The Halting Problem for Deductive Synthesis of Logic Programs

Kung-Kiu Lau
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

Mario Ornaghi
Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano
Via Comelico 39/41, Milano, Italy
ornaghi@imiucca.csi.unimi.it

Sten-Åke Tärnlund
Computing Science Department, Uppsala University
P.O. Box 520, S-751 20 Uppsala, Sweden
stenake@csd.uu.se

Abstract

Deductive synthesis methods derive programs in an incremental manner, and therefore pose a halting problem – when can synthesis stop with a correct program? We give a characterisation of this problem and state a halting principle as a solution. Another characteristic of deductive synthesis is that it may derive several correct programs, giving rise to another question – which correct programs are desirable? We show that the answer is related to the halting problem, via the notion of steadfast, or reusable, programs as desirable programs. Our work also reveals that Clark's idea of the completion of a program is central to deductive synthesis, since it is the basis of our halting principle and our notion of steadfast programs.

1 Introduction

Writing a correct program is a major problem in programming. It is theoretically interesting and practically significant. Incorrect programs may have dire consequences, and are unfortunately related to the software crisis in practice. Methods have been proposed and tried out for solving this problem, for almost half a century. Although some progress has been made, an adequate solution has yet to be found.

Historically, significant progress towards potential solutions of this problem has often followed from theoretical advancements in programming methodology. This experience suggests that further developments of programming methodology are not only sufficient but also necessary for continued progress towards the eventual solution of the correctness problem, as well as the software crisis hopefully.

A classic proposition for writing a correct program is to write a specification S and deduce a program P such that P is correct with respect to S. This is a good idea in its simplest form, but already in logic, difficulties can arise when the axioms in S are too numerous or too complex. This is certainly true for the applied sciences, mathematics, and programming. A natural development of this idea for programming is to develop a more interactive method between a specification S and a program P. Such a deductive synthesis method starts with a specification and derives a program P_1, and then $P_2 \supseteq P_1$, and so on until P_n, such that $P_n \supseteq \cdots \supseteq P_2 \supseteq P_1$, is the entire program P.

There are some important problems that can arise in deductive synthesis, that we shall address in this paper. Firstly, it is possible to derive several different programs which all are correct, but some of which are not necessarily desirable. We shall introduce the concept of a *steadfast program* and show that steadfast programs are desirable.

The deductive synthesis method also leads to a halting problem for program synthesis: when can we stop the synthesis with a correct program P? We shall give a characterisation of and a solution to this halting problem for logic program synthesis.

Before introducing our main ideas, we should first comment on the deductive method in general. The basic idea of the deductive method is neither new nor specific to computing since it was already to be found in Euclides' work 300 B.C. An important development took place around the turn of the century when the Italian mathematician Peano formalised arithmetic, $\mathcal{PA}$ for short, as a model for the deductive method. We shall often use $\mathcal{PA}$ in our framework below.

Several developments of program synthesis have taken advantage of the deductive method, for example Manna and Waldinger [13]. For logic programming, Hogger [8], Hansson and Tärnlund [7], and Lau and Prestwich [10] derived correct logic programs, while Clark and Tärnlund [5] developed a Peano-like theory for logic programs. More recently, Lau and Ornaghi [11] formalised deductive synthesis of logic programs and gave an incompleteness result.

Clark's idea of the completion of a program [3] was originally conceived to justify semantically the inference rule 'negation as failure' in logic programming. As we shall see, the notion of completion turns out to be a profound concept in deductive synthesis. It is the basis for a halting principle for deductive synthesis, and for the notion of steadfast programs.

In Section 2 we describe the basic notation that we shall employ, and define a general specification framework and the notion of synthesising correct, total, and terminating programs. Section 3 takes up Clark's completion of a program. We shall split the completion into two sub-parts, the *if*-part and the *only-if*-part. We use the former to establish correctness criteria, and the latter for a halting criterion for deductive synthesis. In Section 4, we define the halting problem and show a *halting principle* for deductive synthesis. In Section 5, we define steadfast programs, and show that they are desirable programs of a deductive synthesis. We also state a halting principle for the synthesis of steadfast programs.

2 Basic Notation and Terminology

In this section, we describe the basic notation and terminology that we shall use in this paper.

We will use *first-order classical logic with identity* (i.e. the identity axioms will be assumed), interpreted according to the usual model theoretic semantics (see e.g. [2]).

We shall use H, K, R to denote formulas. $H(x)$ will indicate that the *free variables* of H are x. We use x, y, z, w to stand for (tuples of) variables; p, q, r for *predicate symbols*; f, g, h for *function symbols*; t for (tuples of) *terms*; $\forall(H)$ and $\exists(H)$ for the universal and existential closure of H.

$\mathcal{A}, \mathcal{B}, \ldots$, will denote axiomatisations, i.e. sets of closed formulas, and we will use '$\vdash$' to mean provability in classical logic, while $\mathcal{F} \models H$ will indicate that H is a logical consequence of $\mathcal{F}$.

For logic programs, we adopt the definitions and the formalism of standard works such as [4, 9, 12]. Atoms will be denoted by $A, A_1, \ldots$, clauses by $C, C_1, \ldots$, and substitutions by $\sigma, \theta, \ldots$ Programs will be denoted by $P, Q, P_1, \ldots$ For every computed answer substitution θ, we also consider as computed answers the more instantiated substitutions $\theta\gamma$ (by the Lifting Lemma).

2.1 Specifications

In our approach, a specification is given in a general formal framework which axiomatises the problem domain in question, thereby allowing us to specify and synthesise programs for computational problems in that domain. We have already mentioned Peano Arithmetic, which we will denote by $\mathcal{PA}$, as a good example of such a framework for natural numbers. Its language contains identity '$=$', successor 's', sum '$+$' and product '$\cdot$'. In $\mathcal{PA}$ every natural number $\mathbf{n}$ is represented by a *numeral* $s^n(0)$, and every computable function $f(x)$ can be expressed by a formula $F(x, z)$ such that, for every pair of natural numbers $\mathbf{m}$ and $\mathbf{n}$, $\mathbf{m} = f(\mathbf{n})$ iff $F(m, n)$, where m and n are the numerals corresponding to $\mathbf{m}$ and $\mathbf{n}$ respectively, is provable in $\mathcal{PA}$. Thus, in $\mathcal{PA}$ we can specify any computable function by a formula. Other examples of first-order theories interesting for Computer Science are presented in [14].

Thus we define a specification in a general framework as follows:

Definition 2.1 A *specification framework* is a first-order theory with identity, and we shall assume that it is consistent. We will denote specification frameworks by $\mathcal{F}, \mathcal{G}, \ldots$.

In a framework $\mathcal{F}$ with language $\mathcal{L}_\mathcal{F}$, a *specification* consists of:

- a *definition axiom* D_r:

$$\forall\,(r(x) \leftrightarrow R(x))$$

  where r is a new predicate symbol[1] and R is a formula in $\mathcal{L}_\mathcal{F}$. That is D_r defines r, the relation to be computed.

- a set G of *goals* of the form $\exists r(t_0), \exists r(t_1), \ldots$

An *answer* of $\exists r(t) \in G$ in $\mathcal{F}$ is any $r(t\sigma)$ such that $\mathcal{F} \cup D_r \models r(t\sigma)$, where σ is a substitution. $\odot$

Example 2.1 In the framework $\mathcal{PA}$, we can specify the problem of finding the square root, if it exists, of a given numeral as follows:

- $D_{sqrt} \;=\; \forall\,(sqrt(x, y) \leftrightarrow y \cdot y = x)$
- $G = \{\exists z\, sqrt(n, z) \mid n \text{ is a numeral}\}$

The goal $\exists z\, sqrt(s(0), z)$ has the unique answer $sqrt(s(0), s(0))$, for example, but the goal $\exists z\, sqrt(s(s(0)), z)$ has no answer, and so on. $\diamond$

[1]This preserves consistency. We will also use symbols in $\mathcal{L}_\mathcal{F}$, without introducing definition axioms for them.

2.2 Logic Program Synthesis

Given the specification of a problem in a framework $\mathcal{F}$, the goal of logic program synthesis is to find a program P which is *correct*, *total*, and *terminating* for every goal $\exists r(t) \in G$, according to the following definition:

Definition 2.2 A program P is (*partially*) *correct* for a goal $\exists r(t)$ if for every computed answer substitution σ, $\mathcal{F} \cup D_r \models r(t\sigma)$.

P is *total* for a goal $\exists r(t)$ if for every ground substitution σ, $\mathcal{F} \cup D_r \models r(t\sigma)$ entails that σ is a computed answer substitution of P for $\exists r(t)$.[2]

P *terminates* for a goal $\exists r(t)$ iff the *SLD*-tree[3] of P with root $\leftarrow r(t)$ is finitely failed, or it contains at least one success node. $\odot$

For convenience, we shall say P is *totally correct* if it is partially correct and total for each goal in G.

Example 2.2 Consider **Example 2.1** and the following trivial program P

$$\begin{array}{rl} sqrt(0,0) & \leftarrow \\ sqrt(s(0),s(0)) & \leftarrow \end{array}$$

P is correct and terminates for every goal, but it is not total. Indeed, for the goal

$$\exists z\ sqrt(s(s(s(s(0)))),z)$$

the answer in $\mathcal{PA}$ is $sqrt(s(s(s(s(0)))),s(s(0)))$, but P has a finitely failed *SLD*-tree and hence no answer substitution. $\diamond$

Synthesis aims to build a program P which is totally correct and which also terminates for any goal $\exists r(t)$. In this paper, we shall consider totality but not termination. Termination of P is clearly desirable if P is to be of practical use. However, we shall not address the important and non-trivial issue of how to synthesise programs that are guaranteed to terminate. Instead, we shall assume that synthesis gives rise to P that terminates for any goal $\exists r(t)$. This implies the decidability of '$\forall(P) \vdash \exists r(t)$': indeed, '$\forall(P) \vdash \exists r(t)$' is either true or 'finitely' false (see **Proposition 3.1** later). Otherwise, if P does not terminate, then '$\forall(P) \vdash \exists r(t)$' is 'infinitely' false.

If synthesis is done in an iterative manner, i.e. we derive $P_1 \subseteq P_2 \subseteq \cdots$, as in deductive synthesis (see Section 4), then it is natural (and usual) to ensure that partial correctness of each P_i are guaranteed by the synthesis steps. However, totality of P_i cannot be attained until the final step (which yields P_n say). Thus the problem of ensuring that P_n is total is equivalent to the problem of determining when a synthesis can halt. This halting problem is our main concern in this paper.

Definition 2.3 We say that synthesis *successfully halts* if it stops when a total program (P_n) has been synthesised. $\odot$

Thus it is of vital importance to have a suitable criterion for successfully halting synthesis. We will propose one such criterion in Section 3.1.

[2]We use *total* instead of the commonly used *complete*.

[3]We assume a complete search strategy.

3 Correctness Criteria for Deductive Synthesis

In this section, we propose a halting criterion based on Clark's completion $Comp(P)$ of a program P [3]. We normally assume that every predicate symbol $r(x)$ of the current program P is introduced in the framework $\mathcal{F}$ by a definition axiom

$$\forall\, (r(x) \leftrightarrow R(x))$$

We will use D_P to denote the set of such definition axioms for P.

In case some predicate r in P cannot be defined by a definition axiom, then our framework $\mathcal{F}$ is too weak to characterise the relation we want to compute. We will briefly address this and other issues related to weak frameworks in Section 5.1.

3.1 A Halting Criterion based on Clark's Completion

For a program P, we shall decompose its completion $Comp(P)$ into two parts $Comp^+(P)$ and $Comp^-(P)$. First, we define $free(P)$ to be the freeness axioms for the constant and function symbols of P [15]. Informally, these axioms ensure the unique meanings of these symbols. Now we define $Comp^+(P)$ and $Comp^-(P)$ as follows:

- $Comp^+(P) = free(P) \cup Ax(P)$, where $Ax(P)$ is the set $\{Ax(C) \mid C$ is a clause of $P\}$ defined as follows:

  $$\begin{array}{rcl} Ax(A \leftarrow) & \text{is} & \forall(A) \\ Ax(A \leftarrow A_1, \ldots, A_k) & \text{is} & \forall(A \leftarrow A_1 \wedge \cdots \wedge A_k)\ . \end{array}$$

  $Ax(P)$ is logically equivalent to the *if*-part of the completion, and the successful computations of P can be interpreted as proofs in classical logic with identity, using axioms from $Ax(P)$.[4]

- $Comp^-(P)$ is the *only-if*-part of $Comp(P)$, together with $free(P)$. It contains, apart from $free(P)$, for every predicate symbol r in P,

  $$\forall\, (r(x) \rightarrow E_1 \vee \cdots \vee E_k) \tag{1}$$

  which is the *only-if*-part of the completed definition of r. Note that if $k = 0$, i.e. if there is no clause in P with r in its head, then (1) is $\forall \neg r(x)$.

  For convenience, we shall denote (1) by $Comp^-(P, r)$, to be read as 'the *only-if*-part of r in P'. We shall also write $Comp^-(P, r_1, \ldots, r_n)$ for multiple predicate symbols.

  $Comp^-(P)$ is needed to interpret any finitely failed computation of P as a proof in classical logic with identity, of the negation of the goal.[5]

Note that $Comp^+(P)$ is *monotonic*, i.e. $P \subseteq Q$ implies $Comp^+(P) \subseteq Comp^+(Q)$, whereas $Comp^-(P)$ is *non-monotonic*, i.e. there are $P \subseteq Q$ such that $Comp^-(P) \not\subseteq Comp^-(Q)$.

---

[4] We call $Ax(P)$ *success axioms* in [11].

[5] We call $Comp^-(p)$ *failure axioms* in [11], where we use the *contrapositive* form $\forall\, (\neg p(x) \leftarrow \neg E_1 \wedge \cdots \wedge \neg E_k)$.

Example 3.1 Consider the program P:

$$\begin{array}{rcl} sum(x,0,x) & \leftarrow & \\ sum(X,s(y),s(z)) & \leftarrow & sum(x,y,z) \end{array}$$

$free(P)$ contains the freeness axioms for the constant symbol '0' and the function symbol 's':

$$\begin{array}{l} \forall \neg(0 = s(x)) \\ \forall\,(s(x) = s(y) \rightarrow x = y) \end{array}$$

These axioms guarantee the unique meanings of '0' and 's'.

- $Comp^{+}(P) = free(P) \cup Ax(P)$ where $Ax(P)$ contains:

  $$\begin{array}{l} \forall\, sum(x,0,x) \\ \forall\,(sum(x,s(y),s(z)) \leftarrow sum(x,y,z)) \end{array}$$

  These axioms characterise the computations of P that will (successfully) solve $\exists sum(t_1,t_2,t_3)$.

- $Comp^{-}(P)$ contains $free(P)$ as well as:

  $$\begin{array}{l} \forall\,(sum(a,b,c) \rightarrow \exists x\,(a = x \wedge b = 0 \wedge c = x) \vee \\ \quad \exists x,y,z\,(a = x \wedge b = s(y) \wedge c = s(z) \wedge sum(x,y,z)) \end{array}$$

  Computations of P that will finitely fail to solve $sum(t_1,t_2,t_3)$ can be interpreted as proofs of $\neg\exists sum(t_1,t_2,t_3)$ using this axiom and $free(P)$.

$\diamond$

Now, before we give the halting criterion, we note the following proposition on termination, and state a theorem on partial correctness:

Proposition 3.1 If a program P terminates for $\exists r(t)$, then either $Comp(P) \vdash \exists r(t)$ or $Comp(P) \vdash \neg\exists r(t)$.

Proof. This follows readily from the definition of termination and the soundness of completion. $\square$

Theorem 1 *If*

$$\mathcal{F} \cup D_P \models \mathrm{Comp}^{+}(P)$$

then P is partially correct for every goal $\exists r(t)$ for any term t.

Proof.

$$\begin{array}{lll} & \sigma \text{ is a computed answer substitution of } \leftarrow r(t) & \\ \Rightarrow & Comp^{+}(P) \models r(t\sigma) & \text{(by soundness)} \\ \Rightarrow & \mathcal{F} \cup D_P \models r(t\sigma) & \text{(by cut)} \\ \Rightarrow & \mathcal{F} \cup D_r \models r(t\sigma) & (\mathcal{F} \cup D_P \text{ is a conservative extension of } \mathcal{F} \cup D_r) \end{array}$$

$\square$

Now we can state our halting criterion:

Theorem 2 *For a program P, if $\mathcal{F} \cup D_P \models \mathrm{Comp}(P)$, and P terminates for a goal $\exists r(t)$, then P is total for $\exists r(t)$.*

Proof. Since P terminates for the goal $\exists r(t)$, we have either

(a) that there is a finitely failed *SLD*-tree, or

(b) that the *SLD*-tree contains at least one success node.

In the case of (a), $Comp(P) \models \neg\exists r(t)$ holds; since $\mathcal{F} \cup D_P \models Comp(P)$, we have:

$$\mathcal{F} \cup D_P \models \neg\exists r(t)$$

and there is no answer and we are all right.

In the case of (b), assume, by *absurdum*, that

> $\mathcal{F} \cup D_P \models r(t\sigma)$, for some ground σ, but σ is not a computed answer substitution of P.

This means that no success node with computed answer substitution σ exists in the *SLD*-tree. However, this contradicts the completeness of *SLD*-resolution, since $r(t\sigma)$ is a logical consequence of $Comp^+(P)$. The fact that it is a logical consequence of $Comp^+(P)$ can be proved by absurdum as follows.

Assume by absurdum that $Comp^+(P) \not\vdash r(t\sigma)$. Since P terminates and $r(t\sigma)$ is ground, by **Proposition 3.1** $Comp(P) \vdash r(t\sigma)$ or $Comp(P) \vdash \neg r(t\sigma)$, and hence:

$$Comp(P) \vdash \neg r(t\sigma)$$

Since $\mathcal{F} \cup D_P \vdash Comp(P)$ we obtain $\mathcal{F} \cup D_P \vdash \neg r(t\sigma)$, and hence absurdum due to the consistency of $\mathcal{F} \cup D_P$ (indeed a framework $\mathcal{F}$ is consistent by definition and definition axioms give rise to conservative extensions). □

Intuitively, we can explain **Theorem 2** as follows: a partially correct (and terminating) P is not total unless all its failed computations ($Comp^-(P)$) are also logical consequences of $\mathcal{F} \cup D_P$. In other words, if P is partially correct and only fails when it is supposed to, then it is total.

4 The Halting Problem for Deductive Synthesis

In this section, we shall use the halting criterion to state a halting principle which tells us precisely when we can stop synthesis knowing that we have synthesised a total program.

Let $\mathcal{F}$ be a framework and consider a specification consisting of a definition axiom D_0

$$\forall\,(r(x) \leftrightarrow R(x)) \tag{2}$$

and a set G of goals.

A *deductive synthesis* process (for this specification) generates a possibly infinite sequence of programs

$$P_1 \subseteq P_2 \subseteq \cdots \subseteq P_k \subseteq \cdots \tag{3}$$

where, for every P_j, there is a corresponding set D_j of definition axioms, introducing the new predicate symbols of P_j .

Now, for partial correctness, we require that, for every P_j:[6]

$$\mathcal{F} \cup D_j \vdash Comp^+(P_j) \tag{4}$$

[6]Thanks to completeness, we can use provability and hence do synthesis in the theory.

By **Theorem 1**, (4) implies partial correctness. $\mathcal{F} \cup D_j \vdash Ax(P_j)$ alone would be sufficient to ensure partial correctness. However, we require (4) to hold since $\mathit{free}(P)$ is monotonic (if (4) does not hold in a step, then it will never hold in the subsequent steps, in particular the final step).

Example 4.1 Let $\mathcal{F}$ be $\mathcal{PA}$ and consider the specification

$$\begin{array}{rcl} D_{prod} & : & \forall\,(\mathit{prod}(x,y,z) \leftrightarrow x \cdot y = z) \\ G_{prod} & : & \{\exists z\,\mathit{prod}(n,m,z)\,|\,n,m \text{ ground}\} \end{array}$$

A partial deductive synthesis process would be the following:

1. Synthesise a clause for the goal $\mathit{prod}(x,0,z)$. In $\mathcal{PA} \cup D_{prod}$ we can prove:

   $$\mathit{prod}(x,0,z) \leftrightarrow x \cdot 0 = z \leftrightarrow z = 0$$

   Thus $\mathcal{PA} \cup D_{prod} \vdash \mathit{Comp}^{+}(\mathit{prod}(x,0,z) \leftarrow z = 0)$ and we get the program P_1:

   $$\mathit{prod}(x,0,z) \leftarrow z = 0$$

2. Next synthesise a clause for $\mathit{prod}(x,s(y),z)$. In $\mathcal{PA} \cup D_{prod}$ we can prove:

   $$\mathit{prod}(x,s(y),z) \leftrightarrow x \cdot s(y) = z \leftrightarrow x \cdot y + x = z \leftrightarrow \exists w\,(\mathit{prod}(x,y,w) \wedge w + x = z)$$

   From this we could synthesise the clause $\mathit{prod}(x,s(y),z) \leftarrow \mathit{prod}(x,y,w), w + x = z$. However, '+' cannot be used in a program, since it does not satisfy the freeness axioms. Therefore we have to introduce a new relation sum defined by D_{sum}:

   $$\forall(\mathit{sum}(x,y,z) \leftrightarrow x + y = z)$$

   and we get $D_2 = \{D_{prod}, D_{sum}\}$ and we have

   $$\mathcal{PA} \cup D_2 \vdash \mathit{Comp}^{+}(\mathit{prod}(x,s(y),z) \leftarrow \mathit{prod}(x,y,w), \mathit{sum}(w,x,z))$$

   That is our new program P_2 is:

   $$\begin{array}{rcl} \mathit{prod}(x,0,z) & \leftarrow & z = 0 \\ \mathit{prod}(x,s(y),z) & \leftarrow & \mathit{prod}(x,y,w), \mathit{sum}(w,x,z) \end{array}$$

$\diamond$

Now the problem with the program P_2 is that although it is partially correct, it is not total. So we need further synthesis steps. To recognise halting, we use **Theorem 2**, and the halting criterion is the following:

> If $\mathcal{F} \cup D_n \vdash \mathit{Comp}^{-}(P_n, p)$ for every predicate p of P_n, then the synthesis can successfully halt.[7]

Looking at **Example 4.1**, we see that

$$\mathcal{PA} \cup D_2 \vdash \mathit{Comp}^{-}(P_2, \mathit{prod})$$

but

$$\mathcal{PA} \cup D_2 \vdash \neg\mathit{Comp}^{-}(P_2, \mathit{sum})$$

(indeed $\mathit{Comp}^{-}(P_2, \mathit{sum})$ is $\forall\,\neg\mathit{sum}(x,y,z)$).

Thus we have to go on with the synthesis, searching for clauses for sum.

This example shows that the halting criterion can be used in a subtle way. This leads to the following principle:

[7] This is sufficient to obtain $\mathit{Comp}(P_n)$, since $\mathit{Comp}^{+}(P_n)$ has already been proved.

The Halting Principle
If $\mathcal{F} \cup D_n \vdash Comp^-(P_n, p)$ for a predicate p, do not search for any more clauses for p. Furthermore, if this holds for every predicate p of P_n, then synthesis can successfully halt.

It follows directly from **Theorem 2** that this principle holds.

For instance, for **Example 4.1** we can stop synthesis for *prod*, but not for *sum*, and synthesis for *sum* can be summarised as follows:

1. $\mathcal{PA} \cup D_2 \vdash Comp^+(sum(x, 0, x))$, then synthesise P_3 containing this new clause; continue synthesis because $\mathcal{PA} \cup D_2 \vdash \neg Comp^-(P_3, sum)$.

2. $\mathcal{PA} \cup D_2 \vdash Comp^+(sum(x, s(y), s(w)) \leftarrow sum(x, y, w))$, then synthesise P_4 containing this new clause; now synthesis successfully halts because $\mathcal{PA} \cup D_2 \vdash Comp^-(P_4, sum)$.

Our synthesis method has several desirable modularity properties that can be used in practice, for example in an implementation of a synthesis system. $Comp^+(P)$ and $Comp^-(P)$ can be divided into sub-parts, viz. $free(P)$, $Ax(C)$ for every clause C and $Comp^-(P, p)$ for each predicate p in P.

5 Steadfast Programs

Theorem 2 says that the provability of $Comp^-(P_n)$ is a *sufficient* condition for halting synthesis with a totally correct P_n. A natural question then arises as to whether or not it is also a *necessary* condition. In this section, we shall show that the answer is negative for the entire class of synthesised programs. However, if we consider the synthesis of what we shall call *steadfast programs*, which are the programs we consider to be desirable, then the provability of completion is a sufficient and necessary condition for halting synthesis.

The notion of the steadfastness of a program arises in the context of an *open* specification framework, namely one which does not completely specify the program. We shall first informally introduce and discuss the concepts of *open frameworks* and *steadfast programs*, and show by examples that, if in our synthesis we search for steadfast programs, then the provability of completion also becomes a *necessary* condition for successfully halting synthesis. This enables us to state a halting principle for the synthesis of steadfast programs. Then we give a formal treatment of steadfastness in open frameworks, which enables us to establish formally that the provability of completion is indeed a *necessary* condition for successfully halting the synthesis of steadfast programs.

5.1 Open Frameworks and Steadfast Programs

Informally speaking, an *open* (or *loose*) framework leaves open the possibility of different interpretations of the symbols of its language, corresponding to different intended models, while a *closed* framework has a unique intended model, determining a unique interpretation.

In an open framework, it is convenient to distinguish between the *defined* and *defining* predicate symbols of a program P in the following manner. We will say that P is of type $r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$, written $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$, to indicate that $r_1, \ldots, r_k$ are the *defined* predicates of P, namely the ones occurring in the head of at least one clause, and $q_1, \ldots, q_m$ are *defining* predicates, namely the ones occurring only in the bodies.

In such a framework, it is meaningful and interesting to study the possibility of synthesising a program $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ that contains *all* the clauses needed to compute $r_1, \ldots, r_k$, but not necessarily any clauses to compute $q_1, \ldots, q_m$. That is, such a P will use the same set of clauses to compute $r_1, \ldots, r_k$ in a different way for every model of the framework, depending on the interpretation of $q_1, \ldots, q_m$. Such programs will be called *steadfast* in that framework.[8]

We use 'steadfast' in the ordinary sense of being unchanging, because a steadfast program P has the following important characteristic: for every model of our framework, if we have a program Q that correctly computes the predicates not defined by P, then $P \cup Q$ is also a correct program in that model. We note that for different models we have different Q's,[9] whilst P is always the same. Thus $P \cup Q$ represents a *class* of correct programs. Hence a steadfast program has the desirable property of *reusability*.

Now we will show in **Example 5.1** that completion enables us to identify steadfast programs. In other words, completion can be used to state both a *sufficient* and a *necessary* halting condition if our aim is to synthesise steadfast, and hence reusable, programs.

Example 5.1 Consider the (open) framework $\mathcal{F}$:[10]

$$\{q(a) \vee q(b), \forall\,(p(x) \leftrightarrow x = c), \neg a = b, \neg b = c\}$$

and the specification:

$$\begin{array}{lcl} D_r & : & \forall\,(r(x) \leftrightarrow p(x) \vee q(x)) \\ G & : & \{\exists x\, r(x)\} \end{array}$$

We can easily see that the only answer of $\exists x\, r(x)$ that is correct in $\mathcal{F}$ is $r(c)$. Therefore, the program P_1:

$$r(c) \leftarrow$$

is totally correct and terminating for the goal considered and we should halt synthesis immediately. However, $Comp^-(P_1, r)$ is

$$\forall\,(r(x) \rightarrow x = c)$$

and we can easily see that

$$\mathcal{F} \cup D_r \models \neg Comp^-(P_1, r)$$

Thus, the provability of $Comp^-(P_1, r)$ is not a necessary condition for synthesising the totally correct program P_1. This is due to the following.

Our framework $\mathcal{F}$ is *open* wrt the predicate q occurring in the definition axiom of r. Therefore there are many models of $\mathcal{F}$, interpreting q in different ways, and, for every model M, D_r defines an expansion N of M by a new relation r. For

[8] Note that a steadfast program may or may not be total.

[9] For example, Q may be a set of facts in a database.

[10] This example shows that our notion of specification frameworks is very general. In particular, we allow the use of examples as (incomplete) specifications.

example, M_1, M_2, M_3 as defined below are Herbrand models of $\mathcal{F}$ with expansions (by r) N_1, N_2, N_3 respectively:[11]

$$\begin{array}{ll} M_1 = \{p(c), q(a)\} & N_1 = \{p(c), q(a), r(a), r(c)\} \\ M_2 = \{p(c), q(b)\} & N_2 = \{p(c), q(b), r(b), r(c)\} \\ M_3 = \{p(c), q(a), q(f(f(a))\} & N_3 = \{p(c), q(a), q(f(f(a)), r(c), r(a), r(f(f(a))\} \end{array} \tag{5}$$

This illustrates that $\mathcal{F} \cup D_r$ is open wrt r, and r depends on q. In contrast, the completion of P_1 is closed wrt r and gives rise to the interpretation $r(c)$, which is at variance with the possible interpretations stated by the definition axiom D_r.

Another correct program, in which r depends on q, as stated by D_r, is the following program P_2:

$$\begin{array}{rcl} r(x) & \leftarrow & p(x) \\ r(x) & \leftarrow & q(x) \\ p(c) & \leftarrow & \end{array}$$

We consider P_2 to be a better program than P_1 because P_2 is steadfast, which means that P_2 has the following reusability property: P_2 can be extended to a program in each of the expansions N_1, N_2, N_3 in (5) (or more generally, in each model of $\mathcal{F} \cup D_r$), without changing the clauses for computing r and p, but by simply adding the right clauses for q, such that each extended version of P_2 is totally correct and terminating.

Steadfastness of P_2 is guaranteed by the fact the completion axioms of its defined predicates, namely $Comp^-(P_2, r)$ and $Comp^-(P_2, p)$, are provable. (Indeed, they are $\forall\,(r(x) \rightarrow p(x) \vee q(x))$ and $\forall\,(p(x) \rightarrow x = c)$, and are both immediate.) On the other hand, we cannot prove the whole of $Comp^-(P_2)$, since it contains $\forall x \neg q(x)$ and $\mathcal{F} \vdash \exists x\, q(x)$. This corresponds to the fact that $\mathcal{F}$ is open wrt q and therefore we must consider only the part $Comp^-(P_2, r, p)$ which is open wrt q. Therefore, to halt the synthesis of a steadfast program, we have to prove the completion of its defined predicates. ◇

This example shows that in synthesis we can go beyond correctness, i.e. we should and can synthesise the steadfast programs, for which the completion is provable for the symbols defined by them. In the next subsection we shall demonstrate that the converse holds equally, namely that steadfastness implies the provability of the completion of the defined symbols.

Now we can state the following halting principle for the synthesis of steadfast programs:

The Halting Principle (for steadfast programs)
For every predicate p defined by the current program P_n:

(i) if $\mathcal{F} \cup D_n \vdash \neg Comp^-(P_n, p)$, search for other clauses with p in their head;

(ii) if $\mathcal{F} \cup D_n \vdash Comp^-(P_n, p)$, do not search for any more clauses defining p.

If (ii) holds for every defined predicate p of P_n, then synthesis successfully halts.

(If neither (i) nor (ii) holds, then no steadfast program exists for this specification. This can arise in weak frameworks (see below).)

[11] L_{M_3} contains a new function symbol f, i.e. we consider models M of $\mathcal{F}$ with language $L_M \supseteq L_{\mathcal{F}}$.

It is worth pointing out that the Halting Principle applies equally to the synthesis of steadfast programs in both open and closed frameworks. For example, in **Example 4.1**, where we use the closed framework $\mathcal{PA}$, the Halting Principle says that we could successfully halt synthesis wrt *prod*, without synthesising any clause for *sum*. This means that the part synthesised for *prod* works as a steadfast program, never changed by subsequent synthesis steps.

In general, in a closed framework, we try to continue synthesis until we have reached the final totally correct program. By **Theorem 2** this happens when P_n is of type $r_1, \ldots, r_k \Leftarrow$, i.e. when it does not contain defining predicates.

On the other hand, as we saw in **Example 5.1**, in an open framework the Halting Principle tells us to halt synthesis when P_n involves only defining predicates $q_1, \ldots, q_m$ which belong to the language of the framework, and are open in it. The result of the synthesis is a steadfast program $P_n : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$. If $q_1, \ldots, q_m$ have no provable instances in the framework, then P_n is a total program.

In Section 5.2, we will give a formal treatment of the halting problem for synthesis in open frameworks. To conclude this section, it is worthwhile to briefly comment on frameworks which are too weak with respect to a specification. This may happen when the framework is too weak to prove the completion of a program, or when the relation to be computed cannot be specified in the framework by a definition axiom.

In the first case, we reach a program P_n where $\neg Comp^-(P_n, r)$ (for some defined r) is no longer provable. So we try to prove $Comp^-(P_n, r)$, but we discover that $Comp^-(P_n, r)$ too cannot be proved. In this case we can consistently add $Comp^-(P_n, r)$ to the axioms of our framework. By adding it, we eliminate some models of the framework. If the eliminated models are undesirable ones, we are happy to discard them and thereby improve our formalisation of the problem domain. Otherwise we have to reconsider our formalisation of the problem. That is, if and when we discover that no steadfast program exists for this specification, we have to determine whether we have discovered an improvement to the framework or there is some drawback in our formalisation.

In the second case, we cannot define the relation r to be computed by a definition axiom D_r. For example, if our framework is Presburger Arithmetic (where we have successor and sum, but not product), we cannot define product. In this case we have to enrich our framework, adding a 'fresh' symbol '$\cdot$' and new axioms, which codify the usual recursive definition of product. Once we have '$\cdot$', we obtain $\mathcal{PA}$. Note that in $\mathcal{PA}$ this kind of problems no longer occur, since every partial recursive function can be defined in it by a definition axiom.

These two points show that in deductive synthesis there is a feedback mechanism, where synthesis can also help us to understand better, improve and enrich a specification framework. With each loop, hopefully such feedback will become more rare, i.e. step by step, we obtain axiomatisations sufficiently strong for handling larger and larger classes of problems.

5.2 Completion is Necessary for Halting Synthesis

Now we give a formal characterisation of steadfastness, and show formally the link between steadfastness and reusability. Then we discuss the Halting Principle for steadfast programs, and establish formally that the provability of the completion of the defined symbols is a *necessary* condition for halting the synthesis of steadfast programs.

For the sake of simplicity we will consider only *free frameworks*, namely frameworks that satisfy the freeness axioms of the symbols of their language; and *free models*, namely models satisfying the freeness axioms of the symbols of their language.[12]

The notion of steadfast programs is related to a class of models, according to the following definition:

Definition 5.1 A program P is *steadfast* in a class $\mathcal{M}$ of models iff every model $M \in \mathcal{M}$ is a steadfast model (as defined below) of P. $\odot$

Now, we define the steadfast models of P. Intuitively, a steadfast model of $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ is a model which interprets the defined predicates according to P, depending on the interpretation of the defining predicates. To give a precise definition, we use *diagrams* as follows.

Let L_M be a first-order language and M a model for L_M (i.e. a first-order structure interpreting L_M). We say that M is *reachable* (in L_M) if every element of the domain of M is denoted by a ground term of L_M. Henceforth we will deal with reachable models only. This is not restrictive, since we allow 'ideal' L_M's of arbitrary cardinality.

Let r be a relation symbol of L_M. The *positive diagram*, *negative diagram*, and *diagram* of r in M are respectively defined by:

$$diag^+(M, r) = \{r(t) \mid t \text{ are ground terms and } M \models r(t)\}$$

$$diag^-(M, r) = \{\neg r(t) \mid t \text{ are ground terms and } M \models \neg r(t)\}$$

$$diag(M, r) = diag^+(M, r) \cup diag^-(M, r)$$

For many relations, $diag^+(M, r_1, \ldots, r_k)$, $diag^-(M, r_1, \ldots, r_k)$, $diag(M, r_1, \ldots, r_k)$ are defined in the obvious way. For convenience we shall call $diag^+(M)$, $diag^-(M)$, $diag(M)$ collectively *diagrams* of M, i.e. the diagrams of all the relation symbols of L_M.

Since we are only dealing with reachable, free models, $diag^+(M)$ is the Herbrand model corresponding to M, and we define a steadfast model as follows:

Definition 5.2 M is a *steadfast model* of $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ iff the diagram $diag^+(M, r_1, \ldots, r_k, q_1, \ldots, q_m)$ is the minimum Herbrand model of $Comp^+(P)$ $\cup diag^+(M, q_1, \ldots, q_m)$. $\odot$

Example 5.2 Consider the programs P_1 and P_2 of **Example 5.1**. Their types are $P_1 : r \Leftarrow$ and $P_2 : p, r \Leftarrow q$ respectively. The model $N_1 = \{p(c), q(a), r(a), r(c)\}$ is a steadfast model of P_2; indeed, $diag^+(N_1, q) = \{q(a)\}$ and N_1 is the minimum model of $P_2 \cup \{q(a)\}$. A model that is not a steadfast model of P_2 is, for example, $\{p(c), q(a), r(a), r(b), r(c)\}$, since it properly contains N_1 and q has the same diagram in the two models.

P_2 is steadfast in N_1, N_2, N_3, whilst P_1 is not steadfast in any of them (i.e. none of them is a steadfast model of P_1). $\diamond$

Now, steadfastness in a class of models implies reusability. A program Q 'reusing' a steadfast $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ must correctly compute $q_1, \ldots, q_m$, but in general it may contain other predicates. To discuss Q's total correctness, it will be more appropriate to talk about total correctness in a model, as opposed to total correctness in a framework as defined in **Definition 2.2** (in Section 2.2),

[12] This restriction can be avoided by using isoinitial models (see [1]) instead of canonical Herbrand models that we will consider here.

by substituting truth in a model for validity in a framework. Thus, we will say that Q is *totally correct wrt* $q_1, \dots, q_m$ *in a model* M if $q_1, \dots, q_m$ are the predicate symbols of Q belonging to L_M, and Q is totally correct in M wrt $q_1, \dots, q_m$, namely $diag^+(M, q_1, \dots, q_m)$ coincides with the success set of Q restricted to $q_1, \dots, q_m$.

Then we have the property:

Property 5.1 Let a program $P : r_1, \dots, r_k \Leftarrow q_1, \dots, q_m$ be steadfast in a class $\mathcal{M}$ of models and let M be any model belonging to $\mathcal{M}$. For every program Q_M totally correct wrt $q_1, \dots, q_m$ in M, if $P \cup Q_M$ terminates, then $P \cup Q_M$ is totally correct in M wrt $r_1, \dots, r_k, q_1, \dots, q_m$.

In other words, if a program $P : r_1, \dots, r_k \Leftarrow q_1, \dots, q_m$ is steadfast in a class of models, then it can be correctly *re-used* in different larger programs Q_M, which also compute $q_1, \dots, q_m$ according to a chosen model M. Thus steadfastness of P works as a kind of 'parametric correctness' in a class of models, where P assumes in each model M the appropriate behaviour that depends on Q_M.

It is interesting to observe that Deville [6] defined a similar notion of correctness for a logic procedure $LP(p)$ that defines a predicate p. His criterion preserves what he calls the monotonicity of correctness of $LP(p)$, which corresponds to our notion of steadfastness of $LP(p)$.

Now we return to the Halting Principle for steadfast programs. Our aim is to show that the provability of the completion of the defined symbols works as a necessary and sufficient condition for halting with steadfast programs.

First of all, we need a precise definition of closed and open (free) frameworks.

Definition 5.3 A (free) framework $\mathcal{F}$ is *atomically complete* iff $\mathcal{F} \models A$ or $\mathcal{F} \models \neg A$, for every ground atom A.

$\mathcal{F}$ is *closed* if it is atomically complete and has at least one model that is reachable in the language $L_{\mathcal{F}}$. $\odot$

If $\mathcal{F}$ is closed and M is a model of $\mathcal{F}$ that is reachable in $L_{\mathcal{F}}$, then $diag^+(M)$ is the *unique* Herbrand model of $\mathcal{F}$,[13] which we call the *canonical model*.

Moreover, every ground atomic formula $r(t)$ is true in the canonical model iff $\mathcal{F} \models r(t)$; hence the definition of total correctness in $\mathcal{F}$ given in **Definition 2.2** means total correctness in the canonical model of $\mathcal{F} \cup D_r$.[14]

A (free) framework $\mathcal{F}$ is *open* if it is consistent and is not closed. An open framework thus formalises a class of canonical models. Indeed, for every closed (free) framework $\mathcal{F}^* \supseteq \mathcal{F}$ (with $L_{\mathcal{F}^*} \supseteq L_{\mathcal{F}}$), we can consider the canonical model of $\mathcal{F}^*$ as a canonical model of the class axiomatised by $\mathcal{F}$.

Now for classes of models, we have to parameterise the notion of total correctness. From the previous discussion, the 'correct', i.e. desirable, programs are the programs which are *steadfast* in the class of models of the framework (or in a significant subclass).

To state a sufficient condition for halting the synthesis of steadfast programs, we consider the subclass of $(q_1, \dots, q_m)$-*models*, namely the models M such that there is at least a finite program Q totally correct wrt $q_1, \dots, q_m$ in M. This restriction is sensible, since it corresponds to the models where the program P can be used in a finite, totally correct, and terminating program Q. Note that if $diag^+(M, q_1, \dots, q_m)$ is finite (e.g. if it is the set of facts in a data base), then $diag^+(M, q_1, \dots, q_m)$ itself is the finite program Q.

---

[13] We can prove that any two models of a closed framework $\mathcal{F}$ reachable in $L_{\mathcal{F}}$ are isomorphic.

[14] We assume that D_r preserves the canonical model. See also the discussion on closed frameworks at the end of this section.

Furthermore, we shall also need a notion of *parametric termination.*

Definition 5.4 We say a program $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ *parametrically terminates* if, for every program Q which does not contain $r_1, \ldots, r_k$ and terminates for the goals in $q_1, \ldots, q_m$, $P \cup Q$ terminates for the goals in $r_1, \ldots, r_k$. $\odot$

Using this, we can now state the following theorem:

Theorem 3 *Let $P_n : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ be a synthesised program (in a framework $\mathcal{F}$) and D_n the corresponding definition axioms. If $\mathcal{F} \cup D_n \vdash \mathrm{Comp}^-(P_n, r_1, \ldots, r_k)$, and P_n parametrically terminates, then P_n is steadfast in the class of the $(q_1, \ldots, q_m)$-models of $\mathcal{F} \cup D_n$.*

Proof. Let M be a $(q_1, \ldots, q_m)$-model of $\mathcal{F} \cup D_n$ and Q a program correct wrt $q_1, \ldots, q_m$ in M. We have to prove that M is a steadfast model of P_n. Indeed:

Let $p_1, \ldots, p_h$ be the predicate symbols of Q not in L_M. Let M^* be the expansion of M by $p_1, \ldots, p_h$, interpreting the new symbols as the minimum Herbrand model of Q. M^* is a model of $\mathcal{F} \cup D_n \cup Comp(Q)$ and hence of $Comp(P_n \cup Q)$ (since $\mathcal{F} \cup D_n \vdash Comp^+(P_n)$, $\mathcal{F} \cup D_n \vdash Comp^-(P_n, r_1, \ldots, r_k)$, and $q_1, \ldots, q_m$ are defining symbols in P_n)

Now $diag^+(M^*, r_1, \ldots, r_k, q_1, \ldots, q_m, p_1, \ldots, p_h)$ is the minimum Herbrand model of $P_n \cup Q$, since $P_n \cup Q$ terminates and M^* is a reachable model of $Comp(P_n \cup Q)$. Using this fact, we can prove that $diag^+(M, r_1, \ldots, r_k, q_1, \ldots, q_m)$ is the minimum Herbrand model of $P_n \cup diag^+(M, q_1, \ldots, q_m)$, i.e. it is a steadfast model of P_n. $\Box$

This theorem works from the point of view of synthesis. Indeed, if we are synthesising a program $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ in an open framework $\mathcal{F}$, then we are interested in the closed frameworks $\mathcal{F}^* \supseteq \mathcal{F}$ where a program Q for $q_1, \ldots, q_m$ can be found. Thus **Theorem 3** guarantees that the Halting Principle for steadfast programs is a *sufficient condition* for the synthesis of programs that are steadfast at least in the class of the models where correct programs exist that use them.

Our aim now is to prove that our Halting Principle also works as a necessary condition, namely that if a program P steadfast in the class of all the models of the framework exists, then there is a synthesis process that halts with P as the resulting program.

To prove this, let M be a model and let $P : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ be a program. Consider the (possibly infinite) program $P \cup diag^+(M, q_1, \ldots, q_m)$. To avoid infinitary formulas we use the weak completion *Wcomp* defined by:

$$Wcomp(P, M) = Comp^+(P) \cup Comp^-(P, r_1, \ldots, r_k) \cup diag(M, q_1, \ldots, q_m)$$

Then the following property can be proved:

Property 5.2 A steadfast model M of P is also a model of $Wcomp(P, M)$.

We can obtain:

Property 5.3 Let P_n be a synthesised program (in a framework $\mathcal{F}$) and D_n the corresponding definition axioms. If $P_n : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ is steadfast in a non-empty class of models of $\mathcal{F} \cup D_n$, then $Comp^-(P_n, r_1, \ldots, r_k)$ is consistent with $\mathcal{F} \cup D_n$.

Proof. Assume, by *absurdum*, that $Comp^-(P_n, r_1, \ldots, r_k)$ is inconsistent with $\mathcal{F} \cup D_n$. Then every model M of $\mathcal{F} \cup D_n$ is also a model of $\neg Comp^-(P_n, r_1, \ldots, r_k)$.

Since the class of the steadfast models is non-empty, we can choose M in this class. We get absurdum as follows.

M is a model of $Wcomp(P, M)$, since it is a steadfast model of P, by **Property 5.2**, hence absurdum because $Wcomp(P, M)$ contains $Comp^-(P_n, r_1, \ldots, r_k)$ and M is a model of $\neg Comp^-(P_n, r_1, \ldots, r_k)$. □

This property justifies point (ii) of our Halting Principle. However, it guarantees only the consistency of $Comp^-(P_n, r_1, \ldots, r_k)$, not its provability. It could happen that we reach a steadfast P_n but we cannot prove its completion. This will not be the case, however, if we require steadfastness wrt the entire class of models. Indeed, we can prove the following theorem:

Theorem 4 *Let $P_n : r_1, \ldots, r_k \Leftarrow q_1, \ldots, q_m$ be a synthesised program and D_n the corresponding definition axioms. If P_n is steadfast in the class of the models of $\mathcal{F} \cup D_n$, then $\mathcal{F} \cup D_n \vdash \mathrm{Comp}^-(P_n, r_1, \ldots, r_k)$.*

Proof. Assume, by *absurdum*, that $\mathcal{F} \cup D_n \not\vdash Comp^-(P_n, r_1, \ldots, r_k)$. Then there is a model M of $\mathcal{F} \cup D_n \cup \{\neg Comp^-(P_n, r_1, \ldots, r_k)\}$. Since P_n is steadfast on the class of *all* the models of $\mathcal{F} \cup D_n$, M is a steadfast model of P_n, and we get absurdum like in **Property 5.3** □

Theorem 3 and **Theorem 4** explain the rôle of the provability of completion (for the defined symbols) in the Halting Principle for steadfast programs, namely:

- **Theorem 4** shows that it is a *necessary* condition for successfully halting synthesis of programs that are steadfast in the class of models of the framework.
- **Theorem 3** shows that it is also a *sufficient* condition for synthesis of parametrically terminating programs that are steadfast at least in the class of models which are of interest for synthesis.

We conjecture that **Theorem 3** can also be proved for the class of all models, by extending the notion of parametric termination in a suitable way.

Finally, we point out that the provability of the completion of the defined predicate symbols is not sufficient to guarantee that the synthesised programs are total. As we do not have space for an in-depth discussion here, we shall only briefly and informally touch on this issue.

We say that a predicate symbol q is *completely open* in a framework $\mathcal{F}$ if there is no instance $q(t)$ provable (hence valid) in $\mathcal{F}$. To synthesise steadfast programs that are total, the halting principle becomes:

> **The Halting Principle** (for steadfast and total programs)
> For every predicate p defined by the current program P_n:
>
> (i) if $\mathcal{F} \cup D_n \vdash \neg Comp^-(P_n, p)$, search for other clauses with p in their head;
>
> (ii) if $\mathcal{F} \cup D_n \vdash Comp^-(P_n, p)$, do not search for any more clauses defining p.
>
> If (ii) holds for every defined predicate p of P_n and all the defining predicates of P_n are completely open in the framework, then synthesis successfully halts.

The case of closed frameworks, where no open predicate exists, works as a limiting case, where synthesis goes on until all the symbols that have been introduced become defined symbols of the program.

The Halting Principle for steadfast and total programs works in a very wide class of cases. However there are open frameworks where steadfast programs exist, but no steadfast program is total. Clearly, in this case, if we search for steadfast programs, we will not end up with total programs.

In a closed framework, where a canonical model exists, the situation is more fortunate. It can be proved that, if a total program exists, then the canonical model of the framework and the minimum Herbrand model of the program coincide on the common symbols. Therefore, either the completion of the program is provable, or it is consistent and can be added, enriching the framework in a consistent way that also preserves its semantics (i.e. its canonical model). Thus, if in some step, for some defined symbol p, neither (i) nor (ii) can be proved, we simply add $Comp^-(P_n, p)$ to the axioms of the framework and go on with our synthesis process. We will reach a total program P_n, if it exists.

6 Conclusion

In this paper, we have characterised the halting problem for deductive synthesis of logic programs, and stated a halting principle. We have done so for the synthesis of not only correct programs, but also steadfast, or reusable, program modules. Thus we have contributed to the theoretical foundations of deductive synthesis, as well as to practical synthesis of modular programs.

Our halting criteria are for the synthesis of *totally correct* programs. They thus enable us to determine when to stop our iterative synthesis process with a totally correct program. This is an important step forward for deductive synthesis. Hitherto, synthesis could only guarantee partial correctness, and there are no criteria for halting synthesis; total correctness has to be proved after synthesis has been stopped by some arbitrary rules.

The completion of a program plays a central rôle in the halting problem. A program P is totally correct in a framework $\mathcal{F}$ if the answers computable by P coincide with the answers that are valid (and hence provable) in $\mathcal{F}$. The provability of the completion is a sufficient but not a necessary condition for correctness. However, as we have shown in Section 5, total correctness in $\mathcal{F}$ is not adequate to express correctness wrt the models of $\mathcal{F}$. Indeed, different models may require different behaviours of the program. We have shown that here the provability of the completion (of the defined symbols only) becomes a sufficient *and necessary* condition for successfully halting synthesis with steadfast programs.

Steadfastness of programs is related to their reusability. Thus we have shown that in deductive synthesis we can go beyond correctness, and we have provided theoretical results that should enable us to study *object-oriented* synthesis of steadfast logic program modules. Moreover, the precise relationships between steadfastness and correctness are more intricate and complex than we can explain in this paper. In future work, we also intend to address this issue more fully.

The completion of a program also points a way to incrementally improving our starting specification framework as a result of the actual synthesis process itself. This suggests that deductive synthesis is a promising approach to real-world program synthesis or software engineering, where the entire process is typically: unsatisfactory or incorrect specification $\stackrel{\text{synthesis}}{\longrightarrow}$ unsatisfactory program $\rightarrow$ better specification $\xrightarrow{\text{synthesis}}$ better program ... until the specification and the program are both satisfactory. Beyond this 'software cycle', our approach suggests that the more significant specification frameworks should survive any individual program synthesis process, and accumulate the knowledge provided interactively during synthesis processes, thereby becoming more and more complete.

Acknowledgements

We thank the referees for their helpful comments and for their constructive suggestions for future work.

References

[1] A. Bertoni, G. Mauri, and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* **VI**(2):127–170, 1983.

[2] C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.

[3] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293-322. Plenum Press, 1978.

[4] K.L. Clark. Predicate Logic as a Computational Formalism. Research Report 79/59, Dept of Computing, Imperial College, 1979.

[5] K.L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proc. IFIP 77*, pages 939–944. North-Holland, 1977.

[6] Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.

[7] Å. Hansson and S.-Å. Tärnlund. A natural programming calculus. In *Proc. IJCAI-79*, pages 348–355, 1979.

[8] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, **28**:372–392, 1981.

[9] R. Kowalski. Predicate Logic as Programming Language. In *Proc. IFIP 74*, pages 569–574. North-Holland, 1974.

[10] K.K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7^{th} Int. Conf. on Logic Programming*, pages 667–684. MIT Press, 1990.

[11] K.K. Lau and M. Ornaghi. An incompleteness result for deductive synthesis of logic programs. In D.S. Warren, editor, *Proc. 10^{th} Int. Conf. on Logic Programming*, pages 456–477, MIT Press, 1993.

[12] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[13] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, **2**(1):90–121, Jan 1980.

[14] Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming*. Addison-Wesley, 1993.

[15] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19-88. Morgan Kaufmann, 1988.