A Clausal Form for the Completion of Logic Programs

James Harland
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia
jah@cs.mu.oz.au

Abstract

The Clark completion of a program is a way of making explicit the inferences which may be made from the program using the Negation as Failure (NAF) rule. This may be thought of as adding negative information to the program in such a way that an atom fails iff its negation is derivable from the completion of the program. We show how the completion process may be extended to hereditary Harrop formulae, a class of formulae that properly includes Horn clauses, and, more importantly, that the completion of a hereditary Harrop formulae program may be expressed as a hereditary Harrop formulae program. In this way the richer framework of hereditary Harrop formulae allows the completion to be given in a more explicit form than that of Clark. This also allows the completion to be seen as a set of clauses, i.e. a program, rather than just as a formula of first-order logic. With the restriction that programs are *locally consistent*, we show that the completion behaves as expected, in that the computational properties of the program and its completion correspond precisely.

1 Introduction

The most common way to infer negative information from a logic program is to use the Negation as Failure rule [3, 13]: $\neg A$ succeeds iff A fails. One way to make this rule explicit rather than implicit is to use the *completion* of a program, first introduced by Clark [3]. This may be thought of as turning the "if"s in the program into "if and only if"s, and so it is possible to derive negative literals from the completion of a program by the usual rules of first-order logic.

Another way to achieve something similar is to add to the program extra clause which contain appropriate negative information, in contrast to the above approach which converts the clauses of the program into stronger formulae. For example, the completion of the program $A \subset G$ would be

$$\begin{aligned} A &\subset G \\ \neg A &\subset \neg G \end{aligned}$$

rather than $A \leftrightarrow G$. The formula added to the program above states precisely the way that $\neg A$ may be derived from the original program under the Negation as Failure rule, in that A fails iff G fails, and so $\neg A$ succeeds iff $\neg G$ succeeds.

In this paper we show how a form of the completion based on this idea may be given, not just for Horn clauses, but for a larger class of formulae known as *hereditary Harrop formulae* [11, 10], which properly includes Horn clauses. An advantage of this approach is that the formulae added to the program by the completion process may be considered as clauses in this framework, and so the completion of a hereditary Harrop formula program may be considered as a hereditary Harrop formula program. This property does not hold for Horn clause programs. For example, consider the program

$$p \subset \exists x q(x)$$

The completion of this program would involve adding to the program the formula

$$\neg p \subset \forall x \neg q(x)$$

which is clearly not a Horn clause, due to the universal quantification in the body. However, hereditary Harrop formulae may have universal quantifications (and disjunctions) in the bodies of clauses, and hence it seems that hereditary Harrop formulae may provide a framework which is appropriate for the completion of a program.

An interesting observation which may be made at this point is that there is a natural relationship between hereditary Harrop formulae and intuitionistic logic (in the form of *uniform proofs* [11]), and that in intuitionistic logic, $\neg A$ is identified with $A \supset \perp$. This means that for $\neg A$ to be provable, we need to find a proof which explicitly shows that the assumption of A leads to a contradiction, and hence we need an explicit statement of negative information, à la the completion of a program. Hence it seems interesting to investigate the possible connection between an implicit form of a negation (NAF) with an explicit form, such as "negation by implication". One point to note is that in the latter form, there are no negated formulae, but only a distinguished atom $\perp$, and so by giving the completion not only in clausal form, but also in a form which contains no negations, we may eliminate dependence on implicit forms of inference.

In the next section we give the requisite definitions for hereditary Harrop formulae, and discuss our fundamental approach to the problem. Section 3 deals with the definition of our completion, and section 4 with the properties of our completion.

2 Preliminaries

The class of hereditary Harrop formulae was introduced in [10], and the basic properties of this class of formulae were developed in [11]. Below we give the definition of programs and goals extended to the case when negated atoms are allowed in goals.

Definition 2.1 *D and G formulae are given by*

$$D := A \mid D_1 \wedge D_2 \mid \forall x D \mid G \supset A$$
$$G := A \mid \neg A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \forall x G \mid \exists x G \mid D \supset G$$

where A is an atom.

D formulae are intended as the generalised form of a clause, and G formulae as the generalised goals.

Definition 2.2 *Given an atom $A = p(t_1, \ldots t_n)$, we define $name(A) = p$, and for any formula F, $names(F) = \{name(A) \mid A \text{ appears in } F\}$.*

We say an atom A appears positively (negatively) in a formula F as follows:

- *A appears positively in A*
- *A appears positively (negatively) in $F_1 \vee F_2$ iff A appears positively (negatively) in either F_1 or F_2*
- *A appears positively (negatively) in $F_1 \wedge F_2$ iff A appears positively (negatively) in either F_1 or F_2*
- *A appears positively (negatively) in $\exists x F$ iff A appears positively (negatively) in F*
- *A appears positively (negatively) in $\forall x F$ iff A appears positively (negatively) in F*
- *A appears positively (negatively) in $F_1 \supset F_2$ iff A appears positively (negatively) in F_2 or A appears negatively (positively) in F_1*
- *A appears positively (negatively) in $\neg F$ iff A appears positively (negatively) in F*

A program *is a pair $\langle D, N\rangle$ where D is a set of closed definite formulae and N is a pair $\langle N_1, N_2\rangle$ where $N_i \subseteq names(D)$ and $N_1 \cap N_2 = \emptyset$, and which satisfies:*

For all atoms A and B such that $\mathrm{name}(A) \in N_2$, *$G \supset A$ is a formula in D, and B occurs in G, then*

- *If B occurs positively in G then* $\mathrm{name}(B) \in N_2$.
- *If B occurs negatively in G then* $\mathrm{name}(B) \in N_1$.

If $N = \langle N_1, N_2\rangle$, we say $\mathrm{ass}(N) = N_1$, $\mathrm{den}(N) = N_2$.

Intuitively, the two sets of names indicate which predicates are *completely defined* (den(N)) and which are not (ass(N)). The latter set of predicates may be safely assumed (i.e. added to the program), whereas the former set may not. Further discussion on this point is beyond our scope; see [6, 7] for more detail.

An operational notion of provability, denoted by $\vdash_o$ was given in [11] for the above class of formulae when there are no negations present. In order to account for Negation as Failure, we need a more intricate notion of success and failure. We do so by introducing two relations $\vdash_s$ and $\vdash_f$, where the former is used to indicate success and hence will be similar to $\vdash_o$, and the latter is used to indicate failure. Both will be defined over $\mathcal{P} \times \mathcal{G}$, where $\mathcal{G}$ is the larger set of G formulae defined above. There will be some interplay between the two relations, as we wish to identify $P \vdash_s \neg A$ with $P \vdash_f A$, i.e. we identify negation ($P \vdash_s \neg A$) with failure ($P \vdash_f A$).

We will assume that the number of symbols in the Herbrand universe is finite. We also assume the existence of a countably infinite set of variables. Terms are defined in the usual way. We denote the set of all ground terms (the Hebrand universe) by $\mathcal{U}$. We denote the set of all terms other than variables (the *general* Herbrand universe) by $\mathcal{U}'$.

We denote by $A \propto B$ the statement that A is an instance of B. We denote by heads(D) the set of all atoms A such that $G \supset A$ occurs in D.

The definitions of $\vdash_s$ and $\vdash_f$ are given below.

Definition 2.3 *Let D be a closed definite formula. Then we define*

$$
\begin{aligned}
(A) &= \{A\} \\
(\forall x D) &= \bigcup_{t \in \mathcal{U}'} (D[t/x]) \\
(D_1 \wedge D_2) &= (D_1) \cup (D_2) \\
(G \supset A) &= \{G \supset A\}
\end{aligned}
$$

Definition 2.4 *Let $P = \langle D, N \rangle$ be a program. We define the relations $\vdash_s^k$ and $\vdash_f^k$ for any $k \geq 0$ as follows:*

For any $k \geq 0$

- $P \vdash_s^0 A$ *iff* $A \in (D)$
- $P \vdash_f^0 A$ *iff* $\forall B \in (D)$ *and* $\forall G \supset B \in (D)$, $B \not\propto A$
- $P \vdash_s^{k+1} A$ *iff* $P \vdash_s^k A$ *or* $\exists G \supset A \in (D)$ *such that* $P \vdash_s^k G$
- $P \vdash_f^{k+1} A$ *iff* $P \vdash_f^k A$ *or* $\forall B \in (D)$, $B \not\propto A$ *and* $\forall G \supset B \in (D)$ *such that* $B \propto A$, $P \vdash_f^k G$

- $P \vdash_s^{k+1} \neg A$ *iff* $P \vdash_f^k A$ *and* $name(A) \in \mathrm{den}(N)$
- $P \vdash_f^{k+1} \neg A$ *iff* $P \vdash_s^k A$

- $P \vdash_s^k G_1 \vee G_2$ *iff* $P \vdash_s^k G_1$ *or* $P \vdash_s^k G_2$
- $P \vdash_f^k G_1 \vee G_2$ *iff* $P \vdash_f^k G_1$ *and* $P \vdash_f^k G_2$

- $P \vdash_s^k G_1 \wedge G_2$ *iff* $P \vdash_s^k G_1$ *and* $P \vdash_s^k G_2$
- $P \vdash_f^k G_1 \wedge G_2$ *iff* $P \vdash_f^k G_1$ *or* $P \vdash_f^k G_2$

- $P \vdash_s^k \exists x G$ *iff* $P \vdash_s^k G[t/x]$ *for some* $t \in \mathcal{U}'$
- $P \vdash_f^k \exists x G$ *iff* $P \vdash_f^k G[t/x]$ *for all* $t \in \mathcal{U}'$

- $P \vdash_s^k \forall x G$ *iff* $P \vdash_s^k G[t/x]$ *for all* $t \in \mathcal{U}'$
- $P \vdash_f^k \forall x G$ *iff* $P \vdash_f^k G[t/x]$ *for some* $t \in \mathcal{U}'$

- $\langle D, N \rangle \vdash_s^k D' \supset G$ *iff* $\langle D \cup \{D'\}, N \rangle \vdash_s^k G$ *and names(heads(D'))* $\subseteq \mathrm{ass}(N)$
- $\langle D, N \rangle \vdash_f^k D' \supset G$ *iff* $\langle D \cup \{D'\}, N \rangle \vdash_f^k G$ *and names(heads(D'))* $\subseteq \mathrm{ass}(N)$

The relations $\vdash_s$ *and* $\vdash_f$ *are defined as*

$$\vdash_s = \bigcup_{k \geq 0} \vdash_s^k$$

$$\vdash_f = \bigcup_{k \geq 0} \vdash_f^k$$

The above definitions for $\vdash_s$ and $\vdash_f$ are somewhat more intricate than the definition of $\vdash_o$, due to the interdependence between the two caused by the Negation as Failure rule. Note that there are some programs and goals for which neither $P \vdash_s G$ nor $P \vdash_f G$. For example, if P is $p \supset p$ and G is p, then it is clear that $P \not\vdash_s p$ and $P \not\vdash_f p$.

The above definitions of $\vdash_s$ and $\vdash_f$ may be used to derive a proof system by interpreting the left hand side of the iff as the derived sequent and the right hand side as the previous sequent or sequents. Space prevents us developing this further here; the interested reader may consult [7, 5].

The intuitive reading of $P \vdash_s G$ and $P \vdash_f G$ may be given as "G succeeds" and "G fails" respectively.

In [11, 10] the relationship between $\vdash_o$ and provability in intuitionistic logic is discussed, and it is shown that for the definition of $\vdash_o$ given in [10], $P \vdash_o G$ iff $P \vdash_I G$, where $\vdash_I$ denotes intuitionistic provability. The differences between the two are due to the different ways that negations are inferred; in the former case we use an implicit inference rule known as Negation as Failure (NAF), and in the latter case we use an explicit inference rule. Clearly one way to restore the relationship between $\vdash_s$ and $\vdash_I$ is to make the NAF rule explicit, i.e. by using the completion of the program. An interesting point to note here is that one of the well-known features of intuitionistic logic is that $\neg A$ is identified with $A \supset \bot$, which may be thought of as identifying the statement "A is false" with "A can never be true". Hence the natural way to define the completion in this case is in terms of $A \supset \bot$ rather than $\neg A$, and so we may think of the completion as a theory in which $\neg A$ holds iff $A \supset \bot$ holds.

Note that by replacing $\neg A$ with $A \supset \bot$ we eliminate all negative literals from the program, and so we do not need to use Negation as Failure in order to derive negative consequences.

Thus we may add such negative information to the program by allowing $\bot$ as a distinguished atom, as is done in [9].

For example, we may give the completion of the even predicate as the conjunction of

$$\forall x \ \mathrm{even}(x) \subset x = 0 \vee (\exists y \, x = s^2(y) \wedge \mathrm{even}(y))$$

and either

$$\forall x \ \neg\mathrm{even}(x) \subset x = s(0) \vee (\exists y \, x = s^2(y) \wedge \neg\mathrm{even}(y))$$

or

$$\forall x \quad \bot \subset \mathrm{even}(x) \wedge (x = s(0) \vee (\exists y \, x = s^2(y) \wedge \neg\mathrm{even}(y)))$$

Note that if we replace $\neg$even by odd in the first version above, this process has added the usual definition of the odd predicate to the usual definition of the even predicate, which is just what would be expected.

3 Completions and Provability

Clark's completion is given in the definition below.

Definition 3.1 *Let P be a Horn clause program. For each predicate p, of arity n, which is defined by k clauses*

$$p(t_{11}, t_{12}, \ldots t_{1n}) \subset B_1$$
$$\ldots$$
$$p(t_{k1}, t_{k2}, \ldots t_{kn}) \subset B_k$$

where B_i is a conjunction of atoms (or body), the completion $comp(P)$ of P is given by

$$\forall x_1 \ldots x_n \; p(x_1, \ldots x_n) \leftrightarrow \exists(E_1 \wedge B_1) \vee \ldots \vee \exists(E_k \wedge B_k)$$

where E_i is $x_1 = t_{i1} \wedge \ldots \wedge x_n = t_{in}$, $1 \leq i \leq k$, $\exists(E_i \wedge B_i)$ stands for the existential closure of $E_i \wedge B_i$ for all variables of $E_i \wedge B_i$ other than $x_1, \ldots, x_n$, and the variables $\{x_1, \ldots x_n\}$ are new variables, i.e. they do not appear anywhere in the k clauses above.

Note the importance of the equations, which may be thought of as encapsulating the unification process.

In addition, Clark's scheme involves adding the following formula for each predicate p which appears in the body of some clause in the program but not as the head of any clause:

$$\forall x_1, \ldots x_n \neg p(x_1, \ldots x_n)$$

This is just the case when a predicate has no clauses defining it in the program, and so for all terms $t_1, \ldots t_n$ we have that $p(t_1, \ldots t_n)$ is false.

For a Horn clause program P, we refer to the Clark completion as comp(P). Note that for Horn clause programs, the completion is consistent. There are also classes of programs which may contain negations for which the completion is consistent [2, 14]. In this paper we will restrict our attention to *locally consistent* programs. This class of programs is particularly convenient for our technical results, although it is probable that any restriction which leads to the consistency of the completion will suffice. Programs which are locally stratified [12] are locally consistent, but a locally consistent program is not necessarily locally call-consistent [2].

One important property of programs whose completion is consistent is that in any derivation of $A \supset \perp$ from the completion, the assumption of A must be used in the computation. This is justified by the fact that for any consistent program P, $P \not\vdash_s \perp$, and so for $P^c, A \vdash_s \perp$ to hold, we must have that the assumption is used in the derivation. Thus we are not trying to find whether the program is in any way inconsistent, which is the natural interpretation of $P \vdash_s \perp$; we are merely checking whether a given extension to a consistent program makes it inconsistent. Thus we can consider computation of negative information in this way as a specialised form of a consistency check.

It is interesting to note that although the CWA is defined on a program, Clark's completion is defined for a predicate, and the completion of the program is given by the conjunction of the completions of each predicate. Below we give a similar completion procedure for a hereditary Harrop formula program, and our completion will have the added advantage of being defined in an executable language, and so it directly specifies a computational method for NAF. This is done by adding the extra information as clauses, rather than converting clauses into stronger statements, as is done in [3].

The basic idea is the observation that if $G \supset A$ is the only clause whose head matches A, then we know that if G fails, then A fails, and hence we have $\neg A$. Thus the main problem is to express the negation of G in our form of clauses. In order to do this, we introduce below the operator *fails*, which takes a goal and returns another goal such that fails(G) succeeds if the original goal G fails. This is akin to the transformation process of Barbuti et al. [1], and may be thought of as a way of expressing $\neg G$ as a goal.

Definition 3.2 *Let G be a goal. We say G is* negatable *iff all predicates which occur positively in G are completely defined, and all predicates which occur negatively in G are incompletely defined.*

It should be clear that if $G \supset A \in (D)$ and name(A) is completely defined, then G is a negatable goal. This is due to the way that completely defined predicates may depend on other predicates, and so all the bodies of the clauses of the program in which we will be interested will be negatable goals.

The reason that negatable goals are interesting is that we may identify the failure of a negatable goal with the truth of its negation. This may be thought of as an extension of the way that we infer $\neg A$ from the failure of A when name(A) is a completely defined predicate. The reason that all the negatively occurring predicates must be incompletely defined is that we must be able to add the necessary assumptions to the program in order to show that the goal fails, and hence that its negation succeeds. The operator fails may now be defined as follows:

Definition 3.3 *Let G be a negatable goal.*
We define another goal fails(G) *as follows:*

$$\begin{array}{ll}
\text{fails}(A) & = \neg A \\
\text{fails}(\neg A) & = A \\
\text{fails}(G_1 \vee G_2) & = \text{fails}(G_1) \wedge \text{fails}(G_2) \\
\text{fails}(G_1 \wedge G_2) & = \text{fails}(G_1) \vee \text{fails}(G_2) \\
\text{fails}(D \supset G) & = D \supset \text{fails}(G) \\
\text{fails}(\exists x G) & = \forall x \, \text{fails}(G) \\
\text{fails}(\forall x G) & = \exists x \, \text{fails}(G)
\end{array}$$

It is obvious that for any negatable goal G, fails(G) is also a goal. The similarity to the definition of $\vdash_f$ should also be clear.

We may think of this transformation as exploiting the properties of failure and success. For example, the goal $p \wedge q$ fails iff either p fails or q fails, and under Negation as Failure, this is the same as either $\neg p$ succeeds or $\neg q$ succeeds, i.e. $\neg p \vee \neg q$ succeeds.

Note also that fails(fails(G)) = G. It is not hard to show the following result.

Proposition 3.1 *Let P be a program and let G be a goal. Then*

1. $P \vdash_s G \Leftrightarrow P \vdash_f \text{fails}(G)$
2. $P \vdash_f G \Leftrightarrow P \vdash_s \text{fails}(G)$

We may interpret this result as defining a notion of NAF for a larger class of formulae than just atoms. If $P \vdash_s \neg G$ iff $P \vdash_f G$, then from the above proposition we have that $P \vdash_s \neg G$ iff $P \vdash_s \text{fails}(G)$, and so we can think of fails(G) as a way of writing $\neg G$ as a goal. In this way once we have implemented $\neg A$ as a goal, we may derive a more widespread notion of NAF, i.e. one that is applicable to any negatable goal, not just to atoms.

We denote by contr(G) the formula obtained by replacing every occurrence of $\neg A$ in G by $A \supset \perp$.

As mentioned above, we need to restrict the class of programs in order to guarantee the consistency of the completion. This leads us to the definitions below.

Definition 3.4 *Let P be a program.*

We define the P-reliant relation on literals as follows:

1. *A is P-reliant on L iff $\exists G \supset A \in (P)$ such that L appears in G*
2. *$\neg A$ is P-reliant on L if A is P-reliant on L*

We define the P-dependent relation on literals as follows:

1. *A is P-dependent on L iff A is P-reliant on L or there is an L' such that A is P-reliant on L' and L' is P-dependent on L*
2. *$\neg A$ is P-dependent on L if A is P-dependent on L*

We say L is P-self-dependent *if L is P-dependent on L.*

It is easily seen that the P-dependent relation is the transitive closure of the P-reliant relation. Thus in the program below we have that p is P-dependent on p but not on $\neg p$, and that q is P-dependent on $\neg r$ and s.

$$\begin{array}{l} p \subset p \\ q \subset \neg r \\ r \subset s \end{array}$$

These relations are useful in order to formally define the property of local consistency, which is done below.

Definition 3.5 *Let P be a program and let G be a goal.*

P is locally consistent *iff there is no atom A such that A is P-dependent on $\neg A$.*

P and G are a locally consistent pair *if every program which occurs in the computation of G is locally consistent.*

Thus the program given above is locally consistent, whereas the program consisting of the clause $p \subset \neg p$ is clearly not locally consistent.

Note that if P is locally consistent, then so is $P \cup \{A\}$, so that the addition of an atom to a locally consistent program results in a locally consistent program, i.e. the assumption of an atom preserves local consistency.

We need the notion of a locally consistent pair P and G rather than just a locally consistent program because the program may increase during execution, and so we need to ensure that all programs which occur during execution preserve local consistency.

We are now in a position to give a formal definition of our completion.

Definition 3.6 *Let P be a locally consistent program. For each predicate letter p appearing in P, we define p^+ and p^- as follows:*

Let the clauses defining p be the universal closure of

$$p(t_{11}, t_{12}, \ldots t_{1n}) \subset G_1$$
$$\ldots$$
$$p(t_{k1}, t_{k2}, \ldots t_{kn}) \subset G_k$$

We denote by p^+ the clause

$$\forall x_1 \ldots x_n \; p(x_1, \ldots x_n) \subset \bigvee_{i=1}^{k} \exists (E_i \wedge \mathrm{contr}(G_i))$$

where E_i is $x_1 = t_{i1} \wedge \ldots \wedge x_n = t_{in}$, $\exists(E_i \wedge G_i)$ is the existential closure of all free variables of $E_i \wedge G_i$, $1 \leq i \leq k$, and $x_1, \ldots x_n$ are new variables which do not appear in any G_i.

p^- is the clause

$$\forall x_1 \ldots x_n \;\; \perp \subset p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall (\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

If there is no clause in P whose head's name is p, then p^+ is empty, and p^- is the clause

$$\forall x_1, \ldots x_n \;\; \perp \subset p(x_1, \ldots x_n)$$

The completion of a predicate p is $\{p^+, p^-\}$.

The completion P^c of P is the union of the completions of each predicate appearing in P.

We will write often P^c as $P^+ \cup P^-$ to emphasise the nature of our definition of the completion, i.e. the original clauses of the program (in a slightly modified form) together with a set of extra clauses containing the negative information.

Note also that P^c contains the clause

$$\forall x_1, \ldots x_n \perp \subset p(x_1, \ldots x_n)$$

for all completely defined predicates which occur in the body of some clause in the program, but for which there is no clause head whose name is p.

The only thing that prevents us from using the above two clauses directly to compute negated atoms (as distinct from the usually semantic use of comp(P)) is the inequations. We will assume the existence of an idealised interpreter which includes an appropriate solver for both equations and inequations. We may think of this as showing that Negation as Failure may be reduced to the solving of equations and inequations (provided that the program is locally consistent). It should be noted that solving the equations and inequations may be done independently of the resolution process, as the equations and inequations are relations between terms rather than predicates. Hence, we may call the equation/inequation solver at any point in a derivation, and so we shall write $P \vdash_s E$ to denote that the set E of equations and inequations has a solution. This will come in handy when mixing equations, inequations and goals.

Apart from the inequations, the clauses above are definite formulae according to our definition, and so we may think of this form of the completion as some sort of meta-program for the computation of literals. The reason that we consider it a meta-level program is that we consider the programmer as writing the "positive" clauses for the completely defined predicates, and the "negative" clauses as being implicitly understood due to the fact that p is completely defined. Thus this executable form of the completion makes explicit the complement of the definitions written by the programmer.

4 Properties of the Completion

It is not difficult to show that the completion of a locally consistent program behaves in a consistent manner.

Proposition 4.1 *Let P be a locally consistent program. Then it is not the case that*

$$P^c \vdash_s \perp$$

Thus, in order to derive $A \supset \perp$, A must be used in the computation. In fact, we can be more specific; A is only needed to determine which clause to use. For example, consider the completion (below) of the program $p \supset q$, where p and q are completely defined:

$q \subset p$
$\perp \subset (p \supset \perp) \wedge q$
$\perp \subset p$

It is clear that to find a proof of $q \supset \perp$ it is sufficient to determine whether $p \supset \perp$ from the completion of the program, rather than from the completion extended by the assumption of q. A formal statement of this property is given below.

We write the substitution $[x_1 \leftarrow s_1, \ldots x_n \leftarrow s_n]$ as $[x_i \leftarrow s_i]_{i=1}^n$.

Proposition 4.2 *Let P be a locally consistent program, and $p(s_1, \ldots s_n)$ be an atom such that there is a clause*

$$\perp \subset \exists x_1, \ldots x_n\, p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

in P^c. Then

1. *$P^c \cup \{p(s_1, \ldots s_n)\} \vdash_s \perp$ iff $P^c \vdash_s \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$*

2. *If $P^c \cup \{p(s_1, \ldots s_n)\} \vdash_f \perp$ then $P^c \vdash_f \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$*

3. *If $P^c \vdash_f \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^n)$ then $P^c \cup \{p(s_1, \ldots s_n)\} \not\vdash_s \perp$.*

Hence, A is only needed to determine which clause to use, and not as an assumption per se. The above proposition also indicates another "short-cut": when searching for a proof of

$$P^c \cup \{p(s_1, \ldots s_n)\} \vdash_s \perp$$

it is only necessary to consider the clause

$$\perp \subset \exists x_1, \ldots x_n\, p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \mathrm{contr}(\mathrm{fails}(G_i)))$$

rather than all clauses in P^-, as all that can happen by ignoring the other clauses is that some loops are avoided. Hence searching for a proof of $A \supset \perp$ is indeed a specialised form of a consistency check.

The reason that we use 3 above, rather than the the converse to 2, which is stronger, is that there may be a clause in P^- which leads to a loop, even though the relevant clause for $p(s_1, \ldots s_n) \supset \perp$ does not. For example, let P be the program

$$p \subset \neg q$$
$$r \subset r$$

It is clear that

$$P^c \cup \{p\} \vdash_f q \wedge p$$

but that an attempt to find a proof that

$$P^c \cup \{p\} \vdash_s r \wedge (r \supset \perp)$$

leads to a loop. Hence, $P^c \cup \{p\} \not\vdash_f \perp$, but $P^c \cup \{p\} \vdash_f q \wedge p$.

This means that $\vdash_f$ is not quite strong enough for our purposes, and so we need a slightly stronger relation in order to avoid some unnecessary loops.

Hence we define the relations $\vdash_{sc}$ and $\vdash_{fc}$, which are to be used on the completion of programs.

Definition 4.1 *Let P and G be a locally consistent pair. We define the relations $\vdash_{sc}, \vdash_{fc} \subseteq \mathcal{P}^c \times \mathcal{G}^c$, where $\mathcal{P}^c$ is the set of all possible completions of programs, and $\mathcal{G}^c$ is the set of all goals in "contradiction" form (i.e. $contr(G)$), as the smallest relations which satisfy*

1. $\vdash_{sc} = \vdash_s$
2. $\vdash_f \subseteq \vdash_{fc}$
3. $P^c \vdash_{fc} p(s_1, \ldots s_n) \supset \perp$ *iff*

$$P^c \vdash_{fc} (\bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))[x_i \leftarrow s_i]_{i=1}^{n})$$

where there is a clause

$$\perp \subset \exists x_1, \ldots x_n\, p(x_1, \ldots x_n) \wedge \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i)))$$

It is clear that $P^c \vdash_s G$ iff $P^c \vdash_{sc} G$, and by proposition 4.2 we get $P^c \vdash_{sc} p(s_1, \ldots s_n) \supset \perp$ iff

$$P^c \vdash_{sc} \bigwedge_{i=1}^{k} \forall(\neg E_i \vee \text{contr}(\text{fails}(G_i))[x_i \leftarrow s_i]_{i=1}^{k}$$

It is also clear that $P^c \vdash_f G$ implies $P^c \vdash_{fc} G$, but the converse does not hold. As noted above, it is possible that $P^c \cup \{p\}$ may lead to a loop when trying to prove $\perp$ without the short-cut rule, and so it is not the case that $P^c \not\vdash_s p \supset \perp$, but that $P^c \vdash_{fc} p \supset \perp$.

Thus we arrive at the central result, i.e. that the completion has the expected operational behaviour.

Proposition 4.3 *Let P and G be locally consistent pair. Then*

1. $P \vdash_s G \Leftrightarrow P^c \vdash_{sc} \mathrm{contr}(G)$
2. $P \vdash_f G \Leftrightarrow P^c \vdash_{fc} \mathrm{contr}(G)$

In this way we see that our completion preserves the computational behaviour of the program.

5 Conclusion

We have seen how the completion of a hereditary Harrop formula program may be expressed as another hereditary Harrop formula program, and how the computational properties of the original program are preserved in its completion. In particular, we may define the completion in such a way that ambiguities are ignored, and hence inference is not affected, or in such a way that ambiguities are detected via a particular method of checking for consistency. Moreover, for a large class of programs, the locally consistent programs, we have seen that the two notions of completion coincide.

Either of these notions of the completion may be thought of as giving an explicit form to the Negation as Failure rule, and reduces the application of this rule to the solvability of certain types of equations and inequations. Moreover the completion itself is a program, and hence is executable, and so may be considered as a programming tool rather than just a semantic device. An advantage of this approach is that the "executability" of our form of the completion makes it simple to incorporate Negation as Failure with other kinds of negation. In particular, using negations in the heads of clauses in the manner of [4] may be easily integrated using P^c, as the clauses in the completion are written in a similar fashion.

As we have seen, under certain conditions we may eliminate the occurrences of inequations in P^c, and hence preserve the constructs used in the original program. In particular, we may see this as synthesising a definition complementary to the original one, in that both definitions appear in a form that a programmer may have written, and hence may be used as input to a compiler. In this sense our approach to the completion is transformational, in that we take a given program and transform it in various ways before producing another (larger) program. As we have seen, the elimination of inequations is only possible when there are no repeated variables, and this process is greatly simplified when the predicate is defined in such a way that a goal can match at most one clause. This result shows the advantage of using the completion as a programming tool, as it is straightforward to see what kinds of transformations may be performed once the completion is given. Hence it would seem that the proof-theoretic interpretation of the completion, as given in this paper, is perhaps a more natural use of this construct than the traditional semantic use of the completion of a logic program.

6 Acknowledgements

My thanks go to Dale Miller for many stimulating discussions, and to Don Sannella for his constant encouragement and support. Comments from Keith Clark and some anonymous referees were also very helpful.

Much of this research was performed whilst the author was a student in the Department of Computer Science at the University of Edinburgh.

References

[1] R. Barbuti, P. Mancarella, D. Pedreschi and F. Turini, Intensional Negation of Logic Programs, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, 1987.

[2] L. Cavedon, Continuity, Consistency and Completeness Properties for Logic Programs, *Proceedings of the International Conference on Logic Programming* 571-584, Lisbon, June, 1989.

[3] K. Clark, Negation as Failure, in *Logic and Databases*, H. Gallaire and J. Minker (eds.), Plenum Press, 1978.

[4] M. Gelfond and V. Lifschitz, Logic Programs with Classical Negation, *Proceedings of the International Conference on Logic Programming* 579-597, Jerusalem, June, 1990.

[5] J. Harland, A Clausal Form for the Completion of Logic Programs, Technical Report 90/24, Department of Computer Science, University of Melbourne.

[6] J. Harland, A Kripke-like Model for Negation as Failure, *Proceedings of the North American Conference on Logic Programming* 626-642, Cleveland, October, 1989.

[7] J. Harland, *On Hereditary Harrop Formulae as a Basis for Logic Programming*, Ph.D. Thesis, University of Edinburgh, forthcoming.

[8] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[9] D.A. Miller, A Logical Analysis of Modules in Logic Programming, *Journal of Logic Programming* 6:79-108, 1989.

[10] D.A. Miller, G. Nadathur and A. Scedrov, Hereditary Harrop Formulas and Uniform Proof Systems, *Proceedings of the Symposium on Logic in Computer Science* 98-105, Ithaca, June, 1987.

[11] D.A. Miller, G. Nadathur, F. Pfenning and A. Scedrov, Uniform Proofs as a Foundation for Logic Programming, Technical Report MS-CIS-89-36, Department of Computer and Information Science, University of Pennsylvania, June, 1989. To appear in the *Annals of Pure and Applied Logic.*

[12] T. Przymusinski, On the Declarative Semantics of Deductive Databases and Logic Programs, *Foundations of Deductive Databases and Logic Programming* 193-216, J. Minker, (ed.), Morgan Kaufmann, 1988.

[13] R. Reiter, On Closed World Databases, in *Logic and Databases*, H. Gallair and J. Minker (eds.), Plenum Press, 1978.

[14] T. Sato, Completed Logic Programs and their Consistency, manuscript, Electrotechnical Laboratory, Ibaraki, 1988. To appear in the *Journal of Logic Programming.*