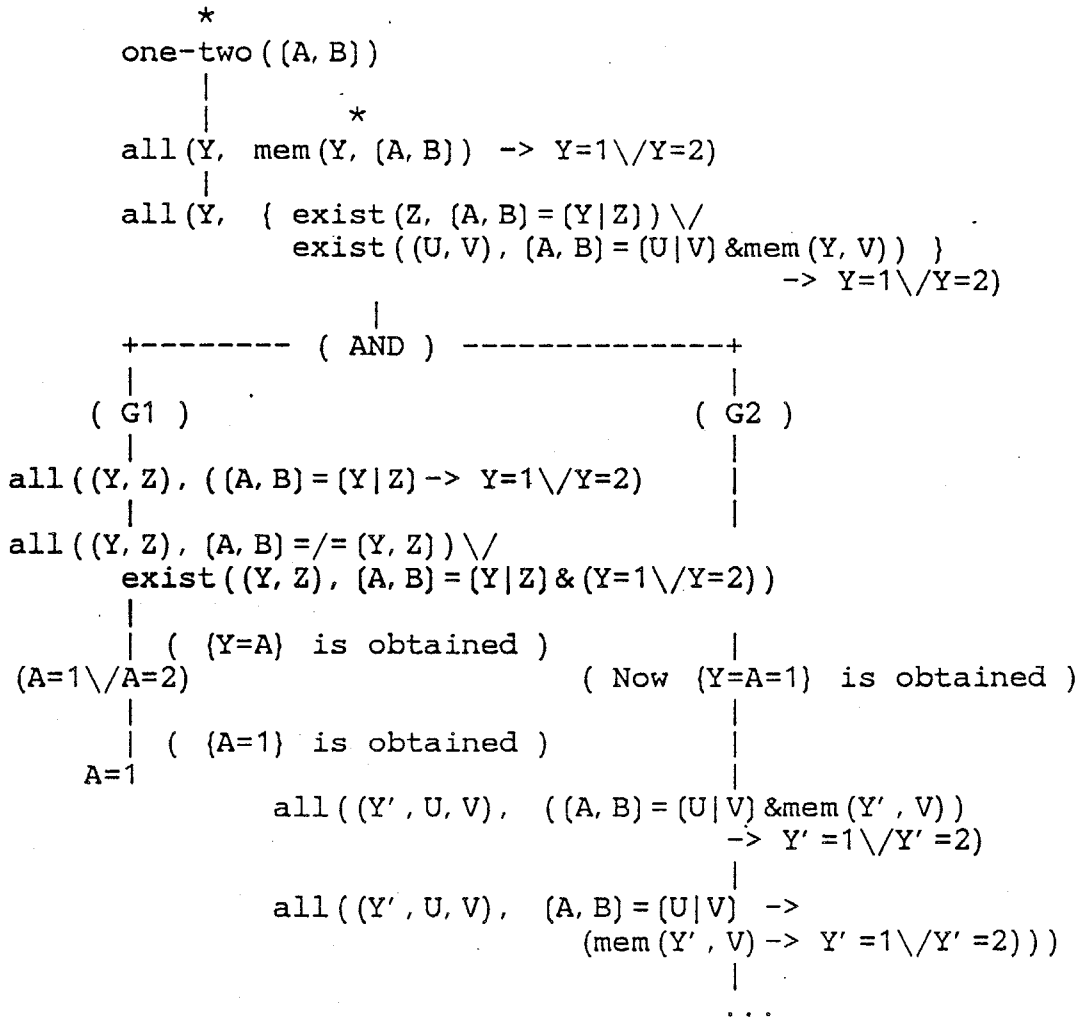
Once the reading has been fixed, it is not difficult to write an interpreter faithful to that reading. We show it using first example:

```
S =

{ one-two(L) <- all(Y, mem(Y,L) -> (Y=1\/Y=2))
  mem(Y, [Y|Z])
  mem(Y, [U|V]) <-mem(Y, V) }.
```

To compute this specification, we prepare iff(S), the iff-definition of S, as follows:

```
iff(S) =

{ one-two(L) <-> all(Y, mem(Y,L) -> (Y=1\/Y=2))
  mem(Y,L) <->
        exist(Z, L=[Y|Z])\/
        exist((U,V), L=[U|V]&mem(Y,V)) }.
```

Let a top goal be <-one-two([A,B]). Then we "unfold" the top goal (the unfold/fold operation is rather broadly definable, but in our context, it is enough to say that given a definition of the form A<->F in which atom A (procedure head) is defined as F (procedure body), a replacement of A by F with a suitable instantiation is called an "unfold" operation. A "fold" operation is the reverse of an "unfold" operation. It is the replacement of F by A). In the following illustration, * denotes an unfolded atom.



First one-two((A, B)) is unfolded to all(Y, mem(Y, (A, B)) -> Y=1\/Y=2) and then the antecedent of this implication is unfolded to a disjunct:

```
exist(Z, (A, B)=(Y|Z))\/
        exist((U, V), (A, B)=(U|V)&mem(Y, V))
```

using the iff-definition of "mem." At this point (denoted by "AND"), by appealing to the propositional law: (A\/B ->C) <-> {(A->C)&(B->C)}, the current goal is reduced to two AND-goals, G1 and G2, each of which has the form of implication. We then take G1 first because it corresponds to the first clause of "mem" in the textual order. Since we can prove that:

```
all((Y, Z), ((A, B)=(Y|Z) -> Y=1\/Y=2)
        <-> all((Y, Z), (A, B)=/=(Y|Z))\/
            exist((Y, Z), (A, B)=(Y|Z)&(Y=1\/Y=2))
```

holds in the Herbrand universe, G1 is reduced to a disjunction.

The first disjunct is found to be false. But the second is solved giving a binding (A=1). Up to this point, we have successfully solved the antecedent mem(Y, [A, B]) with Y=A and have then solved the consequent (A=1\/A=2) with A=1. We return with these bindings and tackle the second goal G2 which corresponds to the second clause in the textual order and so on.

Evidently, this is faithful to our procedural reading of a universally quantified implication. There is a problem however. Suppose that the current goal has the form all(Y, (G1\/..\/Gk ->C)). It is reduced to all(Y, G1->C)&..&all(Y, Gk->C) to form next AND-goals. Universally bound variables, in this case just Y, are shared by the resulting AND-goals although they should be logically independent. So we have to rename them while leaving other variables (global variables) shared.

This requires us firstly to examine which variable in the current goals is universally quantified and secondly to make as many copies as necessary of goals sharing such variables. Since the run-time form of a goal is unpredictable beforehand, these operations must be done at run-time. Consequently, heavy search and copying at run-time become inevitable. An interpreter implemented using run-time copying was found too slow to be usable. It might not be unreasonable to remark that such an interpreter corresponds to the one for definite clause programs which copies an input clause everytime it calls.

To avoid such awkwardness due to search and copying, we have only to stop universally quantified variables being shared. This suggests that instead of computing the value of such variables "eagerly," we should adopt "lazy evaluation" for them and let other variables be computed normally. Or more concretely, it suggests that we should transform an original program to one that delays selectively the actual construction of a term bound to a universally quantified variable until, hopefully, every constituent of the term includes no universally quantified variables. Of course, this is not always possible because it means that in an ordinary evaluation of the original program, a universally quantified variable must eventually become a term only consisting of functors and other non-quantified variables as happened to Y while computing mem(Y, [A, B]). Nevertheless, the idea seems worth trying.

A further question is; in logic programming, how can we selectively delay the computation of terms bound to universally quantified variables? In functional programming, "continuation" passing style programming is known to have the effect of delaying computation. It is therefore interesting to see how this notion is adaptable to logic

programming. Needless to say, as we are looking for "logical continuation," the adaptation must have a logical explanation.

2. 1. 3. Continuation in logic programming

First let us review "continuation." Continuation is, in nature, an abstract object. It approximately means something that represents "the rest of computation." It is an essential device in denotational semantics for imperative languages with jump, go-to, global exit, ... and is proved powerful enough to describe the semantics of a considerable part of, say, Pascal (60).

Continuation has been similarly well known in functional programming schools, but in a more concrete form. Take a function f(X). We introduce another function f' (closure function) with a defining formula:

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

where C is a higher order object (usually a lambda expression) called continuation which denotes "the rest of computation after f(X) has been computed." For example, consider the program append(X, Y):

```
append(X, Y)

= if X=() then Y
  else cons(car(X), append(cdr(X), Y))
```

which concatenates list X with list Y (read according to Lisp convention). We can transform this program to a continuation passing style one mechanically. The result is this:

```
append' (X, Y, C)

= if X=() then C(Y)
  else append' (cdr(X), Y, lambda(Z, cons(car(X), Z))).
```

The top level call is, for example, append((a, b), (c), lambda(Z, Z)). After concatenating (a, b) with (c), no computation remains, hence lambda(Z, Z) as the initial continuation.

We can note two facts. First, the computation of the "cons" operation is delayed by adopting continuation passing style programming as expected. The second fact is that this transformation can be carried out algorithmically using the definition of the closure function and algebraic laws concerning if-then-else and the like.

(Existential continuation):

We first translate the notion of closure and continuation in functional programming into logic programming naively as follows. Suppose a definite clause program is given

which computes a predicate p(X, Y). We want to compute it in continuation passing style.

Corresponding to f' (X, C) = C(f(X)), we introduce two predicates p' (X, C) and cont-p' (C, Z) and require that they satisfy the following relation:

p' (X, C) <-> exist(Y, p(X, Y)&cont-p' (C, Y)).

p' is called the existential closure predicate for p. C is a variable called a continuation variable. It is supposed to carry a term which contains all information needed for the computation after p(X, Y) has been proved. cont-p' is called the existential continuation predicate for p. It is a substitute for a lambda expression and presents the rest of computation using the term bound to C and the value bound to Y computed by p(X, Y).

This is an example of existential continuation form of p(X, Y). It is a direct transfer of continuation into logic programming. We can say, however, that it is more general than the functional counterpart in that it can cope with non-determinacies as well. In fact, using existential continuation, it has been proved that under a certain condition, from a given program that computes p(X, Y), we can transform it into the all-solution program which returns the list of all solutions of Y with respect to p(X, Y). Furthermore, the transformation is justifiable using the equivalence preserving transformation system [64, 65]. As another example of existential continuation, we can that every definite clause program has an equivalent canonical form comprizing only definite clauses of the form A<- or A<-B.

(Universal continuation):

The notion of existential continuation is simple and useful. This does not mean that it is the only form of continuation in logic programming. Another continuation is possible, universal continuation, the dual form of existential continuation. It will be the key notion of this thesis.

A universal continuation form for the predicate p(X, Y) is introduced, for example, by:

p' (X, C) <-> all(Y, p(X, Y)->cont-p' (C, Y)) .FE where p' is called the universal closure predicate for p and cont-p' is called a universal continuation predicate for p. This form is more general than the existential counterpart in the sense that it takes account of every possible value of Y computed by p(X, Y) and computes cont-p' (C, Y), the rest of computation, for each value. Note that by definition, p' (X, C) becomes true if p(X, Y) is false, in contrast with the existential version. Note also, that if there always exists a unique Y with respect to X in the relation p(X, Y), it reduces to an

existential continuation form. Conversely, by negating both sides of the existential continuation form, we can reach the universal one.

In the next subsection, we show how one-two(L) example is compiled into an executable program and describe the role of universal continuation in its justification.

2.1.4. A compilation example

In this subsection, we informally show how to compile one-two(L) specification:

```
one-two(L) <- all(Y, mem(Y, L) -> Y=1\/Y=2)

mem(Y, [Y|Z])
mem(Y, [U|V]) <-mem(Y, V)
```

(a) A source program

The first phase of compilation is "mode labelling." We analyse the program and check whether a consistent "mode assignment" to each occurrence of predicates is possible or not. We start with the (only one) universally quantified implication all(Y, mem(Y, L) -> Y=1\/Y=2). We assign mode mem(-, +) to the antecedent mem(Y, L) seeing that Y is universally quantified and supposing that it is computable from L. It means that we regard Y as the output and L as the input when mem(Y, L) is considered as a function.

As mem(Y, L) can call two clauses, mem(Y, [Y|Z]) and mem(Y, [U|V]) <-mem(Y, V), we assign modes to these potential callees as well. For mem(Y, [Y|Z]), we assign mem(-, +) because it is called from mem(Y, L) with that mode. Then, because it is a unit clause, we must check that the output Y of mem(Y, [Y|Z]) is determined by the input [Y|Z].

For mem(Y, [U|V]) <-mem(Y, V), we assign the head mem(Y, [U|V]) mode mem(-, +) for the same reason as the unit clause and then proceed to the body mem(Y, V). We assign the second argument + because V is determined by [U|V], the second argument of the head which was already assigned +. The remaining arguments, the first argument of mem(Y, V) in this case, are assigned -, because they have no positive reasons to be assigned +. Finally, we must check that the output term of this clause, i. e. Y in the head, occurs somewhere else in the clause.

As no new mode patterns have been generated and no clauses are left to be checked as potential callees with existing mode patterns, we exit the "mode labelling" phase successfully. The result is:

```
                                  mem(-, +)
one-two(L) <- all(Y,  mem(Y, L)  -> Y=1\/Y=2)

  mem(-, +)
mem(Y, [Y|Z]),

   mem(-, +)      mem(-, +)
mem(Y, [U|V])  <- mem(Y, V).
```

(b) After mode labelling

Next comes the "term labelling" phase. We assign a "continuation term" to each atom occurrence labelled by a mode pattern. Suppose that a continuation term t is assigned to an atom occurrence A. Then the functor of t represents that atom occurrence or, more concretely, the return address. Therefore, it must be unique to each atom occurrence. On the other hand, the arguments of t are variables which may be referred to only after A has been computed. The result is:

```
                                  mem(-, +), f0
one-two(L) <- all(Y,  mem(Y, L)  -> Y=1\/Y=2)

  mem(-, +)
mem(Y, [Y|Z]),

   mem(-, +)      mem(-, +), f1(C)
mem(Y, [U|V])  <- mem(Y, V).
```

(c) After term labelling

Here, C is the continuation variable. f0 is found to be a constant because it has no need for carrying variables. This is because after having computed mem(Y, L), the variable whose value is needed to perform the rest of the computation (Y=1\/Y=2) is Y and Y is computed by mem(Y, L). Similarly, f1 becomes a unary function because f1 must carry the continuation information bound to C and that is all that f1 must carry.

The last phase is "translation" into an executable program. Here, we introduce a closure predicate mem' and the continuation predicate cont-mem' for the (only) mode pattern mem(-, +). The result, the compiled program, is as follows:

```
one-two(L)<-mem'(L, f0)                                        ... (1)

mem'(L, C)<- all((Y, Z), L=/=(Y|Z))                           ... (2)
mem'((U|V), C)<-cont-mem'(C, U)& mem'(V, f1(C))                ... (3)

cont-mem'(f0, U)<- U=1\/U=2                                    ... (4)
cont-mem'(f1(C), U)<-cont-mem'(C, U)                           ... (5)
```

(d) The compiled program

This is an executable program. We let a call to all((Y, Z), L=/=(Y|Z)) succeed iff L is ununifiable with a cons pattern (L=() in the anticipated use). Note that although this computation rule is efficient and correct in the Herbrand universe, it is possible, theoretically speaking, to replace all((Y, Z), L=/=(Y|Z)) by another predicate say, non-cons(L) which enumerates all terms ununifiable with the cons-pattern. Hence, we get a totally positive program. It is obvious that such non-cons(L) can be mechanically derivable from all((Y, Z), L=/=(Y|Z)). For example, if a, b, (), (_|_) are the only symbols available, non-cons(L) would be:

```
non-cons(a)
non-cons(b)
non-cons(()).
```

For the query <-one-two((A, B)) for example, this program returns answers (1, 1), (1, 2), (2, 1), (2, 2) and no more.

The reader might notice the fact that this compilation has left (Y=1\/Y=2) intact in all(Y, mem(Y, L) -> Y=1\/Y=2). Therefore, just by replacing U=1\/U=2 with "false", we can compute all(Y, ~mem(Y, L)) with: all(Y, mem(Y, L)->false). In this case, eliminating all((Y, Z), L=/=(Y|Z)) is desirable to get variable bindings.

It must be said that the compiled program is not only operationally correct because it computes the required one-two(L) relation properly, but can be shown to be a logical consequence of the following axioms:

```
one-two(L)<-> all(Y, mem(Y, L) -> Y=1\/Y=2)                    ... (6)

mem(Y, L)<-> exist(Z, L=(Y|Z))                                 ... (7)
             \/exist((U, V), L=(U|V)&mem(Y, V))

mem'(L, C)<->all(Y, mem(Y, L)->cont-mem'(C, Y))                ... (8)
```

```
cont-mem' (f0, U) <-> U=1\/U=2                                  ... (9)
cont-mem' (f1 (C), U) <->cont-mem' (C, U)                       .. (10)
```

(e) Logical source

---

(6) is the initial specification, (7) is the iff-definition of mem, (8) is the universal continuation form corresponding to mode mem(-, +) where mem' is the universal closure predicate for mem(-, +) and (9) and (10) are the auxiliary axioms posed on the continuation predicate cont-mem'.

The derivation is based on the unfold/fold method. First by folding (Y=1\/Y=2) of (6) into cont-mem' (f0, Y) (with suitable pattern matching) using (9), we get:

```
one-two (L) <-> all (Y, mem (Y, L) ->cont-mem' (f0, Y)).
```

Then using (8), this is further folded into:

```
one-two (L) <->mem' (L, f0),
```

thus giving (1). Next we move on to (8). After unfolding (8) at mem(Y, L) and using a valid propositional schema (a formula pattern whose instantiation is always true): (A\/B -> C) <-> ((A->C) & (B->C)), we have:

```
mem' (L, C) <->
        all (Y, exist (Z, L=[Y|Z] -> cont-mem' (C, Y)) &
        all (Y, exist ((U, V), L=[U|V] & mem (Y, V))
                                    -> cont-mem' (C, Y)).
```

Then, by appealing to two scheme:

```
(A&B->C) <-> (A-> (B->C)) and
all (Y, exist (X, F1) ->F2) <->all ((Y, X), F1->F2)
```

and folding cont-mem' (C, Y) into cont-mem' (f1 (C), Y) using (10), this formula is transformed to:

```
mem' (L, C) <->
        all ((Y, Z), L=[Y|Z] -> cont-mem' (C, Y)) &
        all ((U, V), L=[U|V] -> all (Y, mem (Y, V)
                -> cont-mem' (f1 (C), Y)))
```

and then to:

```
mem' (L, C) <->
        all ((Y, Z), L=[Y|Z] -> cont-mem' (C, Y)) &
        all ((U, V), L=[U|V] -> mem' (V, f1 (C)))
```

by folding again the consequence part of the implication on the second conjunct (with a suitable matching) using (8).

Finally, recalling that over a Herbrand universe:

```
all(X, X=t ->F) <-> all(X, ~X=t)\/exist(X, X=t & F)
```

is always true for a term t and a formula F provided X does not occur in t, we get:

```
mem' (L, C) <->
      { all((Y, Z), L=/=(Y|Z))\/
           exist((Y, Z), L=(Y|Z) & cont-mem' (C, Y)) } &
      { all((U, V), L=/=(U|V))\/
           exist((U, V), L=(U|V) & mem' (V, f1(C))) }
```

It is not hard to see that (2) and (3) are derivable from this formula. Other complied clauses, (4) and (5), are apparently derivable from (9) and (10).

Note that we have not yet completed the justification of our compilation example. We have only shown that the complied program is derivable from { (6), (7), (8), (9), (10) }. The proof depends on the existence of predicates mem' and cont-mem' and functions f0 and f1 satisfying {(8), (9), (10)}. How can we assume their existence at all, and why? We have also to check the relationship between the answer computed by a compiled program and the specification because the former is obtained only through continuation predicates and continuation functions whereas the latter has nothing to do with those objects.

In the rest of thesis, we describe the compilation method in a general case and prove that an answer computed by a compiled program is a logical consequence of the completion of the initial specification (now considered as a runnable specification).

2. 1. 5. Related works

As it is possible to regard our extension as one of the program synthesis methods based on the unfold/fold technique, it seems useful to have a look at other synthesis methods.

We understand by "program synthesis" a formal method to derive a runnable program from a (not necessarily runnable) specification. Therefore, depending on choice of the form of a specification, formal languages and techniques available, we have a variety of approaches to the problem of program synthesis. We classify existing approaches as follows (though this may not be comprehensive) :

(1) Deductive approach
(2) Resolution approach
(3) Transformational approach.

The first one is the most formal and has well understood mathematical foundation in both syntax and semantics. It can be dated back to the 1940s. Put simply, given a specification F(X, Y) representing the input-output relation (X -- input, Y -- output) which a program to be synthesized must satisfy, we try to find an existence proof, H |- all(X, exist(Y, F(X, Y))) where H is a set of axioms which the domain being considered must satisfy. If this proof is "constructive," we can extract mechanically a program P from the proof which computes the function f satisfying all(X, F(X, f(X))).

For instance, in a method based on the Godel interpretation [17, 52], H is the axioms for intuitionistic arithmetics and the underlying logic is intuitionistic first order logic. A synthesized program P becomes a "functional of finite type" which takes higher order objects as arguments but always gives a natural number output for natural number inputs. Computation is a kind of reduction of a term to its canonical form. The use of an induction schema yields a recursive program and modus ponens gives the functional composition. If-then-else is introduced naturally instead of "classical or" because in intuitionistic logic, there is no room for the "classical or."

The notion of "realizability" first introduced by Kleene [23, 32] is also available to program synthesis. In this case, we work with recursive functions. From an intuitionistic proof of all(X, exist(Y, F(X, Y))), we can extract a number n which represents a recursive function f satisfying all(X, F(X, f(X))). We read the relationship between n and all(X, (exist(Y, F(X, Y)) as an "object n" belongs to "type all(X, exist(Y, F(X, Y))." This reading has been generalized and developed extensively later in "constructive mathematics" where the interaction among types and objects becomes quite complex [44].

The resolution approach was begun in the late 1960s by (6) after the proposal of the resolution principle and has since been generalized (18). Resolution is an inference rule considered as a generalization of modus ponens but specialized to clauses (a clause is a disjunction of literals. A literal is an atom or a negated atom. Thus, a definite clause is a special case of a clause). With other minor rules, it forms a complete proof procedure for classical first order logic. Therefore, although resolution is an inference rule uniformly applicable to clauses, we have to be careful about the clauses to be resolved upon when aiming at synthesis. Otherwise, we might get an unexecutable program. Like the deductive approach, from the existence proof of a specification satisfying some conditions, we can extract a program in the form of a function. An application of resolution yields an if-then-else program (because resolution is a generalization of modus ponens) and induction gives a recursive program. Synthesized programs are intuitively understandable but no formal semantics seem to have been provided for them.

These two approaches have a common feature in that the existence proof of a specification is required in the first place. This necessarily results in a total function in the case of the deductive approach. But if we want just a runnable program, not the one guranteed to terminate, we do not need to prove all(X, exist(Y, F(X, Y))). On the contrary, we can not find a proof if there exists no such total function f that satisfies all(X, F(X, f(X))). The last approach, the transformational approach, takes advantage of this point. It "transforms" a specification either deductively or another way, until the specification becomes a runnable program. This emerged around the mid 1970s in functional programming (5) and in logic programming (7), and since then has been intensively studied (24, 55, 67). This approach is simple and direct, and hence seems more efficient. It does not require any existence proof. However, as is naturally anticipated, a synthesized program is not necessarily guaranteed to halt. Only partial correctness, i. e. that the computed result satisfies the specification, is guaranteed.

The common feature shared by these three approaches is that they are all highly non-deterministic and synthesis tends to go astray without guides supplied by human intelligence. We can not even assume that the synthesis process will eventually stop.

Our method, when it is seen as a method of program synthesis, belongs to the last category. Therefore, not only can a synthesized program be run in Prolog, but also its partial correctness is guaranteed. What is more, a synthesis process is guaranteed to stop unlike other synthesis methods.

2. 2. Preliminaries

2. 2. 1. General notations and conventions

The underlying logic which will be used throughout this thesis is the classical first order predicate calculus with equality in a language with denumerable variables, function symbols and predicate symbols. By a language, we mean a syntactic system in which usable symbols and usable expressions are exactly specified. As usual, however, we are concerned about only a small part of the language. For instance, when we discuss about a set of formulae S, functions and predicates not appearing in S are irrelevant. So, we use L(S) to refer to the language whose function symbols and predicate symbols are restricted to those occurring in S. If there are two languages L(S) and L(S') and L(S) is a subsystem of L(S'), we use L(S)=<L(S') to denote it.

We take "&", "\/", "->", "all" and "exist" as primitive logical connectives and "false" as a primitive proposition with a null-ary predicate symbol denoting falsity or inconsistency.

Terms are defined as usual. A function symbol at the head of a term is called the functor of the term. We regard a constant symbol as a null-ary function symbol. Variables are represented by strings beginning with an upper case letter. So X, Y, .. are variables. For formulae, we introduce the following syntax sugars:

~F	for	(F->false)
(F2<-F1)	for	(F1->F2)
(F1<->F2)	for	(F1<-F2)&(F2<-F1)

In addition, we define the precedence among connectives as:

{ ~ } < { &, \/ } < { ->, <- } < { <-> }.

Here the left-most symbol has the strongest precedence. So (~A \/ B <-> A->B) stands for ((~A)\/B))<->(A->B).

A formula without free variables is said to be a sentence or said to be ground.

If substituting a term t for a variable X in a formula F does not cause any variable in t to be bound in the resulting formula, t is said to be free for X in F. For instance, f(X) is not free for Y in exist(X, X=Y) because the substitution yields exist(X, X=f(X)) where X in f(X) becomes a bound variable. When we substitute a term t for a variable X in a formula F, we always assume that by renaming bound variables if necessary, t is free for X in F. Otherwise, the meaning of the formula might

be destroyed.

We list other main notational conventions in the following. Let us begin with set notations. For two sets S1, S2, we use:

S1=<S2 to express that a set S1 is a subset of S2,
S1+S2 for the union of two sets S1 and S2,
S1/\S2 for the intersection of two sets S1 and S2 and
S1\S2 for the result of set subtraction of S2 from S1

respectively. The empty set is denoted by {}. The cardinality of S, or the number of elements included in S, is denoted by |S|.

As for a sequence, () is used to denote the null sequence. The concatenation of sequences s1 and s2 is denoted by:

s, t or (s, t).

The length of a sequence s is denoted by:

|s|.

As syntactic variables (all possibly suffixed), we reserve C to refer to variables distinguished as continuation variables. X, Y, Z are used for individual variables. Similarly, v, w, x, y, z are reserved for a sequence of variables. They can be the null sequence.

We use s, t, u, ... for both an individual term and a sequence of terms. They can be the null sequence as well. As for syntactic variables for atoms and formulae, we use A, B for atoms and E, F for formulae.

As an application of these conventions, we introduce a short-hand for quantification as follows:

exist(x, F) for exist(X1, .. exist(Xk, F)..) and
all(x, F) for all(X1, .. all(Xk, F)..)

where x=(X1, .., Xk) (k>=0). By the way, in the context where an expression such as:

p(s)

is used where p is a predicate symbol or a function symbol and s is a sequence of terms, we always understand that |s| coincides with the arity of p. We also use another short hand:

s=t

to denote the formula (s1=t1 & ... & sk=tk) where

s=(s1,...,sk) and t=(t1,..,tk) for some integer k and s1,..,sk,t1,..,tk are all individual terms. By definition, if s or t is the null sequence, s=t becomes true. In addition to s=t, we also introduce:

s=/=t

to denote the expression ~(s=t). Consequently, when |s|=|t|=0, s=/=t denotes false.

As we mentioned before, when the distinction between sets and sequences is immaterial, we deliberately confuse them. For example, if t is a sequence of n different individual terms (t1,..,tn), t also can denote the set {t1,..,tn}. Similarly, an expression like:

sorted(t)

allows both t as set and t as sequence and denotes the sequence obtained by sorting t in alphabetic order.

Let S be a term, a sequence of terms, a formula or a sequence of formulae. S may have free variables. Therefore we use:

Fvar(S)

to denote the set of all free variables occurring in S. This notation naturally applies to E as a set. So we have F({})=F(())={}.

When we use:

E(X1,..,Xm),

we stipulate that E is an expression whose free variables are among X1,..,Xm. We also stipulate that for j(1=<j=<m), Xj denotes all occurrences of Xj in E if it actually occurs in E. Similarly when we use the notation like:

E(t),

we understand that there is an expression E(X) such that t is free for X in E(X) and E(t) is the result of substitution of t for X in E(X). This notation automatically extends to the case where t is a sequence.

Formally, a substitution T is a mapping from a finite set of variables to their values but is usually identified with and denoted by, for example,

T= {X1<-t1,..,Xk<-tk}
where X1,..,Xk are all pair-wise distinct variables

and no Xi occurs in the value tj (1=<i, j=<k, i=/=j).

We may use instead T= (x<-t) where x=(X1,..,Xk) and t=(t1,..,tk). The application of T to an expression E is denoted by (E)T. It means the simultaneous replacement of X1 by t1, .. and Xk by tk for all occurrences of X1,..,Xk in E. For the composition of two substitutions T1 and T2, we use T1*T2. (E)T1*T2 = ((E)T1)T2 always holds for any expression E.

2. 2. 2. Proof theoretic notions

One of the fundamental notions in logic is the notion of proof. The role of a proof is to illustrate how a conclusion is derived from its premises. Therefore, to define a proof system is to define what can be a premise and what is available to derive a conclusion. There are two types embodying different definitions.

On one hand, we have the sequent calculus developed by Gentszen which starts from the statement that A entails A and manipulates both the premise and the conclusion in a sequent with so many inference rules. On the other hand, we have a proof system adopted by Hilbert which starts with so many logical axioms and uses only a few inference rules. Since both are equivalent but the latter seems simpler when we talk about a proof, our proof system is supposed to be a Hilbert-style one (we use a because there can be several or more Hilbert-style proof systems. But the difference does not matter). A proof in our system starts with logical axioms and non-logical axioms. Logical axioms are predetermined formulae such as A->A which are considered as necessarily true. We include equality axiom(s) in logical axiom. Therefore, X=X, X=Y->f(X)=f(Y), ... are all logical axioms. By contrast non-logical axioms are formulae whose truth depends on how we interpret them. For example, the set of axioms for Peano arithmetic is true over the domain of natural numbers but not so over that of real numbers.

Starting with logical and non-logical axioms considered as being already proved and applying inference rules recursively to the already proved formulae, we can prove new formulae. We assume that modus ponens (infer B from A and A->B) and universal generalization (infer all(X, A) from A) are basic inference rules. We also assume "false" is a basic proposition and (false->F) for any formula F is an axiomatic scheme. An axiomatic scheme is a pattern any instance of which becomes an axiom. We use:

A1, . . , An |- F

to denote that a formula F is provable from the non-logical axioms A1, . . , An. We say F is a logical consequence of A1, . . , An. In case of n=0, F is provable only from logical axioms, thereby valid (=true regardless of the way we interpret the symbols in F). The completeness of the first order predicate calculus means the reverse direction. Namely every valid sentence is provable. An example of such a system is found in (51).

Throughout this thesis, for the sake of compactness, we do not distinguish sets and sequences when they the same members. We also stipulate that unless otherwise stated, every

free variable appearing in a formula F which is used as an axiom, is universally quantified at the front of F. But this is not the case when we apply the deduction theorem:

if A1,..,An |- B then A2,..,An |- A1->B.

In such a context, we always understand that free variables in A1 are not quantified and treated as if they were constants in the deduction. We often say that B is a logical consequence of A1 under the non-logical axioms A2,..,An if we have A2,..,An |- A1->B.

In general, we prefer set to sequence. For example, we will write S |- F where S={A1,..,An} instead of A1,..,An |- F. Let S1 and S2 be a set of formulae. We write:

S1 |- S2

to assert that S1 |- F holds for every formula F in S2. It maight be convenient to bear in mind that in our system,

S |- F iff S |- all(X,F)

holds. Therefore, there is no difference between proving F and proving F', the universal closure of F, which is obtained by universally quantifying all free variables of F at the front of F. Note also that if a term t is free for X in F, we have:

S |- F(X) implies S |- F(t)

because S |- F(X) implies S |- all(X,F(X)) and S |- all(X,F(X)) implies S |- F(t). Consequently, proving a formula and proving its alphabetic variant is the same thing.

We further extend our notation. We use an expression such as E(F) to denote that F is a subformula of E. Suppose then, there is a formula of the form:

A<->F.

And also suppose A is an atom and F is a formula. We consider it as a definition of the procedure where A is the procedure head (definiand) and F is the procedure body (definiens) in analogy to a function definition. Finally, suppose that there is a formula:

E(A)

which has an occurrence of A. Replace an occurrence of A in E(A) (a procedure call of A in E) by F (the procedure body). We call it an unfolding of A using A<->F. The reverse operation, the replacement of an occurrence of F in F(F) by A, resulting F(A), is called a folding operation. Note that we have:

A<->F |- E(A)<->E(F).

A theory is a set of formulae such that no new formulas are deduced from the set by any inference rule. Accordingly, there are many ways of defining a theory. But the most typical way is to declare a set S of formulae as non-logical axioms and identify S with the set formulae provable from S, which is naturally closed w.r.t. every inference rule. In this sense, we use a set of formulae and a theory interchangeably.

(Conservative extension):

Let S be a theory and p be a new n-ary predicate symbol alien to L(S). Extend S by adding to it a formula:

p(x) <-> F(x)

such that x is a sequence of n new variables, F is a formula in L(S) and the set of free variables occurring in F coincides with x as set. The resulting new theory S'=S+{p(x)<->F(x)} is called an extension of S by the introduction of a new predicate p with the defining formula p(x)<->F(x).

It is well known (for example [59]) that if S is a theory and S' is an extension by an introduction of new predicate, then, for any formula F in L(S),

S |- F iff S' |- F.

This means that the extension does not change the set of provable formulae as far as L(S) is concerned. In general, given two theories S and S' such that L(S)=<L(S') and S=<S', we say S' is a conservative extension of S when the above relation holds. So an extension by the introduction of new predicates is a (very simple) example of conservative extension. As is evident, conservative extension is a transitive relation among theories.

2. 2. 3. First order programs

In this subsection, we introduce first order programs. They are extension of definite clause programs and are intended to play the role of (hopefully runnable) specifications. They allow a certain type of first order formulae to appear in the body of a clause, thereby enhancing expressive power. A formula which can be the body of a clause is called an admissible formula.

(Admissible formulae):

The class of admissible formulae is defined inductively as follows:

(A-1) An atom is an admissible formula.

Let F1 and F2 be admissible formulae. Then,

(A-2) F1&F2, F1\/F2 and exist(x, F1) are admissible formulae.

(A-3) all(x, F1->F2) is an admissible formula if x=<Fvar(F1) holds. When |x|=0, this is equivalent to an implication (F1->F2).

(System/user predicate):

There are predicates such as "false" and "=" whose meanings are inherently determined. In other words, they are not expected to express other meanings or we should not be able to change their meanings. In this sense, we call "false" and "=" system predicates. Of course, it is possible to enlarge a list of system predicates by, for example, adding ">" (greater-than relation). On the other hand, there are predicates which we can or we want to define through a first order program (see below). Such a predicate is called a user predicate.

Later we introduce other classes of predicates. They will be called closure predicates and continuation predicates. They must be disjoint to the class of system predicates and that of user predicates. But the boundaries among classes are rather arbitrary so that there is freedom concerning how to divide available predicates into such disjoint classes. Therefore, for simplicity, we first regard every predicate other than system predicates as a user predicate and when closure and continuation predicates are introduced, we see to it that the predicates in the latter class are disjoint to system and user predicates, thereby retaining disjointness.

(System/user atom):

An atom containing a user predicate is called a user atom. An atom containing a system predicate is called a system atom.

(First order program):

A first order program is a finite set of first order clauses. A first order clause is a formula A <- F such that

(F-1) A is a user atom and
(F-2) F is an admissible formula.

A is called the head, F the body. A predicate contained in the head is called the head predicate of the clause. A clause whose head predicate is p is called a clause about p. It is convenient here to introduce more terminologies relating to first order programs which are intensively used in this thesis.

(Iff-definition):

The iff-definition of an n-ary predicate p in a first order program P, denoted by iff(p), is defined as follows:

Let p(u1)<-E1,..,p(ul)<-El (l>=0) be an enumeration of clauses in P about the relation p and yj(1=<j=<l) be an enumeration of free variables in p(uj)<-Ej. Prepare a sequence of n new variables and let it be x. Then, if l>0, let:

$$iff(p) = p(x) <-> exist(y1, (x=u1 \ \& \ E1)) \backslash/ \ldots \backslash/ exist(yl, (x=ul \ \& \ Ek)).$$

Otherwise, put:

$$iff(p) = p(x) <-> false.$$

For a program P, put:

$$iff(P) = \{ iff(p) \mid p \text{ is a predicate symbol occurring in } P \}$$

and call iff(P) the iff-definition of P.

(Equality theory):

Let S be a set of function symbols or a program. The weak equality theory for S, denoted by Ew(S), is defined as:

$$\{ f(x)=f(y) -> x=y, \ f(x)=/=g(y) \mid (x,y) \text{ is a sequence of different variables. } f,g \text{ are distinct function symbols in } S \}$$

For a program P, we also introduce Eq(P), the equality theory for P by putting:

$$Eq(P) = Ew(P) + \{\, X =/= t(X) \mid t \text{ is a term in } L(P) \text{ and } X \text{ is a variable occurring in } t \,\}$$

(Completion):

For a program P, comp(P), the completion of the program is a set of formulae defined by:

$$comp(P) = Eq(P) + iff(P)$$

(Semantics):

As we can not expect a first order program to have a canonical model like the least model except in rare cases, we consider comp(P) as the logical content of the program P. In other words, we understand the denotation of P as comp(P). This is compatible with what has been standard in logic programming (9, 38).

2. 3. Conversion to basic programs

In this section, we show every first order program can be transformed to a more restricted form called a basic program.

(Basic program) :

A basic program is a finite set of basic clauses. A basic clause is a first order clause which fits into either one of the following forms.

A0 <- A1 &. . & Ak
where A0, . . , Ak (k>=0) are atoms and
none of A1, . . , Ak is "false. "

This is nothing other than a definite clause.

A0 <- all(y, A1->A2)
where y=<Fvar(A1) and A0, A1 and A2 are atoms.

This type is called an extended clause.

A goal of the form all(y, A1-> A2) where A1 and A2 are atoms and y=<Fvar(A1) is called an extended goal. An atom and an extended goal are called a basic goal. Likewise, a conjunction of basic goals is called a basic body.

For a basic program P, Pe denotes the set of extended clauses included in P. It is called the extended part of P. The rest P\Pe, denoted by Pd, is called the definite part of P.

Transformation from a first order program to a basic program is achieved by the introduction of new predicates. Let Nb(P) be the total number of occurrences of non-basic subformulae in the body of a clause in a first order program P.

(Lemma 2. 3. 1)
Let P be a first order program. Then there are first order programs P' and P'' such that:

(1) iff(P') is a conservative extension of iff(P) by introduction of (possibly no) new predicates,
(2) iff(P') |- iff(P'') and iff(P'') |- iff(P'),
(3) Nb(P'')<N(P) if Nb(P)>0,
(4) L(P'')>=L(P) and
(5) P'' and P have the common function symbols.

(Proof) Suppose A<-E is a clause in P whose body E is not basic.

(B-1) E= E1&E2.
(B-1-1) Both of E1 and E2 are an extended goal or neither of E1 nor E2 is basic. Introduce new predicates p1, p2 and put

```
P' = {p1(x1)<-E1, p2(x2)<-E2}+P  and
P'' = P\{ A<-E }+
        { A<-p1(x1)&p2(x2), p1(x1)<-E1, p2(x2)<-E2 }
```

where xi=Fvar(Ei), i=1,2.

Since iff(P') (={p1(x1)<->E1, p2(x2)<->E2}+iff(P)) is the introduction of new predicates p1 and p2 to iff(P) with defining formulae E1 and E2 resectively, (1) is obvious.

The difference between P' and P'' is that of A<-E and A<-p1(x1)&p2(x2) and furthermore,

```
iff(P')  |- A<-E <-> A<-p1(x1)&p2(x2)   and
iff(P'') |- A<-E <-> A<-p1(x1)&p2(x2)
```

holds. Thereby (2) holds too. For (3), we have:

```
Nb(P'') =Nb(P\{A<-E})+Nb({ A<-p1(x1)&p2(x2),
                           p1(x1)<-E1, p2(x2)<-E2 })
        =Nb(P)-Nb({A<-E1&E2})+Nb({ p1(x1)<-E1, p2(x2)<-E2 })
        =<Nb(P)-1
```

(4) and (5) are apparent.

(B-1-2) E1 is an extended goal and E2 is a conjunction of atomic goals. Introduce a new predicates p1 and put:

```
P' ={p1(x1)<-E1}+P and
P'' =P\{ A<-E }+{ A<-p1(x1)&E2, p1(x1)<-E1 }
```

where x1=Fvar(E1). The proof is similar to (B-1-1).

(B-2) E=E1\/E2
Put P'=P and P''=P\{ A<-E }+{ A<-E1, A<-E2 }. (1), (2), (4), (5) are obvious from iff(P'')=iff(P')=iff(P). (3) follows from:

```
Nb(P'') = Nb(P\{ A<-E }+{ A<-E1, A<-E2 })
        = Nb(P)-(Nb({A<-E1\/E2})-Nb({ A<-E1, A<-E2 })
        =< Nb(P)-1.
```

(B-3) E=exist(y,E1)
Put P'=P and P''=P\{A<-E}+{A<-E1}. The proof is similar to (B-2).

(B-4) E=all(y,E1->E2)
Introduce new predicates p1, p2 and put:

```
P' ={ p1(x1)<-E1, p2(x2)<-E2 }+P and
P'' =P\{ A<-E }+{ A<-all(y,p1(x1)->p2(x2)) }
```

where xi=Fvar(Ei), i=1,2. As x1=Fvar(E1)>=y, A <- all(y,p1(x1) -> p2(x2)) is an extented clause. The proof is similar to (B-1-1).

{ Theorem 2.3.1 }

For a first order program P0, there exists a basic program P such that comp(P) is a conservative extension of comp(P0) and P and P0 have the same function symbols.

(Proof)

{ Lemma 2.3.1 } says that for every non-basic first order program P', there is another program P'' such that iff(P'') is a conservative extension of iff(P'), Nb(P'') < Nb(P') and P' and P'' have the same function symbols. So if P0 is not already a basic program, we can obtain a series of programs P0, P1, P2, .. Pn where Nb(Pn)=0 and for 1=<i=<n, iff(Pi) is conservative extension of iff(Pi-1), Nb(Pi)>Nb(Pi-1) and Pi-1 and Pi have the same function symbols.

Nb(Pn)=0 means Pn is a basic program. Moreover, as the relation of conservative extension is transitive, iff(Pn) becomes a conservative extension of iff(P0). Obviously P0 and Pn have the same function symbols. Put P=Pn.

Then from Eq(P0)=Eq(P) and iff(P) is a conservative extension of iff(P0), it is easy to see comp(P) (= iff(P)+Eq(P)) is a conservative extension of comp(P0) (= iff(P0)+Eq(P0)).

End of theorem 2.3.1.

We present here an algorithm, based on the theorem, which transforms a first order program P0 to a basic program P such that P0 and P have the common function symbols and comp(P) is a conservative extension of comp(P0).

```
----------------( Preprocessing Algorithm )--------------------

        Let P0 be a first order program.

( step 1 )
        Put S=P0;
        Until S becomes a basic program
        Do      Take a non-basic clause A<-E in S;

        ( case 1 )      E=E1&E2 and both of E1 and E2 are
                        an extended goal or neither of E1 nor E2
                        is basic.
        Introduce new predicates p1, p2;
        Put S=S\{ A<-E }+{ A<-p1(x1)&p2(x2),
                                p1(x1)<-E1, p2(x2)<-E2 }
                where xi=sorted(Fvar(Ei)), i=1,2.

        ( case 2 )      E=E1&E2 and E1 is an extended goal and
                        E2 is a conjunction of atomic goals.
        Introduce a new predicates p1;
        Put S=S\{ A<-E }+{ A<-p1(x1)&E2, p1(x1)<-E1 }
                where x1=sorted(Fvar(E1))

        ( case 3 )      E=E1\/E2
        Put S=S\{ A<-E }+{ A<-E1, A<-E2 }

        ( case 4 )      E=exist(y,E1)
        Put S=S\{A<-E}+{A<-E1}

        ( case 5 )      E=all(y,E1->E2)
        Introduce new predicates p1, p2;
        Put S=S\{ A<-E }+{ A<-all(y,p1(x1)->p2(x2)),
                                        p1(x1)<-E1, p2(x2)<-E2 }
                where xi=sorted(Fvar(Ei)), i=1,2.


        oD

( step 2 )
        Put P=S;
        Return P

----------------------( End of Algorithm )----------------------
```

2. 4. Outline of compilation process

In this section, we describe our compilation process globally to help the reader get a global view which is often lost in the forest of details. As is usual, we begin with definitions.

(Executable goal/clause/program) :

An executable goal is an atom or a formula of the form all(x, s=/=t). An executable clause is one whose body includes only executable goals. An executable program is a set of executable clauses. The result of compilation becomes an executable program.

(User program/clause/function) :

We need to distinguish an original program P supplied by a user intended for compilation from the compiled one. So we call P a user program to emphasize that P is a user written, not a compiler generated program. Likewise, we distinguish the symbols included in P from other symbols which are brought into P at compile time.

Any function symbol (including constant symbol by our convention) occurring in a user program P is called a user function symbol and or a user function for short. The set of user function symbols is denoted by Uf(P) or Uf when P is understood. Similarly, as we defined before, all predicate symbols except system predicates "false" and "=" occurring in P are called user predicates and an atom containing a user predicate is called a user atom. A user clause is one whose head is a user atom. Therefore, every clause in a user program is a user clause. Later we will have closure predicates, continuation predicates and continuation functions as other types of symbols all of which must be alien to P but might be included in the compiled program.

We would like to regard a compilation process as one from a basic program to an executable program, not one from a first order program P0. This is firstly because the preprocessing algorithm automatically converts P0 into a basic program and secondly because that phase is not a main concern in this thesis.

Let P be a basic program. The compilation of P goes through 3 phases. The first phase is a "mode labelling" phase. It analyzes P and assigns a "mode pattern" such as p(+, -) to an atom which may be involved in the computation of a universally quantified goal.

This phase is intended to check the compilability of a source program. We would like the reader to note that "our mode

pattern" has little to do with mode declaration in DEC-10 Prolog. It is designed to be an approximation to an atom's instantiation pattern at run-time. On the other hand, ours stems from the neeed to indicate which argument of a predicate may have a term including universally quantified variables.

Suppose for example that the mode pattern p(+,-) is assigned to the specific occurrence of atom p(s,t). Then it means that s, the term corresponding to +, must contain only global (not universally quantified) variables at the time of execution, whether it is located at the head or in the body of a clause. So we can say that s is in the input position of p(+,-). Similarly as for t in p(s,t), the pattern means that it is in the output position and after computing p(s,t), it must not include any universally quantified variables even though it may include such variables before the computation. When the assignment of such mode patterns is impossible, we have to give up compiling P. But once P has passed this phase, the rest is assured to succeed. The same predicate may be assigned different modes as a result of this phase.

The second phase is a "term labelling" phase. It labels goals further. This time a goal labelled with a mode pattern is labelled with a term. Other atoms remain intact.

Suppose an atom occurrence A is labelled with a term f(t1,..,tk). Generally it can be interpretable as:

> (t1,..,tk-1) is a frame pushed on to the existing stack "tk" with the return address f.

The functor symbol f is therefore chosen uniquely w.r.t. that occurrence. because a return address must be unique. In practice, we prepare a new function symbol f for each occurrence of an atom. The arguments t1,..tk are actually variables. They are to bound to terms representing all information required to continue computation after A has been computed.

We use t-m(P) to refer to the result of the second phase. A new function symbol introduced at this phase is called a continuation function symbol or continuation function for short. The set of continuation function symbols in t-m(P) is denoted by Cf(t-m(P)) or Cf when t-m(P) is understood.

The last phase is "translation." Each labelled clause in t-m(P) is translated into one or more executable clauses using new predicates called closure predicates and continuation predicates. Other clauses bearing no labels remain intact during translation. The point at this phase that the computation of all(y, A1->A2) is reducible to, say all(y, x=t -> A2) under a certain condition (which was already checked at mode labelling phase) and the latter is equivalent to all(y, x=/=t)\/exist(y, x=t & A2) in a Herbrand universe

if y=<Fvar(t) (this was also checked at mode labelling phase). We use Pc to refer to the result of the last phase.

Since every goal appearing in Pc is an atom or a formula of the form all(y, s=/=t) and the latter is correctly computable by negation-as-failure rule, Pc is actually executable and can be seen, for example, as a Prolog program.

Here, we have to mention the soundness of the compiled program Pc. Roughly speaking, it is possible to prove that an atom derivable from Pc is a logical consequence of comp(P), the completion of the original program P. It is also provable that an atom which a top-down interpreter finitely fails to prove from P, is refutable from comp(P). But there is an exception where P has no functional symbols other than constants. In this case, we require that the completion of P is "model independent" to mentain the soundness of our method. The precise definition of "model independentness" is given in section 2. 10. .

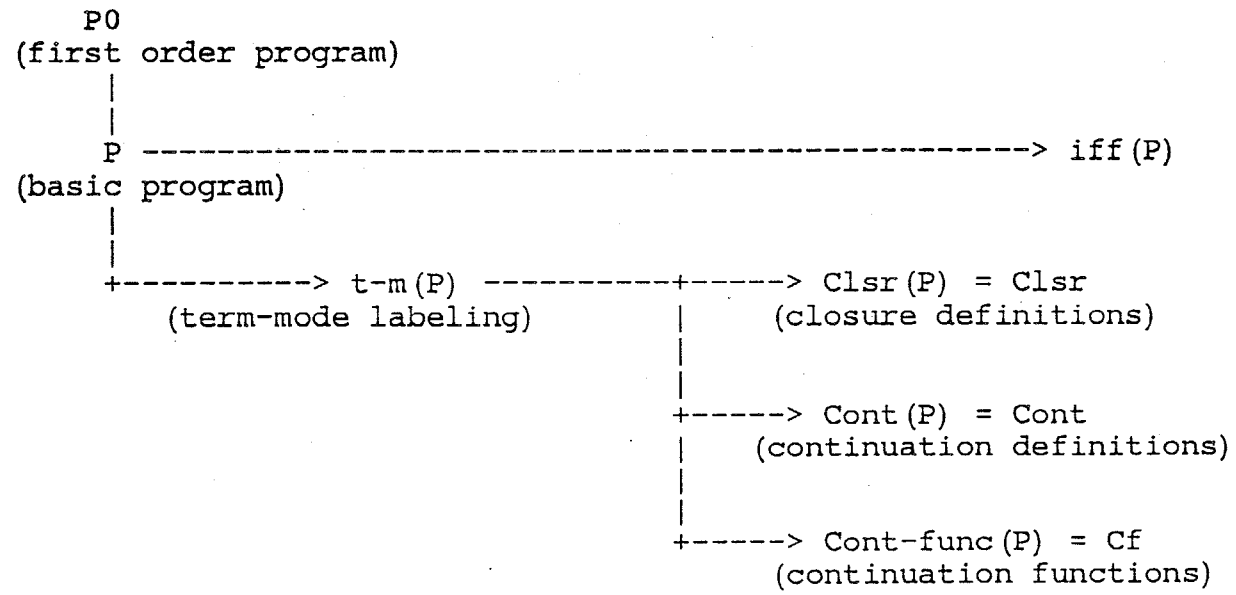
P0 : first order program P : basic program =(Pe+Pd)

Clsr : the set of clauses about closure predicates for t-m(P)
Cont : the set of clauses about continuation predicates for t-m(P)
Aux : { A<->F | A<-F is in Cont }

P' : the set of definite clauses about user predicates in P
Clsr' : the set of executable clauses about closure predicates
Cont' : the set of executable clauses about continuation predicates

Uf : user function symbols in P.
Cf : continuation function symbols introduced during term-mode labelling.

Pc : compiled program



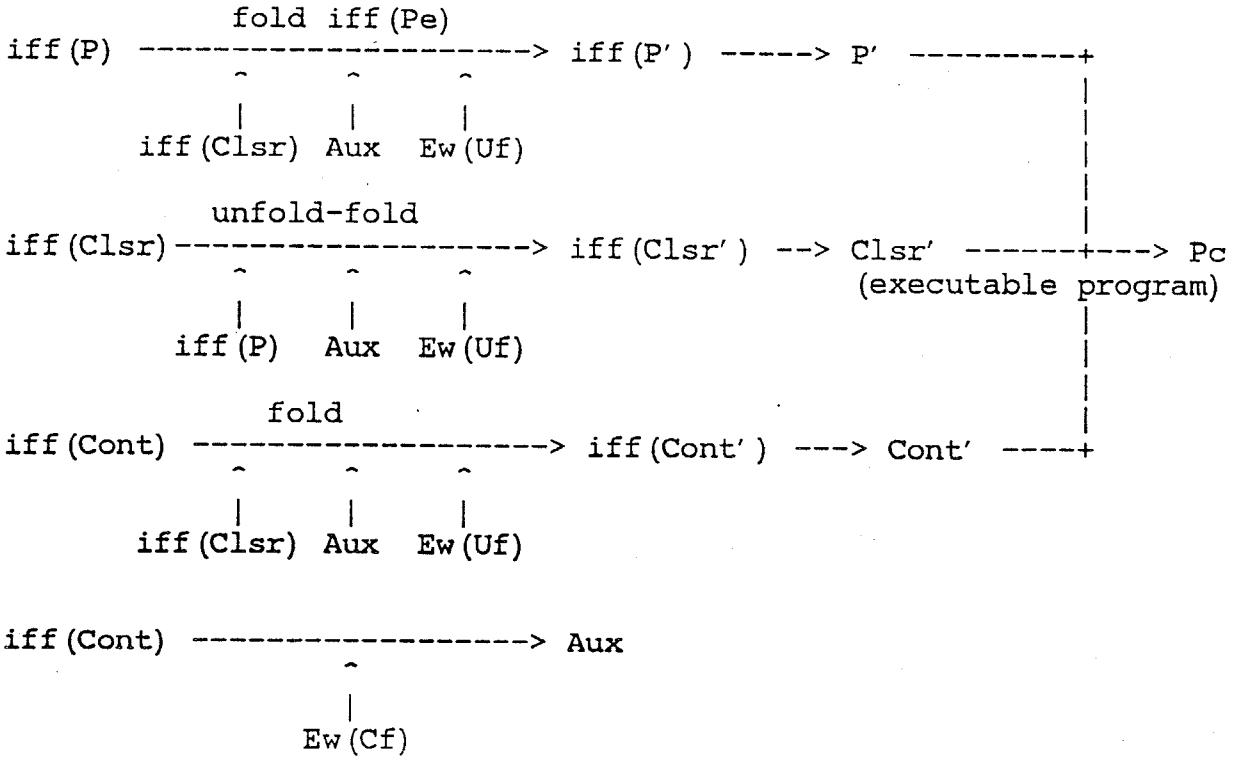


Fig. 2.4.1. A diagram of compilation process

2. 5. Mode labelling

In this section, we describe the first phase of compilation. The output of this phase is failure, in this case we have to abort compilation, or a user program annotated with mode patterns. To describe the mode labelling algorithm, we need some terminologies.

(Mode pattern) :

Let r be an n-ary predicate symbol. A mode pattern for r is a pattern r(e1,..,en) such that ei="+" or ei="-" (1=<i=<n). Each ei (1=<i=<n) is called a mode indicator. Consider an atom r(u1,..,un) and it i-th argument ui (1=<i=<n). We say ui is an input (output) argument under the mode pattern r(e1,..,en) if a mode indicator ei corresponding to ui is "+" ("-") respectively.

(Moded atom) :

An atom labelled by a mode pattern for its predicate is called a moded atom.

(Input/Output sequence) :

Let

r(e1,..,en)
r(u1,..,un)

be a moded atom. Pick up all input (output) arguments from (u1,..,un) preserving their order. The resulting sequence is called the input (output) sequence of r(u1,..,un) under r(e1,..,en) respectively. For example, if we have:

between(+, -, +)
between(a, b, c),

the input sequence is (a, c) and the output sequence is (b).

The I/O notation of a moded atom $\begin{matrix} r(e1,..,en) \\ r(u1,..,un) \end{matrix}$ is an expression defined as :

$$\begin{matrix} r(e1,..,en) \\ r(s1,..,sk,,t1,..,tl) \end{matrix}$$

where (s1,..,sk) is the input sequence and (t1,..,tl) is the output sequence of r(u1,..,un) under the mode pattern r(e1,..,en).

In the above example, its I/O form notation is:

between(+, -, +)
between(a, c, , b).

It is easy to see that a moded atom and its I/O form notation have one-to-one correspondence. From this section on, for brevity, we adopt a short hand notation for a sequence of terms and I/O form notation for a moded atom. For example,

$$r^{m}(s, , t)$$

automatically means that s (t) is the input (output) sequence of the original atom r(u1,..,un) under a mode pattern m.

(Well moded clause):

A moded clause is a basic clause with mode patterns on its atoms (not necessarily all). A well moded clause is one which fits one of the following patterns:

type 1:

$$A0 \leftarrow all(y, (q^{m1}(s1, , t1) \rightarrow A2))$$
where $y = <Fvar(t1)$

type 2:

$$p^{m0}(s0, , t0) \leftarrow all(y, (A1 \rightarrow A2^{m2}))$$
where $Fvar(t0) + Fvar(all(y, (A1 \rightarrow A2))) = <Fvar(s0)$

type 3:

$$p0^{m0}(s0, , t0) \leftarrow p1^{m1}(s1, , t1) \& \ldots \& pk^{mk}(sk, , tk)$$
where for i $(0 = <i = <k)$
$Fvar(si) = <Fvar(s0, s1, t1, .., si-1, ti-1)$ and
$Fvar(t0) = <Fvar(s0, s1, t1, ..., sk, tk)$

m0 is called a head mode pattern or a head mode for short. A condition posed on the variables of a clause is called the variable condition on that clause. As is suggested by the presentation of variable conditions, we consider a definite well moded clause as an ordered clause. Namely goals in a definite clause are ordered in the left-to-right textual order.

----------(Mode labelling algorithm)----------

Let P be a basic program and Pe be the extended part of P.

(step 1)
This step constructs a set Pe' of moded clauses from P. Pe' consists of moded clauses of type 1. It is defined as:

```
                    q(e1,..,en)
{ A0 <- all(y,  q(u1,..,un) -> A2) |
        A0 <- all(y, (q(u1,..,un) -> A2)) is in Pe and
        for i(1=<i=<n), define,
        ei='-' if ui includes some variable in y
        ei="+" otherwise }
```

Recall that in an extended clause A0 <- all(y, A1->A2), each variable in y must have an occurrence in A1. Therefore, Pe' actually becomes a set of well moded clauses of type 1. Mode labelling algorithm is one to try to find such S that:

(M-1) S is a set of well moded clauses,
(M-2) Pe' =<S and
(M-3) Let m be a mode pattern appearing in S and
suppose it is for predicate r. Then,
for every clause Cl in P whose head predicate is r,
S includes the well moded clause such that
its head predicate is r and it is obtained by labelling Cl.

Though there may be a more sophisticated way to find such set, we present a naive algorithm.

(step 2)
This step is iteration until saturation.

1: Let S be Pe' ;

2: Let I be the set of mode labels occurring in Pe' ;
Until I saturates or "failure" is reported
Do
for every pair of a mode pattern m0 in I and
a clause Cl in P such that m0 is a mode pattern
for the head predicate of Cl,

/* let m0=p(e1,..,en) and Cl=p(t1,..,tn)<-E */

3: Label the head of Cl by m0;

4: Let Vin be the set of variables occurring in the input sequence of p(t1,..,tn) under m0;

5: Let Vout be the set of variables occurring in the output sequence of p(t1,..,tn) under m0;

```
( case 1 ) /* a well moded clause of type 2 is generated */
        Cl is an extended clause. Put:
        Cl = p(t1,..,tn)<- all(y, A1->r(s1,..,sk)).

6:      If Vout+Fvar(all(y, A1->r(s1,..,sk)))=<Vin
        then    Do
7:                      Define m2, a mode pattern for r as:

                        m2= r(e1,..,ek) where e1=..=ek="+"

8:                      Add m2 to I;
                                m0                      m2
9:                      Add p(t1,..,tn)<- all(y, A1->A2) to S
                oD
10:     else    stop and report "failure"
(end of case 1)

( case 2 ) /* a well moded clause of type 3 is generated */
        Cl is a definite clause.
        Let it be p(t1,..,tn)<-A1&..&Ak (0=<k)

        For i=1 to k
        Do      /* let Ai=pi(s1,..,sl) */
11:             Define mi, a mode pattern for pi, as:

                mi=pi(d1,..,dl)
                        where  for each j(1=<j=<l),
                        dj="+" if Fvar(sj)=<Vin
                        dj="-" otherwise;

12:             Add mi to I;
13:             Add Fvar(Ai) to Vin;
14:             Label Ai by the mode pattern "mi"
        oD;
15:     If Vout =< Vin and "=" is not assigned mode =(-,-)
16:     then    Add Cl ( with labelled mode patterns ) to S
17:     else    Stop and report "failure"
(end of case 2)

    oD;
    Return S
```

----------(End of mode labelling algorithm)----------

We remark that this mode labelling algorithm always halts because when predicate symbols are finite, there can be only finitely many mode patterns. As is easily checked, a successful return not only means that the output S is a set of well moded clause including Pe ((M1), (M2)), but also means that since S is obtained by saturation, S is closed w.r.t. a new mode clause generation thereby (M3).

(Compilability):

If the mode labelling algorithm applied to a basic program P halts successfully, P is said to be compilable. We use m(P) to denote the set of well moded clauses generated by the algorithm when P is compilable. Hereafter, we will only deal with compilable programs.

2. 6. Term labelling

Given m(P), a set of well moded clauses obtained by the mode labelling algorithm, t-m(P), a set of clauses labelled further by terms, is obtained by the term labelling algorithm. Terms are chosen so that they include necessary and sufficient information to make it possible to carry out the remaining computation after the atom labelled by the term has been computed.

----------(Term labelling algorithm)----------

(step 1)
Rename variables so that no two clauses included in m(P) has a common variable.

(step 2)
Prepare a new variable and let it be C.

(step 3)
Define t-m(P) as:

$$\{\ A0 \overset{m1,\ f1(z1)}{\leftarrow} all(y,\ q(s1,,t1) \overset{m1}{\rightarrow} A2)\ \mid$$

A0<- all(y, (q(s1,,t1)->A2)) is in m(P),
"f1" is a new function symbol and
z1 = sorted((Fvar(t1)+Fvar(A2))\y) }+

$$\{\ A0 \overset{m0}{\leftarrow} all(y,\ (A1 \overset{m2,\ f2}{\rightarrow} A2))\ \mid$$

A0<-all(y, (A1->A2)) is in m(P) and
"f2" is a new 0-ary function (constant) symbol }+

$$\{\ p0(s0,,t0) \overset{m0}{\leftarrow} p1(s1,,t1) \overset{m1,\ f1(z1,C)}{\&\ldots\&} pk(sk,,tk) \overset{mk,\ fk(zk,C)}{\mid}$$

p0(s0,,t0) <- p1(s1,,t1) &...& pk(sk,,tk) (k>=1)
is in m(P),
"f1",.., "fk" are new function symbols and
zi = sorted(Fvar(s0, s1, t1,.., si-1, ti-1, si)
/\Fvar(ti, si+1, ti+1,.., sk, tk, t0))
for i (1=<i=<k) }

----------(End of term labelling algorithm)----------

(Continuation variable/function):

A new variable which is introduced at (step 2) is

called the continuation variable for t-m(P). We usually use C to stand for the variable.

New function symbols introduced by the term labelling algorithm are called continuation function symbols. Cf(t-m(P)) denotes the set of all continuation function symbols occurring in t-m(P). When t-m(P) is understood, we use Cf for Cf(t-m(P)).

2.7. Closure and continuation definitions

This section is not a part of the compilation algorithm. Rather, it provides definitions for section 2.9. and section 2.10.. We introduce a pair of new predicate symbols for each mode pattern appearing in t-m(P) with their defining formulae.

(Closure/continuation predicate):

Let m be a mode pattern for an n-ary predicate p and x (y) be the input (output) sequence of the moded atom p(X1,..,Xn) under m respectively where X1,..,Xn are n new variables. Let us write it in I/O form:

$$p^{m}(x,,y)$$

We introduce the universal continuation form which correspond to pattern m:

$$p'(x,C) \leftarrow\!\rightarrow \mathrm{all}(y,\ p(x,,y) \rightarrow \mathrm{cont\text{-}p'}(C,y)).$$

p' is a |x|+1 -ary new predicate symbol called a universal closure predicate for m while cont-p' is a |y|+1 -ary new predicate symbol called a universal continuation predicate for m. They are assumed to be selected uniquely to m. Hereafter, we omit the word "universal" from terms such as universal closure predicate, universal continuation predicate and so on. Let C be the continuation variable for t-m(P).

(Closure definition):

The closure definition by t-m(P), denoted by Clsr(t-m(P)) or Clsr when t-m(P) is understood, is a set of first order clauses defined as:

$$\{\ p'(x,C) \leftarrow \mathrm{all}(y,\ p^{m}(x,,y) \rightarrow \mathrm{cont\text{-}p'}(C,y))\ \mid$$

m is a mode pattern for p appearing in t-m(P),
$p^{m}(x,,y)$ is the I/O form of $p^{m}(X1,..,Xn)$ where
X1,..,Xn are n new variables,
p' is the closure predicate for m and
cont-p' is the continuation predicate for m }

This is a set of universal continuation forms represented in the form of a first order program. In practice, as iff(Clsr) = { A<->E | A<-E is in Clsr } holds, iff(Clsr) becomes a set of universal continuation forms.

(Continuation definition):

The continuation definition by t-m(P), denoted Cont(t-m(P)) or Cont when t-m(P) is understood, is a set of first order clauses defined as:

```
{ cont-q' (f1(z1), y1)<-all(y, y1=t1 -> A2)        |

                        m1, f1(z1)
        A0<-all(y, q(s1, , t1) -> A2) is in t-m(P),
        y1 is a sequence of |t1| new variables,
        cont-q' is the continuation predicate for m1,
        if |t1|=0, replace all(y, y1=t1 -> A2) by A2 }+

{ cont-r' (f2)<-false     |

        m0                    m2, f2
        A0<- all(y, A1->r(s, , )) is in t-m(P),
        cont-r' is the continuation predicate for m2 }+

{ cont-pi' (fi(zi, C), yi)<-
        all(vi,  yi=ti & pi+1(si+1, , ti+1) &. . & pk(sk, , tk)
                        -> cont-p0' (C, t0))       |

             m0      m1, f1(z1, C)      mk, fk(zk, C)
        p0(s0, , t0)<-p1(s1, , t1)&. . & pk(sk, , tk)  (k>=1) is
        in t-m(P), and for each i (1=<i=<k),
        pi' and cont-pi' are the closure and continuation
        predicate for the mode pattern mi respectively,
        yi is a sequence of |ti| new variables,
        vi = sorted(Fvar(ti, si+1, ti+1, . . , sk, tk, t0)
                \Fvar(s0, s1, t1, . . , si-1, ti-1, si)) and
        if i=k and |tk|=0, replace the body by
        cont-p0' (C, t0)}

/* We call vi local variables and zi global ones.
For the characterization see our remark below. */
```

We remark that it might be useful to keep in mind the following characterization of variables vi and zi. For a clause:

```
p0(s0, , t0) <- p1(s1, , t1)&. . &pk(sk, , tk)  (k>0),
```

consider the following sequence:

```
s0, p1(s1, , t1), . . , pk(sk, , tk), t0.
```

Then, vi (1=<i=<k) is exactly the set of variables occurring only to the right of pi(si, , -), i.e. the variables occurring only in its subsequence:

```
ti, pi+1(si+1, , ti+1), . . , pk(sk, , tk), t0.
```

On the other hand, zi (1=<i=<k) is characterized as the

set of variables common in the two subsequences:

s0, p1(s1, , t1), . . , pi-1(ti-1, si-1), si and
ti, pi+1(si+1, , ti+1), . . , pk(sk, tk), t0.

Note that Fvar(A)=Fvar(F) holds for every clause A<-E in Cont.

{ Auxiliary definition }:

The auxiliary definition for t-m(P) is denoted by Aux(t-m(P)) or Aux when t-m(P) is understood, and defined as:

{ A<->F | A<-F is in Cont(t-m(P)) }

Note that Fvar(A)=Fvar(F) for every A<-F in Cont(t-m(P)).

{ Lemma 2.7.1 }
Let P and t-m(P) be a compilable basic program and the result of its mode-term labelling, Cf the set of continuation function symbols in t-m(P), Cont the continuation definition for t-m(P) and Aux the auxiliary definition for t-m(P) respectively. Then,

iff(Cont)+Ew(Cf) |- Aux

(Proof)
Let cont-q' be an arbitrary continuation predicate in t-m(P) and cont-q'(u1)<-E1, . . , cont-q'(ul)<-El be an enumeration of clauses about cont-q' in Cont. We suppose the arity of cont-q' is n. In order to prove the lemma, we have to show that for each j (1=<j=<l), cont-q'(uj)<->Ej is a logical consequence of iff(Cont)+Ew(Cf).

By inspecting the definition of Cont, we know that each head cont-q'(uj) (1=<j=<l) can be written as:

cont-q'(f(z), y)

where f is a continuation function symbol which does not occur anywhere else, (z, y) is a sequence of different variables and every free variable in Ej is also that of cont-q'(uj). Therefore, iff(cont-q') can be written as:

cont-q'(x) <-> exist((z1, y1), x=(f1(z1), y1) & E1)\/
. . .
exist((zl, yl), x=(fl(zl), yl) & El)

Here, x is a sequence of n new variables. In what follows, we show that this formula, included in iff(Cont), is logically equivalent to cont-q'(uj)<->Ej for each j(1=<j=<l) under the non-logical axiom Ew(Cf).

Rename (zj,yj) as (zj',yj') where (zj',yj') is a sequence of new variables. Then, by substituting (fj(zj),yj) for x, we have:

```
cont-q' (fj(zj'),yj')
    <-> exist((z1,y1), (fj(zj'),yj')=(f1(z1),y1)&E1) \/
                                ...
        exist((zj,yj), (fj(zj'),yj')=(fj(zj),yj) & Ej) \/
                                ...
        exist((zl,yl), (fj(zj'),yj')=(fl(zl),yl) & El).
```

First, by appealing to the formula:

```
fi(xi)=/=fj(xj)  ( i=/=j, 1=<i=<l and (xi,xj) is
                     a sequence of different variables )
```

which is included in Ew(Cf), each formula:

```
(fj(zj'),yj')=(f1(zi),yi) & Ei   ( i=/=j, 1=<i=<l )
```

is proved to be false. Therefore, we get:

```
cont-q' (fj(zj'),yj')<->
        exist((zj,yj), (fj(zj'),yj')=(fj(zj),yj) & Ej)
```

Secondly, by appealing to the formula:

```
fj(xj)=fj(zj)->xj=zj     ( 1=<j=<l and (xj,zj) is a sequence
                             of different variables )
```

for the continuation function symbol "fj" included in Ew(Cf), we see the formula (fj(zj'),yj')=(fj(zj),yj) is logically equivalent to (zj',yj')=(zj,yj) so that we have:

```
cont-q' (fj(zj'),yj') <->
        exist((zj,yj), (zj',yj')=(zj,yj) & Ej(zj,yj))
```

which is in turn equivalent to:

```
cont-q' (fj(zj'),yj') <-> Ej(zj',yj').
```

Since this is an alphabetic variant of cont-q' (fj(zj),yj) <-> Ej, i.e. the formula we have to prove, we are done.

For later use, we introduce some terminologies.

{ Closure/Continuation atom/clause }:

An atom containing closure/continuation predicates is called a closure/continuation atom respectively. If a clause has a closure/continuation atom as its head,

it is called a closure/continuation clause respectively. An atom is said to be normal if it is either a system atom or a user atom.

2. 8. Translation

A basic program P now labelled with terms and modes is finally translated into an executable program. This section describe the translation algorithm.

The output of translation, denoted by Pc, will be the union of a set of executable clauses for user predicates denoted by P', that for closure predicates denoted by Clsr' and that for continuation predicates denoted by Cont'.

----------(Translation algorithm)----------

Let P be a compilable basic program and t-m(P) the output of the term-mode labelling algorithm. Recall no two clauses in t-m(P) share free variables thanks to the renaming step of the term labelling algorithm. Also recall that t-m(P) has a distinguished variable C called the continuation variable for t-m(P).

```
{ step 1 }
        For each mode pattern appearing in t-m(P),
        prepare two new predicate symbols, one as a closure
        predicate and the other as a continuation predicate.

{ step 2 }      /* Definite user clauses remain unchanged. */
1:      Let P' be Pn, the definite part of P;
2:      Let Clsr' be {};
3:      Cont' be {}

{ step 3 }      /* Extended user clauses are folded and
                   continuation clauses are generated. */

        For every extended clause Cl of type 1 in t-m(P),
        do the following.
                                        m1, f1(z1)
        /* let Cl be A0 <- all(y, q(s1,, t1)->A2)          */

1:      Let y1 be a series of new variables of the length |t1|,
        q' be the closure predicate for m1 and
        cont-q' be the continuation predicate for m1;

2:      Add the following clause to P' :

                A0 <- q' (s1, f1(z1))

        /* This clause will be referred to as a translated
        user clause */;
3:      Add the following clause(s) to Cont' :

                cont-q' (f1(z1), y1) <-
                        all(y, y1=/=t1) \/ (y1=t1 & A2)
                if |t1|=0, delete the formulae all(y, y1=/=t1)
```

```
                and (y1=t1)

        /* This formula will be referred to as a daughter
        clause with Cl being its brother. */;

{ step 4 } /* clauses for closure and continuation predicates are
                generated. */

        For every m0, a head mode pattern occurring in t-m(P),
        do the following:

        /* let m0=p(e1,..,en) */

1:      Prepare n new variables X1,..,Xn and
                                                          m0
        let x (y) be the input (output) sequence of p0(X1,..,Xn)
        respectively;
                                    m0                    m0
        So  the I/O form of p0(X1,..,Xn) becomes p(x,,y).

2:      Let Body be {} and
        C be the continuation variable for t-m(P);

        For every clause Cl in t-m(P) whose head mode is m0
    Do

{ case 1 } Cl is an extended clause of type 2.

                        m0                          m2,f2
        /* Let Cl be p(s0,,t0) <- all(y, A1 -> r(s2,,))  */

3:      Let cont-p' be the continuation predicate for m0,
        r' be the closure predicate for m2;

4:      Add the following formula to Body:

                all(y0, x=/=s0) \/
                  exist(y0,y), x=s0 & A1 & r'(s2,f2))\/
                    exist(y0, x=s0 & cont-p'(C,t0))
                where y0 =sorted(Fvar(s0))

        /* This formula will be referred to as the translated
        body of Cl. */;

5:      Add the following clause to Cont':

                cont-r'(f2) <-false

        /* This clause will be referred to as a daughter clause
        with Cl being its brother. */;

6:      Go-to 15
{ end of case 1 }
```

```
{ case 2 } Cl is a unit clause.

                              m0
        /* Let Cl be: p(s0,,t0)   */

7:      Let cont-p' be the continuation predicate for m0;
8:      Add the following formula to Body.

              all(y0, x=/=s0) \/ exist(y0, x=s0 & cont-p' (C, t0))
              where y0 =sorted(Fvar(s0))

        /* This formula will be referred to as the translated
        body of Cl. */;
10:     Go-to 15
{ end of case 2 }

{ case 3 }  Cl is a non-unit definite clause.

                                m0      m1, f1(z1, C)       mk, fk(zk, C)
        /* Let Cl be:    p(s0,,t0)<-p1(s1,,t1)&...& pk(sk,,tk)   */

11:     Let v be Fvar(Cl), i.e. the set of all free variables
        in Cl and for each i (1=<i=<k), let
        pi' and cont-pi' be the closure and
        continuation predicate for mi respectively;

12:     Add the following formula to Body.

              all(y0, x=/=s0) \/
                      exist(y0, x=s0 & p1' (s1, f1(z1, C)))
              where y0 =sorted(Fvar(s0))

        /* This formula will be referred to as the translated body
        of Cl. */;

13:     If k>=2, then
        Add the following clause set to Cont' with each
        body separated.

        { cont-pi' (fi(zi, C), yi)<-
                all(vi, yi=/=ti)\/
                        (yi=ti & pi+1' (si+1, fi+1(zi+1, C))        |

                for i (1=<i=<k-1),
                yi is a sequence of |ti| new variables,
                vi = sorted(Fvar(ti, si+1, ti+1, . . , sk, tk, t0)
                                \Fvar(s0, s1, t1, . . , si-1, ti-1, si))
                and if |ti|=0, replace the body by
                pi+1' (si+1, fi+1(ti+1, C)) };

13' :   Add the following clause set to Cont' with
        each body separated.
```

```
        { cont-pk' (fk(zk, C), yk) <-
                all(vk, yk=/=tk) \/ (yk=tk & cont-p0' (C, t0))
          where yk is a sequence of |tk| new variables,
                vk = sorted(Fvar(tk, t0)
                        \Fvar(s0, s1, t1, . . , sk-1, tk-1, sk))
                and
                if |tk|=0, replace the body by cont-p0' (C, t0) }

        /* Clauses added at step 13 and at step 13' will be
        referred to as a daughter clause with Cl being its brother.
        They have a continuation variable C.   */;

14:     Go-to 15
{ end of case 3 }

15:     Let F1, . . , Fl (l>=0) be an enumeration of the formulae
        accumulated in Body;
16:     If l=0  /* p may be a system predicate. */ then
17:       if  p is a user predicate or p is "false"
18:       then Add { p' (x, C) } to Clsr'
          else /* p must be "=" */
19:         if  m0 = "=(+, -)"
20:         then    Add { =' (x, C) <-cont-=' (C, x) } to Clsr'
21:         else    /* m0 = =(+, +) */
                    Add { =' ' (X1, X2, C) <- (X1=/=X2) \/
                                ((X1=X2) &cont-=' ' (C)) } to Clsr'
        else
22:         Rename bound variables in F1, . . , Fl so that they
            do not have common bound variables;

        /* For each j (1=<j=<l), Fj is a disjunction of formulae
        of the form, all(y, x=/=t), exist(y, x=t&A1) or
        exist(y, x=t&A1&A2). In addition, Fvar(Fj) = (x, C) holds
        as set */

23:         Make a formula p' (x, C) <- F1&. . &Fl
            where p' is the closure predicate for m0;
24:         Convert the body F1&. . &Fl into a disjunctive form
            G1\/. . . \/Gm.

        /* Each Gi (1=<j=<m) is a conjunction of formulae of
        the form, all(y, x=/=t), exist(y, x=t&A1) or
        exist(y, x=t&A1&A2). For each j (1=<j=<m), Fvar(Gj) = (x, C)
        holds as set */

24:         Convert p' (x, C) <-G1\/. . \/Gm into a set of clauses
            { p' (x, C) <-G1, . . , p' (x, C) <-Gm } ;

25:         Remove all existentially quantified variables from
            G1, . . , Gm and let the result be G1', . . , Gm' ;
26:         Add { p' (x, C) <-G1', . . , p' (x, C) <-Gm' } to Clsr'

    oD
```

(end of step 4)

(step 5)
1: Put Pc =P' +Clsr' +Cont' ;
2: Return Pc.

----------(End of translation algorithm)----------

We say that P is compiled into Pc via t-m(P). Pc is said to be the compiled program of P.

(Continuation variable) :

The distinguished variable C which is inherited from t-m(P) is called the continuation variable for Pc. Suppose a clause Cl in Pc holds it somewhere. It is said to have a continuation variable and C is called the continuation variable of Cl. We stipulate that the name of continuation variable survives renaming. That is, if the continuation variable of Cl is renamed to C' when Cl is renamed, C' becomes the continuation variable of the renamed clause. A continuation variable purports to convey a continuation term that represents the rest of the computation.

Observe that, for each mode pattern m, at least one closure clause and one continuation clause are generated. This may cause redundancy in a compiled program. Because for instance, if there is no clause bearing m at its head, we know that any call to an atomic goal labelled by m always fails and any call to all(y, A1->A2) in which A1 is labelled by m always succeeds. One might think that we should adopt a more sophisticated translation algorithm which takes such cases into account, but we would like to point out that our primary concern in this thesis is not to write a smart compiler but to establish a theoretical basis for a compiler of that kind. For that purpose, we prefer simplicity and hence deliberately ignore the apparent crudeness of our algorithm.

Logically speaking, what the translation phase has done is none other than unfolding and folding operations applied to iff(P)+iff(Clsr)+iff(Cont). This will be confirmed rigorously in section 2.10..

2. 9. Executing compiled programs

So far we have been describing how to compile P, a basic program into Pc, an executable program. Our next task is to describe how to run Pc. We can say, however, that we need no more than a usual Prolog interpreter. For only executable goals appear in Pc. By definition, an executable goal is either an atom or a formula of the form all(x, s=/=t).

To execute an atom, we expand it by a matched clause, as usual. If it is a system predicate, we execute it directly. For example, if it is "false", we let the computation fail and start backtracking. Or if it is of the form s=t, we unify s and t. To cope with all(x, s=/=t), we adopt negation-by-failure computation rule. That is, we let all(x, s=/=t) succeed iff s and t are ununifiable. **Therefore, from a practical point of view, the Prolog interpreter is enough to deal with the rest of the job.**

Nonetheless, if we want to somehow relate an original program P to the result of computation by the compiled program Pc, it is obvious that we must sooner or later describe the computation rule formally and precisely to the required preciseness as will be done in the following. Throughout the rest of this section, we fix P and stipulate that Pc denotes the compiled program, Uf the set of user functions symbols in Pc and Cf the set of continuation function symbols in Pc respectively.

2.9.1. An interpreter for compiled programs

We describe how an interpreter responds to a user query. We assume a query is always an atom.

The behaviour of an interpreter is best described as a state transition. A state of an interpreter is a triple <(G1,..,Gn), A, T> such that (G1,..Gn) is a sequence of executable goals called a current goal sequence, A is an atom called a top goal queried by a user and T is a substitution (variable bindings) which has been accumulated up to this state. We stipulate that a top goal A is an atom in L(P). Namely, A has no continuation predicate or continuation function.

An initial state is defined to be:

<A, A, e> where A is a top goal queried by a user and e is an empty substitution.

As computation proceeds, an interpreter nondeterministically selects a goal from the current goal sequence and then executes it directly or searches Pc for a matching clause to expand it. When a matched clause is found, the goal is replaced by the body of the matched clause with suitable substitution. The interpreter changes its state according to the result as a result of its action.

Let <(G1,..,Gn), A, T> be a current state. A state is said to be terminal if no more transition is possible. We describe below the possible state transitions:

(1) n=0
A successful terminal state. No more transition.
T is called an answer substitution.

(2) n>0
Select nondeterministically a goal from the current goal sequence. Let G1 be the selected goal for simplicity.

(2-1) G1=(s=t) for some terms s and t
If s and t are unifiable
then next state is <(G2,..,Gn)T1, A, T*T1>
where T1 is the mgu of s and t (T*T1 stands for a composition of two substitution T and T1)
else next state is a failed terminal state <failure, A, e>

(2-2) G1=all(y, s=/=t)
If s and t are not unifiable
then next state is <(G2,..,Gn), A, T>
else if all(y, s=/=t) is ground, namely s=/=t has no variables except those in y
then next state is a failed terminal state <failure, A, e>

else next state is an aborted terminal
state <aborted, A, e>

(2-3) G1=false
Next state is a failed terminal state <failure, A, e>

(2-4) Otherwise
Select nondeterministically a clause A0<-A1&..&Am
(renaming is always assumed) in Pc such that A0 and
G1 have a mgu (most general unifier) T1
If there is a such clause
then next state is <(A1,..,Am,G2,..,Gn)T1, A, T*T1>
else next state is a failed terminal state <failure,A,e>

An interpreter is nondeterministic due to goal selection and clause selection. So its behaviour can be highly complex. But to say when an interpreter succeeds w.r.t. a user query is a fairly simple matter. If there is a sequence of state transitions from the initial state <A,A,e> to some successful terminal state <(), A, T>, we say a computation of A in Pc halts successfully with an answer substitution T. We write that fact as:

Pc ?- <(),A,T>

On the other hand, to say when an interpreter fails is much subtler because it is quite probable that some goal selection leads to a failed state while some other goal selection never leads to a terminal state. So the notion of "finite failure" is very sensitive to goal selection rule. Therefore, for the sake of simplicity, we simply assume hereafter that an interpreter always selects a left-most goal from the current goal sequence. As a result, nondeterminacies lie only in clause selection.

Then we can say that if every possible state transition starting from <A,A,e> ends in the failed terminal state <failure, A, e>, the computation of A in Pc finitely fails and write:

Pc ?- <failure, A, e>.

The remaining case is one in which either an interpreter reaches the state where the computation is aborted or it moves from a state to another forever without reaching a terminal state.

2. 9. 2. Soundness of successful computation

In this subsection, we prove a theorem stating the relationship between the result of computation and Pc. It is about successful computation.

(Theorem 2. 9. 1)

For an atom A in L(P),

Pc ?- <(), A, T> implies Pc+Eq(Uf) |-(A)T

and (A)T is also an atom in L(P).

For the reader who is already familiar with the theory of logic programming, this may seem nothing special at first glance. But we would like to call his/her attention to the non-logical axiom for function used there. It is Eq(Uf) contrary to the expected one, Eq(Cf+Uf). No equational theory for the continuation functions is included though they play an essential role during computation.

The reason is this:

When we logically relate the output of an interpreter to the original program P as we shall see in later sections, we are required to prove the existence of continuation functions satisfying Ew(Cf) over the domain being considered. Fortunately, it is not difficult to find such functions. But if Eq(Cf+Uf) were adopted, we would have to prove the existence of continuation functions which can satisfy it instead of Ew(Cf). Mostly, Eq(Cf+Uf) is an infinite set and much more complicated than Ew(Cf). So it is very likely that we would hardly be able to prove the existence of such continuation functions, thereby it would become extremely difficult to prove the correctness of our compilation method.

Hence, Ew(Cf) is adopted. Consequently, we must make sure that every unification done by an interpreter can be deductively simulated by such a reduced set of equality axioms.

Our proof depends on the fact that a certain structural property (well-formedness) of the current goal sequence is preserved through computation. "Well formedness" implies that a continuation function behaves according to the discipline of a stack and more to the point, the pattern of appearance of continuation functions in a current goal sequence obeys a certain law. We first classify terms and atoms in the language L(Pc) as follows.

(User/Continuation term):

A term made out of variables and symbols in Uf is

called a user term. The class of continuation term, on the other hand, is defined inductively as follows:

(1) f(u) is a continuation term if f is a continuation function symbol and u is a sequence of user terms.
(2) g(u, t) is a continuation term if u is a sequence of user terms and t is a continuation term.

We can then define "well formedness."

(Well formed atom/goal):

A user/system atom is well formed if it includes no continuation function symbols. Apparently, every atom in L(P) is well formed. A closure atom p'(u, t) is well formed if u is a sequence of user terms and t is a continuation term. A continuation atom cont-p'(t, u) is well formed if t is a continuation term and u is a sequence of user terms. An atom is well formed if it is a well formed system, user, closure or continuation atom.

A goal is well formed if it is a well formed atom or a formula of the form all(y, s=/=t) such that s and t are user terms. A sequence of well formed goals is called a well formed goal sequence.

During computation we have to be careful about the name clash between universally bound variables included in a goal all(y, s=/=t) and those substituted by a unifier. We avoid it by stipulating that previous to substitution, all bound variables in a goal sequence are always renamed alien to those in the substitution.

Let us turn our attention to the compiled program Pc. By inspection we know that every clause in Pc falls into one of the following three categories.

(C1) A0<-A1 &..& Am
m>=0. A0,..,Am are all well formed user atom.

(C2) A0<-A1
A0 is a well formed user atom and
A1 is a well formed closure atom that has no continuation variable.

(C3) B0<-B1 &..& Bm
m >0. B0 contains a closure predicate or a continuation predicate. A continuation variable might be included in the clause. In such case, we use C to denote it. (C3) is further sub-classified.

The head B0 is:

(CH-1) a closure atom p' (x, C) such that variables in (x, C) are pair-wise distinct or
(CH-2) a continuation atom cont-p' (t, y) such that t is a continuation term, variables in Fvar(t)+y are all distinct and if C appears in the atom, it only appears in t with the form f(V1, . . , Vk, C) where t=f(V1, . . , Vk, C) and f is a continuation function symbol.

Each goal Bi (1=<i=<n) in the body is:
(CB-1) s=t such that s and t are user terms and neither of them includes C,
(CB-2) all(y, x=/=t) such that x and y has no common variables, t is a sequence of user terms and neither of x nor t includes C,
(CB-3) false,
(CB-4) a well formed user atom which does not include C,
(CB-5) a well formed closure atom p' (s, f(w, C)) such that C occurs in neither s nor w and variables in (w, C) are pair-wise different or
(CB-6) a continuation atom cont-p' (C, t) such that t is a user term sequence and C does not occur in t.

Then, three lemmas follow.

(Lemma 2. 9. 1)
If G is a well formed goal and T substitutes only user terms for variables, then (G)T is a well formed goal.

(Proof)
The proof is done by the induction on the maximum depth of continuation terms included in G.

(Lemma 2. 9. 2)
Suppose that A0<-A1&. . &Am in Pc has no continuation variable and a substitution T substitutes only user terms. Then, (A1, . . , Am)T is a well formed goal sequence.

(Proof) Obvious from (C1) and (C2).

(Lemma 2. 9. 3)
Suppose that B0<-B1 &. . & Bm in Pc has a continuation variable C and a substitution T substitutes a continuation term for C and user terms for the other variables. Then, (B1, . . , Bm)T is a well formed goal sequence.

(Proof) Obvious from (CB-1) to (CB-6) and the definition of a continuation term.

We get back to the proof of (theorem 2. 9. 1).

(Proof of theorem 2. 9. 1)

Suppose that there is a sequence of state transitions from the initial state <A, A, e> to some successful terminal state. We inductively show that three invariants:

[I1] (G1,..,Gn) is a well formed goal sequence,
[I2] iff (Pc)+Eq(Uf)+Ew(Cf) |- G1&..&Gn -> (A)T and
[I3] (A)T is a well formed user atom

hold for every intermediate state <(G1,..,Gn),A,T>. At a successful terminal state, there is no goal any more. So by putting n=0, we obtain the desired conclusion.

[I1], [I2] and [I3] apparently hold for the initial state <A, A, e>. For (A)e<-A is identical to a tautology A<-A and A is a well formed user atom by the premise of the theorem.

Suppose that [I1], [I2] and [I3] hold for the current state <(G1,..,Gn),A,T>. We have to show that they hold for the next state. The proof is separately done according to the form of G1.

(1) G1=(s=t)
As we supposed that the entire computation was successful, s and t must be unifiable. Let T1 be the mgu of s and t. Since s and t are user terms by [I1], it can be written as:

{X1<-u1,..,Xk<-uk}
where k>=0,
X1,..,Xk are pair-wise different variables,
u1,..,uk are individual user terms
and no Xi occurs in uj (i=/=j, 1=<i, j=<k).

We consider T1 as a set of equations {X1=u1,..,Xk=uk}. First we have:

Eq(Uf) |- s=t <-> (X1=u1 &..& Xk=uk)

because it has already been proved that Eq(Uf) is enough to simulate the unification between user terms deductively [1, 9]. Combining this with [I2], (remembering A<-B&C is equivalent to (A<-B)<-C) we have:

Pc+Eq(Uf) |- ((A)T<-G2&..&Gn) <- (X1=u1 &..& Xk=uk).

Since we have assumed the renaming of bound variables of the current goal sequence in advance of substitution, we can safely apply:

F[X]<-(X=t) is logically equivalent to F[t],
if t is free for X in F

to ((A)T<-G2&..&Gn) <- (X1=u1 &..& Xk=uk), giving:

Pc+Eq(Uf) |- ((A)T<-G2&..&Gn)T1

as a result. So the next state <(G2,..,Gn)T1, A, T*T1> satisfies (I2). On the other hand, since T1 substitutes only user terms, it follows from (lemma 2.9.1) that (G2,..,Gn)T1 is a well formed goal sequence and (A)T*T1 is a well formed user atom because (A)T is a well formed user atom by (I3). Hence (I1) and (I3) for the next state.

(2) G1=all(y, s=/=t)

In this case, s and t must be ununifiable as we have adopted negation-by-failure computation rule. In addition, s and t must be user terms by (I1). So we can prove:

Eq(Uf) |- all(y, s=/=t).

Therefore, from (I2), we have:

Pc+Eq(Uf) |- (A)T<-G2&..&Gn

This means (I2) for the next state <(G2,..,Gn), A, T>. On the other hand, from (I1), (G2&..&Gn) is a well formed goal sequence. Hence follows (I1). (I3) is obvious.

(3) G1 can not be "false" by our assumption.

(4) There is a selected clause A0<-A1&..&Am in Pc such that A0 and G1 have a mgu T1. Next state is <(A1,..,Am,G2,..,Gn)T1, A, T*T1>. Owing to our assumption on renaming, we can assume that no variables in the selected clause occur in (G1,..,Gn), A or T.

(4-1) G1 is a user atom.

A0 must be a user atom. So T1 substitutes only user terms for variables in A0 and G1. Therefore, it follows from (lemma 2.9.1) and (lemma 2.9.2) that (A1,..,Am,G2,..,Gn)T1 is a well formed goal sequence and (A)T*T1 is a well formed user atom. The rest is similar to (1).

(4-2) G1 is a closure atom.

Let G1 be p'(s1,..,sk,t) where p' is a closure predicate and s1,..,sk,t are all individual terms. By (I1), s1,..,sk are all user terms and t is a continuation term. A0 must be a closure atom. From (CH-1), A0 can be written as p'(X1,..,Xk,C) where X1,..,Xk,C are all different variables and C is the continuation variable of the selected clause. Then, T1 becomes:

T1={X1<-s1,..,Xk<-sk,C<-t}.

Therefore, we can conclude that T1 substitutes the continuation term t for the continuation variable C and user terms for other variables while nothing for the variables in G2,..,Gn or (A)T (we assumed the selected clause has been renamed).

Obviously we have (A)T*T1=(A)T and (G2,..,Gn)T1=(G2,..,Gn). { I3 } on the current state combined with (A)T*T1=(A)T gives { I3 } for the next state. On the other hand, we note that it follows from { lemma 2.9.2 } that (A1,..,Am)T1 is a well formed goal sequence. Combining this with (G2,..,Gn)T1=(G2,..,Gn), we can see that (A1,..,Am,G2,..,Gn)T1 (= (A1,..,Am)T1,G2,..,Gn) is a well formed goal sequence. Hence { I1 } for the next state.

From { I2 } on the current state, (A1)T1=G1 and Pc |- (A0<-A1&..&Am)T1, we have:

Pc+Eq(Uf) |- (A)T <- (A1&..&Am)T1&G2&..&Gn.

Using (A)T*T1=(A)T and (G2,..,Gn)T1=(G2,..,Gn) again, we get:

Pc+Eq(Uf) |- (A)T*T1 <- (A1&..&Am&G2&..&Gn)T1,

namely { I2 } for the next state.

(4-3) G1 is a continuation atom

A0 must be a continuation atom, thereby, from (CH-2), it can be written as:

cont-p' (f(V1,..,Vl),Y1,..,Yk)
where l>0, k>=0 and V1,..,Vl,Y1,..,Yk are all different variables.

for some continuation predicate cont-p'. Also from (CH-2), we can assume that "Vl" is the continuation variable if the selected clause has a continuation variable.

Let G1 be cont-p' (t,s1,..,sk). By { I1 }, t must be a continuation term and s1,..,sk are all user terms. The unification of cont-p' (t,s1,..,sk) and cont-p' (f(V1,..,Vl),Y1,..,Yk) reduces to that of r and f(V1,..,Vl). In addition, as a continuation term can not be a variable and as t is unifiable with f(V1,..,Vl), t must has the form f(t1,..,tl). where t1,..,tl-1 are user terms and tl is either a user term or a continuation term. Then, T1 apparently becomes:

T1={V1<-t1,..,Vl<-tl,Y1<-s1,..,Yk<-sk}.

We can draw three conclusions. Firstly T1 does not substitute for variables not in A0<-A1&..&Am. Secondly if the selected clause has no continuation variable, T1 substitutes only user terms. Thirdly, suppose that the selected clause has a continuation variable. Then it must coincides with Vl so that T1 substitutes a continuation term for the continuation variable and user terms for other variables. Using these three, the rest can be done similarly to (4-2).

END of theorem 2.9.1

2. 9. 3. Soundness of failed computation

In this subsection, we deal with failed computation. We show that if an interpreter finitely fails to solve a user query A, ~A (= A->false) is provable from iff(Pc) with additional non-logical axioms for functions.

(Theorem 2. 9. 2)
For an atom A in L(P),

Pc ?- <failure, A, e> implies iff(Pc)+Eq(Uf)+Ew(Cf) |- ~A.

(Proof)
We show that the following invariant (I4) also holds for an arbitrary state <(G1,..,Gn), A, T> between the initial state and any failed terminal state.

(I4) iff(Pc)+Eq(Uf)+Ew(Cf) |- G1&..&Gn -> false

By applying (I4) to the initial state where n=1 and G1=A, we obtain the desired theorem. The proof is done by induction on the maximum number of state transitions from the current state to a failed terminal state. That is, our induction proceeds from a failed terminal state to the initial state.

First of all, we distinguish a direct failure from an indirect failure. A state <(G1,..,Gn), A, T> directly fails if all of its succeeding states are failed terminal states. In other words, it directly fails if:

(1) G1=(s=t) and s and t are ununifiable,
(2) G1=all(y, s=/=t), G1 is ground and s and t are unifiable,
(3) G1=false or
(4) G1 has no clause in Pc that has a unifiable head with G1.

A state <(G1,..,Gn), A, T> indirectly fails if it has at least one non-terminal succeeding state and every possible state transition ends with a failed state.

We show that (I4) holds for every directly failed state (base case) then show it also holds for every indirectly failed state in the computation (induction step). Note that since every state transition satisfies the invariant (I2) until it reaches a terminal state, we can assume (G1,..,Gn) is a well formed goal sequence.

2. 9. 3. 1 Direct failure

Let <(G1,..,Gn), A, T> be the current state and suppose that it directly fails. To show (I4), it is enough to

prove ~G1 is a logical consequence of iff(Pc)+Eq(Uf)+Ew(Cf).

(1-1) G1=(s=t).
By our assumption, s and t are ununifiable user terms. So,

Eq(Uf) |- s=t <-> false.

(1-2) G1=all(y, s=/=t)
s and t are unifiable user terms. And since the failed computation includes no abortion, all(y, s=/=t) must be ground, which implies that neither s nor t includes variables other than those in y. Therefore, it is possible to make s and t identical expressions only by substituting for variables in y. It is easy to verify:

|- exist(y, s=t)

holds in this case. By the equivalence of exist(y, s=t) and ~all(y, ~(s=t)), we obtain:

|- all(y, s=/=t)<-> false.

Therefore, we have:

|- G1<->false.

(1-3) G1=false. This case is obvious.

(1-4) Otherwise. G1 can be a user, closure or continuation atom and there is no clause in Pc whose head is unifiable with G1 by our assumption. Note that G1 can not be a closure atom because for any closure atom, there is always a clause whose head is unifiable with G1 (see CH-1 in subsection 2.9.2.).

(1-4-1)
Suppose G1 is a user atom p(t). No clause head in Pc is unifiable with p(t). Let p(s1)<-E1,..,p(sk)<-Ek (k>=0) be an enumeration of clauses about p. If k=0, iff(Pc) includes p(x)<->false and we can prove iff(Pc) |- p(t)<->false. So suppose otherwise. Then iff(Pc) includes the iff definition of p:

p(x)<->exist(v1, x=s1 & E1)\/
..\/exist(vk, x=sk & Ek) (k>0).

By substituting t for x, we have:

iff(Pc) |- p(t)<->exist(v1, t=s1 & E1)\/
..\/exist(vk, t=sk & Ek)

Since t and each si (1=<i=<k) are ununifiable user terms, we can prove:

Eq(Uf) |- t=si <-> false.

Therefore, combining the two, we have:

iff(Pc)+Eq(Uf) |-
p(t)<->exist(v1, false & E1)\/
..\/exist(vk, false & Ek)

Consequently, we have:

iff(Pc)+Eq(Uf) |- G1 <-> false.

(1-4-2)
Suppose G1 is a continuation atom cont-p'(t, s). Since G1 is a well formed goal, we can assume that t is a continuation term g(u) (|u|>=0) and s is a sequence of user terms. Recall that from (CH-2), every clause head in Pc containing cont-p' can be written as cont-p'(f(v), y) such that f is a continuation function symbol and variables in (v, y) are pair-wise distinct. Therefore, the unification of cont-p'(g(u), s) and cont-p'(f(v), y) is equivalent to the following two independent unifications. Namely, one between g(u) and f(v) and the other between s and y. As the unification between s and y never fails (all variables in y are pair-wise distinct), the ununifiability of G1(= cont-p'(g(u), s)) and cont-p'(f(v, C), y) is ascribed to that of g(u) and f(v). Recalling that v is a sequence of different variables again, we conclude that f and g must be different continuation function symbols.

Since Ew(Cf) includes f(x)=/=g(y) where (x, y) is a variable sequence without repetition, we have:

Ew(Cf) |- g(u)=f(v) <-> false

Thus,

Ew(Cf) |- (t, s)=(f(v), y) <-> false.

Since this holds for an arbitrary clause about cont-p', the rest can be done parallel to the case of user atoms.

2.9.3.2 Indirect failure

Let the current state be <(G1, . . , Gn), A, T>. This state has at least one non-terminal succeeding state and every possible state transition ends with a failed terminal state. By the induction hypothesis, (I4) holds for every non-terminal succeeding state. We have to show that (I4) also holds for the current state.

(2-1) G1=(s=t).

s and t must be unifiable user terms by our assumption. The next state is <(G2,..,Gn)T1, A, T*T1> where T1 is the mgu of s and t. Let T1 be {X1<-u1,..,Xk<-uk}. From (I4) applied to the succeeding state, we have:

iff(Pc)+Eq(Uf)+Ew(Cf) |- (G2&..&G)T1 -> false

On the other hand, since none of X1,..,Xk occur in u1,..,uk by the nature of a unifier, we can safely apply a valid schema:

exist(x, x=t & F(x))<->F(t)

to this case. As a result, we have:

|- (G2&..&Gn)T1 <->
exist((X1,..,Xk), G2&..&Gn&(X1=u1)&..&(Xk=uk))

and further more, from Eq(Uf) |- s=t <-> X1=u1&..&Xk=uk, we also have:

Eq(Uf) |- (G2&..&Gn)T1 <->
exists((X1,..,Xk), G2&..&Gn&(s=t)).

This, combined with iff(Pc)+Eq(Uf)+Ew(Cf) |-(G2&..&Gn)T1 -> false, gives us:

iff(Pc)+Eq(Uf)+Ew(Cf)
|- exist((X1,..,Xk),(s=t)&G2&..&Gn)->false.

Using a valid scheme: exist(x,A)->B <-> all(x,A->B) if x does not occur in B, we conclude that:

iff(Pc)+Eq(Uf)+Ew(Cf)
|- all((X1,..,Xk), (s=t)&G2&..&Gn ->false).

Since it is sound to infer F(x) from all(x,F(x)), we obtain:

iff(Pc)+Eq(Uf)+Ew(Cf) |- (s=t)&G2&..&Gn->false.

Thus (I4) for the current state.

(2-2) G1=all(y, s=/=t)

The current state must have a succeeding non-terminal state by the induction hypothesis. So s and t must be ununifiable user terms. Next state is <(G2,..,Gn),A,T>. The application of (I4) to that state gives:

iff(Pc)+Eq(Uf)+Ew(Cf) |- (G2&..&Gn -> false).

Using a valid scheme (A->(B->A)) in propositional calculus, we have:

iff(Pc)+Eq(Uf)+Ew(Cf) |- G1->(G2&..&Gn -> false),

which is equivalent to (I4) for the current state:

iff(Pc)+Eq(Uf)+Ew(Cf) |- G1&G2&..&Gn -> false.

(2-3) G1= false.

Since the current state is supposed to have a non-terminal succeeding state by the induction hypothesis, this case can not occur.

(2-4) Otherwise.

G1 is a user, closure, or continuation atom. Let G1 be r(t). There is at least one clause in Pc whose head is unifiable with r(t). Let r(s1)<-E1,..,r(sk)<-Ek (k>0) be an enumeration of clauses about r and its iff-definition be:

r(x)<->exist(v1, x=s1 & E1)\/..\/exist(vk, x=sk & Ek) ... (A)

where vi is the set of all free variables in r(si)<-Ei(1=<i=<k).

Take an arbitrary clause r(si)<-Ei (1=<i=<k). If r(si) is a user atom, the unification of G1 and r(si) can be completely simulated by Eq(Uf). If r(si) is a closure atom, si must be a sequence of different variables. So the unification between G1 and r(si) never fails. Consider the case when r(si) is a continuation atom.

As can be seen from the discussion at (4-3) of the proof of (theorem 2.9.1), the successful unification of r(si) and G1 can be reduced to that between two continuation terms. We note that Ew(Cf) is strong enough to simulate deductively the unification between continuation terms so long as it is successful. This is because no occur check is needed in the deductive simulation of a successful unification. We also note that as was proved at (1-4-2) of direct failure, unification failure between a well formed continuation atom and a clause head is caused only by symbol clash and this can also can be simulated deductively using the formulae in Ew(Cf).

Therefore, in summary, whatever predicate G1 may contain and whatever the result of the unification between G1 and r(si) may be, Eq(Uf)+Ew(Cf) is strong enough as non-logicals axiom to simulate the unification deductively. In other words, for each i (1=<i=<k), if G1(=r(t)) and r(si) is unifiable, we can prove:

Eq(Uf)+Ew(Cf) |- t=si <-> (X1=u1)&..&(Xk=uk) ... (B)
where T1= {X1<-u1,..,Xk<-uk} is a mgu of t and si

or otherwise,

```
Eq(Uf)+Ew(Cf) |- t=si <-> false.                    ... (C)
```

Bearing this in mind, we return to [I4]. Let r(si)<-Ei be an arbitrary clause about r (1=<i=<k) and suppose that G1 calls r(si)<-Ei successfully and next state becomes:

```
<(Ei&G2&..&Gn)T1, A, T*T1>
```

where T1 is a mgu of G1(=r(t)) and r(si). By applying [I4] to this state, we have:

```
iff(Pc)+Eq(Uf)+Ew(Cf) |- (Ei&G2&..&Gn)T1 -> false.
```

By (B) and the same inference done at (2-1), we can obtain:

```
iff(Pc)+Eq(Uf)+Ew(Cf) |- (t=si)&Ei&G2&..&Gn -> false.
```

So we have:

```
iff(Pc)+Eq(Uf)+Ew(Cf)
        |- all(vi, (t=si)&Ei&G2&..&Gn -> false)
where vi is the set of free variables in r(si)<-Ei.
```

Because of renaming, we can assume that vi and G2&..&Gn have no common variables. Then using the valid scheme (exist(x, A)->B) <-> all(x, A->B) if x does not occur in B, we obtain:

```
iff(Pc)+Eq(Uf)+Ew(Cf)
        |- exist(vi, (t=si)&Ei)&G2&..&Gn -> false
                                                    ... (Di)
```

Suppose that G1 calls r(si)<-Ei but G1 and r(si) could not be unified. Then, by (C), t=si becomes false and (Di) holds this case too. Thus we can conclude that (Di) holds for every i (1=<i=<k).

We put together all (Di) (1=<i=<k) into (D) using the valid scheme (A->C)&(B->C) <-> (A\/B ->C), having:

```
iff(Pc)+Eq(Uf)+Ew(Cf)
        |- ( exist(v1, (t=s1)&E1)\/...
             \/ exist(vk, (t=si)&E) )&G2&..&Gn -> false
                                                    ... (D)
```

Next by substituting t for x in the head r(x) of the iff-definition of r which belongs to iff(Pc), we obtain:

```
iff(Pc) |-
        r(t)<->exist(v1, t=s1 & E1)\/
               ..\/exist(vk, t=sk & Ek)             ... (E)
```

From (D) and (E), we finally have:

iff(Pc)+Eq(Uf)+Ew(Cf) |- r(t)&G2&..&Gn -> false

Since G1=r(t), this is (I4) for the current state. So we are done.

END of theorem 2.9.2

2. 10. Correctness of compilation

2. 10. 1. Preliminaries

In this subsection, we discuss the correctness of the whole compilation process. So it is convenient to fix a compilable basic program P again and assume that it is compiled into Pc via t-m(P). Throughout this subsection, accordingly, we always understand that Uf denotes the set of user function symbols in Pc, Cf the set of continuation function symbols in Pc, Clsr the closure definition by t-m(P) and Cont the continuation definition by t-m(P) respectively.

By "correctness" we mean the soundness of our system. More precisely, we prove that if an interpreter successfully returns a user query A (atom) with an answer substitution T, then (A)T is a logical consequence of comp(P) and if it finitely fails, ~A is a logical consequence instead. The proof is fairy long. So we would like to describe it step by step.

The first part is proof theoretical. We show that iff(Pc) is deducible from comp(P) if there exist appropriate continuation functions and continuation predicates satisfying certain axioms. Then, in the succeeding part, we show that these axioms can be replaced by a simpler one which states that the domain of a model of comp(P) is infinite (note that this is unconditionally satisfied as long as P includes a non-constant function symbol). The reasoning adopted there is rather model theoretical. For example, we prove the existence of the required continuation predicates over a given domain based on the (well-known) fact that a monotonous function over a complete lattice has a fixed point. Then we deal with the case where the domain is not infinite, therefore the axiom requiring an infinite domain does not hold. To deal with such case, a class of "domain independent" programs is introduced. It is shown that the soundness theorem holds again for a domain independent program. Some syntactic sufficient conditions for a program to be domain independent is also given.

First of all, we have to prove the following lemma which will be used many times.

(Lemma 2. 10. 1)
Given a first order language L, define E as:

E= { f(x)=f(y) -> x=y | f is a function symbol in L,
(x, y) is a variable sequence without repetition }.

Then for any formula F and terms s, t in L such that y=<Fvar(t) and y/\Fvar(s)={},

E |- all(y, s=t -> F) <->
all(y, s=/=t)\/exist(y, s=t & F)

(Proof)

In this proof, we stipulate that non-logical axioms are presented as they are and we do not assume variables in them are universally quantified. Also to avoid the immaterial aspects of formal deduction, some details of the deduction will be deliberately not mentioned.

Since "->" direction is easy (note that:

all(y, s=t -> F) <-> all(y, s=/=t)\/exist(y, s=t & F)

is a valid formula), we show only "<-" direction.

Let y=(Y1,..,Ym). {Y1,..,Ym} is a subset of Fvar(t) by our assumption. We always understand that t(y), F(y) denote all occurrences of variables in y if they actually occur in t, F. Prepare new variables y'=(Y1',...,Ym'). Let t(y') (F(y')) denote the result of simultaneous replacement of all occurrences of Yj by Yj' (1=<j=<m) in t (F) respectively. Then we can show:

s=t(y), s=t(y'), E |- y=y'

by induction on the maximum complexity of terms included in s and thereby

s=t(y), s=t(y'), E, F(y) |- F(y').

by the equality axiom. Then, using the deduction theorem, we have:

s=t(y), E, F(y) |- s=t(y')->F(y').

Since none of Y1,..,Ym appears in s=t(y')->F(y') because y/\Fvar(s)={} from our assumption, we have:

exist(y, s=t(y))&F(y)), E |- s=t(y')->F(y')

Therefore, by renaming Y1',...,Ym' to Y1,..,Ym, we obtain:

exist(y, s=t&F), E |- all(y, s=t->F) ... (1)

On the other hand, from s=/=t, s=t |- false, we have:

all(y, s=/=t), s=t |- F.

Therefore, by the deduction theorem,

all(y, s=/=t) |- s=t -> F.

Using universal generalization, we obtain:

```
all(y, t=/=s) |- all(y, t=s-> F)                    ... (2)
```

From (1) and (2), we have:

```
all(y, s=/=t)\/exist(y, (s=t)&F), E |- all(y, s=t->F).
```

Finally, by the deduction theorem again:

```
E |- all(y, s=t -> F) <-
                all(y, s=/=t)\/exist(y, s=t & F).
```

End of lemma 2.10.1

2. 10. 2. Proving proof theoretic relation

Now we can state the longest proposition (not the content but its proof) in this thesis.

{ Proposition 2. 10. 1 }

iff(P)+iff(Clsr)+iff(Cont)+Ew(Cf)+Ew(Uf) |- iff(Pc)

(Proof)

Note that to prove the proposition, we may prove:

iff(P)+iff(Clsr)+Aux+Ew(Uf) |- iff(Pc)

because iff(Cont)+Ew(Cf) |- Aux holds by { lemma 2. 7. 1 } where Aux = { A<->E | A<-E is in Cont }.

As can be seen from the compilation process, a compiled program Pc becomes the union of three sets P', Clsr' and Cont' where P' is a set of clauses about user predicate, Clsr' about closure predicates and Cont' about continuation predicates.

Accordingly, formulae in iff(Pc) are divided into 3 groups:

(Case 1) an iff-definition that belongs to iff(P')
(Case 2) an iff-definition that belongs to iff(Clsr')
(Case 3) an iff-definition that belongs to iff(Cont')

We consider three cases separately.

2. 10. 2. 1 (Case 1) an iff-definition that belongs to iff(P')

We show that an iff-definition about a user predicate in iff(P') is a logical consequence of iff(P)+iff(Clsr)+Aux. Let p be an n-ary user predicate,

p(u1)<-E1, . . , p(ul)<-El (l>=0)

be an enumeration of clauses in P about the relation p, and the iff-definition about p in P be

p(x)<->false (l=0) or

p(x)<-> exist(v1, (w=u1 & E1)) \/
. . . \/exist(vl, (w=ul & El)).
where vj=sorted(Fvar(p(uj)<-Ej)) (1=j=<l)

As is evident from the compilation process about user clauses (see the translation algorithm, { step 3 }), if there is no clause about p in P, there is no clause about p in P' either. Thus, iff(P') includes p(x)<-> false and so does iff(P).

Suppose l>0.

Remember that every user clause in P is transferred to P' unchanged or undergoes a replacement of its body (this case we refer to it as a translated user clause). Consequently without loss of generality, we can assume that there is an enumeration of clauses about p in P' :

```
p(u1)<-E1', . . , p(ul)<-El'   (l>0)
```

such that Ej' corresponds to Ej (1=<j=<l). Then, the iff-definition of p in P' becomes:

```
p(w) <-> exist(v1', (w=u1 & E1')) \/
                              . . . \/exist(vl', (w=ul & El')).
where vj' =sorted(Fvar(p(uj)<-Ej'))  (1=j=<l)
```

and this belongs to iff(P'). To prove (case 1), it is sufficient to show:

(a) iff(Clsr)+Aux |- Ej<->Ej' and
(b) vj = vj'

hold for every j (1=<j=<l).

Let p(uj)<-Ej be a clause in P. If it is a definite clause, it is transferred to P' unchanged during translation. Therefore, Ej=Ej' and vj=vj so that (a) and (b) hold trivially.

Suppose otherwise. Then p(uj)<-Ej is an extended clause. Let us write it as:

```
p(uj) <- all(y, q(uj')->A2).
```

Suppose this clause has been labelled as follows:

```
                        m1, f(z1)
p(uj) <- all(y, q(s1,, t1) -> A2)
where z1 = sorted(Fvar(t1, A2)\y)
```

by the term-mode labelling algorithm, translated into:

```
p(uj)<- q' (s1, f1(z1))
where q' is the closure predicate for m1.
```

and finally transferred to P'. As a result, P' includes:

```
p(uj)<- q' (s1, f1(z1)).
```

Recall that Aux has the following formula about cont-q' :

```
( a1-1 ):        cont-q' (f1(z1), y1) <-> all(y, y1=t1 -> A2)
```

where cont-q' is the continuation predicate for m1

because cont-q' (f1(z1), y1) <- all(y, y1=t1 -> A2) is included in the continuation definition Cont and Aux includes "individual completion" of each clause in Cont. Also recall that the variable condition:

y/\Fvar(s1) = {}

is satisfied since

$$p(uj) \leftarrow all(y,\ q(s1,,t1)^{m1,f(z1)} \rightarrow A2)$$

is a well-moded clause. We then have the following deduction under the non-logical axiom { a1-1 }.

all(y, q(s1,,t1) -> A2)

<-> all(y, exist(y1, q(s1,,y1)&y1=t1) -> A2)
where y1 is a sequence of |t1| new variables.

<-> all((y,y1), q(s1,,y1)&y1=t1 -> A2)

<-> all((y,y1), q(s1,,y1)->(y1=t1 -> A2))

<-> all(y1, q(s1,,y1) -> all(y, y1=t1 -> A2))
/* y/\Fvar(s1)={} by the variable condition */

<-> all(y1, q(s1,,y1) -> cont-q' (f1(z1),y1))
/* { a1-1 } is used to fold: all(y, y1=t1 -> A2)
into: cont-q' (f1(z1),y1). */

<-> q' (s1,f1(z1))
/* The formula:

all(y1, q(s1,,y1) -> cont-q' (f1(z1),y1))

is folded into:

q' (s1,f1(z1))

by the iff-definition of q' :

q' (x',C)<->all(y1, q(x',y1) -> cont-q' (C,y1))

which is included in iff(Clsr). */

Thus, we have shown that

iff(Clsr)+Aux |- Ej<->Ej'

holds for Ej=all(y, q(s1,,t1) -> A2)) and Ej'=q' (s1,f1(z1)).

Thus (a) has been proved. Next let compare vj and vj'. Recall that vj are free variables of

```
p(uj) <- all(y, q(s1,,t1) -> A2))
```

and vj' are those of

```
p(uj) <- q'(s1,f1(z1))
where z1 = sorted((Fvar(t1)+Fvar(A2))\y).
```

Since Fvar(s1) and y have no intersection owing to the variable condition, we have:

```
vj =   Fvar(uj)+Fvar(all(y, q(s1,,t1) -> A2))
   =   Fvar(uj)+((Fvar(s1)+Fvar(t1)+Fvar(A2))\y)
   =   Fvar(uj)+Fvar(s1)+(Fvar(t1,A2)\y)
   =   Fvar(uj)+Fvar(s1)+z1
   =   vj'
```

So (b) is proved too, which completes [case 1].

2.10.2.2 (Case 2) an iff-definition that belongs to iff(Clsr')

We show that every iff-definition about a closure predicate in iff(Clsr') is a logical consequence of iff(P)+iff(Clsr)+Aux+Ew(Uf). For reference, see the translation algorithm, [step 4].

Let m0 be a mode pattern for an n-ary predicate p, x (y) be the input (output) sequence of p(X1,..,Xn) under mode m0 where X1,..,Xn are n new variables and C be the continuation variable.

Let p' and cont-p' be the closure and continuation predicates for m0 respectively,

```
                          m0
p'(x,C)<->all(y,p(x,,y)->cont-p'(C,y))
```

be the iff-definition of p' included in iff(Clsr).

We first deal with the case when P includes no clauses about p. Note that p can be a system predicate, "false" or "=."

According to the translation algorithm, when p is a user predicate or "false", iff(Clsr) must include:

```
p'(x,C)
```

as the iff-definition of p'. On the other hand,

```
p(x,,y)<->false
```

is provable from iff(P) whether p be a user predicate or not. As

p(x,,y)<->false, p'(x,C)<->all(y, p(x,,y)->cont-p'(C,y))
|- p'(x,C)

is a valid inference, we have:

iff(Clsr)+iff(P) |- p'(x,C).

The other case is p = "=." For simplicity, we only deal with the case where the mode assigned to "=" is =(+,-). In this case, iff(Clsr) includes:

='(X,C) <-> all(Y, (X=Y) -> cont-='(C,Y))

and iff(Clsr') includes:

='(X,C) <-> cont-='(C,X)

where =' and cont-=' are the closure predicate and continuation predicate for =(+,-). But both formulae are apparently equivalent. So the case l=0 is done. Suppose otherwise and let

m0 m0
p(s1,,t1)<-E1, .., p(sl,,tl)<-El (l>0)

be an enumeration of clauses in t-m(P) about the relation p with head mode pattern m0. Without loss of generality, we can assume that this is also an enumeration of the clauses about p in P because both are only notationally different. In what follows, mode patterns are omitted as the context is clear.

Before going into the detail, it seems beneficial to review the translation process for closure predicates. First for each j (1=<j=<l), we take Ej, the body of a user clause p(sj,,tj) <- Ej, and translate it into Fj. We called Fj the translated body. All the translated bodies are accumulated in "Body" (during translation, when p(sj,,tj)<-Ej is an extended clause or a non-unit definite clause, some continuation clauses are added to Cont'). Let Body be {F1,..,Fl}. A formula:

p'(x,C)<- F1&..&Fl

is formed then. Since each Fj (1=<j=<l) is a disjunction of formulae of the form all(y,x=/=t), exist(y,x=t&A1) or exist(y,x=s&A1&A2), it is always possible to convert F1&..&Fl into a disjunctive-conjunctive form G1\/..\/Gm. Furthermore, as Fvar(Fj)=<(x,C) (1=<j=<l) holds as set, we can also assume Fvar(Gk)=<(x,C) holds for every k (1=<k=<m). The final clauses about p' included in Clsr' become:

{ p' (x, C) <-G1' , . . , p' (x, C) <-Gm' }

where Gk' is a formula obtained by removing all existentially quantified variables from Gk (1=<k=<m). The iff-definition of p' in Clsr' must be constructed using these clauses.

Therefore, every free variable in Gk' (1=<k=<m) must be the one in Gk or must come from the existentially quantified variable in Gk. As Fvar(Gk) =< (x, C) holds as set, we can therefore assume that all free variables in Gk' other than (x, C) must come from the existentially quantified variable of Gk. As a consequence, we are able to take:

p' (x, C) <->G1\/. . \/Gm

as the iff-definition formed by { p' (x, C) <-G1' , . . , p' (x, C) <-Gm' }. In other words, we can adopt it or its logical equivalent:

p' (x, C) <->F1&. . &Fl

as the iff-definition of p' belonging to iff(Clsr'). For this reason, we regard the last formula as the iff-definition of p'. Therefore, (case 2) is reduced to prove:

iff(P) +iff(Clsr) +Aux+Ew(Uf) |- p' (x, C) <->F1&. . &Fl.

This is further reduced to prove:

iff(P) +iff(Clsr) +Aux+Ew(Uf) +Ew(Cf) |-
all(y, p(x, , y) ->cont-p' (C, y)) <-> F1&. . &Fl.

This is because

p' (x, C) <->all(y, p(x, , y) ->cont-p' (C, y))

belongs to iff(Clsr). Now we can start proving. We first deductively rewrite:

all(y, p(x, , y) ->cont-p' (C, y))

using the iff-definition of p:

p(x, , y) <-> exist(v1, (x=s1 & y=t1 & E1)) \/
. . .
exist(vl, (x=sl & y=tl & El))

where vj=Fvar(p(sj, , tj) <-Ej) for j (1=<j=<l) as follows:

all(y, p(x, , y) ->cont-p' (C, y))

<-> all(y, { exist(v1, x=s1 & y=t1 & E1) \/
. . . \/ exist(vl, x=sl & y=tl & El) }

```
                                        -> cont-p' (C, y))

        /* Unfolding, i.e., the iff-definition of p is
        substituted for p(x,,y) in the definition of p'. */

<->     all((y,v1), x=s1 & y=t1 & E1 -> cont-p' (C, y)) &
                                ...
        all((y,vl), x=sl & y=tl & El -> cont-p' (C, y))

        /* (A\/B -> C) and (A->B)&(A->C) are logically
        equivalent.      */

<->     all(v1, x=s1 & E1 -> all(y, y=t1 -> cont-p' (C, y))) &
                                ...
        all(vl, x=sl & El -> all(y, y=tl -> cont-p' (C, y)))

        /* This is because no variable in y appears
        in s1,..,sl,E1,..,El and (A&B->C) and (A->(B->C))
        are logically equivalent.        */

<->     all(v1, x=s1 & E1 -> cont-p' (C, t1)) &
                                ...
        all(vl, x=sl & El -> cont-p' (C, tl))

        /* This is because the scheme all(y, y=t -> F(y))<->F(t)
        is valid if no variable in y appears
        in a term t or a formula F.      */
```

Then, we get:

```
iff(P) |- all(y, p(x,,y)->cont-p' (C, y))

          <-> all(v1, x=s1 & E1 -> cont-p' (C, t1)) &
                                ...
              all(vl, x=sl & El -> cont-p' (C, tl))          ... (A)
```

as a result. Next we show that under the non-logical axioms iff(Clsr)+Aux+Ew(U), each conjunct (1=<j=<l):

```
        all(vj, x=sj & Ej -> cont-p' (C, tj))
```

in the right hand side of (A) which corresponds to a clause p(sj,,tj)<-Ej (hereafter called the source of a conjunct) is logically equivalent to the formula Fj which was referred to as the translated body of the clause. We prove it by case-analysis.

(case 2-1)

Suppose the source of a conjunct is an extended clause Cl of type 2 in t-m(P). Let Cl be:

```
            m0                          m2, f2
        p(s0,,t0) <- all(y, A1 -> r(s,,))
```

where m2 =r(+,..,+) and f2 is a new constant symbol

and the conjunct be:

all(v, x=s0 & all(y, A1->r(s,,)) -> cont-p' (C,t0))
where v=Fvar(s0,all(y, A1->r(s,,)),t0).

Since Cl is well-moded, we can assume the variable condition:

Fvar(s0)>=v or equivalently, Fvar(s0)=v

is satisfied. We note,

{ a2-1-1 }: cont-r' (f2)<->false
where cont-r' is the continuation predicate for m2.

belongs to Aux since cont-r' (f2)<-false is included in the continuation definition Cont. Using Ew(Uf)+Ew(Cf) and { a2-1-1 } as the non-logical axiom, we deductively rewrite the conjunct as follows:

all(v, x=s0 -> { all(y, A1->r(s,,))->cont-p' (C,t0) })

<-> all(v,x=/=s0) \/
exist(v, x=s0 & {all(y, A1->r(s,,))->cont-p' (C,t0)})

/* by the variable condition on Cl, Ew(Uf) and
{ lemma 2.10.1 } */

<-> all(v,x=/=s0) \/
exist(v, x=s0 & {exist(y, A1 & ~r(s,,))
\/ cont-p' (C,t0)})

<-> all(v,x=/=s0) \/
exist(v, x=s0 & {exist(y, A1 & (r(s,,)->false))
\/ cont-p' (C,t0)})

<-> all(v,x=/=s0) \/
exist(v, x=s0 &
{exist(y, A1 & (r(s,,)->cont-r' (f2)))
\/ cont-p' (C,t0)})

/* Using { a2-1-1 } included in Aux, "false" is
folded into cont-r' (f2). */

<-> all(v,x=/=s0) \/
exist(v, x=s0 & {exist(y, A1&r' (s,f2))
\/ cont-p' (C,t0)})
where r' is the closure predicate for m2.

/* The formula: r(s,,)<-cont-r' (f2)
is folded into: r' (s,f2)
using the iff-definition of r' in iff(Clsr):

```
                                    m2
                    r' (x, C) <-> (r(x) ->cont-r' (C)        */

<->     all(v, x=/=s0) \/
          exist((v, y), x=s0 & A1 & r' (s, f2)) \/
            exist(v, x=s0 & cont-p' (C, t0))
```

Thus we have proved:

```
iff(Clsr)+Aux+Ew(Uf) |-
        all(v, x=s0 & all(y, A1->r(s,,)) -> cont-p' (C, t0))
        <-> all(v, x=/=s0) \/
              exist((v, y), x=s0 & A1 & r' (s, f2)) \/
                exist(v, x=s0 & cont-p' (C, t0))                    ... (B)
```

Confirm that:

```
        all(v, x=/=s0) \/
          exist((v, y), x=s0 & A1 & r' (s, f2)) \/
            exist(v, x=s0 & cont-p' (C, t0))
```

is exactly the translated body of p(s0,,t0) <- all(y, A1 -> r(s,,)) accumulated in "Body" during the translation process (note that v=Fvar(s0) holds).

(case 2-2)

Suppose the source of a conjunct is a definite clause Cl. Let it be:

```
            m0          m1, f1(z1, C)   mk, fk(zk, C)
        p(s0,,t0) <-  p1(s1,,t1)&..&pk(sk,,tk)

        where for each i (1=<i=<k),
        zi = sorted(Fvar(s0, s1, t1, .., si-1, ti-1, si)
                                    /\Fvar(ti, si+1, ti+1, .. sk, tk, t0))
```

and suppose that it corresponds to a conjunct:

```
        all(v, x=s0 & p1(s1,,t1) &..& pk(sk,,tk) ->cont-p' (C, t0))
        where v=Fvar(s0, s1, t1, .., sk, tk, t0).
```

Since Cl is well-moded, we can assume the variable condition on Cl:

```
        Fvar(t0) =<Fvar(s0, s1, t1, .., sk, tk) and
        Fvar(si) =<Fvar(s0, s1, t1, .., ti-1, si-1)  (1=<i=<k)
```

is satisfied. Recall that when k>0, the following formula belongs to in Aux (remember Aux={ A<->F | A<-F in Cont } and the definition of Cont).

(a2-2-1):

```
        cont-p1' (f1(z1, C), y1)
                <->all(v1, y1=t1 & p2(s2, , t2)&. . &pk(sk, , tk)
                                                ->cont-p' (C, t0))
        where
        y1 is a sequence of |t1| new variables,
        z1=sorted(Fvar(t1, s2, t2, . . , sk, tk, t0)/\Fvar(s0, s1))
        v1=sorted(Fvar(t1, s2, t2, . . , sk, tk, t0)\Fvar(s0, s1))
```

Using non-logical axioms Ew(Uf) and [a2-2-1], we rewrite the conjunct deductively as follows: First we deal with the case where k=0.

```
        all(v,  x=s0 -> cont-p' (C, t0))

<->     all(v0, x=s0 -> cont-p' (C, t0))
        where v0=Fvar(s0)

        /* Because the variable condition
        on C1: Fvar(t0)=<Fvar(s0) is satisfied, we have:
        v=Fvar(s0, t0)=Fvar(s0)=v0          */

<->     all(v0, x=/=s0) \/ exist(v0, x=s0 & cont-p' (C, t0)

        /* By v0=Fvar(s0), Ew(Uf) and [ lemma 2.10.1 ] */
```

Thus, we have proved:

```
        Ew(Uf) |-
            all(v, x=s0 -> cont-p' (C, t0))
            <-> all(v0, x=/=s0) \/
                    exist(v0, x=s0 & cont-p' (C, t0))          . . . (C)
        where v0=Fvar(s0)
```

Here we note that all(v0, x=/=s0) \/ exist(v0, x=s0 & cont-p' (C, t0)) is the translated body of p(s0, , t0) accumulated in "Body" during the translation process. Next we deal with the case where k>=1.

```
        all(v, x=s0 & p1(s1, , t1) &. . & pk(sk, , tk)
                                        -> cont-p' (C, t0))

<->     all(v, x=s0 ->
                (p1(s1, , t1)&. . &pk(sk, , tk)
                                        ->cont-p' (C, t0)))

<->     all(v0, x=s0 ->
                all(v1, p1(s1, , t1) ->
                        (p2(s2, , t2)&. . &pk(sk, , tk)
                                        ->cont-p' (C, t0))))
        where v0=Fvar(s0) and
        v1=Fvar(t1, . . , sk, tk, t0)\Fvar(s0, s1)

        /* This is sound since as set, v1
```

```
        is equal to v\v0, which is confirmed by:

        v\v0
        = Fvar(s0, s1, t1, . . , sk, tk, t0)\Fvar(s0)
        = Fvar(s0, s1, t1, . . , sk, tk, t0)\Fvar(s0, s1)
          ( since the variable condition on Cl is satisfied,
            Fvar(s0)>=Fvar(s1) holds. )
        = Fvar(t1, . . , sk, tk, t0)\Fvar(s0, s1)
        = v1     */

<->     all(v0, x=s0 ->
          all((v1, y1), p1(s1, , y1) ->
                          (y1=t1 -> (p2(s2, , t2)&. . &pk(sk, , tk)
                                       ->cont-p' (C, t0))))

        /* This is sound since y1 is a sequence
        of new variables. */

<->     all(v0, x=s0 ->
          all(y1, p1(s1, , y1) ->
            all(v1, y1=t1 -> (p2(s2, , t2)&. . &pk(sk, , tk)
                                       ->cont-p' (C, t0))))
        /* This is sound because,
        v1/\Fvar(s1)
        = (Fvar(t1, . . , sk, tk, t0)\Fvar(s0, s1))/\Fvar(s1)
        = {}     */

<->     all(v0, x=s0 ->
          all(y1, p1(s1, , y1) ->
            all(v1, (y1=t1)&p2(s2, , t2)&. . &pk(sk, , tk)
                                ->cont-p' (C, t0))))

<->     all(v0, x=s0 -> all(y1, p1(s1, , y1)
                                -> cont-p1' (f1(z1), y1)))

        /* The formula:
                all(v1, y1=t1 & p2(s2, , t2)&. . &pk(sk, , tk)
                                       ->cont-p' (C, t0))))
        is folded into:   cont-p1' (f1(z1, C), y1) where
        z1=sorted(Fvar(t1, s2, t2, . . , sk, tk, t0)/\Fvar(s0, s1))
        by the formula ( a2-2-1 ) included in Aux.       */

<->     all(v0, x=s0 -> p1' (s1, f1(z1, C)))

        /* The formula:
                all(y1, p1(s1, , y1) ->cont-p1' (f1(z1, C), y1))

        is folded into:
                p1' (s1, f1(z1, C))

        by the iff-definition of p1' included in iff(Clsr) */

<->     all(v0, x=/=s0) \/ exist(v0, x=s0 & p1' (s1, f1(z1, C)))
```

/* By v0=Fvar(s0), Ew(Uf) and { lemma 2.10.1 } */

Therefore, we have, for k>=1:

iff(Clsr)+Aux+Ew(Uf) |-

all(v, x=s0 & p1(s1,,t1)&..&pk(sk,,tk) -> cont-p'(C,t0))
<-> all(v0,x=/=s0) \/
exist(v0, x=s0 & p1'(s1,f1(z1,C))) .. (C')

We again note that all(v0,x=/=s0) \/ exist(v0, x=s0 & p1'(s1,f1(z1,C))) exactly coincides with the translated body p(s0,,t0) <- p1(s1,,t1)&..&pk(sk,,tk) which is accumulated in "Body" during the translation process.

So we have proved (A), (B), (C) and (C'). (A) says that: under the non-logical axiom iff(P),

m0
all(y, p(x,,y)->cont-p'(C,y))

is logically equivalent to:

all(v1, (x=s1 & E1 -> cont-p'(C,t1)) &
...
all(vl, (x=sl & El -> cont-p'(C,tl))

m0
where p(sj,,tj)<-Ej (1=<j=<l) is an enumeration of the clauses about p in t-m(P) with head mode m0.

On the other hand, (B), (C) and (C') say that under non-logical axiom iff(Clsr)+Aux+Ew(Uf), each conjunct:

all(vj, x=sj & Ej -> cont-p'(C,tj)) (1=<j=<l)

is logically equivalent to Fj, the formula accumulated in "Body" as the translated body of p(sj,,tj)<-Ej. Combining the two, we have:

iff(P)+iff(Clsr)+Aux+Ew(Uf) |-
all(y, p(x,,y)->cont-p'(C,y)) <-> F1&..&Fl.

As we pointed out before, this is enough to complete { case 2 }.

2.10.2.3 (Case 3) an iff-definition that belongs to iff(Cont')

We show that every iff-definition of a continuation predicate included in iff(Cont') is a logical consequence of iff(P)+iff(Clsr)+iff(Cont)+Ew(Uf).

Firstly, let us review how continuation clauses in Cont' and those in Cont are generated. Let cont-q' be the continuation

predicate for a user predicate q under some mode pattern. A close look at the translation process reveals that the continuation clauses about cont-q' included in Cont and those included in Cont' come from the common clauses in t-m(P), which allows us, corresponding to an enumeration of cont-q' clauses in Cont:

cont-q' (u1, C) <-E1, . . , cont-q' (um, C) <-Em,

to enumerate, by allowing a body to be a disjunction, cont-q' clauses in Cont' :

cont-q' (u1, C) <-F1\/F1' , . . , cont-q' (um, C) <-Fm\/Fm'

in such way that for every j (1=<j=<m), cont-q' (uj) <-Fj\/Fj' (referred to as a daughter clause here) and cont-q' (uj) <-Ej (referred to as a brother clause here) come from the same non-unit clause (referred to as the mother clause here) belonging to t-m(P).

Secondly note that when we construct an iff-definition, in general, we reach the same formula whether or not the body of a clause is unified using "\/" or separated into several clauses and whether or not some variables occurring in the body of a clause are existentially quantified.

Thirdly, as mentioned before, there exists at least one continuation clause for each mode pattern. So m>0.

As a result, we can conclude that in order to prove that the iff-definition of cont-q' included in iff(Cont') is a logical consequence of iff(P)+iff(Clsr)+iff(Cont)+Ew(Uf), it suffices to prove that: for every j (1=<j=<m),

iff(Clsr)+Aux+Ew(Uf) |- Ej<->Fj\/exist(wj, Fj')

holds where wj is some free variables appearing in Fj', assuming that cont-q' (uj) <-Fj\/Fj' is a brother of cont-q' (uj) <-Ej and both have the same mother clause.

(case 3-1)

A mother clause is a type-1 extended clause (see (step 3) in the translation algorithm). Let it be:

$$A0 \leftarrow \mathrm{all}\left(y,\ \overset{m,\,f(z)}{q(s,,t)} \rightarrow A2\right).$$

It generates a daughter clause:

cont-q' (f(z), y1) <-all(y, y1=/=t) \/ ((y1=t) &A2)
where y1 is a sequence of |t| new variables and
if |y1|=0, all(y, y1=/=t) \/ ((y1=t) &A2)
is replaced by A2.

and this is added to Cont' during translation. In this case, Cont includes the brother clause:

```
cont-q' (f1(z1), y1)<-all(y, y1=t -> A2).
(if |y1|=0, replace all(y, y1=t -> A2) by A2).
```

Let us compare the bodies of these clauses. Since the variable condition on Cl:

```
y=<Fvar(t)
```

is satisfied, it follows from (lemma 2.10.1) that:

```
Ew(Uf) |- all(y, y1=t -> A2).
          <-> all(y, y1=/=t)\/exist(y, (y1=t)&A2) ... (A)
```

So this case is done.

(case 3-2)

A mother clause is a type-2 extended clause. Let it be:

```
m0                  m2, f2
A0<- all(y, A1->q(s, , )).
```

It generates a daughter clause:

```
cont-q' (f2)<-false
```

which is added to Cont' during translation. In this case, Cont includes the brother clause:

```
cont-q' (f2)<-false.
```

Therefore, since

```
|- false<->false                                          ... (B)
```

holds trivially, this case has finished too.

(case 3-3)

A mother clause is a non-unit definite clause that has a goal containing q. Let it be:

```
      m0          m1, f1(z1, C)      m, fh(zh, C)       mk, fk(zk, C)
p0(s0, , t0) <- p1(s1, , t1) &..& q(sh, , th) &..& pk(sk, , tk)
where k>=1
```

and yh be a sequence of |th| new variables. When h<k, it generates a daughter clause:

```
cont-q' (fh(zh, C), yh) <-
        all(vh', yh=/=th)\/
                ((yh=th)&ph+1' (sh+1, fh+1(zh+1, C)))
```

```
        where   vh' = sorted(Fvar(th) \
                                Fvar(s0, s1, t1, . . , sh-1, th-1, sh)),
                ph+1' is the closure predicate
                for ph+1 under mh+1.
```

Its brother clause in Cont is:

```
        cont-q' (fh(zh, C), yh) <-
                all(vh, (yh=th & ph+1(sh+1, , th+1) &. . & pk(sk, , tk))
                                        -> cont-p0' (C, t0))

        where   vh = sorted(Fvar(th, sh+1, th+1, . . , sk, tk, t0)
                                \Fvar(s0, s1, t1, . . , si-1, ti-1, sh)).
```

Similarly when h=k, namely when q(sh, , th) is the right most goal of the body, it generates a daughter clause:

```
        cont-q' (fk(zk, C), yk) <-
                all(vk', yk=/=tk) \/ ((yk=tk) &cont-p0' (C, t0))

        where   vk' = sorted(Fvar(tk) \
                                Fvar(s0, s1, t1, . . , sk-1, tk-1, sk)),
                cont-p0' is the continuation predicate for m0 and
                if |tk|=0, replace the body by cont-p0' (C, t0).
```

Its brother clause in Cont is:

```
        cont-q' (fk(zk, C), yk) <- all(vk, yk=tk -> cont-p0' (C, t0))

        where   vk = sorted(Fvar(tk, t0) \
                                Fvar(s0, s1, t1, . . , sk-1, tk-1, sk))
                and
                if |tk|=0, replace the body by cont-p0' (C, t0).
```

In either case, the generated daughter clause is added to Cont'. Recall that Aux is defined as Aux={ A<->F | A<-F is in Cont} and we proved iff(Cont)+Eq(Uf) |- Aux. So to prove the proposition in this case, we show in the following that the bodies of daughter and brother clauses are equivalent under iff(Clsr)+Aux+Ew(Uf), or symbolically,

```
        iff(Clsr)+Aux+Ew(Uf) |-
                all(vh,  yh=th & ph+1(sh+1, , th+1) &. . & pk(sk, , tk)
                                -> cont-p0' (C, t0))
                <-> all(vh', yh=/=th) \/
                        exist(vh', (yh=th) &
                                ph+1' (sh+1, fh+1(zh+1, C))))

        where   vh' = sorted(Fvar(th) \
                                Fvar(s0, s1, t1, . . , sh-1, th-1, sh))
                and
                ph+1' is the closure predicate
                for ph+1 under mh+1
```

holds when h<k and

```
iff(Clsr)+Aux+Ew(Uf)  |-
        all(vk,  yk=tk -> cont-p0' (C, t0))
        <-> all(vk',  yk=/=tk) \/
                        exist(vk', (yk=tk)&cont-p0' (C, t0))

where   vk'  = sorted(Fvar(tk) \
                        Fvar(s0, s1, t1, . . , sk-1, tk-1, sk)),
        cont-p0' is the continuation predicate for m0
        and
        if |tk|=0, replace both bodies by cont-p0' (C, t0)
```

holds when h=k. We treat the simpler case h=k first.

When |tk|=0, what we have to prove is:

iff(Clsr)+Aux+Ew(Uf) |- cont-p0' (C, t0) <->cont-p0' (C, t0)

and this is vacantly true. So suppose |tk|>0. Remember that because Cl, the mother clause, satisfies the variable condition:

```
Fvar(t0) =<Fvar(s0, s1, t1, . . , sk, tk),
```

we have:

```
vk
= Fvar(tk, t0) \Fvar(s0, s1, t1, . . , sk-1, tk-1, sk)
= (Fvar(tk) \Fvar(s0, s1, t1, . . , sk-1, tk-1, sk))
        + (Fvar(t0) \Fvar(s0, s1, t1, . . , sk-1, tk-1, sk))
= (Fvar(tk) \Fvar(s0, s1, t1, . . , sk-1, tk-1, sk)) + {}
= Fvar(tk) \Fvar(s0, s1, t1, . . , sk-1, tk-1, sk)
= vk'
=< Fvar(tk)
```

as set. Since yk is a sequence of |tk| new variables, it follows from vk=<Fvar(tk), Ew(Uf) and { lemma 2.10.1 } that:

```
Ew(Uf)  |-
        all(vk,  yk=tk -> cont-p0' (C, t0))
        <-> all(vk',  yk=/=tk) \/
                exist(vk',  (yk=tk)  & cont-p0' (C, t0)).
```

So this case is finished. The case h<k is a bit longer. We calculate as follows:

```
        all(vh,  (yh=th)&ph+1(sh+1, , th+1)&. .&pk(sk, , tk)
                                                ->cont(C, t0)))

<->     all(vh,  yh=th ->  (ph+1(sh+1, , th+1)&. .&pk(sk, , tk)
                                                ->cont(C, t0)))

<->     all(vh',  yh=th ->
```

```
                all(vh+1,  ph+1(sh+1,, th+1)  &..& pk(sk,, tk)
                                                  ->cont-p(C, t0)))

        /* This is sound. For, as a set,
        vh' = Fvar(th)\Fvar(s0, s1, t1,..., sh-1, th-1, sh)) and
        vh  = Fvar(th, sh+1, th+1,..., sk, tk, t0)\
                                  Fvar(s0, s1, t1,..., sh-1, th-1, sh).
        Therefore,
        vh\vh'
        =( Fvar(th, sh+1, th+1,..., sk, tk, t0)\
                Fvar(s0, s1, t1,..., sh-1, th-1, sh)  )
                  \( Fvar(th)\Fvar(s0, s1, t1,..., sh-1, th-1, sh)  )
        =(Fvar(th, sh+1, th+1,.. tk, sk, t0)\Fvar(s0, s1, t1,..., th, sh))
                                                  \Fvar(th)
        = Fvar(th, sh+1, th+1,.. tk, sk, t0)\
                 (Fvar(s0, s1, t1,..., th, sh)\/Fvar(th))
        = Fvar(th, sh+1, th+1,.. tk, sk, t0)\Fvar(s0, s1, t1,..., th, sh, th)
        = Fvar(sh+1, th+1,.. tk, sk, t0)\Fvar(s0, s1, t1,..., th, sh, th)
        = Fvar(th+1,.. tk, sk, t0)\Fvar(s0, s1, t1,..., th, sh, th, sh+1)
           ( because Fvar(sh+1)=<Fvar(s0, s1, t1,..., th, sh, th)  holds
             by variable condition on Cl.  )
        = vh+1   */

<->     all(vh',  yh=th ->
          all((vh+1, yh+1),  ph+1(sh+1,, yh+1) ->
            (yh+1=th+1  & ph+2(sh+2,, th+2)&..& pk(sk,, tk)
                                                  ->cont-p(C, t0))))

        /* This is sound since yh+1 is a sequence of |th+1| new
         variables.      */

<->     all(vh',  yh=th ->
          all(yh+1,  ph+1(sh+1,, yh+1) ->
            all(vh+1,  yh+1=th+1  & ph+2(sh+2,, th+2)&..& pk(sk,, tk)
                                                  ->cont-p(C, t0)))

        /* For vh+1/\Fvar(sh+1)={}  by the definition of vh+1   */

<->     all(vh',  yh=th ->
          all(yh+1,  ph+1(sh+1,, yh+1)
                                  ->cont-ph+1'(fh+1(zh+1, C), th+1)))

        /* using the non-logical axiom ( a2-2-h+1 )  included
        in Aux,
                all(vh+1,  yh+1=th+1 &
                                  ph+2(sh+2,, th+2)  &..& pk(sk,, tk)
                                                  ->cont-p(C, t0)))
        is folded into:
                cont-ph+1'(fh+1(zh+1, C), th+1)     */

<->     all(vh',  yh=th -> ph+1'(sh+1, fh+1(zh+1, C)))

        /* This is a folding by the iff-definition of ph+1'
```

```
        in iff(Clsr)     */

<->     all(vh', yh=/=th) \/
                exist(vh', (yh=th)&ph+1' (sh+1, fh+1(zh+1, C)))

        /* Recall that yh is a sequence of |th| new variables.
        so that we have vh' /\yh ={}. The rest is by Ew(Uf) and
        { lemma 2.10.1 } */.
```

Thus, we have proved:

```
        iff(Clsr)+Aux+Ew(Uf) |-
                all(vh, yh=th & ph+1(sh+1,,th+1) &..& pk(sk,,tk)
                                        -> cont-p0' (C, t0))
                <-> all(vh', yh=/=th) \/
                        exist(vh', (yh=th)&
                                        ph+1' (sh+1, fh+1(zh+1, C)))
```

Therefore, { case 3-3 } has finished and so has the entire proof finally.

END of proposition 2.10.1

2. 10. 3. Proving model theoretic relation

In this subsection, we relate iff(P)+iff(Clsr)+iff(Cont) +Ew(Cf)+Ew(Uf) to comp(P) by model theoretic means and thus relate comp(P) to iff(Pc). Let us start with model theoretic definitions:

(Interpretation):

An interpretation I for a language L means a mapping from the function and predicate symbols in L to some functions and relations over a non-empty set D called the domain of I. Let f be an m-ary function symbol and r be an n-ary predicate symbol in L. We use <f> and <r> to refer to the result of mapping of f and r by I. <f> must be an m-ary function and <r> must be an n-ary relation over D. We say that f, r is interpreted as <f>, <r> respectively by I. Note that by definition, there are only two 0-ary relations, one called true, the other false and the 0-ary constant symbol "false" must be interpreted as false.

As usual, an interpretation I over D for atoms is extended to that of any formula F(x), depending on the assignment of elements in D to x, giving true or false. Put x=(X1,..,Xm) and d=(d1,..,dm) where d1,..,dm are individual elements in D. Let F(d) denote a formula resulting from the substitution {X1<-d1,..,Xm<-dm} applied to F(x). We write:

$$F(d) = F(x)\ \{X1 \leftarrow d1, \ldots, Xm \leftarrow dm\}$$

and call F(d) a closed formula over D. If F(d) becomes true in I, we denote it as:

$$I \models F(d).$$

We stipulate that $I \models F(x)$ iff $I \models all(x, F(x))$ when F includes free variables x. An interpretation I is called a model of F if $I \models F$. Similarly I is a model of a set of formulae S if I is a model of every formula in S. Letters M, M'.. are used for models. The completeness of first order logic means that given two sentences A and B, if $M \models B$ holds for every model M of A, then holds $A \vdash B$.

When interpreting a formula F, symbols not appearing in it have evidently nothing to do with the interpretation. On the other hand, one usually deals with a formula, or a set of formulae which includes only a tiny part of the whole set of available symbols. Therefore, it is convenient to let the word "interpretation" hereafter mean not one that interprets every symbol but one that only interprets the symbols relevant to the formulae or a set of formulae in question. Accordingly, when we say an interpretation I for a formula F or a set of formulae S, we understand that I only interprets the function

symbols and predicate symbols that actually occur in F or S. Other symbols are not given any interpretation by I.

It sometimes happens that we want to focus on the interpretation of some specific predicate symbols r1,..,rm in L. In such case, we use I(<r1>,..,<rm>) to emphasize that I interprets a predicate symbol "rj" as <rj> for each j(1=<j=<m). Therefore, this notation does not necessarily mean that r1,..,rm are the only predicates I interprets.

It also happens that we extend a model M to another model M' that can interpret new function symbols f1,..,fk and new predicate symbols r1,..,rm as well by adding to M corresponding new functions <f1>,..,<fk> and corresponding new relations <r1>,..,<rm>. In such case, we denote the relationship between M and M' by:

M' =M[<g1>,..,<gk>,<r1>,..,<rm>]

[Product]:

For a set D, let prod(n,D) denote n-cartesian products of D. For an n-ary function f over D, namely a mapping from prod(n,D) into D, the range of function Range(f) is defined by:

Range(f)= { a | a=f(b) for some b in prod(n,D) }.

[Lemma 2.10.2]
Let c1,c2,.. be an infinite sequence of new constants and put

Inf = { ci=/=cj | i=/=j }.

A model M of comp(P)+Inf can be extended to that of comp(P)+Ew(Cf) over the same domain as M by adding to M suitable interpretations for the continuation function symbols in Cf.

(Proof)
Let M be an arbitrary model of comp(P)+Inf. The domain D of M must be infinite because of Inf. And let { f1,..,fk } (k>0) be Cf, the set of continuation function symbols introduced by compilation.

Ew(Cf) consists of two parts. One is:

{ fi(x)=fi(y) -> x=y
| 1=<i=<k, (x,y) is a variable sequence without repetition }.

This means fi (1=<i=<k) must be a one-to-one mapping. The

other one is:

$$\{\ fi(x) =/= fj(y) \mid 1=<i, j=<k,\ i=/=j,\ (x, y) \text{ is a variable sequence without repetition }\}.$$

This requires that $Range(fi) /\backslash Range(fj) = \{\}$ if $fi=/=fj$ $(1=<i, j=<k)$.

Since D is infinite, we can partition D into k-disjoint sets D1,..,Dk such that each Di $(1=<i=<k)$ is infinite and has the same cardinality as D.

Let Ni $(1=<i=<k)$ be the arity of function symbol fi. Then there is a one-to-one and onto mapping <fi> from prod(Ni, D) to Di because prod(Ni, D) and Di has the same cardinality.

Extend M to M(<f1>,..,<fk>) by adding to M the interpretation of { f1,..,fk } with fi being interpreted as <fi> $(1=<i=<k)$.

For each i $(1=<i=<k)$ <fi> is a one-to-one by definition and $Range(<fi>) /\backslash Range(<fj>) = \{\}$ (i=/=j) follows from $Range(<fi>) = Di$, $Range(<fj>) = Dj$ and $Di/\backslash Dj = \{\}$ for $1=<i, j=<k$, i=/=j. Therefore, M(<f1>,..,<fk>) is the required model.

END of lemma 2.10.2

Next we prove that given an interpretation I for the functions and user predicates concerned, there always exist continuation predicates satisfying iff(Cont). The proof depends on the existence of a fixed point in some complete lattice of the monotonous function derived from iff(Cont). Therefore, it is convenient to introduce terminologies used to define such function.

{ Non-positive/Non-negative formula }:

A formula F is non-negative (non-positive) w.r.t. a predicate symbol r iff:

(1) F is an atom (F is an atom not containing r)
(2) F= F1&F2 or F= F1\/F2 and F1 and F2 are both non-negative (non-positive) w.r.t. r,
(3) F= ~F1 (= F->false) and F1 is non-positive (non-negative) with respect to r,
(4) F= F1->F2, this case is reduced to F= (~F1\/F2) or
(5) F= exist(x, F1) or F=all(x, F1) and F1 is non-negative (non-positive) with respect to r.

We say a formula F is non-negative (non-negative) with respect to {r1,..,rk} if F is non-negative (non-negative) with respect to every rj (1=<j=<m).

(Lemma 2.10.3)
Suppose that there are two interpretations I(<r1>,..,<rm>) and I'(<r1>',..,<rm>') over D which are different only in the interpretation of the predicate symbols r1,..,rm. Also suppose

<rj> =< <rj>' for every j (1=<j=<m)

holds. Then we have for any closed formula F over D:

(P) if F is non-negative w.r.t. {r1,..,rm} and I |= F, then I' |= F and
(N) if F is non-positive w.r.t. {r1,..,rm} and I' |= F, then I |= F.

This is proved by induction on the number of logical connectives included in F. We only show the crucial case here. Suppose (P) and (N) already hold for F. We show that they hold for ~F(= F->false).

Suppose F->false is non-negative with respect to {r1,..,rm} and I |= F->false. It follows F is non-positive with respect to {r1,..,rm} and Not(I |= F). So, by (N), we have Not(I' |= F), which means I' |= F->false. Hence (P) for F.

Suppose F->false is non-positive with respect to {r1,..,rm} and I' |= F->false. Then F must be non-negative with respect to {r1,..,rm} and Not(I' |= F). So, this time by (P), we have Not(I |= F), which means I |= F->false. Hence (N) for F.

END of Lemma 2.10.3

For set D, let Pow(D) denote the power set of D, the set of all subsets. It is well known that Pow(D) is a complete lattice under the set inclusion ordering. Let us consider Prod(m, Pow(D)), m cartesian products of Pow(D). It is also well known that Prod(m, Pow(D)) becomes a complete lattice under the ordering introduced by:

(a1,..,am) =< (b1,..,bm)
iff aj=<bj as set for every j (1=<j=<m).

We use this simple and useful fact in lemma 2.10.4.

(Lemma 2.10.4)
An interpretation I of symbols in iff(Cont) other than continuation predicates can be extended to a model I' of

iff(Cont) over the same domain by adding to I appropriate interpretations for the continuation predicates included in Cf.

(Proof)
It is easy to check that every formula in Cont can be written as:

```
cont-p'(s) <- all(x, s=t -> A1),
cont-p'(s) <- false or
cont-p'(s) <- all(x, s=t & A1&..&Am -> cont-q'(u))
```

where cont-p' and cont-q' are some continuation predicates, m>=0 and A1,..,Am are all user atoms. So every antecedent part of a clause about continuation predicate is non-negative with respect to all continuation predicates.

Let cont-p' be an arbitrary continuation predicate appearing in Cont and cont-p'(s1)<-E1,..,cont-p'(sk)<-Ek be an enumeration of clauses about cont-p'. Then we can say, for every i (1=<i=<k), Ei is non-negative w.r.t. all continuation predicates. Consequently, so is the right hand side of the iff-definition of cont-p':

```
cont-pi'(x) <-> exist(v1, x=s1 & E1) \/
                                ..\/exist(vk, x=sk & Ek).
```

Let cont-p1',..,cont-pm' (m>0) be an enumeration of continuation predicates appearing in Cont and put

```
iff(Cont) =
   { cont-p1'(x1) <->F1(cont-p1',..,cont-pm')(x1),
                    ...
     cont-pm'(xm) <->Fm(cont-p1',..,cont-pm')(xm) }.
```

The expression Fj(cont-p1',..,cont-pm')(xj) (1=<j=<m) means that some of cont-p1',..,cont-pm' can possibly appear in the formula Fj and xj is the set of all free variables in Fj.

What we have to prove is that, given an interpretation I over a domain D which interprets the functions and predicates other than cont-p1',..,cont-pm', we can extend it to an interpretation I' over D such that: for all j (1=<j=<m),

```
I' |= all(xj, cont-pj'(xj)
                        <->Fj(cont-p1',..,cont-pm')(xj)).
```

To make matters simple, we suppose all continuation predicates are unary (generalization is easy). For each j (1=<j=<m), define Kj(R1,..,Rm), a subset of D, as:

```
Kj(R1,..,Rm) =
        { d | I(R1,..,Rm) |= Fj(cont-p1',..,cont-pm')(d),
              d is an element in D }.
```

When R1,..,Rm considered as parameters, this defines a mapping Kj form Prod(m, Pow(D)) to Pow(D). We then introduce K, a mapping over Prod(m, Pow(D)), by putting:

(R1',..,Rm') =
K(R1,..,Rm)
where Rj' = Kj(R1,..,Rm) for each j(1=<j=<m).

Since each Fj(1=<j=<m) is non-negative with respect to { cont-p1', .., cont-pm' }, it follows from { lemma 2.10.3 } that for all j (1=<j=<m), Kj is monotonous in the sense that:

(R1,..,Rm) =< (R1',..,Rm')
implies Kj(R1,..,Rm) =<Kj(R1',..,Rm')

thereby,

(R1,..,Rm) =< (R1',..,Rm')
implies K(R1,..,Rm) =<K(R1',..,Rm').

In other words, K is a monotonous mapping over the complete lattice Prod(m, Pow(D)). Therefore, K has a fixed point in it. Take one of the fixed points and let it be (R1,..,Rm) and consider the following interpretation I' over D:

I' =I(<cont-p1'>,..,<cont-pm'>)
where <cont-pj'> = Rj for each j (1=<j=<m),

This is an extension of I obtained by adding to it interpretation of continuation predicates. As (R1,..,Rm) is a fixed point of K, we have:

(R1,..,Rm) =K(R1,..,Rm).

This means for each j (1=<j=<m),

Rj =Kj(R1,..,Rm)

or equivalently, by the definition of Kj,

d belongs to Rj
iff I(R1,..,Rm) |= Fj{cont-p1',..,cont-pm'}(d),
.FE
for any element d in D. As Rj is an interpretation of cont-pj' by I' for each j (1=<j=<m), the above means:

I' |= all(x, cont-pj'(x) <->Fj(x))

for all j (1=<j=<m). Therefore, I' is the required interpretation.

END of lemma 2.10.4

{ Proposition 2.10.2 }

Let c1, c2, .. be an infinite sequence of new constants not included in L(P). Put

Inf = { ci=/=cj | i=/=j }.

Then, for any formula F in L(P), we have:

comp(P)+Inf |- F
if comp(P)+Ew(Cf)+iff(Clsr)+iff(Cont) |- F

(Proof)

Since comp(P)+Ew(Cf)+iff(Clsr)+iff(Cont) can be regarded as a conservative extension of comp(P)+Ew(Cf)+iff(Cont) by introduction of closure predicates with the set of defining formulae iff(Clsr), we have only to prove:

comp(P)+Inf |- F if comp(P)+Ew(Cf)+iff(Cont) |- F

holds for a formula F in L(P).

Let M1 be an arbitrary model of comp(P)+Inf and its domain D. M1 gives an interpretation over D for the user predicates, user functions and the constants in Inf. By { lemma 2.10.2 }, M1 can be extended to a model M2 over D of comp(P)+Ew(Cf) by adding to M1 appropriate interpretations for the continuation functions over D. Then, by { lemma 2.10.4 }, M2 can again be extended to a model M3 over D of comp(P)+Ew(Cf)+iff(Cont) by adding to M2 appropriate interpretations for the continuation predicates over D.

It follows from M3 |= comp(P)+Ew(Cf)+iff(Cont) that M3 |= F is derived. As M1 and M3 give the same interpretation over the symbols included in F, we have M1 |= F. Thus, every model of comp(P)+Inf is a model of F. Hence, by the completeness of first order logic, comp(P)+Inf |- F.

END of proposition 2.10.2

2. 10. 4. First soundness theorem

(Lemma 2. 10. 5)

Suppose P includes at least one non-constant function symbol. Let F be a formula in L(P) and c1, c2, .. be an infinite sequence of new constants not included in L(P). Put Inf = { ci=/=cj | i=/=j }. Then,

comp(P) |- F if comp(P)+Inf |- F.

(Proof)

Suppose that comp(P)+Inf |- F holds. Let M be an arbitrary model of comp(P). Since comp(P) includes Eq(Uf) and Eq(Uf) includes the set { X=/=t(X) | t is a term in L(P) which include a variable X }, the domain D of M must be infinite. So, we can pick up denumerably many elements <c1>, <c2>, ... in D such that <ci>=/=<cj> if i=/=j. Extend M to an interpretation M' of comp(P)+Inf by taking <ci> as the interpretation of ci (i=1, 2, ...). Then, M' satisfies comp(P)+Inf, thereby F. We conclude that M satisfies F because and M and M' have the same interpretation for the symbols included in F. Therefore, F is true in every model of comp(P). In other words, comp(P) |- F.

END of lemma 2. 10. 5

Now we are in a position to state the first soundness theorem and prove it.

(Theorem 2. 10. 1)

Let P0 be a first order program including at least one non-constant function symbol, P be a basic program which is the output of the preprocessing algorithm applied to P0 and Pc be the compiled program of P. Then,

(1) comp(P0) |- (A)T if Pc ?- <(), A, T>
(2) comp(P0) |- ~A if Pc ?- <failure, A, e>

for every atom A in L(P0).

(Proof)

We only show (1). (2) is treated similarly. Let A be an arbitrary atom in L(P0) and suppose Pc ?- <(), A, T>. As P is obtained from P0 by the preprocessing algorithm, comp(P) is a conservative extension of comp(P0). Furthermore, P0 and P have the same function symbols. So P has at least one non-constant function symbol. A is an atom in L(P) because L(P)>=L(P0). Let Inf be defined as in (lemma 2. 10. 5). Then we have:

Pc ?- <(), A, T>

=> iff(Pc)+Eq(Uf) |- (A)T

```
        ( by { theorem 2.9.1 }. A is an atom in L(P) )
=>      comp(P)+Ew(Cf)+iff(Clsr)+iff(Cont ) |- (A)T
        ( by { proposition 2.10.1 }. (A)T is an atom in L(P)
        by { theorem 2.9.1 } )
=>      comp(P)+Inf |- (A)T
        ( by { proposition 2.10.2 } )
=>      comp(P)  |- (A)T
        ( by { lemma 2.10.5 } )
=>      comp(P0) |- (A)T
        ( comp(P0) is a conservative extension of comp(P) ).
```

END of theorem 2.10.1

Note that in our proof system, proving F means proving all(v, F) where v=Fvar(F).

The reverse direction does not necessarily hold. For instance, put P0 =P ={ p(x)<-(q(x)<-q(x)), q(x)<-q(x) }. Iff(P0) |- p(x) immediately follows. But we can not have Pc ?- <(), p(x), T> because of infinite loop by q(x)<-q(x). To assure the reverse direction, Pc must satisfy some conditions. For example, consider the following condition:

Not (Pc ?- <(), A, T>) implies Pc ?- <failure, A, e>.

It is equivalent to say that an interpreter always halts with success or failure. An executable program satisfying this condition is said to be totally-terminating. Then it is obvious that comp(P0) is consistent and Pc is totally-terminating, we have:

```
(1)     comp(P0) |- (A)T        iff     Pc ?- <A, A, T>
(2)     comp(P0) |- ~A          iff     Pc ?- <failure, A, e>.
```

for any atom A in L(P0).

2. 10. 5. Second soundness theorem

We have to mention the case when a user program P does not have any non-constant function symbol. In this case, comp(P) may have a finite model that can not be extended to a model of comp(P)+Inf and hence the reasoning we adopted may no longer remain valid. We get around this difficulty by imposing some conditions on P. It is related to the notion of "domain independentness" which we will explain now.

It arouse in the research of data base theory (40). There we often meet a situation where the domain of discourse is enlarged by, for example, a new customer, a new employee or a new born baby. As a result, some propositions about the data base may fail to hold whereas others may continue to hold after the enlargement.

Consider a world where a single father John is living with a girl baby Jane and they are the only inhabitants in that world. So the world consists of { john, jane } where john and jane are entities corresponding to the names John and Jane. We have:

```
is-parent(john, jane),
```

and

```
human-baby(jane).
```

about this world. As we can say an inhabitant is either a male baby or someone's father, we also have:

```
all(X, human-baby(X)\/exist(Y, is-parent(X, Y))).
```

Suppose then, one day John got a little dog from somewhere when he went for a walk. So that dog enlarges the world as a new inhabitant. Now the new world D' consists of { john, jane, * } where * denotes the dog, a new entity (John did not give it a name). The up-dated world must look like:

```
is-parent(john, jane),
human-baby(jane),
~is-parent(john, *),
~human-baby(*).
```

Note that both of "is-parent" and "human-baby" have preserved their extensions through this enlargement. Consequently, a formula such as is-parent(john, jane) remains true in the new world. But there are other formulae that can not keep their truth values even though the predicates concerned have kept their extensions.

```
all(X, human-baby(X)\/exist(Y, is-parent(X, Y)))
```

is an example which fails to hold in the up-dated world because

* is neither a human baby nor a parent.

We call a formula F(x) "domain independent" if its extension remains the same in both old and new world. More precisely, let F(x) be a formula including only constants and variables and I be an interpretation for F over a domain D. Define *-extension as follows:

Let p1,..,pk be the list of predicates included in F. Enlarge the domain D by a new element * and put D' = D+{*}. Extend I to I', an interpretation over the new domain D' in such a way that:

for every i (1=<i=<k) and d, a sequence of elements in D',

(1) if d does not include *, put
I' |= pi(d) iff I |= pi(d)

(2) or else, put I' |= ~pi(d).

I' is called the *-extension of I to the domain D+{*}. It is a very special case of model extension known in model theory (59). I' gives the same interpretation for predicates as I as long as the new element * is not concerned and it assigns the value "false" to every atomic predicate including *.

(Domain independentness):

F(x) is said to be domain independent iff

{ d | I |= F(d), d is a sequence of elements in D }
=
{ d' | I' |= F(d), d' is a sequence of elements in D' }

holds for any interpretation I with domain D and its *-extension I'. When F is a sentence, we call it domain independent sentence. We use I* to denote a *-extension of I.

It is immediately obvious that if there is a successive *-extension I1,..,Ik such that Ii+1 = Ii* (1=<i=<k-1) and if a formula F(x) is domain independent, we have:

{ d | I1 |= F(d), d is a sequence of elements in D1 }
=
{ d' | Ik |= F(d'), d' is a sequence of elements in Dk }.

Especially, when F is a domain independent sentence, we have:

I1 |= F iff Ik |= F.

Note that since being domain independent is a model

theoretic property, any formula logically equivalent to a domain independent sentence is also domain independent.

Now we can get back to the original problem.

{ Lemma 2.10.6 }
Suppose that P has no function symbols except constants. Also suppose that there is a domain independent sentence F and comp(P) is a conservative extension of F. Let c1, c2, .. be an infinite sequence of new constants not included in L(P) and put Inf = { ci=/=cj | i=/=j }. Then,

comp(P) |- A if comp(P)+Inf |- A
and
comp(P) |- ~A if comp(P)+Inf |- ~A

for any atom A in L(P).

(Proof)
We prove the first case. The second case is similar. Suppose for an atom A in L(P), we have:

comp(P)+Inf |- A.

Since only finite formulae are used in the deduction, there is some k such that:

comp(P)+Inf(k) |- A

where Inf(k)={ci=/=cj | i=/=j, i,j=<k }. Take the conjunction of all formulae in Inf(K) and write it as J(c1,..,ck). By the deduction theorem, we have:

comp(P) |- J(c1,..,ck)->A.

None of c1,..,ck appears in comp(P) or A. Therefore, we can transform this deduction by replacing c1,..,ck by new different variables X1,..,Xk into another deduction:

comp(P) |- J(X1,..,Xk)->A.

Since iff(P) is a conservative extension of F by our assumption and J(X1,..,Xk)->A is a formula in L(F), we have:

F |- J(X1,..,Xk)->A.

By introducing existential variables to J(X1,..,Xk) and universal variables to A, we obtain:

F |- exist(x,J(x))->all(v,A)

where x=(X1,..,Xl) and v=Fvar(A). Recall that exist(x,J(x)) is a sentence stating that there are at least k different

elements in the domain. In the following, we show F |- A.

Let M0 be an arbitrary model of F. Using *-extension as many times as necessary, we can enlarge M0 to M0' so that it can satisfy exist(x, J(x)) while keeping M0' as a model of F. This is possible because F is a model independent sentence by our assumption. From M0' |= F and F |- exist(x, J(x))->all(v, A) we have M0' |= exist(x, J(x))->all(v, A). Then, from M0' |= exist(x, J(x)), we have M0' |= all(v, A). Since M0' and M0 assign the same relation to the same predicate and A is an atom, M0 |= all(v, A) follows. Since we supposed M0 was an arbitrary model of F, F |- all(v, A) follows from the completeness of first order logic. Finally remembering again iff(P) is a conservative extension of F, we reach:

comp(P) |- all(v, A).

End of lemma 2.10.6

(Theorem 2.10.2)

Let P0 be a first order program including only constants, P be a basic program which is the output of the preprocessing algorithm applied to P0 and Pc be the compiled program of P. Then, if comp(P0) (considered as a sentence) is domain independent, we have:

(1)	comp(P0) |- (A)T	if	Pc ?- <(), A, T>
(2)	comp(P0) |- ~A	if	Pc ?- <failure, A, e>

for every atom A in L(P0).

(Proof)

The proof is completely parallel to that of (theorem 2.10.1) except for the use of (lemma 2.10.6) where F=comp(P0) instead of (lemma 2.10.5). Note comp(P0) is a finite set and by converting it to a conjunction, we can consider it as a single sentence.

The rest of the problem is how to know if comp(P0), or more generally, a first order formula, is model independent. Unfortunately it is proved that deciding whether a formula is model independent or not is recursively unsolvable (40). Nevertheless there are some syntactically recognizable, yet large enough classes of domain independent formulae. As describing them is another topic, we are satisfied with giving a simple idea about when comp(P0) becomes domain independent for a first order program P0.

(Lemma 2.10.7)

Let G(x) be a basic goal that has no function symbols except constants and x be an enumeration of free variables in it. And let I and I' be an interpretation over D and its *-extension to D'=D+{*} respectively. Then for any d, a sequence of elements

of D such that |d|=|x|, we have:

I |= G(d) iff I' |=G(d)

where G(d) = G(x) {x<-d}.

(Proof)
When G is an atomic goal, this directly follows from the definition of *-extension. So let G(x) = all(y, A1(y, x) ->A2(y, x)) where A1 and A2 are atoms. We first show:

I' |= ~G(d) implies I |= ~G(d).

Suppose I' |= ~G(d). Then there is some sequence d' of elements of D' such that |d'|=|y| and I' |= A1(d', d) and I' |= ~A2(d', d).

If d' does not include *, this means, by the definition of *-extension, I |= A1(d', d) and I |= ~A2(d', d). Hence I |= ~G(d).

If d' includes *, it follows from I' |= A1(d', d) that * does not occur in A1(d', d). This in turn means * does not occur in A2(d', d) either. For since there is no constant that designates *, * occurring in A2(d', d) but not in A1(d', d) means an existence of a variable that only occurs in A2. But it contradicts the variable condition y=<Fvar(A1) which all(y, A1(y, x) ->A2(y, x)) has to satisfy as an extended goal.

So A2(d', d) has no occurrence of *. Therefore, we have I |= ~A2(d', d) from I' |= ~A2(d', d). We also have I |= A1(d', d) from I' |= A1(d', d). These two in conjunction imply I |= ~G(d).

We then show the reverse direction:

I |= ~G(d) implies I' |= ~G(d).

Suppose I |= ~G(d). This assumption means there must be some sequence d' of elements of D such that |d'|=|y| and I |= A1(d', d) and I |= ~A2(d', d). So from the definition of *-extension, we have I' |= A1(d', d) and I' |= ~A2(d', d), thereby I' |= ~G(d). So we are done.

End of lemma 2.10.7

(Lemma 2.10.8)
Let E(x) be a conjunction of basic goals with x being an enumeration of the free variables in E. Suppose E(x) has no function symbols except constants. And also suppose every free variable in E(x) occurs in some atom in E. Then, E is domain independent.

(Proof)

Let I be any interpretation with domain D and I' be its *-extension to the domain D' (=D+{*}). From (lemma 2.10.7),

```
{ d | I |= E(d), d is a sequence of elements in D and
        |d|=|x| }
=<
{ d' | I' |= E(d'), d' is a sequence of elements in D'
        and |d'|=|x| }
```

is immediate. To show the reverse direction, suppose I' |= E(d') for some sequence d' of elements in D'. Then, d' can not include * because if d' includes *, as every free variable must appear in some atomic goal in E(d'), there must be some atomic goal in E(d') which includes *, thereby false in I'. This contradicts I' |= E(d'). Since * is not included in d', we conclude I |= E(d') by (lemma 2.10.7). So we are done.

End of lemma 2.10.8

For example, conjunctions such as q(x,y)&r(y,z) and all(y, q(x,y)->r(y,z))& s(x,z) are domain independent.

(Proposition 2.10.3)

Let P0 be a first order program with no function symbols other than constants. If the following holds for every clause in P0, comp(P0) becomes domain independent.

(a) A unit clause is ground.
(b) Let A0<-E be a non-unit clause. Then,
(b-1) E is a conjunction of basic goals.
(b-2) Every free variable in the clause appear in some atomic goal in E.

(Proof)

We first show comp(P0) is model independent if so is iff(P0). But this is obvious from comp(P0)=iff(P0)+Eq(P0) and Eq(P0) includes only a formula of the from a=/=b where a and b are different constants. So, it is enough to show iff(P0) is model independent.

Let p(u1)<-E1,..,p(u)<-El (l>=0) be an enumeration of clauses in P0 about p. Iff(p) becomes:

```
all(x, p(x)<->
        exist(v1, x=u1 & E1(v1)) \/
                        ..\/exist(vl, x=ul & El(vl))
```

where x is a sequence of new variables and vi is the list of all free variables appearing in p(ui)<-Ei (0=<i=<l). Let I be an interpretation of P over the domain D and I' be its *-extension to the domain D+{*}.

When l=0, iff(p)= all(x, p(x)<->false) and this is

surely domain independent. So we assume l>0. To show iff(p) is domain independent, we are required to show:

```
I |= iff(p)      implies          I' |= iff(p)
```

and vice versa. Since both directions can be treated similarly, we show only one direction. Namely, we assume:

```
I |= all(x, p(x) <-> exist(v1, x=u1 & E1) \/
                          ..\/exist(vl, x=ul & El))
```

holds and then show:

```
I' |= all(x, p(x) <-> exist(v1, x=u1 & E1) \/
                    ..\/exist(vl, x=ul & El))
```

holds. Therefore, we assume the following sentence is true in I:

```
p(d) <->exist(v1, d=u1 & E1) \/..\/exist(vl, d=ul & El))
```

where d is an arbitrary sequence of elements of D' whose length is |x| and prove it is true too in I'. First we deal with "<-" direction. Suppose exist(v1, d=u1 & E1) \/..\/exist(vl, d=ul & El)) is true in I'. We have:

```
        I' |= exist(v1, d=u1 & E1) \/..\/exist(vl, d=ul & El)).
=>      I' |= exist(vi, d=ui & Ei) for some i (1=<i=<l)
=>      I' |= d=ui(di) & Ei(di) for some sequence di of elements
        of D' such that |di|=|vi|
```

We deal the case when Ei is empty first. Then from the condition (a), vi is the null sequence and ui is a sequence of ground terms. Therefore, ui, when interpreted by I', denotes a sequence of elements of D, thereby so does d from I' |= d=ui(di). Thus we can conclude that neither d or di includes *.

Next we consider the case when Ei is non-empty. In this case, we notice that I' |= Ei(di) means that * is not included in di because otherwise * must appear in Ei(di), especially in some atomic goal G(di) in Ei by the condition (b-2). But then G(di) becomes false in I' and so does Ei(di), contradicting I' |= Ei(di).

We also notice that * can not be included in d either. For if * is included in d, it must be included in si(di) from I' |= d=si(di). Moreover, since our language being considered lacks any constant that can designate *, the presence of * in ui(di) means that it is brought into si(vi) by substituting di for vi. It means that di must include *. But this contradicts again that * is not included in di. In summary, whether Ei is empty or not, .FB => * is not included in d or di. => I |= d=ui(di) & Ei(di) by (lemma 2.10.6) => I |= exist(vi, d=ui

& Ei) => I |= exist(v1, d=u1 & E1) \/..\/exist(vl, d=ul & El)) => I |= p(d) => I' |= p(d). .FE Hence "<-" direction. Next we show "->" direction is also true in I'.

```
        I' |= p(d)
=>      d must be a sequence of elements of D
=>      I  |= p(d)
=>      I  |= exist(v1, d=u1 & E1) \/..\/exist(vl, d=ul & El)
        by our assumption
=>      I  |= exist(vi, d=ui & Ei) for some i (1=<i=<l)
=>      I  |= d=ui(di) & Ei(di)
                        for some sequence di of elements of D
=>      I' |= d=ui(di) & Ei(di)  by ( lemma 2.10.6 )
=>      I' |= exist(vi, d=ui & Ei)
=>      I' |= exist(v1, d=u1 & E1) \/..\/exist(vl, d=ul & El)
```

So both directions are shown to hold.

End of proposition 2.10.3

2. 11. System predicates

In a practical situation, we can not do without system predicates such as "=/=", "<", "=<", "length(L, N)." Usually they are not supposed to be used arbitrarily. Instead they have their "intended mode patterns" and if an actual mode pattern violates the intended one, the system recognizes it as an error and takes some action; for example, issuing a warning message. In this section, we show the treatment of system predicates other than "false" and "=."

The principle is simple. We treat them in the same way as other predicates at the mode labelling phase and at the term labelling phase. If the modes assigned to these predicates are found illegal w. r. t. their allowable modes, we have to stop compiling with failure.

Take length(L, N) for instance which computes the relation that N is the length of list L and suppose that it has the allowable mode patterns { length(+, -), length(+, +) }. If at the mode labelling phase, length(L, N) is labelled:

```
length(-, -)
length(L, N),
```

we stop the compilation with failure. Suppose otherwise and it is labelled as:

```
f1(Z, C), length(+, -)
    length(L, N).
```

Then we introduce length'(X, C) and cont-length'(C, Y) as the closure predicate and continuation predicate for length(+, -) respectively as usual. But at the translation phase, the difference between system predicates and non-system predicates must be taken care of. A system atom located in the body of a clause is treaded similarly to a non-system atom. To generate clauses about continuation predicates introduced by system predicate, we do not need special treatment either. When it comes to generate closure clauses for system predicates however, we must pay special attention.

In case of length(L, N), the universal continuation form for the mode pattern length(+, -) is:

```
length'(X, C) <->all(Y, length(X, Y) ->cont-length'(C, Y)).
```

Assuming that length and cont-length' are already given, we generate clauses about length' so that they can compute this form. As length(X, Y) defines a (partial) function X |---> Y, this form is logically equivalent to:

```
length'(X, C) <-> all(Y, ~length(X, Y)) \/
```

exist(Y, length(X, Y)&cont-length' (C, Y))

So, we generate:

```
length' (X, C) <- all(Y, ~length(X, Y))
length' (X, C) <- length(X, Y) & cont-length' (C, Y)
```

and include them in a compiled program as translated clauses for length'.

The treatment of all(Y, ~length(X, Y)) depends on the user and the system s/he uses. If one uses DEC-10 Prolog (69), the goal \+length(X, Y) can take the place of all(Y, ~length(X, Y)).

Similarly, if the mode pattern assigned to length is length(+, +), we introduce the closure predicate length'' and the continuation predicate cont-length'' corresponding to it as usual. From the universal continuation form:

```
length'' (X, Y, C) <-> (length(X, Y) ->cont-length'' (C, X, Y)),
```

we generate:

```
length'' (X, Y, C) <- ~length(X, Y)
length'' (X, Y, C) <- length(X, Y)&cont-length'' (C, X, Y)
```

as the translated clauses about length''.

As another example, suppose we have "=/=" (not-equal) as a primitive with the allowable mode pattern set { =/=(+, +) }. In this case, we have:

```
=/=' (X, Y, C) <-> (=/=(X, Y) -> cont-=/=' (C, X, Y))
```

as the universal continuation form for =/=(+, +) where =/=' and cont-=/=' is the closure predicate and continuation predicate for =/=(+, +). Then, as this is equivalent to:

```
=/=' (X, Y, C) <-> (X=Y \/ cont-=/=' (C, X, Y)),
```

we have:

```
=/=' (X, X, C)
=/=' (X, Y, C) <- cont-=/=' (C, X, Y)
```

as the translated clauses about =/='.

2. 12. More compilation examples

2. 12. 1. A tiny data base

Suppose we have a very small data base D:

```
rel(a, b)
rel(c, d)
```

and an "all-solution" program A:

```
all-sol(X, L) <- all(Y, rel(X, Y)->mem(Y, L)) & list(L)

mem(Y, [Y|Z])
mem(Y, [H|Z]) <-mem(Y, Z)
list([])
list([H|L]) <-list(L).
```

P0 (= D+A) is a first order program. To compile it, we transform P0 to the basic program P:

```
all-sol(X, L) <- search(X, L) & list(L)
search(X, L) <- all(Y, rel(X, Y)->mem(Y, L))

... definite clauses for "mem", "list" and "rel" ...
```

(a) Basic program P

---

using the preprocessing algorithm. Pe, the extended part of P, is { search(X, L) <- all(Y, rel(X, Y)->mem(Y, L)) }. The mode labelling algorithm starting with Pe ends successfully. So P is compilable. The only mode pattern generated is rel(+, -). The term labelling algorithm at the second phase gives:

```
                                f0(L), rel(+, -)
search(X, L) <- all(Y, rel(X, Y) -> mem(Y, L))

rel(+, -)   rel(+, -)
rel(a, b)   rel(c, d)
```

(b) After term-mode labelling

---

where f0 is a continuation function. The universal continuation form for rel(+, -) is introduced:

```
rel'(X, C) <-> all(Y, rel(X, Y) -> cont-rel'(C, Y))
```

where rel' and cont-rel' are the closure predicate and the continuation predicate for mode rel(+, -). The translation algorithm at the last phase translates it into:

```
search(X, L) <-rel' (X, f0(L))
rel' (X, C) <- (X=/=a \/ (X=a & cont-rel' (C, b)) &
                (X=/=c \/ (X=c & cont-rel' (C, d))
cont-rel' (f0(L), Y) <-mem(Y, L).

... clauses for mem, list, rel ...
```

(c) After translation

Here C is the continuation variable. After simplification of rel' clause, we have:

```
all-sol(X, L) <- search(X, L) & list(L)
search(X, L) <-rel' (X, f0(L))

rel' (X, C) <- X=/=a & X=/=d
rel' (a, C) <- cont-rel' (C, b)
rel' (c, C) <- cont-rel' (C, d)

cont-rel' (f0(L), Y) <-mem(Y, L)

... clauses for mem, list, rel ...
```

(d) The final output

---

as a final program. When considered as a Prolog program, and if a cut (!) is inserted like:

```
list((H|L)) <- ! & list(L)
```

in the second clause about "list", it can return different all-solutions:

```
?- all_sol(X, L)
        X=a, L=(b) ;
        X=c, L=(d)
```

2. 12. 2. Fibonacci series

The following program P0 is a first order program to compute a fibonacci series according the recurrence relation Fib(0)=0, Fib(1)=1, Fib(n+2)=Fib(n+1)+Fib(n).

```
fib(L) <-
        L=[0, 1|R] &
        all((Y, Z, W),
          exist((X, U), append(X, [Y, Z, W|U], L) -> W is Y+Z))

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

(a) Fibonacci source program P0

This program is intended to compute the fibonacci series 0, 1, 1, 2... by choosing adjacent elements in L by three (denoted by Y, Z, W) in turn moving from left-to-right through L. As the reader notices, the order of choice depends on the textual order of append clauses.

We suppose that "is" is a binary system predicate is(X, Y) which computes the value of the expression Y and unifies it with X. So the allowable modes are { is(+, +), is(-, +) }.

After moving existentially quantified variables to the front and applying the preprocessing algorithm, we have the basic program P:

```
fib(L) <- L=[0, 1|R] & fib1(L)
fib1(L) <-
     all((X, Y, Z, U, W), append(X, [Y, Z, W|U], L) -> W is Y+Z))

... clauses for append ...
```

The mode labelling algorithm labels the (only one) extended clause as follows:

```
fib1(L) <-                          append(-, -, +)
     all((X, Y, Z, U, W), append(X, [Y, Z, W|U], L) -> W is Y+Z))
```

then propagates it through the whole programs successfully. P is therefore a compilable program. The term labelling algorithm gives:

```
                                                append(-, -, +), f0
fib1(L) <- all((X, Y, Z, U, W), append(X, [Y, Z, W|U], L)
                                                    -> W is Y+Z))

 append(-, -, +)
append([], Y, Y)

 append(-, -, +)                    append(-, -, +), f1(H, C)
append([H|X], Y, [H|Z])  <- append(X, Y, Z)
```

(b) After term-mode labelling

The universal continuation form for append(-, -, +):

```
append'(Z, C)
        <->all((X, Y), append(X, Y, Z)->cont-append'(C, X, Y))
```

is introduced. Then, via the intermediate translation:

```
fib(L) <- L=[0, 1|R] & fib1(L)

fib1(L) <- append'(L, f0)

append'(X, C) <-
        { all(Y, X=/=Y) \/
                exist(Y, X=Y & cont-append'(C, [], Y)) } &
        { all((H, Z), X=/=[H|Z]) \/
                exist((H, Z), X=[H|Z] &
                                append'(Z, f1(H, C)) }

cont-append'(f0, Y', Z') <-
        all((X, Y, Z, W, U), (Y', Z')=/=(X, [Y, Z, W|U])) \/
        exist((X, Y, Z, W, U), (Y', Z')=(X, [Y, Z, W|U])) &
                                                W is Y+Z)

cont-append'(f1(H, C), Y', Z') <-
        all((X, Y), (Y', Z')=/=(X, Y)) \/
        exist((X, Y), (Y', Z')=(X, Y) &
                                cont-append'(C, [H|X], Y))
```

(c) After translation

we reach the following final output:

```
fib(L) <- L=[0, 1|R] & fib1(L)

fib1(L) <- append'(L, f0)

append'(Y, C) <-
        all((H, Z), Y=/=[H|Z]) &
        cont-append'(C, [], Y)

append'([H|Z], C) <-
        cont-append'(C, [], [H|Z]) &
        append'(Z, f1(H, C))

cont-append'(f0, Y', Z') <- all((Y, Z, W, U), Y' =/= [Y, Z, W|U])
cont-append'(f0, X, [Y, Z, W|U]) <- W is Y+Z
cont-append'(f1(H, C), X, Y) <- cont-append'(C, [H|X], Y)
```

(d) The final output

In a usual case, it is favourable to replace all((H, Z), Y=/=[H|Z]) in the first append' clause by Y=[], which will give append'([], C) <- cont-append'(C, [], Y). A top call such as <- fib([0, 1, F3, F4, F5]) calculates the fibonacci series up to the fifth element. Note that the compiled program executes append clauses faithfully according to the execution order taken by Prolog with backtracking.

2. 12. 3. Intuitionistic truth predicate

We show how the truth definition for a subset of intuitionistic propositional logic is compiled into an executable form. It is an example of a definition using universal quantifications. This sort of definition is sometimes found in a text book but takes time to write a correct program for it. Our compilation method enables us to test such definitions quickly without errors.

A class of propositional formulae we deal with consists of atoms (propositional letters) and "&" and "=>" (intuitionistic implication). For convenience, we assume that they are represented by a term of the form atom(X), (X & Y) or (X => Y).

Truth definition in intuitionistic logic can not be the same as that in a classical logic because proof theoretically speaking, for example, intuitionistic logic does not accept the law of excluded middle A\/~A as an axiom. There can be a couple of semantics suitable to such logic but the most intuitive one would be Kripke semantics defined in terms of possible worlds. So the truth definition we adopt is based on that semantics.

First imagine a situation where a finite set of possible worlds W1, W2,.. Wk are given with "seeing relation" can-see(W, W') among them. The relation can-see(W, W') means that we can "see" world W' from world W. We suppose that the seeing relation is reflexive and transitive and presented by a set of ground unit clauses (this is not obligatory).

For each world W, we have an assignment, a set of propositional letters which are true at W. A letter not included in the set considered as false at W. Accordingly, we put the truth definition for an atom into:

```
true(atom(X), W) <- assg(atom(X), W)
```

where "assg(atom(X), W)" means that an atom atom(X) is true at world W. We suppose that the assignment relation is given by a set of ground unit clauses. We further require that the assignment satisfies the condition that if an atom is true at W, then it is true at every world which can be seen from W.

In general, we use "true(F, W)" to denote that formula F is true at world W. The truth definition of a conjunction is simple. At world W, (X&Y) is true if X is true at W and so is Y at W.

```
true((X&Y), W) <- true(X, W)&true(Y, W).
```

But that of an implication formula is not so simple because to

decide the truth of an implicational formula, we need to refer to other worlds which can be seen from W. It is defined as:

```
true((X=>Y),W) <-
        all(W', can-see(W,W')&true(X,W') -> true(Y,W')).
```

It says that a formula (X=>Y) is true at W if for every world W' which can be seen from W, X being true at W' implies Y also being true at W'.

So, a source program for the truth predicate over a Kripke model with the seeing relation can-see(a,a'), can-see(b,b'), ... and with the assignment relation assg(atom(p),a), assg(atom(p), a'), assg(atom(q),b),... would look like:

```
true(atom(X),W) <- assg(X,W)
true((X & Y),W) <- true(X,W)&true(Y,W)
true((X =>Y),W) <- all(W', can-see(W,W')&true(X,W')
                                        ->true(Y,W'))

can-see(a,a'), can-see(b,b'), ...
assg(atom(p),a), assg(atom(p), a'), assg(atom(q),b),...
```

(a) Source program

Although the definition for each case is expressed using only "<-", this actually means "<->" since heads containing "true" predicate are ununifiable. By introducing a new predicate true1(W,X,W) <-> can-see(W,W') & true(X,W'), we transform P0 into the following basic program P:

```
true(atom(X),W) <- assg(atom(X),W)
true((X & Y),W) <- true(X,W)&true(Y,W)
true((X => Y),W) <- all(W', true1(W,X,W') -> true(Y,W'))
true1(W,X,W') <- can-see(W,W')&true(X,W')

can-see(a,a'), can-see(b,b'), ...
assg(atom(p),a), assg(atom(p), a'), assg(atom(q),b),...
```

(b) Basic program by the preprocessing algorithm

The mode labelling first labels the only one extended clause as:

```
true1(+,+,-)
```

```
true((X => Y), W) <- all(W', true1(W, X, W') -> true(Y, W'))
```

then propagates mode patterns successfully. The following term-labelling gives:

```
                                        true1(+, +, -), f0(Y)
true((X => Y), W) <- all(W', true1(W, X, W') -> true(Y, W'))

 true1(+, +, -)   can-see(+, -), f1(X, C)  true(+, +), f2(W', C)
true1(W, X, W') <-  can-see(W, W')   &   true(X, W')

   true(+, +)          true(+, +), f3(Y, W, C)   true(+, +), f4(C)
true((X & Y), W) <-         true(X, W)     &      true(Y, W)

   true(+, +)                                                 true(+, +), f5
true((X => Y), W) <- all(W', true1(W, X, W') -> true(Y, W'))

   true(+, +)          assg(+, +), f6(C)
true(atom(X), W) <- assg(atom(X), W)

can-see(+, -)     can-see(+, -)
can-see(a, a'), can-see(b, b'), ...

    assg(+, +)            assg(+, +)            assg(+, +)
assg(atom(p), a), assg(atom(p), a'), assg(atom(q), b), ...
```

where C is the continuation variable.

(c) After term-mode labelling

As a result, the universal continuation forms introduced becomes:

```
true1'(W, X, C) <-> all(W', true1(W, X, W')
                                  ->cont-true1'(C, W'))
 true'(X, W, C) <-> (true(X, W)->cont-true'(C))
can-see'(W, C) <-> all(W', can-see(W, W')
                                  ->cont-can-see'(C, W'))
 assg'(X, W, C) <-> (assg(X, W)->cont-assg'(C)).
```

The clauses other than the definite clauses in P which are generated by the translation algorithm are:

```
true((X=>Y),W) <- true1'(W,X,f0(Y))
cont-true1'(f0(Y),W') <- true(Y,W')

true1'(W,X,C) <-can-see'(W,f1(X,C))
can-see'(W,C) <-
        { W=/=a \/ (W=a & cont-can-see'(C,a')) } &
        { W=/=b \/ (W=a & cont-can-see'(C,b')) } &
        ...
cont-can-see'(f1(X,C),W') <- true'(X,W',f2(W',C))
cont-true'(f2(W',C)) <- cont-true1'(C,W')

true'((X&Y),W,C) <- true'(X,W,f3(Y,W,C))
cont-true'(f3(Y,W,C)) <- true'(Y,W,f4(C))
cont-true'(f4(C)) <-cont-true'(C)

true'((X=>Y),W,C) <-
        { true1(W,X,W')&true'(Y,W',f5) } \/ cont-true'(C)
cont-true'(f5) <-false

true'(atom(X),W,C) <- assg'(atom(X),W,f6(C))
cont-assg'(f6(C)) <- cont-true'(C).

assg'(atom(X),W,C) <-
        { (atom(X),W) =/= (atom(p),a)
                \/ ((atom(X),W) = (atom(p),a) &
                        cont-assg'(C)) } &
        { (atom(X),W) =/= (atom(p),a')
                \/ ((atom(X),W) = (atom(p),a') &
                        cont-assg'(C)) } &
        ...
```

(d) Generated clauses for
`true((X => Y),W) <- all(W', true1(W,X,W') -> true(Y,W'))`

As was expected, this program can check that a given formula F is true at a specific world and more, can tell in which world F becomes true. Unfortunately, this virtue is flawed by the relatively large space required by the compiled program.

2. 13. Optimization

In this section, we would like to show some optimization techniques. Although the first one is very minor, the other two is very significant in a pragmatic sense.

2. 13. 1. False goal

First it is very obvious that we can remove a clause of the form:

A <- false

from a compiled program without affecting theorems proved in the previous sections. Thus cont-true' (f5) <-false is removable from the compiled program for the intuitionistic truth predicate in the previous section.

2. 13. 2. Tail recursive optimization

The second technique is "tail recursive" optimization. It is quite similar to the one for a functional language. We can modify the translation algorithm so that it allows to generate fewer and simpler clauses. Look at step 12, step 13 and step 13' of { case 3 } in the translation algorithm. We modify these steps as follows:

12: Add the following formula to Body.

```
all(y0, x=/=s0) \/
        exist(y0, x=s0 & p1' (s1, f1(z1, C)))

where y0 =sorted(Fvar(s0))
and if p1(s1, , t1) is the right most goal,
    m0=m1, t0=t1 as sequence, t1 is a sequence of
    different variables and z1={},
    then, REPLACE f1(z1, C) by C;
```

13: If k>=2, then
Add the following clause set to Cont' with
each body separated.

```
{ cont-pi' (fi(zi, C), yi)<-
        all(vi, yi=/=ti)\/
                (yi=ti & pi+1' (si+1, fi+1(zi+1, C))        |

        for i (1=<i=<k-1),
        yi is a sequence of |ti| new variables,
        vi = sorted(Fvar(ti, si+1, ti+1, . . , sk, tk, t0)
                        \Fvar(s0, s1, t1, . . , si-1, ti-1, si))
        and if |ti|=0, replace the body by
            pi+1' (si+1, fi+1(ti+1, C))
        and if m0=mk, t0=tk as sequence,
            tk is a sequence of different variables
            and zk={},
            then REPLACE fk(zk, C) by C }+
```

13' : Add the following clause set to Cont'
with each body separated.

```
{ cont-pk' (fk(zk, C), yk)<-
        all(vk, yk=/=tk)\/(yk=tk & cont-p0' (C, t0))
  where yk is a sequence of |tk| new variables,
        vk = sorted(Fvar(tk, t0)\
                        Fvar(s0, s1, t1, . . , sk-1, tk-1, sk))
        and if |tk|=0, replace the body by cont-p0' (C, t0)
        and if m0=mk, t0=tk as sequence, tk is a sequence
            of different variables and zk={},
            then NOT ADD this formula to Cont' }
```

Fig. 2.13.1. Modification to the translation algorithm

This modification does not affect the soundness theorems in section 2.10. at all (a proof omitted). It reflects the fact that there is a case where we can make a short cut to obtain closure clauses. For simplicity, consider the following clause:

```
        m0        m1, f1(z1, C)
p(s0, , t0) <- p(s1, , t1)                                    ... (1)
```

where m0=m1, t0=t1 as sequence, t1 is a sequence of different variables and z1={}, is included in a compilable program P. The corresponding universal continuation form for m0 becomes:

```
                                m0
p'(x, C) <-> all(y, p(x, , y)->cont-p'(C, y)).                ... (2)
```

When deriving clauses about p', the contribution by clause (1) can be represented as:

```
p'(x, C) <->
..& all(v0, x=s0 ->
        all(y1, p1(s1, , y1)->
                all(v1, y1=t1 ->cont-p'(C, t0))))&...
                                                              ... (3)
```

where v0= Fvar(s0), v1=Fvar(t1, t0)\Fvar(s0, s1), y1 is a sequence of |t1| new variables. At this stage, in an ordinary case, we fold, as a preliminary folding, all(v1, y1=t1 -> cont-p'(C, t0)) into cont-p1'(f1(z1, C), y1) where z1=Fvar(t1, t0)/\Fvar(s0, s1), assuming:

```
cont-p1'(f1(z1, C), y1) <->all(v1, y1=t1 ->cont-p'(C, t0))
                                                              ... (4)
```

is included in Aux for P. Then we have:

```
p'(x, C) <->
..& all(v0, x=s0 ->
      all(y1, p1(s1, , y1)->cont-p1'(f1(z1, C), y1))) &...
```

as a result. And this is further folded by (2) into:

```
p'(x, C) <->..& all(v0, x=s0 -> p1'(s1, f1(z1, C))) &...
                                                              ... (5)
```

On the other hand, if a special conditions below:

m0=m1, t0=t1 as sequence, t1 is a sequence of different

variables and z1={}

holds for clause (1), we need not the preliminary folding of (3) by (4). First, in (3), we can put p1=p and y1=t0=t1 thanks to the conditions, giving:

```
p' (x, C) <->
..& all(v0, x=s0 ->
        all(y1, p(s1,, y1)-> cont-p' (C, y1))) &..    ... (3')
```

Then using (2) directly, this is folded into:

```
p' (x, C) <-> ...& all(v0, x=s0 -> p' (s1, C)) &...    ... (5')
```

without appealing to the preliminary folding by (4). Compare (5) with (5'). The continuation term f1(z1, C) in (5) is relaced by C in (5'). Moreover, as (5') is obtained without (4), we do not need generate continuation clauses from (4).

An optimized compiler applied to the truth predicate example in section 2.12. would not generate:

```
cont-true' (f4(C)) <- cont-true' (C)
```

and would generate:

```
cont-true' (f3(Y, W, C)) <- true' (Y, W, C)
```

instead of:

```
cont-true' (f3(Y, W, C)) <- true' (Y, W, f4(C)).
```

2. 13. 3. Transformation

When compilation finished, the result is not always satisfactory from the algorithmic point of view. In this example, we show how optimization by transformation becomes possible. Since a systematic description of transformation system will be given at large in the next chapter together with examples, we only show the outline. A program chosen for this purpose is a maximum program (see section 3. 1. for terminologies).

```
max(M, L) <- mem(M, L) & all(Y, mem(Y, L) -> Y=<M)

mem(Y, [Y|Z])
mem(Y, [U|V]) <-mem(Y, V)
```

Fig. 2. 13. 2. A source program

We use the optimized compiler for compilation. After reducing all((Y, Z), L=/=[Y|Z]) to L=[], we obtain:

```
max(M, L) <-mem(M, L) & mem'(L, f0(M))                    ... (1)
mem'([], C)                                               ... (2)
mem'([U|V], C) <-cont-mem'(C, U) & mem'(V, C)             ... (3)
cont-mem'(f0(M), U) <- U=<M                               ... (4)

mem(Y, [Y|Z])                                             ... (5)
mem(Y, [U|V]) <-mem(Y, V)                                 ... (6)
```

Fig. 2. 13. 3. The compiled program

We then appeal to the equivalence preserving transformation system for definite clause programs [64, 65] to eliminate the continuation function "f0" from the compiled program. First we define leqall(L, M) by:

```
leqall(L, M) <- mem'(L, f0(M))                            ... (7)
```

and unfold it at the goal mem'(L, f0(M)). We obtain:

```
leqall([], M)                                             ... (8)
leqall([U|V], M) <-
        cont-mem'(f0(M), U) & mem'(V, f0(M))              ... (9)
```

Then by unfolding clause (8) at cont-mem'(f0(M), U), we have:

```
leqall([U|V], M) <- U=<M & mem'(V, f0(M))                 ... (10)
```

We fold the goal mem' (V, f0(M)) of clause (10) into leqall(V, M) using (7). It gives:

```
leqall((U|V), M) <- U=<M & leqall(V, M)                    ... (11)
```

Finally, we fold the goal mem' (L, f0(M)) of (1) into leqall(L, M):

```
max(M, L) <-mem(M, L) & leqall(L, M).                      ... (12)
```

Thus we have successfully eliminated the continuation function "f0" from the compiled program while keeping program equivalence. The resulting program for max is:

```
max(M, L) <- mem(M, L) & leqall(L, M)                      .. (12)

leqall([], M)                                              ... (8)
leqall((U|V), M) <- U=<M & leqall(V, M)                    .. (11)

mem(Y, [Y|Z])                                              ... (5)
mem(Y, [U|V]) <-mem(Y, V)                                  ... (6)
```

Fig. 2.13.4. After transformation: no continuation function

This is not satisfactory since it computes the maximum element M in list L in quadradic order with respect ro |L|. From this form however, it is not difficult by transformation to obtain max(M, L) program which can compute M in proportion to |L|.

3. Transformational Logic Programming

We have so far been describing how to compile first order programs into definite clause porgrams. In this chapter, we describe a transformation system for definite clause programs consisting of unfold/fold transformation rules and other minor rules. The system was first designed with a view to optimizing programs but later it turned out usable as a preprocessing tool for negation technique which provides another path from first order programs (specifications) to definite clause programs. Since program transformation may not be familiar, we would like first to mention the background briefly. Program transformation, if interpreted as general term, is not new. It can mean what an Assembler does to macro expansion requirement, or what a Fortran compiler does to a Fortran program. Recent tendency however, seems to use it in a slightly restricted way reflecting the growing interest in "transformational programming," one of the programming pardigms for constructing error free but efficient programs, hence contribution to solving software crisis (16).

Transformational programming is possible in principle in any programming language. It roughly means a programming style starting from a clear program whose correctness is self-evident, then using so called source-to-source program transformation, transforming it into another but efficient program to obtain a correct and efficient program. A closest example which may fit into this category is loop optimization performed by a compiler such as constant code removal. Tail recursive optimization can be an example as well. But in order for the concept of transformation to be a real breakthrough, it can not remain at the level of traditional opimization techniques. It must be embodied as a total system based on the understanding of nature of programming with firm theoretical foundations.

As developing such a system is apparently a long term project, we have to set up the initial condition very carefully. If we start with a syntactically complex language, essential parts may be lost in manupulating syntax. If we start with a semantically complex language, it will be hard for transformation to preserve the correcness of a program. We therefore have chosen definite clause programs as starting point in view of syntactic and semantic simplicity. Then how about formalism for our transformation system. We fortunately have two candidates which are theoretically interesting and practically meaningful. One is partial evaluation. The other is unfold/fold transformation. Both are applicable to wide range of programs as means of program optimization whether they are declarative or not. We however must say in advance that despite the difference in their origins, later development seems to have blurred the distinction between

the two (14, 15, 29, 31, 33, 61, 64, 65).

We explain partial evaluation in a narrow sense first. Before going into the details, as we will manupulate programs as syntactic objects, we need a couple of stipulations to prevent possible confusion. By "a program e" we mean the text of a program which consists of e, a sequence of characters or something equivalent. For convenience we use e(x, y) to mean that e has x and y as arguments and we let them denote all of their occurrences in the expression e. Then, we adopt a notation {e} (x, y) to denote the value returned by the program e for inputs x and y. In other words, e denotes a syntactic object whereas {e} denotes the function e computes. It can naturally happen that two programs e and e' compute the same function, namely {e} (x, y) = {e'} (x, y) holds. The equality must be interpreted as the left hand side has a value iff so does the right hand side and they have the same value. We use "==" for syntactic identity.

Then imagine a situation where a program e(x, y) is given and it is known that the value of x is always a, whenever it is invoked. Then {e} (a, y) is a function of y and e' (y), the program e with x being fixed to a, equal to e(a, y) in notation, computes it. So {e'} (y) = {e} (a, y) holds. There is, however, a hope of getting another program e'' such that {e''} (y) = {e} (a, y), whose performance is better than e'.

Partial evaluation means program transformation that purports to obtain, from the original program e and known data a, an efficient program e'' such that {e''} (y) = {e} (a, y) holds. Take e(x, y) ==x*y and put a=1 for instance. Although {e} (1, y) is computable using a program e' (y) (==e(1, y) ==1*y), by multiplying 1 and y, it is evident that the identity program e'' (y) ==y computes {e} (1, y) more efficiently than 1*y.

Partial evaluation, or mixed computation (14), has been known in the name of constant propagation say, as one of the practical optimization techniques. We would like not to use it in this narrow sense though, because it gives us a far reaching insight beyond technical matters into the nature of programming, as we describe in the following (15). To start with, we have to note that the origin of partial evaluation probablely can be traced to recursion theory, especially to S-m-n theorem (32). In recursion theory, the domain of discourse is natural numbers and there is a theorem stating that any (general) recursive function f(x, y) is represented as {e} (x, y) where e is a program text, actually a natural number called index caluculated through Godel numbering, of a Turing machine that computes f. There also is another theorem, S-m-n theorem, which states in this case that there is a total recursive function s1(u, v) satisfying:

$$\{e\}(x, y) = \{s1(e, x)\}(y). \qquad \ldots (1)$$

When x is fixed to a, we have:

$$\{e\}(a, y) = \{s1(e, a)\}(y) = \{e'\}(y) \quad \text{where } e' = s1(e, a)$$

This equation says that there is a fixed function s1 which, supplied e and a, yields another program e', a specialization of e with respect to the first argument x to the value a. Therefore, s1 can be called partial evaluator. Since it is a recursive function, it has an index k such that:

$$s1(u, v) = \{k\}(u, v) \quad \ldots (2)$$

Now consider a (partial) recursive function $\{x\}(y)$ and suppose it has an index i:

$$\{x\}(y) = \{i\}(x, y) \quad \ldots (3)$$

With x being fixed, $\{x\}(y)$ denotes a function of y computed by x. Then, $\{i\}(x, y)$ means that the program i, taking a program x and an input data y as arguments, comuputes the value $\{x\}(y)$, the result of executing the program x for the input y. Therefore, $\{i\}$ works as an interpreter. For this reason, we can regard i as an interpreter program. Combining (1) (2) and (3), we have:

$$\begin{aligned}\{x\}(y) &= \{i\}(x, y)\\ &= \{s1(i, x)\}(y)\\ &= \{\{k\}(i, x)\}(y)\\ &= \{\{s1(k, i)\}(x)\}(y)\\ &= \{\{c\}(x)\}(y) \quad \text{where } c = s1(k, i)\end{aligned}$$

reading that a program $\{c\}(x)$ denotes the same function as $\{x\}$ and it is automatically computed from x by the program c. So $\{c\}$ functions as compiler and $\{c\}(x)$ is interpreted as compiled codes for program x. And what is more, c, a compiler program, is obtained by partially evaluating the partial evaluator program k with respect to the interpreter i. On the other hand, combining $c = s1(k, i)$ and (2), we reach:

$$\begin{aligned}c &= s1(k, i)\\ &= \{k\}(k, i)\\ &= \{s1(k, k)\}(i)\\ &= \{cc\}(i) \quad \text{where } cc = s1(k, k) \quad \ldots (4)\end{aligned}$$

This can be read that when cc, the result of partial evaluation of a partial evaluator program k with respect to itself (using itself because $cc = s1(k, k) = \{k\}(k, k)$), is available, the compiler c is automatically obtained from the interpreter i as $\{cc\}(i)$. So cc behaves as compiler-compiler [28]. Thus, once we have constructed a partial evaluator, it can be used to automatically obtain a correct compiler from an iterpreter and a correct complier-compiler from the partial

evaluator itself as well as to automatically get a specialized program. And if, this is a big if, we have succeeded in constructing such good partial evaluator that a partially evaluated program runs much more efficiently than the original one, a program with some imputs being fixed to known values, not only will we have a correct compiler and a correct compiler-compiler but we will have usable ones automatically.

It is obvious that this story reamins valid when we move to any other programming language capable of dealing with programs as data. However, we have to remember that whether it becomes someday a real story or not entirely depends on the performance of a partial evaluator. In Lisp, a program s1(f, y) = lambda((z), apply(f, (y, z))) may work as a partial evaluator for specialzing a two argument function with respect to the first argument, but alas, as s1(f, a), the specialzation of y to a, returns just lambda((z), apply(f, (a, z))). It is meaningless to use such partial evaluator for program optimization. At present, we do not seem to have a partial evaluator usable enough to automatically derive a usable compiler or a usable compiler-complier (28, 29). It should not be taken negatively however. It should be taken instead positively as the indication of the need for intensive research for a good (language and) partial evaluator.

The development of a partial evaluator has considerable significance as exemplified above. It is possible however to generalize the notion of partial evaluation because it is only a special case of program optimization when some information is available about the circumstance in which programs are used. Take h(x, y) = f(x, g(y)) where f(x, y) = x+y for example. This program can not be optimized when we know nothing about g. So h(x, x) must be computed using f and g. But once the program for g(y) is known to be -y, then h(x, x) = f(x, g(x)) = x+(-x) holds and we can transform the program x+(-x) to 0, giving an optimized program h'(x, x) = 0 instead of x+(-x). More generally, when functions or procedures are combined, there can be a redundancy, hence the room for optimization. Partial evaluation in a broad sense hereafter includes program transformation which makes use of information about such program use as well as about data.

Now we have to mention unfold/fold transformation in relation to this generalized sense. Unfolding is one step symbolic execution of a program. Putting it differently, it is a replacement of a procedure call by the procedure body, or that of a definiendum by the definiens. Folding is the reverse operation, a replacement of a procedure body by the procedure call or that of a definiens by the definiendum. It may not be clear how this can be used to optimize a program. So we give a simple example in a simple setting.

Suppose we have a pure functional language with if-then-else construct, recursion etc and append2(x, y) program for appending list y to the tail of list x whose definition is:

```
append2 (x, y)
      = if x=nil then y
        else cons (car (x) , append2 (cdr (x) , y) )          ... (1)
```

where nil is the empty list, car(x) gives the first element of x and cdr(x) gives the remaining list. Using the knowledge such as cons(x, y) =/=nil, car(cons(x, y)) =x and cdr(cons(x, y)) =y, we have:

```
append2 (cons (x, y) , z)
    = if cons (x, y) =nil then z
      else cons (car (cons (x, y) ) , appends (cdr (cons (x, y) ) , z) )
    = if false then z
      else cons (x, append2 (y, z) )
    = cons (x,  append2 (y, z) )                              ... (2)
```

by succeesive symbolic execution (unfolding). Then suppose we write appned3(x, y, z) as follows to append x, y and z:

```
append3 (x, y, z) =append2 (append2 (x, y) , z)               ... (3)
```

This program needs 2*|x|+|y| cons operations (|x| is the length of list x). So we get another program that only requires |x|+|y| cons operations. First, we replace a function call append2(x, y) in append2(append2(x, y), z) by its definition (unfolding), resulting:

```
append3 (x, y, z) =
        append2 (if x=nil then y
                 else cons (car (x) , append2 (cdr (x) , y) ) , z)
```

Since f(if p then q else r, z) =if p then f(p, z) else f(r, z) holds in general, we have:

```
append3 (x, y, z) =
        if x=nil then append2 (y, z)
        else append2 (cons (car (x) , append2 (cdr (x) , y) ) , z) .
```

Then from (2), we get:

```
append3 (x, y, z) =
        if x=nil then append2 (y, z)
        else cons (car (x) , append2 (append2 (cdr (x) , y) , z) )
```

Noting that append2(append2(cdr(x), y), z)) is equal to append3(cdr(x), y, z) by definition (1), we replace the former by the latter (folding), giving:

```
append3 (x, y, z) =
        if x=nil then append2 (y, z)
        else append3 (cdr (x) , y, z) .                    ... (4)
```

This program together with (1) appends 3 lists x, y and z with |x|+|y| cons operations, much more efficient than the original program (3).

An unfold/fold transformation for functional programming language such as one used here was first proposed by Burstall and Darlington (5, 11) with many optimizing examples. What they showed was that given a program P, unfold/fold transformation could produce a better program P'. It is evident that their unfold/fold transformation can be simulated as deduction in the logic of equality so that the transformed program P' is always partially correct, meaning that if P' terminates, the returned value is the same as P, the original program, does. But there remains a question of termination because unconditional use of folding may destroy termination. Take append3 example again. Starting from (3):

```
append3 (x, y, z) =append2 (append2 (x, y) , z)            ... (3)
```

and by immediately folding the right hand side using (3), we reach:

```
append3 (x, y, z) =append3 (x, y, z)
```

which never terminiates for any input. So it is clear that we need some conditions on the application of transformation rules if we want the system to preserve correctness. This problem was solved by Kott (34) and later by Scherlis (58) more precisely though in a slightly different formulation.

Unfold/fold transformation in the deductive framework was first attempted by Clark and Sickel (7) and subsequently developed by Hogger (24) and others. They have convincingly demonstarated that it is possible to deduce definite clause programs from specifications written in first order predicate calculus as well as to optimize logic programs by unofld/fold transformation.

Since transformation is considered as logical deduction in their frame work, replacement of a formula A by another formula B when A<->B has been proved, the partial correctness of a transformed program is self-evident, simply because every result returned by the program is a logical consequence of the initial formula. Thus deductive unfold/fold transformation guarantees partial correcness for free. This is a valuable property, but regrettably, not a satisfactory answer to the problem of correctness.

Suppose a definite clause program P is transformed

deductively to another program P'. M(P), the set of all ground (variable free) atoms provable from P, has been considered as the standard meaning of P because it gives the least model of P over the Herbrand universe generated from the symbols in which P is written, thereby characterizing P most from the user's point of view. So we say definite clause program P1 and P2 are equivalent iff M(P1)=M(P2). Then what partial correctness assures is only M(P')=<M(P), not vice versa. Accordingly, we can not escape from the danger that P' may fail to provide an answer which P can. We therefore have to check their equivalence every time transformation has finished.

We have developed a program transformation system for logic programs which is the reformulation of unfold/fold transformation system initiated by Burstall and Darlington (20). Later we have augmented it with other rules (21). This system is not deductive in the sense that transformation rules are not considered as inference rules of special type adapted to transformational setting but they are formulated as solely syntactive manupulation. The point is that it is proved to preserve the least models of logic programs as long as we observe certain conditions associated to each tranformation rule. We think it is the most distinctive feature of our system.

By virtue of this property for example, we are free from the need for case-by-case verification of program equivalence, which as we mentioned above has been indispensable to deductive formulation of unfold/fold transformation (8, 10, 11). It also allows us to prove theorems such as the associativity of addition without induction, when they are presented as a problem of proving the equivalence of two programs.

Many examples we attempted have shown that unfold/fold transformation is very effective for logic program optimization as it was for functional program optimization. We also have found that it often greatly reduces non-determinacy of programs. This is paticularly interesting, because while nondeterminism helps us write succinct programs, it has not yet widely spread since people in software engineering often doubt the performance of nondeterministic programs. But the performance can be improved later, in logic programming, by unfold/fold transformation without the fear of destroying program equivalence. It follows therefore that logic programming offers good scaffolding to transformational programming.

In addition to unfold/fold transformation, there is another type of transformation, probably peculiar to logic programming, called "negation technique" (57). It is an equivalent of taking the complement of a set. Given a program S, to be precise, it automatically generates another program S' which computes the relations complementary to those computed by S (under certain

conditions). Using negation technique for example, we can automatically get a program specifying odd numbers, once we write a program specifying even numbmers. Since neagation (-as-failure) implemented in Prolog is something illogical as we pointed out somewhere else, the technique can be considered as new logical power added to Prolog. Moreover, as will be shown in the following, it gives yet another way of synthesizing definite clause programs from first order specifications.

In the subsequent sections, we describe our transformation system with brief introduction and with a representative example which suggests the feasiblity of transformational logic programming. Then negation technique follows. Finally we show how it is used to derive logic programs from first order specifications.

3. 1. Equivalence Preserving Transformation System

3. 1. 1. What is transformation like?

We illustrate a small example.

```
append([], Y, Y)                                        ... (1)
append([H|X], Y, [H|Z]) <- append(X, Y, Z)              ... (2)

initial(X, Z) <- append(X, Y, Z)                        ... (3)
```

Fig. 3. 1. 1. "initial" program

A set of definite clauses {(1), (2)} is a logic program for the relation append(X, Y, Z) defined as appending list Y to list X yields Z. Using "append" relation, "initial" relation is defined by the clause (3). initial(X, Z) reads that X is the initial segment of list Z. Note the occurrence of an internal variable Y (an internal variable of clause C is one that appears in the body and not in the head of C). We call S0={(1), (2)} base program, D1={(3)} definition set and S1={(1), (2), (3)} initial program. S1 is transformed by transformation rules as follows.

```
initial(X, Z) <- >>append(X, Y, Z)<<                    ... (3)
   initial([], Z).                                      ... (4)
   initial([H|X], [H|Z]) <- <<append(X, Y, Z)>>         ... (5)
     initial([H|X], [H|Z]) <- initial(X, Z)             ... (6)
```

Fig. 3. 1. 2. Transformation process

The transformation process begins by unfolding the goal append(X, Y, Z) of (3). Unfolding means goal expansion by input clauses. we surround an unfolded goal by >> and << as shown in Fig. 3. 1. 2. This unfolding yields two clauses (5) and (6) as illustrated by indentation in the figure (if clauses C1, . . , Ck are obtained from a clause C0 by transformation, C1, . . , Ck are always indented from C1). We have a new program S2 = {(1), (2), (5), (6)}. Then we note that the body append(X, Y, Z) of (5) is an instance of that of the definition clause (3). So we replace append(X, Y, Z) by the corresponding clause head of (3) suitably instantiated. This is called folding and we say goal append(X, Y, Z) of (5) was folded into initial(X, Z). Folded goals are surrounded by << and >> as shown in Fig. 3. 1. 2. Folding is, so to speak, replacement of a procedure body by the corresponding procedure call. Now we have a new program S3={(1), (2), (4), (6)}.

Since (6) is obtained from (5), it is also relatively indented in the figure. At this stage, we have obtained a self recursive "initial" program that does not depend on "append" program any more.

```
initial([], Z)                                          ... (4)
initial([H|X], [H|Z]) <- initial(X, Z)                  ... (6)
```

Fig. 3. 1. 3. Output

Two facts must be mentioned. First it is assured that S1 and S3 are equivalent in such sense that they compute the same append and initial relations, hence equivalent. Second a new program {(4), (6)} for "initial" includes no internal variables. Internal variable elimination by transformation is used as a preprocess for application of negation technique.

3. 1. 2. Transformation System

We need some definitions to describe our transformation system precisely. Some of them are not new but repeated here for the sake of the reader's convenience.

(Program syntax):

A logic program is a set of definite clauses in a sorted first order language. A definite clause is a formula of the form A0<-A1&..&Am(m>=0) where Ai(0=<i=<m) is an atom(an atomic formula). We assume variables are universally quantified. A0 is called head, A1&..&Am body and each Ai(1=<i=<m) goal respectively. We assume symbols other than variables are finite in the language.

(Program semantics):

We adopt least model semantics for logic programs as usual (1). Let success(S) be the set of ground atoms provable from a program S. Then success(S) determines an interpretation M(S) by:

M(S) |= A iff A is in success(S).

M(S) thus defined becomes the least model of S over the Herbrand universe. Since M(S) and success(S) have one-to-one correspondence, we use them interchangeably in the sequel. In the light of the least model semantics, we define programs S1 and S2 are equivalent iff M(S1)=M(S2).

(Definition set D with respect to a base program S):

A base program S is one that is used to supply basic relations like "append" which are building blocks for new relations like "initial." So a predicate that appears in S is said to be old. A definition set D wtih respect to S is a set of definite clause A0<-A1&..&Am (m>0) where the predicate symbol contained in A0 does not appear anywhere other than A0 itself and the predicate of Ai is old for every i(1=<i=<m). D is used to define new relations. Predicates symbols appearing in clause heads in D are called new predicates.

(Initialization):

Given a base program S and a definition set D, initialization for transformation is done as follows:

(a) Mark every clause in S "foldable."

(b) Put S1=S+D. S1 is called an initial program for transformation.

(c) To each goal appearing in S1, assign 1 as group identifier and {} as the set of identifiers of ancestor goals. In the following, id(G) and ances(G) denote the identifier of G, and the set of identifiers of ancestors of G respectively. We call a pair < id(G), ances(G) > the state of goal G. A set of goals is a foldable goal set (f.g.s.) iff for every two goals in the set, say G1 and G2, either id(G1)=id(G2) holds or ances(G1) and ances(G2) have no intersection. If we do not use "goal addition rule" in the transformation process, there is no need for bookkeeping id(G) and ances(G). Moreover, any goal set becomes a foldable goal set.

(d) Put D1=D. D1 is called the initial definition set for transformation.
Make a pair <S1, D1>. A pair of program and definition set is called a configuration.

(Rank) :

For every ground atom A provable from the initial program S1, let smallest-size(A) be the size of the smallest proof tree of A in S1 and define rank(A) by:

if A contains an old predicate
then rank(A) = samllest-size(A)
else rank(A) = samllest-size(A) -1

Further, define rank for And-goals by:

rank(exist(y, B1(a, y)&...&Bm(a, y)) =
the minimum of rank(B1(a, b))+...+rank(Bm(a, b))
where b runs over ground terms vectors of
length |y|

Let S0, S1, D1 be a base program, an initial program and an initial definition set respectively. Suppose that, starting from <S1, D1>, we have obtained a sequence of configurations <S1, D1>, ..., <Si, Di>. Next configuration <Si+1, Di+1> is obtained by applying one of the following transformation rules to <Si, Di> nondeterministically.

---------- Transformation Rules ----------

(1) Introduction:

Let C be a clause p(t1, ..., tk) <-A1&...&An (n>=0) such that p is a new predicate not used so far and every Aj (1=<j=<n) contains an old predicate. Then, Do not mark C "foldable." Generate a new group identifier N. Set id(Aj) = N, ances(Aj) = {} for every Aj (1=<j=<n) of C. Calculate rank(p(t1, ..., tk)) for later use where t1, ..., tk are all ground. Set Si+1=Si+{C} and Di+1=Di+{C}.

(2) Unfolding:

Select a clause C (=A0<-A1&...&An (n>0)) from Si and such goal Aj (1=<j=<n) that Aj contains an old predicate. Then, Select Sh (1=<h=<i). Resolve Aj against all clauses in Sh. Let the resulting clauses be {C1,..,Ck,..}. Mark every clause in {C1,..,Ck,..} "foldable." Set id(G)=id(Aj) and ances(G)=ances(Aj) for every goal G of a clause in {C1,..,Ck,..} introduced by the resolution. Set Si+1= (Si\{C})+{C1,..,Ck,..} and Di+1=Di.

(3) Folding:

Select a clause C (=A0<-A1&...&An (n>0)) from Si, its f.g.s. (foldable goal set) {B1,..,Bm}, and a clause C'=P<-G1,...&Gm in the definition set Di such that:

< Folding condition >
(a) m<n or C is marked "foldable."
(b) {B1,...,Bm}={G1,...,Gm}T for some substitution T.
(c) T substitutes terms only for variables in {G1,...,Gm}.
(d) Distinct internal variables of C' appearing in {G1,...,Gm} are substituted by distinct internal variables Vs of C appearing in {B1,...,Bm}.
(e) In addition, Vs must not appear anywhere else other than in {B1,...,Bm}.

Put {B1',...,Bl'} = {A1,...,An}\{B1,...,Bm}, C'' =(A0 <- P & B1' &..& Bl')T. Mark C'' "foldable" only when C is marked "foldable." Generate a new group identifier N. Set id(PT) = N, ances(PT) = ances(B1)+..+ances(Bm). For every goal other than PT, inherit its state from C. Set Si+1=(Si\{C})+{C''}, Di+1=Di.

(4) Goal Replacement:

Select a clause C= A0<-A1&..&An in Si and a f.g.s. of {B1,..,Bm} of C. Suppose there is a formula exist(y,B1'&...&Bk') and

< Goal replacement condition >
(a) C is marked "foldable."
(b) M(S1)|= exist(y, B1'&...&Bk')<->exist(x, B1&...&Bm)
(c) rank(exist(y, B1'&...&Bk'))=< rank(exist(x, B1&...&Bm))
where x is a vector of internal variables of C which appear nowhere except in {B1,...,Bm}

is satisfied. Let {B1'',..,Bb''}={A1,..,An}\{B1,..,Bm}. Put C'= A0<-B1'&..&Bk'&B1''&..&Bl''. Mark C' "foldable." Generate a new group identifier N. Set id(Bj')=N, ances(Bj') = ances(B1)+..+ances(Bm) for every Bj' (1=<j=<n) in C'. For every goal in {B1'',..,Bl''}, inherit its state from C. Set Si+1=(Si\{C})+{C'}, Di+1=Di.

(5) Goal Addition:

Select a clause C= A0<-A1,..&An from Si and a f.g.s. {B1,...,Bm} of C. Suppose there is a formula exist(y, B1'&...&Bk'). Suppose also that:

< Goal addition condition >
(a) C is marked "foldable."
(b) M(S1) |= exist(y, B1'&...&Bk') <-exist(x, B1&...&Bm)
(c) rank(exist(y, B1'&...&Bk')) =< rank(exist(x, B1&...&Bm))
where x is a vector of internal variables of C which appear nowhere except in {B1,...,Bm}.

is satisfied. Let {B1'',...,Bl''} = {A1,..,An}\{B1,...,Bm} and C' = A0 <- B1&..&Bm&B1'&..&Bk'&B1''&..&Bl''. Mark C' "foldable." Generate a new group identifier N. Set id(Bj')=N, ances(Bj') = {N}+ances(B1)+..+ances(Bn) for every Bj' (1=<j=<k) in C'. Add add {N} to ances(Bj) for every Bj (1=<j=<m) in C'. For every goal in {B1'',...,Bl''}, inherit its state from C. Set Si+1=(Si\{C})+{C'}, Di+1=Di.

(6) Goal Deletion:

Select a clause C from Si. Suppose there are such goal sets {B1,..,Bm} and {B1',...,Bk'} of C that

< Goal deletion condition >
(a) {B1,..,Bm} and {B1',...,Bk'} are disjoint.
(b) M(S1) |= exist(x, B1'&..&Bk') <-B1&..&Bm
where x is a vector of internal variables appearing only in {B1',...,Bk'}.

Delete {B1',...,Bk'} from C. Let the resulting clause be C'. Mark C' "foldable." For every goal other than B1',...,Bk', inherit its state from C. Set Si+1=(Si\{C})+{C'}, Di+1=Di.

(7) Function Merge:

Select a clause C=A0<-A1&A2&..&An in Si and such different two goals A1(x, y) and A2(x, z) of C that:

< Functionality condition >
(a) M(S1) |= all((x, y, z), A1(x, y)&A2(x, z) -> y=z)

Delete A2 from C and substitute y for z in C. Let the resulting clause be C'. Mark C' "foldable." For every goal other than A2, inherit its state from C. Set Di+1=Di, Si+1=(SiC})+{C'}.

(8) Case Split:

Select a clause C=A0<-A1&..&An in Si and a pair of atomic formulas B1, B2 such that

< Case split condition >
(a) C is marked "foldable"
(b) M(S1) |= all(x, B1\/B2)) & all(x, ~(B1&B2))
    where x is a vector of variables appearing in {B1, B2}
(c) No proof trees of B1(a) and B2(a) do not use C
    where a is a vector of ground terms of length |x|

is satisfied. Let C1=A0<-B1&A1&..&An and C2=A0<-B2&A1&..&An. Mark C1, C2 "foldable." Generate a new group identifier N. Set id(B1)= id(B2) =N, ances(B1) = ances(B2) = {N}+ances(A1)+ Suppose B1 contains predicate "p1", B2 "p2." From now on, no goal containing "p1" or "p2" is allowed to be resolved upon. No transformation rules are allowed to be applied to clauses on which "p1" or "p2" may depend either. Set Si+1=(Si\{C})+{C1, C2}, Di+1=Di.

---------- End of Transformation Rules ----------

Laws (5) like the associativity of addition is subsumed by the "goal replacement" rule. It subsumes goal merge, (A & A) ==> A as well. Our equivalence preservation theorem is stated as follows (64):

(Theorem 3.1.1)
Suppose, given a base program S, an initial definition set D, we have constructed a sequence of configurations <S1, D1>, ..., <Sk, Dk> for some k where S1=S+D, D1=D. Then M(Si)=M(S+ Di) for every i(1=<i=<k). Especially, if no introduction rule is used in the sequence, we have M(Si)=M(S+D) for every i(1=<i=<k).

Rules might seem complicated due to conditions posed on rule applications. This is because whatever interactions unfolding/folding rules and other ones may occur, we want to keep program equivalence.

Unfolding/folding rules are the same as those in functional languages except in that they are adapted to logic programs. But in case of transformation where only unfolding/folding rules are used, conditions for preserving the least model of programs are simple and possible to check mechanically unlike the functional counter parts (34, 58).

Rules other than unfolding/folding ones are not new either (36). They are effective to obtain efficient programs. But it is easy to show that their unconditional use sometimes destroys program equivalence. What we have done is to give rigorous conditions for preserving program equivalence.

3. 1. 3. An optimization example

We would like to illustrate how the goal addition rule interacts with the unfolding/folding rules to drastically reduce search spaces of logic programs. As an example, we take "insertion sort" program given by:

```
perm([], [])                                          ... (1)
perm([A|X], Y) <- perm(X, Z) &del(Y, A, Z)            ... (2)
del([H|X], H, X)                                      ... (3)
del([H|X], Y, [H|Z]) <- del(X, Y, Z)                  ... (4)
ord([])                                               ... (5)
ord([A])                                              ... (6)
ord([A, B|X]) <- (A=<B) &ord([B|X])                   ... (7)

isort(X, Y) <-perm(X, Y) &ord(Y)                      ... (8)
```

Fig. 3. 1. 4. The initial program for "insertion sort"

A base program is {(1)..... (7)}. It defines three basic relations, namely, perm(X, Y) -- the list Y is a permutation of the list X, del(X, Y, Z) -- Z is the result of deletion of an element Y from list X and ord(X) -- X is an ascending ordered list. "perm", "del" and "ord" are old predicates. Definition set is {(8)}. It defines the new predicate "i_sort" where isort(X, Y) reads Y is the sorted list of X. Initial program S1={(1),..., (7), (8)} is a naive sorting program with time complexity of factorial order in case of mode isort(+, -). We show a derivation of another but equivalent "insertion sort" program of cubic order by transformation.

To help inuitive understanding, we display the transformation process in a tree like diagram and do not mention the legality of rule applications.

```
i_sort(X, Y) <->>perm(X, Y) <<&ord(Y)                          ... (8)
  i_sort([], []) <->>ord([]) <<                                ... (9)
    i_sort([], [])                                             .. (10)
  i_sort([H|X], Y) <- perm(X&Z) &del(Y, H, Z) &ord(Y)          .. (11)
    i_sort([H|X], Y) <- <<perm(X, Z), ord(Z) >>, del(Y, H, Z) &ord(Y)  (12)
      i_sort([H|X], Y) <-i_sort(X, Z) &<<del(Y, H, Z) &ord(Y) >>   .. (13)
        i_sort([H|X], Y) <-i_sort(X, Z) &insert(H, Z, Y)           .. (14)

insert(H, Z, Y) <-del(Y, H, Z) &ord(Y)                             .. (15)
```

Fig. 3. 1. 5. Transformation process of "insertion sort" program

We unfold the goal perm(X, Y) of clause (8) first. The result contains two clauses (9) and (11). We have < { (1), .., (7), (9), (11) }, { (8) } > as configuration. By unfolding perm([], []) of clause (10), i_sort([], []) is obtained. Next configuration becomes <{(1),..., (7), (10), (11)}, {(8)}>. Here, we note that:

(a) M(S1) |= ord(Z)<-exist((Y, H), del(Y, H, Z)&ord(Y)), and

(b) the computation of "ord(Z)" is always a part of that of exist((H, Z), del(Y, H, Z)&ord(Y)).

So we can add ord(Z), according to goal addition rule, to the body of clause (11). Next state becomes < {(1),..., (7), (10), (12)}, {(8)} >. Although this addition reduces the search space of the program to some extent, it can not improve time complexity. Instead, it gives rise to a chance of folding. We fold perm(X, Z)&ord(Z) into i_sort(X, Z). Clause (12) becomes clause (13) and the configuration is now <{(1), ..., (7), (10), (13)}, {(8)}>. By this folding, we have achieved the improvement of time complexity since the program {(1),..., (7), (10), (13)} is a sorting program of cubic order. To complete the program, we introduce a new predicate insert(H, Z, Y) by clause (15) which reads an insertion of H into list Z yields Y and fold the goals del(Y, H, Z)&ord(Y) in clause (13) into clause (14). Configuration becomes < {(1),..., (7), (10), (14)}, {(8), (15)} >. The sorting program now looks like:

```
i_sort([], [])                                          .. (10)
i_sort([H|X], Y)<- i_sort(X, Z)&insert(H, Z, Y)         .. (14)
insert(H, Z, Y)<-del(Y, H, Z)&ord(Y)                    .. (15)
  clauses for "del" and "ord"
```

Fig. 3.1.6. Cubic order insertion sort program

This is a sorting program, cubic order in the length of X in i_sort(X, Y), which is assured to be equivalent to the initial program {(1),..., (7), (8)}. If we think an insertion program by clause (15) should be optimized because insert(H, Z, Y) is a square order algorithm in case of mode insert(+, +, -), it is not difficult to obtain an insertion program of linear order by transformation, though we do not show the transformation process here.

We have attempted other examples: a fibonacci number program, a string matcher program, a same-fringe program (comparing leaves of two trees), a selection sorting program, an 8-queens program and so on. In almost cases, unfolding/folding transformation did not suffice to achieve

significant program improvement. Improvement was achieved, for example, with the help of function merge, case split and goal addition rules. But we have found that a small number of rules seemed sufficient in a simple setting to improve programs and have come to believe that it would be possible to build an automatic program transformation system in a specific area. Along this line, we have started developing "append optimizer" intended to support such declarative programming that naive use of "append" does not obstruct writing efficient programs but only serves writing clear ones. It is based on unfolding/folding rules, laws such as associativity of "append" relation and a few heuristics to introduce new definitions. The experimental results are encouraging. We expect it will evolve into a new type of "compiler" in logic programming.

3. 2. Negation Technique

In a set theory, we have an operation that takes the complement of a set. In logic programming, the same can be done by writing a first order program. If we want a program for ~p(x) relation, the following program:

not_p(x) <-(p(x)->false)

would do the job aside from obvious theoretical limitation (for there is a set whose complement is not recursively enumerable so that no program can compute the complement of it). If we note a negation program is always of the form not_p(x)<-(p(x)->false), it is natural to seek for a method which takes advantage of this specific form.

"Negation technique", explained in the sequal, is such a method. It transforms a definite clause program S into another iprogram S' such that the relations computed by S' are complementary to those computed by S. Moreover, S' does not include extraneous symbols like continuation function symbols and/or continuation predicate symbols. Therefore, it gives negated programs far less compilcated than those obtained through negation by first order programming.

If each predicate p in S has a corresponding predicate "not_p" in S' and,

Not(S |- p(a)) if S' |- not_p(a)

holds for every ground atom p(a) and not_p(a), S' is said to be a dual program of S. If "if" can be replaced by "iff", it is said to be a complementary program of S. Negation technique is a method to derive a dual/complementary program S' from S. We name the negation through negation technique "procedural negation" for the reason that it is, in contrast with "logical negation", a procedural representation of failed computations by means of definite clause programs. To make the point clear, we illustrate a small example.

3. 2. 1. Beginning negation technique

Suppose we have an "even" program S:

```
even(0)
even(s(s(X)))<-even(X)
```

Fig. 3. 2. 1. "even" program S

S defines the set of even numbers. The complement is the set of odd numbers. We show intuitively how to derive program S' from S that can calculate the set of odd numbers. The idea is to mimic the failed computations of S and represent them positively. It is negation-as-failure without negation.

Suppose that a ground call <-even(a) occurs. We examine, by case-by-case analysis, when it fails. It fails when both of the call to even(0) and the call to even(s(s(X)))<-even(X) fail, namely:

(1) even(a) does not match any clause head or,
(2) even(a) does not match even(0) and
it matches even(s(s(X))) and
the call even(X) fails.

By introducing a new predicate not_even(X) that is supposed to represent the failure of call to even(X), we can describe each case by:

(1') not_even(A)<- A=/=0, all(X, A=/=s(s(X)))
(2') not_even(A)<- A=s(s(X))&not_even(X)

This is a logic program since we can consider all(X, A=/=s(s(X))), representing unification failure between A and s(s(X)), as the relation computed by some logic program (we have assumed functor symbols are finite). Moreover, if X in even(X) is known to range over a natural number sort { 0, s(0), s(s(0)) ...}, this program is simplified into S':

```
not_even(s(0)).
not_even(s(s(X)))<- not_even(X).
```

Fig. 3. 2. "not_even" program S'

This simplification is leagal because one of A=0, A=s(0) and exist(X, A=s(s(X))) always holds for any A so that we

can prove all(X, A=/=s(s(X))) is equivalent to A=s(0). We also can prove that a ground atom not_even(a) is provable from S' iff every SLD proof of even(a) in S fails and that even(a) always terminates with success or failure in S. Therefore, not_even program S' computes exactly the complement of "even" relation.

3. 2. 2. Formalization

The process of negation described above was intuitive. But it can be formalizable at the cost of notational complexities. We next present a formalized version of negation technique which is easy to justify. Suppose we have a logic program S to be negated.

----------(Negation Algorithm : CASE A)----------

(CASE A : No clause in S has an internal variable)
We use "even" example.

(step 1)
Make an iff definition (1, 9) of S.

```
even(A) <-> A=0 \/
            exist(X, A=s(s(X))&even(X))
```

(step 2)
Negate both sides. Use the fact that all(x, A=/=t(X) \/ ~p(X)) is equivalent to all(X, A=/=t(X))\/exist(x, A=t(x)& ~p(x)) in Herbrand universe.

```
~even(A) <-> A=/=0 &
             (all(X, ~A=/=s(s(X))) \/
              exist(X, A=s(s(X))&~even(X)))
```

(step 3)
Convert the right hand side into a disjunctive form.

```
~even(A) <-> (A=/=0 & all(X), A=/=s(s(X)))) \/
             (A=/=0 & exist(X, A=s(s(X))&~even(X)))
```

(step 4)
Change names (p --> not_p) and make a form of iff definition.

```
not_even(A) <-> (A=/=0 & all(X, A=/=s(s(X))) \/
                 exist(X, s(s(X))=/=0) &
                          A=s(s(X)) &
                          not_even(X))
```

(step 5)
Simplify the parts concerning unification.

```
not_even(A) <-> (A=/=0 & all(X, A=/=s(s(X)))) \/
```

exist(X, A=s(s(X)) & not_even(X))

{ step 6 }
Convert to a set of definite clauses.

```
not_even(A) <-  A=/=0 & all(X, A=/=s(s(X)))
not_even(s(s(X)) <- not_even(X)
```

{ step 7 }
Simplify clauses using sort information.

```
not_even(s(0))
not_even(s(s(X)) <- not_even(X)
```

----------{ End of Negation Algorithm A }----------

We would like to state the correctness of this algorithm. Recall that success(S) is the set of ground atoms provable from S. Dually we define failure(S) as:

failure(S) = { A | Proof of ground goal A by a top down interpreter with a fair rule selection rule fails in S } (38)

success(S) and failure(S) are disjoint. For any ground atom, there are three possibilities:

(1) It belongs to success(S).
(2) It belongs to failure(S).
(3) It belongs to neither success(S) nor failure(S).

If every ground atom belongs to either success(S) or failure(S), S is said to be dichotomous, or totally terminating. For, this means that the third possibility is excluded and any SLD proof of a ground atom terminates with success or failure (1). Note that since both of success(S) and failure(S) are recursively enumerable, if success(S) are not recursive, there must be an ground atom A such that any attempt of SLD proof of A does not terminate.

{ Theorem 3.2.1. }
Let S be a logic program in which no internal variable occurs and S' the result of { step 6/7 }. Then, for any ground atom p(a), not_p(a) is in success(S') iff p(a) is in failure(S). Therefore, S' is a dual program of S. If S is dichotomous, S' is a complementary program of S.

We give an intuitive proof. Let comp(S) be the union of iff definitions for every predicate in S, equality axioms and the inequality axioms for terms in Herbrand universe. By the result of (27), we know, comp(S) |- ~p(a) iff ~p(a) is in failure(S). So it is enough to prove comp(S) |-

~p(a) iff comp(S') |- not_p(a). But this is obvious since both proofs are mutually recoverable by exchanging predicate names (p<--> not_p).

In some cases we can relax no internal variable condition. Consider the following multiplication program S.

```
mult(0, Y, 0)
mult(X+1, Y, Z) <- mult(X, Y, W) & add(W, Y, Z)

add(0, Y, Y)
add(X+1, Y, Z+1) <- add(X, Y, Z)
```

Fig. 3.2.3. "mult" program.

This is not internal variable free. The second clause has an internal variable W. But we note that W is unique with respect to <Y, Z> because add(W, Y, Z) defines a partial function from <Y, Z> to W. The domain of this function is such <Y, Z> that exist(W, add(W, Y, Z)) and the complement of the domain is such <Y, Z> that Y>Z.

We say an internal variable Y of a clause C in a logic program S has a function part p(x, Y) for Y if:

(1)
p(x, Y) is a goal in C such that every variable in x appears in the head of C,

(2)
Y is unique w.r.t. x, i.e. M(S) |= b=b' <- p(a, b)&p(a, b') for any ground terms a, b, b' and

(3)
There exists a predicate (non-domain predicate) non_dom(x) computed by some logic program such that

M(S) |= non_dom(a) iff M(S) |= all(Y, ~p(a, Y))

So W in Fig. 3.2.3. has a function part add(W, Y, Z) and a non-domain predicate Y>Z. If every internal variable in S has a function part and a non-domain predicate, negation technique extends to this case as follows:

----------(Negation Algorithm : CASE B)----------

(step 1)
Make an iff definition.

```
mult(A, B, C) <-> exist(Y, <A, B, C>=<0, Y, 0>) \/
```

```
                    exist((X, Y, Z, W),
                            <A, B, C>=<X+1, Y, Z> &
                            mult(X, Y, W) & add(W, Y, Z))
```

{ step 2 }
Negate both sides of the definition. Use the fact that all((x, Y), A=/=t(X) \/ ~p(X, Y) \/ ~q(X, Y)) is equivalent to all(x, A=/=t(X)) \/ exist(X, A=t(X)&non_dom(X)) \/ exist((X, Y), A=t(X)&p(X, Y)&~q(X, Y))) in Herbrand universe when p(x, Y) is a function part for Y with a non-domain predicate non_dom(X).

```
        ~mult(A, B, C) <->
                all(Y, <A, B, C>=/=<0, Y, 0>) &
                { all((X, Y, Z), <A, B, C>=/=<X+1, Y, Z>) \/
                  exist((X, Y, Z),
                        <A, B, C>=<X+1, Y, Z> & Y>Z) \/
                  exist((X, Y, Z),
                        <A, B, C>=<X+1, Y, Z> &
                        add(W, Y, Z) &
                        ~mult(W, Y, Z)) }
```

{ step 3 }
Make the right hand side a disjunctive form.

{ step 4 }
Change predicate names (p --> not_p) and make a form of iff definition.

```
        not_mult(A, B, C) <->
                ( all(Y, <A, B, C>=/=<0, Y, 0>) &
                  all((X, Y, Z), <A, B, C>=/=<X+1, Y, Z>) ) \/
                exist((X, Y, Z),
                        <A, B, C>=<X+1, Y, Z> &
                        all(Y', <X+1, Y, Z>=/=<0, Y', 0>) & Y>Z) \/
                exist((X, Y, Z, W),
                        <A, B, C>=<X+1, Y, Z> &
                        all(Y', <X+1, Y, Z>=/=<0, Y', 0>) &
                        add(W, Y, Z) &
                        not_mult(W, Y, Z))
```

{ step 5 }
Simplify the parts concerning unification.

```
        not_mult(A, B, C) <->
                (<A, C>=/=<0, 0> & all(X, A=/=X+1)) \/
                exist((X, Y, Z),
                        <A, B, C>=<X+1, Y, Z> & Y>Z) \/
                exist((X, Y, Z, W),
                        <A, B, C>=<X+1, Y, Z> &
                        add(W, Y, Z) &
                        not_mult(W, Y, Z))
```

{ step 6 }

Convert to a clause set.

(step 7)
Simplify using sort information.

```
not_mult(0, B, C) <- C=/=0
not_mult(X+1, Y, Z) <- Y>Z.
not_mult(X+1, Y, Z) <- add(W, Y, Z) & not_mult(W, Y, Z)
```

Fig. 3.2.4. "not_mult" program

----------(End of Negation Algorithm B)----------

(Theorem 3.2.2.)
Let S be a logic program and S' the result of (step 6/7) above. Then, for every ground atom p(a) and not_p(a), if not_p(a) is in success(S'), p(a) is not in success(S). So S' is a dual program of S. Since if p(a) is in failure(S), not_p(a) is in success(S'), S' is a complementary program of S if S' is dichotomous.

We outline a proof. Let M' = { p(a) | not_p(a) is not in success(S') }. M', considered as an interpretation of predicate symbols in S, satisfies comp(S). This entails the theorem.

When we are faced with a program S0 for which neither (CASE A) nor (CASE B) is applicable, we transform S0 to S1, a program equivalent to S0 but to which negation technique is applicable. For example, an initial program S0:

```
initial(X, Z) <- append(X, Y, Z)
append( (), Y, Y)
append( (H|X), Y, (H|Z)) <- append(X, Y, Z)
```

precludes negation technique because of the internal variable Y in the first clause. But as shown before, we can obtain the following internal variable free "initial" program S1:

```
initial( (), Z)
initial( (H|X), (H|Z)) <- initial(X, Z)
```

Assuming "initial" has a sort pattern initial(list, list), we get S1' by negation technique:

```
not_initial( (A|X), ())
not_initial( (A|X), (B|Z)) <- A=/=B
not_initial( (A|X), (A|Z)) <- not_initial(X, Z)
```

Fig. 3.2.5. "not_initial" program S1'

Every SLD proof of initial(a, b) in S1 terminates when a, b are ground. So S1' is dichotomous and complementary to S1. Since S1 is equivalent to S0, we can conclude that S1' is complementary to S0.

3. 2. 3. Double negation technique

Although negation technique is just a transformation algorithm applied to definite clause programs, it gives birth to yet another way to definite clause program derivation (synthesis) from first order programs (specifications) when combined with equivalence preserving transformation. We confine the domain of discourse to somewhat restricted non-recursive first order programs, firstly because in general, the new method can not handle a recursive specification of the form p(s, t) <- all(y, p(u, y) ->p(y, v)) where a predicate to be difined has both positive and negative occurrences in the body (we met this type of specification before when we dealt with the intuitionistic truth predicate example). Secondly because the restriction instead makes it possible to discuss the semantic aspect of program derivation more precisely than the compiling approach.

We therefore give a formulation of logic program synthesis adapted to the one from non-recursive specifications. We also give a definition of (partial) correctness of the derived progam with respect to the specification.

Suppose we have a definite clause program S. It provides us with basic relations, called old relations, each computed by S, available for constructing a new relation. Every predicate appearing in S is called an old predicate. A new relation p is defined by a specification. It is, by definition, a first order formula of the form p(x) <->F(x) where p(x) is an atomic formula and F(x) is a general formula containing only old predicates. The problem of logic program synthesis is to find a definite clause program S' for p(x) such that M(S) |= F(a) if/iff M(S') |= p(a) for any ground terms a. Then S' is said to be partially/totally correct with respect to the specification according to if/iff in the statement, and in either case, it is said to S' realizes the specification.

Take a specification of the form p(x) <->all(y, F(x, y)). To find a program that realizes it, we first realize another specification not_p(x) <->exist(y, ~F(x, y)) by some program S', with the help of transformation system, eliminate existential variables y and finally apply negation technique. The detail follows:

----------(Double Negation Technique)----------

(step 0)
An initial specification using "all" is given.
L is supposed to be a list here.

leqall(L, M) <->all(H, mem(H, L) -> H=<M)

where "mem" is given by:

```
mem(X, [X|Y])
mem(X, [H|Y]) <-mem(X, Y)
```

(step 1)
Obtain a specification by logical negation.

```
not_leqall(L, M) <->exist(H, mem(H, L) & H>M)
```

(step 2)
Realization of the second specification by a logic program S'.

```
not_leqall(L, M) <-mem(H, L) &H>M

and clauses for "mem"
```

(step 3)
The elimination of internal variables from S' by equivalence preserving transformation. S'' obtained.

```
not_leqall([H|L], M) <-H>M
not_leqall([H|L], M) <-not_leqall(L, M)
```

(step 4)
Take the procedural negation of S'' by applying negation technique to S''. S obtained.

```
leqall([], M)
leqall([H|L], M) <-H=<M&leqall(L, M)
```

Fig. 3.2.6. Process of double negation

----------(End of Double Negation Technique)----------

It is easy to see that if the program at (step 3) is dichotomous, the final program at (step 4) is totally correct with respect to the initial specification and otherwise, only partial correctness is assured. Therefore, in this case, S is totally correct with respect to the initial specification leqall(L, M) <-> all(H, mem(H, L) -> H=<M).

Realization of a universally quantified specification p(x) <-> all(y, F(x, y)) using logical negation, transformation and procedural negation by negation technique in this way is called "double negation technique."

Negation technique and double negation technique enable us to have a systematic approach to definite clause program synthesis from non-recursive first order specifications. Suppose a(x), b(x), c(x, y) are predicates defined by some logic program S through its least model. Then, in each row in the figure below, a specification in the left side is realized by a program in the right side where p(x) is a predicate to be specified.

p(x)<->a(x) & b(x)	...	{ p(x)<-a(x)&b(x) }+S
p(x)<->a(x) \/ b(x)	...	{ p(X)<-a(X), p(X)<-b(X) }+S
p(x)<->exist(y, c(x, y))	...	{ p(X)<-c(X, Y) }+S
p(x)<->~a(x)	...	by negation technique
p(x)<->all(y, c(x, y))	...	by double negation technique

Fig. 3. 2. 7. One step synthesis

For a general non-recursive specification p(x)<->F(x), we apply the above table recursively to subformulae of F(x). The resulting program is totally correct provided that, in the synthesis, every intermediate program to which negation technique was applied is dichotomous.

The program synthesis method presented here is general and we hope it is also usable. But liability can reside in double negation techinique bacause it always forcess two operations, logical negation and the subsequent procedural negation, which seemingly should be cancelled out. So one may well suspect that it can be done more directly. Such a direct method was proposed by Kanamori and Seki (31) as generalization of unfold/fold transformation. In contrast with ours which is only capable of dealing with atoms, their system allows to unfold and fold more general formulae, thus eliminates the need for double negation to treat universal quantification. Conditions on the application of unfolding and folding are duly more complex than ours, but the meanings of formulas, hence, specifications, are assured to be preserved.

3. 3. Getting usable programs

Up until now, we have proposed two approaches to logic program synthesis. One, the compiling approach, leads to first order programming. The other, nondeterministic one, makes use of unfold/fold transformation. Both tend to yield rather naive definite clause programs, though they are certainly runnable. For our formalism to be effective, however, it is requested to give more than runnable programs. In what follows, we would like to show how to derive "usable" programs.

We have chosen, as an example, the problem of deriving a usable maximum program. It typically illustrates multi-staged programming methodology where we first write a declarative program (a specification) of generate-and-test type, then, by compilation or transformation, derive a runnable program and finally, by appealing to transformation to reduce nonderminacies in the program, reach a usable program. Now follows a specification:

```
max(M, L) <-> mem(M, L) & all(X, mem(X, L) -> X=<M)

        mem(X, [X|Y])
        mem(X, [H|Y]) <-mem(X, Y)

        and clauses for "=<" and "<"
```

Fig. 3. 3. 1. spec-1: A specification of "max" relation

This is the same as one used in section 2. 13. (see Fig. 3. 3. 2.). max(M, L) reads M is the maximum element of list L. mem(M, L) reads M is an element of list L. We assume that the clauses "=<" and "<" consists of infinite unit clauses. Since those unit clauses play only a secondary role in the synthesis and subsequent transformation, they are omitted in the following.

The subformula all(X, mem(X, L) -> X=<M) is not executable. So we introduce a sub-specification below:

```
leqall(M, L) <-> all(X, mem(X, L) ->X=<M)
```

Fig. 3. 3. 2. spec-2: A sub-specification for "leqall"

This specification is a compilable first order program. But we prefer synthesis by double negation technique for the

aethetic reason, giving:

```
leqall( [], M)
leqall( [H|L], M) <-H=<M & leqall(L, M)
```

Fig. 3.3.3. A runnable program for "leqall"

This is totally correct with respect to spec-2 in Fig. 3.3.2. as mentioned in the preceding section. Thus, we have successfully derived a program S (see Fig. 3.2.6.):

```
max(M, L) <- mem(M, L) &leqall(L, M)                    ... (1)

leqall( [], M)                                          ... (2)
leqall( [H|L], M) <-H=<M&leqall(L, M)                   ... (3)

mem(X, [X|Y])                                           ... (4)
mem(X, [H|Y]) <-mem(X, Y)                               ... (5)
```

Fig. 3.3.2. A runnable "max" program S

which is totally correct with respect to spec-1 in Fig. 3.3.1.. S coincides with the program shown in Fig. 3.3.4. in section 2.13., which was derived by compilation of the specification and subsequent optimization -- elimination of continuation functions by unfold/fold transformation.

S works under the mode max(-, +). But computing max(M, L) takes the time complexity of square order in the length of L. Therefore it should be called only a "runnable" program. So next problem is to derive a "usable" one by means of equivalence preserving program transformation. It starts from <S1, D1> with an initial program S1 = S+{(1)} and a definition set D1={(1)}.

```
max(M, L) <- mem(M, L) &>>leqall(L, M) <<               ... (1)
```

We first unfold the second goal and obtain the only one resolvent:

```
max(M, [X|L]) <- mem(M, [X|L]) &X=<M&leqall(L, M)       ... (6)
```

Since this can be seen as a definition for "max", we abolish the preceding definition (1) and start again a new transformation process with the configuration <S1', D1'> where S1'={(6), (2), ..., (5)} and D1'={(6)}. We show the transformation history below:

```
max(M, [X|L]) <- mem(M, [X|L]) &X=<M&>>leqall(L, M) <<                    ... (6)
  max(M, [X]) <- >>mem(M, [X]) <<&X=<M
   --- subsequent unfolding ---
    max(M, [M])                                                           ... (7)
  max(M, [X, Y|L]) <- mem(M, [X, Y|L]) &X=<M&Y=<M&leqall(L, M)            ... (8)

    ( case split :  X=<Y )
    max(M, [X, Y|L])
        <- mem(M, [X, Y|L]) &X=<M&Y=<M&X=<Y&leqall(L, M)                  ... (9)
      max(M, [X, Y|L])
          <- X=<Y&<<mem(M, [Y|L]) &Y=<M&leqall(L, M) >>                   ... (10)
        max(M, [X, Y|L]) <-X=<Y&max(M, [Y|L])                            ... (11)

    ( case split :  X>Y )
    max(M, [X, Y|L])
        <- mem(M, [X, Y|L]) &X=<M&Y=<M&Y<X&leqall(L, M)                   ... (12)
      max(M, [X, Y|L])
          <-Y<X, <<mem(M, [X|L]) &X=<M&leqall(L, M) >>                    ... (13)
        max(M, [X, Y|L]) <- Y<X&max(M, [Y|L])                            ... (14)
```

Fig. 3.3.3. Transformation process of S

Unfolding leqall(L, M) of (6) gives two resolvents. One leads to (7) by the subsequent unfolding. The other is (8). We apply the case split rule to (8) using predicates "=<" (greater or equal to) and "<" (greater than). In the first case, (8) becomes (9). Since:

```
( mem(M, [X, Y|L]) & X=<M & Y=<M & X=<Y )
                    <-> ( mem(M, [Y|L]) & X=Y & Y=<M )
```

is true in M(S), the least model defined by S, the goal replacement rule is applicable. The application yields clause (10). It has a foldable goal set { mem(M, [Y|L]), Y=<M, leqall(L, M) }, (11) is obtained by folding. Similarly, we obtain clause (14) from the second case. Although we did not mention the conditions of rule applications, it is not difficult to check them. The final program S' becomes:

```
max(M, [M])                                                   ... (7)
max(M, [X, Y|L]) <-X=<Y&max(M, [Y|L])                         .. (11)
max(M, [X, Y|L]) <-Y<X&max(M, [Y|L])                          .. (14)
```

Fig. 3.3.4. A usable "max" program S'

S' can find the maximum element M of list L with linear order time complexity. So it is entitled to be called a

usable program. Moreover, it is totally correct with respect to the initial max specification because S' is equivalent to S and S is totally correct with respect to the initial specification spec-1 in Fig. 3. 3. 1..

```
( spec-1 ) :
max(M, L) <-> mem(M, L) & all(X, mem(X, L) -> X=<M)
|    mem(X, [X|Y])
|    mem(X, [H|Y]) <-mem(X, Y)
|
|     ( spec-2 ) :
|     leqall(M, L) <- all(X, mem(X, L) -> X=<M)
|     |
|     | <---(double negation technique)
|     |
|     leqall program
|        leqall( [], L)
|        leqall( [H|L], M) <-H=<M & leqall(L, M)
|
S : square order "max" program
|        max(M, L) <- mem(M, L) &leqall(L, M)
|        leqall( [], M)
|        leqall( [H|L], M) <-H=<M&leqall(L, M)
|        mem(X, [X|Y])
|        mem(X, [H|Y]) <-mem(X, Y)
|
| <---(equivalence preserving transformation)
|
S' : linear order "max" program
        max(M, [M])
        max(M, [X, Y|L]) <-X=<Y&max(M, [Y|L])
        max(M, [X, Y|L]) <-Y<X&max(M, [Y|L])
```

Fig. 3. 3. 5. "max" program derivation as a whole

The whole derivation process is summarized in Fig. 3. 3. 5.. Generally speaking, one of the crucial points in unfold/fold transformation is the introduction of new predicates. To get more chances of folding, a new predicate must be defined as general as possible. On the contrary, however, in order to get an efficient program, it is desirable to introduce a new predicate as specific as possible. Simple-minded introduction of a new predicate usually leads to astray. We have developed a couple of generalization strategies for new predicate introduction. In [57], the problem of synthesis of 8 queens program is treated where an example of generalization using "buffer predicate" is shown.

4. Conclusion

Our basic point of view is that a program is amalgamation of three parts: logical part, control part and data structure and the programming which encodes the logic part first, i.e. logic programming, must be taken seriously as an alternative to the prevailing ones if we want to establish bug-free program development methodology. We particularly insist on the need for developing a highly expressive logic programming language in conjunction with a meaning preserving transformation system for logic programs because the former facilitates to write quickly correct and understandable programs while the latter provides the means for optimization without fear of disturbing correctness, thereby achieving both correctness and efficiency.

Aiming at a highly expressive declarative language, we have extended definite clause programs to first order programs in which universal quantifications are allowed in the form of all(y, p(x, y)->q(y, z)). We then have shown how to compile them into definite clause programs, faithfully to the procedural reading we gave to the construct. The proposed method is simple, deterministic and logically correct. It enables us to write programs which have virtually been impossible due to the drawbacks inevitable in implementing universal quantifiers through negation-by-failure combined with backtracking. It should be emphasized that not only can a compiled program be run in Prolog correctly without checking negative goals, but we are now almost certainly able to get variable bindings from negative goals (in the form of an implicational formula (p->false)) through compilation.

When viewed from a theoretical standpoint, our compilation is a deductive unfold/fold method applied to universal continuation forms. Mysteriously, the notion of continuation has not been introduced to logic programming in a formal way though it was used as a kind of programming technique (68). We have introduced it to logic programming by means of the existential and universal continuation forms. The former, exist(y, p(x, y)&cont-p(C, y)) where C is a term representing continuation, is just a direct translation of the equivalent already familiar in functional programming, but the latter, all(y, p(x, y)->cont-p(C, y)), can be considered as a proper generalization. In reality, were it not for the generalization, we could hardly have a logical justification of our compilation method. Since both forms are first order formulae, we can say now that continuation in logic programming is "a first order object" as well as "a first class object."

The proof theoretical part of the justification is based on the deductive unfold/fold method (7, 21, 25, 55). If there is a contribution to the method made by the author, it

is the recognition that there is a class of first order formulae to which the unfold/fold method is deterministically applicable to produce significant results. We know that the unfold/fold method is rather weak as a method of program synthesis compared with others. It focuses only on the partial correctness of a synthesized program. It can not tell whether a synthesized one terminates or not. Therefore, it is no wonder if it can synthesize a program "quickly." However, the attempts made so far along this line are all highly nondeterministic and hence have no prospect of automatization although there should be some benefits which can compensate for the loss of assurance of termination of a synthesized program. If this discussion makes sense, the method proposed in this thesis seems reasonable in that it is completely mechanical. We hope that this property can counterbalance the need for proving individually the termination of a synthesized program.

The extension of current logic programming to full first order logic programming still has a long way to go. But if we are to establish methodology capable of leading a programmer correctly to a program realizing his/her intention and if this intention is best expressed in terms of logical formulae, the extension to first order programming must be pursued with ever increasing efforts.

Devising an algorithm which can mechanically derive runnable programs from first order formulae is not the end of our story. For, as a result of the procedural interpretation we gave to all(y, p(x, y)->q(y, z)), compiled programs might be too naive to use. None will use "correct programs" which run terribly slow.

There may be a number of optimization techniques applicable to such naive programs. But only a few seem to have wide range applicablity. Unfold/fold transformation belongs to such few. It is simple but has the potential of automatically generating a compiler and a compiler-compiler from an interpreter. We therefore have adopted unfold/fold transformation as a basic framework for logic program optimization.

In the context of deriving correct programs, it is essential to formulate unfold/fold transformation rules (together with other transformation rules) in such a way that they always preserve the meaning of programs, or put differently, programs before transformation and those after transformation are the same except for efficiency. Otherwise, they would be too dangerous to use. This requirement may sound innocent and seem easy to satisfy. But once we actually try to formulate them for imperative languages, it does not take time to recognize the formidable difficulties due to the invisibility of logical content of programs and the semantic complications of imperative features.

On the contrary in logic programming, logic part is visible and indepedent of control part so that it is easy to define the semantics -- the least model semantics -- without concerning control strategies, but yet useful enough for describing what we intend to express by a program. We have formulated a logic program transformation system which includes many powerful transformation rules as well as unfold/fold transformation rules, then have proved that it always preserves the least models of programs whatever rules and transformation sequence we may use (64, 65).

Getting back to optimization, it should be noted that there are three types of optimization, one concerning logical part, one concerning control part and one concerning data structure. The first type of optimization can be done through our meaning preserving transformation system as shown in the preceding chapter. The effect of optimization, in particular, the elimination of that of nondeterminacies, was noticeable. It appears that the power of unfold/fold transformation is most convincingly revealed in logic programming. The third type of optimization, e.g. data-structure mapping (21) is also possible. But the second type, the optimization of control part, seems beyond our system at the moment. This is because the least model semantics to which unfold/fold transformation nicely fits, is too insenseitive to reflect individual control strategies so that we can not expect it to preserve such meaning of a program that takes the control detail into account. Take cut in Prolog for example. Since it affects the clauses remaining as alternatives, local rules such as unfold/fold rules concerning only clauses in question do not necessarily preserve program equivalence. In case of concurrent (comitted choice) logic programming languages, even unfolding might completely destroy operational semantics. We think control optimization of logic programs should be a future research topic.

Independently of unfold/fold transformation, we introduced another transformation -- negation technique. It provides a short cut to get a negated program from the positive program. It is, say, negation-without-failure. The technique is not universal because a certain condition, for instance, no internal variables condition, must be met by a program to be negated and not all programs satisfy it. However, we soon found that unfold/fold transformation can do something for the negation technique. For example, it can eliminate internal variables from programs, thus ensuring the applicablity of the negation technique. Being motivated by the discovery, we then introduced "double negation techinique" which offers another way to deal with all(y, p(x, y)) specification by the sequence of logical negation, internal variable elimination by unfold/fold transformation, and the application of negation technique. It is

easy to see negation technique and double negation technique in conjunction with "&" and "\/" constructs already available in Prolog lead to a nondeterministic logic program synthesis method from first order specifications.

Finally we would like to make a few remarks on the overall effect of these individual devices -- logic programming, compilation, unfold/fold transformation, negation technique -- on program development. First logic programming enables us to divide the problem of program synthesis into that of runnable logic part and that of giving better control. Getting a runnable logic part can be done either by the deterministic compilation or by the nondeterministic combination of negation technique, transformation and double negation techinique. Program optimization can be achieved through transformation preserving partial correctness. One of the typical optimization effects is the reduction of nondeterminacy to the extent that the final program runs deterministically on Prolog, as exemplified by the "max" example. Thus, instead of trying to synthesize a "max" function all at once for example, our formulation allows to first divide the problem into smaller pieces such as the problem of obtaining a runnable "max" program and that of its optimization, and then tackle each piece. Our claim is that such two (or multi-) staged program development can reduce the search space at each stage, thereby making each stage manageable to a computer, hence so can be the whole problem. We hope our formulation provides programming development methodology which uses abundant computer power effectively not only for the kind of book keeping but for the essential task, search of solutions.

{ Acknowledgements }

This work started in Japan and has been developed when the author stayed at Imperial College of Science and Technology in London. He therefore wishes to express his thanks to Yoshihiko Futamura for initiating the research into transformational programming and to Robert Kowalski for giving him the opportunity to work in his group. Thanks are also due to the British Council, which supported the author's stay at the College. He is deeply indebted to Hisao Tamaki for the encouragement and perspicuous discussions without which this thesis would never been completed. He has benifited as well from the discussions with the members of Machine Inference Section of Electrotechnical Laboratory and the working groups of the Fifth Generation Computer Project. Last of all, I would like to thank Mr. Tom Routen and Clive Spenser for patient proof reading.

(References)

(1) Apt, K. R. & van Emden, M. H., "Contributions to the Theory of Logic Programming," J. ACM 29-3, 1982.
(2) Barstow, D., "Automatic Programming for Streams," 9th IJCAI, 1985.
(3) Bibel, W., "Syntax-Directed, Semantics-Supported Program Synthesis," Artificial Intelligence 14, North-Holland 1980.
(4) Bowen, K. A., "Programming with full first-order logic," Machine Intelligence 10, Ellis Horwood Ltd., 1982.
(5) Burstall, R. M. & Darlington, J., "A Transformation System for Developing Recursive Programs," J. ACM Vol. 24 No. 1, 1977.
(6) Chang, C. L. & Lee, R. C. T., "Symbolic Logic and Mechanical Theorem Proving," Academic Press, New York, 1973.
(7) Clark, K. L. & Sickel, S., "Predicate Logic:A Calculus for Deriving Programs," 5th IJCAI, 1977.
(8) Clark, K. L. & Tarnlund, S-A., "A First Order Theory of Data and Programs," IFIP 77, North-Holland, 1977.
Deriving Programs," 5th IJCAI, 1977.
(9) Clark, K. L., "Negation as Failure," in Logic and Database, Plenum Press, New York, 1978.
(10) Clocksin, W. F. & Mellish, C. S., "Programming in Prolog," Springer-Verlag, 1981.
(11) Darlington, J., "An Experimental Program Transformation Synthesis System," Artificial Intelligence 16, 1981.
(12) Darlington, J., "A Synthesis of Several Sorting Algorithm," Acta Informatica 11, Springer, 1978.
Synthesis System," Artificial Intelligence 16, 1981.
(13) Eriksson, A. & Johanson, A. L., "Computer_Based Synthesis of Logic Programs," 5th International Symposium on Programming, Lec. Note in Comp. Sci. 137, Springer, 1982.
(14) Ershov, A. V., "Mixed Computation:Potential Applications and Problems for Study," Theoretical Comp. Sci. 18, North-Horlland, 1982.
(15) Futamura, Y., "Partial Computation of Programs," RIMS Symposia, Lecture Note in Comp. Sci. 147, Springer 1982.
(16) Gabbay, D. M. & Reyle, U., "N-Prolog: An Extension of Prolog with Hypothetical Implications. 1.," J. of Logic Programming, 1984.
(17) Goto, S., "Program Synthesis from Natural Deduction Proof," 6th IJCAI, 1979.
(18) Green, C., "Application of Theorem Proving to Problem Solving," 1st IJCAI, 1969.
(19) Gresse, C., "Automatic Programming from Data Types Decomposition Patterns," 8th IJCAI, 1983.
(20) Guiho, G., "Program Synthesis from Incomplete Specifications," 5th Conf. on Automated Deduction, Lecture Note on Comp. Sci. 87, Springer, 1980.
(21) Hansson, A. and Tarnlund, S-A., "A Natural Programming Calculus," 6th IJCAI, 1979.
(22) Haridi, S. & Sahlin, D., "Evaluation of Logic Programs

Based on Natural Deduction," TRITA-CS-8305,
The Royal Institute of Technology, Stockholm, 1983.
(23) Hayashi, S., "PX:a system extracting programs from proofs," 86-PL-8, Information Processing Society of Japan, 1984.
(24) Hogger, C. J., "Derivation of Logic Programs," J. ACM Vol. 28 No. 2, 1981.
(25) Hogger, C. J., "Introduction to Logic Programming," Academic Press, 1984.
(26) Honda, M. et al, "On Automatic Synthesis of Programs" (in Japanese), Tsuusin-gakkai, Gihou AL 75-28, 1975.
(27) Jaffar, J., Lassez, J. and Lloyd, J., "Completeness of the Neagation as Failure Rule," 8th IJCAI, 1983.
(28) Jones, N. D. et al, "An experiment in Partial Evaluation: The generation of a Compiler Generator," ACM SIGPLAN NOTICIES Vol. 20 Num. 8, 1985.
(29) Kahn, K. & Carlsson, M., "The Compilation of Prolog programs without The Use of a Prolog Compiler," FGCS 1984, Tokyo, 1984.
(30) Kanamori, T. and Seki, H., "Verification of Prolog Programs Using An Extension of Execution," ICOT TR-093, Tokyo, 1985.
(31) Kanamori, T. and Horiuch, K., "Construction of Logic Programs Based on Generaliaed Unfold/Fold Rules," ICOT TR-094, Tokyo, 1985.
(32) Kleene, S. C., "Introduction to Metamathematics," North-Holland, 1971.
(33) Komorowski, H. J., "Partial Evaluation as a Means for inferencing data structures in an applicative language," 9th POPL, 1982.
(34) Kott, L., "Unfold/Fold Program Transformations," Research Rep. 155 Centre de Rennes, INRIA, 1982.
(35) Kowalski, R. A., "Predicate Logic as Programming Language," IFIP-74 Congress, 1974.
(36) Kowalski, R. A., "Logic for Problem Solving," North-Holland, New York, 1979.
(37) Lassez, J. L. and Maher, M. J., "Optimal Fixedpoints of Logic Programs," Theoretical Computer Science 39, North-Holland 1985.
(38) Lloyd, J. W., "Foundations of Logic Programming," Symbolic Computation Series, Springer, 1984.
(39) Lloyd, J. W. & Topor, R. W., "Making PROLOG More Expressive," Journal of Logic Programming Vol. 1, No. 3, 1984.
(40) Lloyd, J. W. & Topor, R. W., "A Basis for Deductive Database Systems," J. of Logic Programming Vol. 2, 1985.
(41) Manna, W. & Waldinger, R., "Toward Automatic Program Synthesis," C. ACM Vol. 14 No. 3, 1971.
(42) Manna, W. & Waldinger, R., "Automatic Synthesis of Systems of Recursive Programs," 5th IJCAI, 1977.
(43) Manna, W. & Waldinger, R., "Deductive Synthesis of the Unification Algorithm," STAN-CS-81-855, Dept. of Comp. Sci, Stanford Univ., 1981.
(44) Martin-Lof, P., "Constructive Mathematics and Computer

Proggramming," 6th International Congress for Logic, Methodology of Science, Hannover, 1979.

(45) Mizikami, T et al, "Extension of a Program Synthesis Based on the Intuitionistic Natural Number Theory to List-Data Processing" (in Japanese), Tsuusin-Gakkai Ronbun-shi D-8, 1981.

(46) Mutoh, Y. & Shimura, M., "Program Synthesising Algorithm for Generating Both Branching and Iterations" (in Japanese), Tsuusin-gakkai, Gihou AL 78-58, 1978.

(47) Nakagawa, H., "Prolog Program Transformation of Tree Manipulation Programs," Chisiki Kougaku to Jinkou Chinou 36-5, Information Processing Society of Japan, 1984.

(48) Nordstrom, B., "Programming in Constructive Set Theory: Some examples," ACM FP Symposium, 1982.

(49) Partsch, H. & Steinbruggen, R., "Program Transformation Systems," ACM Comp. Surveys Vol. 15 No. 3, 1983.

(50) Paige, R. A., "Formal Differentiation: A Program Synthesis Method," UMI Research Press, 1979.

(51) Robbin, J. W., "Mathematical Logic," BenjaminInc., New York, 1969.

(52) Sato, M., "Towards A Mathematical Theory of Program Synthesis," 5th IJCAI, 1979.

(53) Sato, T., "Negation and Semantics of Prolog Programs," 1st International Logic Programming Conf., Marceille, 1982.

(54) Sato, T., "An Algorithm for Intelligent Backtracking," Lecture Note in Comp. Sci. 147, Springer 1982.

(55) Sato, T. & Tamaki, H., "A Framework for Deductive Logic Program Synthesis," The Transactions of the IECE of Japan, Vol. E67, No. 10, 1984.

(56) Sato, T., "Enumeration of Success Patterns in Logic Programs," Theoretical Computer Sci. No. 34, North-Holland, 1984.

(57) Sato, T. & Tamaki, H., "Transformational Logic Program Synthesis," FGCS 1984, Tokyo, 1984.

(58) Scherlis, W. L., "Expression Procedures and Program Derivation," STAN-CS-80-818 Dep. of Comp. Sci., Stanford Univ., 1980.

(59) Shoenfield, R. S., "Mathematical Logic," Addison-Wesley, 1967.

(60) Stoy, J. E., "Denotational Semantics:The Scott-Strachey Approach to Programming Lnaguage Theory," The MIT Press, 1977.

(61) Takeuchi, S., "Applying Partial Evaluation to Meta-Programming," (in Japanese), Software-kisoron 13-2, Information Processing Society of Japan, 1985.

(62) Tamaki, H. & Sato, T., "Program Transformation Through Meta-shifting," New Generation Computing No. 1, OHMSHA, 1983.

(63) Tamaki, H. & Sato, T., "Append Optimizer," Proc. of Japan Logic Programming Conf. '84 (in Japanese), 1984.

(64) Tamaki, H. & Sato, T., "Unfold/Fold Transformation of Logic Programs," 2nd International Logic Programming Conf., Uppsala, 1984.

(65) Tamaki, H. & Sato, T., "A Generalized Correctness Proof of The Undold/Fold Logic Program Transformation," Techinical Rep. No. 86-4, Ibaraki Univ., 1986.
(66) Tamaki, H. & Sato, T., "OLD resolution with Tabulation," 3rd International Logic Programming Conf., London, 1986.
(67) Tarnlund, S-A, "Logic Programming Language Based on A Natural Deduction System," UPMAIL Tech. Report NO. 6 (1981).
(68) Ueda, K., "Making Exhaustive Search Deterministic"', Proc. of 3rd ILPC, London, 1986.
(69) Warren, D., Pereira, L. M. & Pereira, F., "User's Guide to DEC system-10 Prolog," occasional paper 15, Dept. of AI, Edinburgh Univ., 1979.
(70) Yonezawa, A., "A Method for Synthesizing Data Retrieving Programs," Journal of Information Processing, Vol. 5, No. 2, 1982.