Structural Properties of Logic Programs

James Harland
Key Centre for Knowledge Based Systems
Department of Computer Science
University of Melbourne
Parkville, 3052
Australia
e-mail: jah@cs.mu.oz.au
Pho9ne: +61 3 344 5229
Fax: +61 3 348 1184

October 29, 1990

Abstract

Miller has shown that disjunctions are not necessary in a large fragment of hereditary Harrop formulae, a class of formulae which properly includes Horn clauses. In this paper we extend this result to include existential quantifications, so that for each program D, there is a program D' which is operationally equivalent, but contains no disjunctions or existential quantifications. We may think of this process as deriving a normal form for the program. This process is carried out by pushing the connectives outwards from the body of a clause, and this process leads to a normal form for goals as well. The properties of the search process used to find uniform proofs of goals (which generalises SLD-resolution) together with the normal form allow successful goals to be converted into program clauses, and so we may add successful goals to the program. The stored form of the goal requires a larger class of formulae, i.e. full first-order hereditary Harrop formulae, and so this leads to a natural separation into an object level and a meta-level. This separation seems intuitively natural and to correspond to aspects of programming methodology. Finally we discuss some possibilities for amalgamation of the object and meta-level languages.

1 Introduction

It is well known that variables which appear in Horn clauses may be considered to be universally quantified at the front of the clause, so that the Prolog clause

```
p(X) :- q(X,Y)
```

may be considered a shorthand for the formula

$$\forall x \forall y \; p(x) \subset q(x,y)$$

However, we may also consider variables which do not appear in the head of the clause as being existentially quantified at the front of the *body*, so that the above clause is equivalent to

$$\forall x \, p(x) \subset \exists y \, q(x,y)$$

Thus we may consider Horn clauses to be defined in the latter style (i.e. in which existential quantifiers are allowed in the bodies of clauses), but that we need only implement the former style. In this way we may think of the more restrictive definition as a normal form for programs corresponding to the second definition.

A similar property may be discerned for disjunction, in that given the following program

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

it is clear that p succeeds if either q succeeds or r succeeds, and hence the above program is equivalent to

$$p \subset q \vee r$$

Hence, as far as Horn clauses are concerned, we may consider existential quantification and disjunction as redundant constructs, in that given any program containing either of them, there is an (operationally) equivalent program which contains neither construct. Thus we may think of the standard definition of Horn clauses (i.e. the universal closure of formulae of the form $A \subset B$ where A is an atom and B a conjunction of atoms) as a normal form for a larger class of formulae. Using the terminology of [12], this larger class of formulae may be given as follows:

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

where A denotes an atom, a program is a set of D formulae and a goal is a G formula.

As shown in [12, 13], there are larger classes of formulae which may be used as logic programming languages, among them being the following class of programs and goals.

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

where A denotes an atom, a program is a set of D formulae and a goal is a G formula.

In [12], it was shown how disjunctions may be removed from programs in the above class, in that for any program which contains a disjunction, there is a program with the same operational behaviour, which, although usually much larger, contains no disjunctions. We may think of this as removing redundancies from the program, and so programs may be given a simpler definition, in that there is a smaller class of formulae with the same expressive power. In this paper we extend this result by showing that existential quantifiers may be similarly removed from programs. Thus, the normal form will be a program which contains no disjunctions and no existential quantifications. We also show some stronger results for the disjunction case than those in [12], and show that these may be extended to the existential case as well.

This means that the following class of programs and goals may be thought of as a normal form for the latter class above.

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

where A denotes an atom, a program is a set of D formulae and a goal is a G formula.

Another interesting property of Horn clauses is the way that successful goals may be *memoised* [11], that is, how a successful goal may be converted into a set of clauses and stored as a part of the program. This is due to the fact that a successful goal is a conjunction of atoms (instantiated by an answer substitution), which may be added to the program as a set of facts. This allows later computations to make use of earlier work. The larger class of programs exhibited above has a similar property, although a slightly more sophisticated approach is needed in order to store a successful goal as a program clause. We show how this may be done, and we discuss several features of the classes of programs involved which leads to a natural separation of object level and meta-level programs.

This paper is organised as follows. In section 2 we introduce the basic definitions and properties of our language, and in the following section we show how the normal forms may be derived. Section 4 discusses the possibilities for memoisation, and in section 5 we consider the applications of the ensuing division into an object level and a meta-level.

2 Fundamentals

We will have cause to consider the three classes of programs and goals given below.

Definition 2.1 *D_{normal} and G_{normal} formulae satisfy*

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

D_{mod} and G_{mod} formulae satisfy

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

D_{hhf} and G_{hhf} formulae satisfy

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

In all cases, we refer to D formulae as *definite* formulae, and to G formulae as *goal* formulae. Also, a program is a set of closed definite formulae, and a goal is a closed goal formula. In [14] and [13] it is shown how an operational notion of proof may be given for all the above classes of programs and goals. Such proofs are known as *uniform* proofs, and may be defined as follows.

We assume the existence of a finite set of constant and function symbols, and a countable set of variables.

Definition 2.2 *We refer to the set of all ground terms as the* Herbrand universe*, denoted by $\mathcal{U}$.*

We refer to the set of all terms (not necessarily ground) as the general Herbrand universe*, denoted by $\mathcal{U}'$.*

As noted in [13], we may think of a definite formula in much the same way that we think of a Horn clause. The transformation (), defined below, is evidence for the validity of this statement. This is a similar to the [] transformation of [12].

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

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

We also define $(\{D_1, \ldots D_n\}) = \bigcup_{i=1}^{n} (D_i)$.

In this way we may think of a program as a representation of all its instances. The reason that this is useful is that by using (D) in place of D, we may omit details of the unification process, which simplifies the technicalities somewhat. Below we show how this transformation may be used to define an operational notion of proof known as a *uniform* proof. The properties of uniform proofs and relationships to other proof systems are discussed in [12, 13, 7]; for now we note that uniform proofs generalise the notion of an SLD-derivation. The definition of the relation $\vdash_o$, given below, may be thought of as one characterisation of uniform proofs.

Definition 2.4 *Let P be a program. Then we define*

- $P \vdash_o A$ *iff* $A \in (P)$ *or* $\exists G \supset A \in (P)$ *such that* $P \vdash_o G$
- $P \vdash_o G_1 \vee G_2$ *iff* $P \vdash_o G_1$ *or* $P \vdash_o G_2$
- $P \vdash_o G_1 \wedge G_2$ *iff* $P \vdash_o G_1$ *and* $P \vdash_o G_2$
- $P \vdash_o \exists x G$ *iff* $P \vdash_o G[t/x]$ *for some* $t \in \mathcal{U}'$
- $P \vdash_o \forall x G$ *iff* $P \vdash_o G[y/x]$ *where y is not free in P or G*
- $P \vdash_o D' \supset G$ *iff* $P \cup \{D'\} \vdash_o G$

It was shown in [13] that $P \vdash_o G$ iff $P \vdash_I G$ where $\vdash_I$ denotes intuitionistic provability. It is also shown that classical logic is too strong to have this property, i.e. $P \vdash_C G$ does not imply that $P \vdash_o G$, and hence the above definition of $\vdash_o$ requires us to think in terms of intuitionistic logic rather than classical logic. Essentially this is due to the greater number of equivalences in classical logic, and so whilst in intuitionistic logic each connective corresponds to a distinct search operation, in classical logic there is no such independence. More on this point may be found in [12, 13].

In the next section, we show how D_{normal} programs may be considered as a normal form for D_{mod} programs, and in section 3 how we may use this result to store G_{normal} goals as D_{hhf} programs.

3 A Normal Form for Programs

In this section we show how any D_{mod} program P is operationally equivalent to a D_{normal} program P', i.e. that $P \vdash_o G$ iff $P' \vdash_o G$. This process is inspired by the intuitionistic equivalences

$$(G_1 \vee G_2) \supset A \equiv_I (G_1 \supset A) \wedge (G_2 \supset A)$$
$$(\exists x G) \supset A \equiv_I \forall x (G \supset A)$$

where x does not appear in A.

The second of these equivalences is well-known and may be found in [9], for example. A proof of the first may be found in the appendix.

As in [12], dnf(G) and dfnf(P) denote respectively the *disjunctive normal form* of a goal G and the *disjunction-free normal form* of a program P. The definitions in [12] are given in a slightly modified form here, so that we may consider dnf(G) as a formula, rather than a set. We introduce along similar lines the *existential normal form* of a goal and the *existential-free normal form* of a program, here denoted by enf and efnf respectively. We will assume that each existentially quantified variable is unique, so that a goal such as $\exists x\, p(x) \wedge \exists x\, q(x)$ is written as $\exists x\, p(x) \wedge \exists y\, q(y)$. Note that goals must be closed formulae, and so cannot contain free variables.

Definition 3.1 *Let D be a definite formula and let G be a goal. Then we define* dfnf(D), efnf(D), dnf(G) *and* enf(G) *as follows:*

$$
\begin{array}{lcl}
\mathrm{dnf}(G) & = & \bigvee \mathrm{dnf}'(G) \\
\mathrm{dnf}'(A) & = & \{A\} \\
\mathrm{dnf}'(G_1 \vee G_2) & = & \mathrm{dnf}'(G_1) \cup \mathrm{dnf}'(G_2) \\
\mathrm{dnf}'(G_1 \wedge G_2) & = & \{G' \wedge G'' \mid G' \in \mathrm{dnf}'(G_1), G'' \in \mathrm{dnf}'(G_2)\} \\
\mathrm{dnf}'(\exists x G) & = & \{\exists x G' \mid G' \in \mathrm{dnf}'(G)\} \\
\mathrm{dnf}'(D \supset G) & = & \{\mathrm{dfnf}(D) \supset G' \mid G' \in \mathrm{dnf}'(G)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{dfnf}(A) & = & A \\
\mathrm{dfnf}(G \supset A) & = & \bigwedge \{G' \supset A \mid G' \in \mathrm{dnf}'(G)\} \\
\mathrm{dfnf}(D_1 \wedge D_2) & = & \mathrm{dfnf}(D_1) \wedge \mathrm{dfnf}(D_2) \\
\mathrm{dfnf}(\forall x D) & = & \forall x\ \mathrm{dfnf}(D)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{enf}(G) & = & \exists \tilde{x}\ \mathrm{enf}'(G) \text{ where } \tilde{x} \text{ are all the existentially quantified variables of } G \\
\mathrm{enf}'(A) & = & A \\
\mathrm{enf}'(G_1 \vee G_2) & = & \mathrm{enf}'(G_1) \vee \mathrm{enf}'(G_2) \\
\mathrm{enf}'(G_1 \wedge G_2) & = & \mathrm{enf}'(G_1) \wedge \mathrm{enf}'(G_2) \\
\mathrm{enf}'(\exists x G) & = & \mathrm{enf}'(G) \\
\mathrm{enf}'(D \supset G) & = & \mathrm{efnf}(D) \supset \mathrm{enf}'(G)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{efnf}(A) & = & A \\
\mathrm{efnf}(G \supset A) & = & \forall \tilde{x}\ (\mathrm{enf}'(G) \supset A) \text{ where } \tilde{x} \text{ are all the existentially quantified variables of } G \\
\mathrm{efnf}(\forall x D) & = & \forall x\ \mathrm{efnf}(D) \\
\mathrm{efnf}(D_1 \wedge D_2) & = & \mathrm{efnf}(D_1) \wedge \mathrm{efnf}(D_2)
\end{array}
$$

Let P is a program. Then we define $\mathrm{dfnf}(P) = \{\mathrm{dfnf}(D) \mid D \in P\}$, *and* $\mathrm{efnf}(P) = \{\mathrm{efnf}(D) \mid D \in P\}$.

An example of each of these transformations is given below.

P	dfnf(P)	efnf(P)
$s \subset (\exists x\, p(x) \vee q(x)) \wedge r$	$s \subset \exists x\, p(x) \wedge r$	$\forall x\, s \subset (p(x) \vee q(x)) \wedge r$
	$s \subset \exists x\, q(x) \wedge r$	

We may think of the process described by $\mathrm{dnf}(G)$ as pushing all disjunctions to the top level of the goal, so that $\mathrm{dnf}(G)$ is a disjunction of disjunction-free formulae. We may then apply the identity above to obtain the disjunction-free program. A similar remark applies to $\mathrm{enf}(G)$; all existential quantifiers are pushed to the top of the formula, and so we may apply the corresponding identity to obtain a program free of existential quantifications. Thus the above equivalences imply that

$$\mathrm{dfnf}(G \supset A) \equiv_I \bigwedge\{G' \supset A \mid G' \in \mathrm{dnf}'(G)\} = \mathrm{dnf}(G) \supset A$$
$$\mathrm{efnf}(G \supset A) \equiv_I \forall(\mathrm{enf}'(G \supset A)) = \mathrm{enf}(G) \supset A$$

In this way we may specify the program equivalent to D which contains no occurrences of $\exists$ or $\vee$ as $\mathrm{efnf}(\mathrm{dfnf}(D))$. We will refer to this program as $\mathrm{normal}(D)$. Similarly we will refer to $\mathrm{enf}(\mathrm{dnf}(G))$ as $\mathrm{normal}(G)$. The formal results which establish the desired operational equivalence of D and $\mathrm{normal}(D)$ are given below.

PROPOSITION 3.2 *Let P be a D_{mod} program, and let G be a G_{mod} goal. Then*

$$P \vdash_o G \Leftrightarrow \mathrm{dfnf}(P) \vdash_o G \Leftrightarrow P \vdash_o \mathrm{dnf}(G) \Leftrightarrow \mathrm{dfnf}(P) \vdash_o \mathrm{dnf}(G)$$

The equivalence of the first and the fourth cases was shown in [12].

Proof: It is clear that the fourth case will follow immediately from the other three, and so it will suffice to show that $P \vdash_o G \Leftrightarrow \mathrm{dfnf}(P) \vdash_o G \Leftrightarrow P \vdash_o \mathrm{dnf}(G)$.

It should also be clear that the result will follow if we can show that for any element D of P, $P, D \vdash_o G \Leftrightarrow P, \mathrm{dfnf}(D) \vdash_o G \Leftrightarrow P, D \vdash_o \mathrm{dnf}(G)$.

We proceed by induction on the length of the proof of G.

It should be clear that $A \in (P)$ iff $A \in (\mathrm{dfnf}(P))$ and that $\exists G \supset A \in (P)$ iff $\exists G' \supset A \in (\mathrm{dfnf}(P))$.

The base case occurs when G is an atom A, and as $A \in (P)$ iff $A \in \mathrm{dfnf}(P)$, and $\mathrm{dnf}(A) = A$, it is clear that the proposition holds in this case.

Hence we assume that the proposition is true for all programs and for all goals whose proof is less than a given size. There are five cases:

A: As $\mathrm{dnf}(A) = A$, it suffices to show that $P, D \vdash_o A \Leftrightarrow P, \mathrm{dfnf}(D) \vdash_o A$. Now if the base case does not hold, then $P, D \vdash_o A$ iff $\exists G \supset A \in (P \cup \{D\})$ such that $P, D \vdash_o G$ which by the hypothesis is equivalent to $\exists G' \supset A \in (P \cup \{\mathrm{dfnf}(D)\})$ such that $P, \mathrm{dfnf}(D) \vdash_o G'$ which is equivalent to $P, \mathrm{dfnf}(D) \vdash_o A$.

$G_1 \vee G_2$: $P, D \vdash_o G_1 \vee G_2$ iff $P, D \vdash_o G_1$ or $P, D \vdash_o G_2$.
By the hypothesis this is just $P, \mathrm{dfnf}(D) \vdash_o G_1$ or $P, \mathrm{dfnf}(D) \vdash_o G_2$, which in turn is just $P, \mathrm{dfnf}(D) \vdash_o G_1 \vee G_2$.
From the hypothesis we may also deduce that the above is equivalent to $P, D \vdash_o \mathrm{dnf}(G_1)$ or $P, D \vdash_o \mathrm{dnf}(G_2)$ which is just $P, D \vdash_o \mathrm{dnf}(G_1) \vee \mathrm{dnf}(G_2)$, i.e. $P, D \vdash_o \mathrm{dnf}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: $P, D \vdash_o G_1 \wedge G_2$ iff $P, D \vdash_o G_1$ and $P, D \vdash_o G_2$.
By the hypothesis this is just $P, \mathrm{dfnf}(D) \vdash_o G_1$ and $P, \mathrm{dfnf}(D) \vdash_o G_2$, which in turn is just $P, \mathrm{dfnf}(D) \vdash_o G_1 \wedge G_2$.

From the hypothesis we may also deduce that the above is equivalent to $P, D \vdash_o \mathrm{dnf}(G_1)$ and $P, D \vdash_o \mathrm{dnf}(G_2)$ which is just $P, D \vdash_o \mathrm{dnf}(G_1) \wedge \mathrm{dnf}(G_2)$, i.e. $P, D \vdash_o (\bigvee\{G' \mid G' \in \mathrm{dnf}'(G_1)\}) \wedge (\bigvee\{G'' \mid G'' \in \mathrm{dnf}'(G_2)\})$ which is equivalent to $P, D \vdash_o \bigvee\{G' \wedge G'' \mid G' \in \mathrm{dnf}'(G_1), G'' \in \mathrm{dnf}'(G_2)\}$, which is just $P, D \vdash_o \mathrm{dnf}(G_1 \wedge G_2)$.

$\exists xG$: $P, D \vdash_o \exists xG$ iff $P, D \vdash_o G[t/x]$ for some $t \in \mathcal{U}'$.

By the hypothesis this is equivalent to $P, \mathrm{dfnf}(D) \vdash_o G[t/x]$, which is just $P, \mathrm{dfnf}(D) \vdash_o \exists xG$.

From the hypothesis we may also deduce that the above is equivalent to $P, D \vdash_o \mathrm{dnf}(G[t/x])$, which is $P, D \vdash_o \bigvee\{G' \mid G' \in \mathrm{dnf}'(G[t/x])\}$ which in turn is equivalent to $P, D \vdash_o G'[t/x]$ for some $G' \in \mathrm{dnf}'(G)$, and this is clearly equivalent to $P, D \vdash_o \exists xG'$, which in turn is just $P, D \vdash_o \bigvee\{\exists xG' \mid G' \in \mathrm{dnf}'(G)\}$, i.e. $P, D \vdash_o \mathrm{dnf}(\exists xG)$.

$D' \supset G$: $P, D \vdash_o D' \supset G$ iff $P, D, D' \vdash_o G$.

By the hypothesis this is equivalent to $P, \mathrm{dfnf}(D), D' \vdash_o G$, which is just $P, \mathrm{dfnf}(D) \vdash_o D' \supset G$.

From the hypothesis we may also deduce that the above is equivalent to $P, D, D' \vdash_o \mathrm{dnf}(G)$ which by another application of the hypothesis is equivalent to $P, D, \mathrm{dfnf}(D') \vdash_o \mathrm{dnf}(G)$, which in turn is just $P, D, \mathrm{dfnf}(D') \vdash_o \bigvee\{G' \mid G' \in \mathrm{dnf}'(G)\}$. Now this is equivalent to $P, D, \mathrm{dfnf}(D') \vdash_o G'$ for some $G' \in \mathrm{dnf}'(G)\}$, which in turn is equivalent to $P, D \vdash_o \mathrm{dfnf}(D') \supset G'$, which is just $P, D \vdash_o \bigvee\{\mathrm{dfnf}(D') \supset G' \mid G' \in \mathrm{dnf}'(G)\}$, i.e. $P, D \vdash_o \mathrm{dnf}(D' \supset G)$.

$\square$

We now turn to the corresponding result for existential quantifications.

Proposition 3.3 *Let P be a D_{mod} program, and let G be a G_{mod} goal. Then*

$$P \vdash_o G \Leftrightarrow \mathrm{efnf}(P) \vdash_o G \Leftrightarrow P \vdash_o \mathrm{enf}(G) \Leftrightarrow \mathrm{efnf}(P) \vdash_o \mathrm{enf}(G)$$

Proof: It is clear that the fourth case will follow immediately from the other three, and so it will suffice to show that $P \vdash_o G \Leftrightarrow \mathrm{efnf}(P) \vdash_o G \Leftrightarrow P \vdash_o \mathrm{enf}(G)$.

It should also be clear that the result will follow if we can show that for any element D of P, $P, D \vdash_o G \Leftrightarrow P, \mathrm{efnf}(D) \vdash_o G \Leftrightarrow P, D \vdash_o \mathrm{enf}(G)$.

We proceed by induction on the length of the proof of G.

It should be clear that $A \in (P)$ iff $A \in (\mathrm{efnf}(P))$ and that $\exists G \supset A \in (P)$ iff $\exists G' \supset A \in (\mathrm{efnf}(P))$.

The base case occurs when G is an atom A, and as $A \in (P)$ iff $A \in \mathrm{efnf}(P)$, and $\mathrm{enf}(A) = A$, it is clear that the proposition holds in this case.

Hence we assume that the proposition is true for all programs and for all goals whose proof is less than a given size. There are five cases:

A: As $\mathrm{enf}(A) = A$, it suffices to show that $P, D \vdash_o A \Leftrightarrow P, \mathrm{efnf}(D) \vdash_o A$. Now if the base case does not hold, then $P, D \vdash_o A$ iff $\exists G \supset A \in (P \cup \{D\})$ such that $P, D \vdash_o G$ which by the hypothesis is equivalent to $\exists G' \supset A \in (P \cup \{\mathrm{efnf}(D)\})$ such that $P, \mathrm{efnf}(D) \vdash_o G'$ which is equivalent to $P, \mathrm{efnf}(D) \vdash_o A$.

$G_1 \vee G_2$: $P, D \vdash_o G_1 \vee G_2$ iff $P, D \vdash_o G_1$ or $P, D \vdash_o G_2$.

By the hypothesis this is just $P, \text{efnf}(D) \vdash_o G_1$ or $P, \text{efnf}(D) \vdash_o G_2$, which in turn is just $P, \text{efnf}(D) \vdash_o G_1 \vee G_2$.

From the hypothesis we may also deduce that the above is equivalent to $P, D \vdash_o \text{enf}(G_1)$ or $P, D \vdash_o \text{enf}(G_2)$ which is just $P, D \vdash_o \text{enf}(G_1) \vee \text{enf}(G_2)$, which in turn is just $P, D \vdash_o \exists\tilde{x}\, \text{enf}'(G_1) \vee \exists\tilde{y}\, \text{enf}'(G_2)$ which is clearly equivalent to $P, D \vdash_o \exists\tilde{x}\exists\tilde{y}\, \text{enf}'(G_1) \vee \text{enf}'(G_2)$, i.e. $P, D \vdash_o \text{enf}(G_1 \vee G_2)$.

$G_1 \wedge G_2$: $P, D \vdash_o G_1 \wedge G_2$ iff $P, D \vdash_o G_1$ and $P, D \vdash_o G_2$.

By the hypothesis this is just $P, \text{efnf}(D) \vdash_o G_1$ and $P, \text{efnf}(D) \vdash_o G_2$, which in turn is just $P, \text{efnf}(D) \vdash_o G_1 \wedge G_2$.

From the hypothesis we may also deduce that the above is equivalent to $P, D \vdash_o \text{enf}(G_1)$ and $P, D \vdash_o \text{enf}(G_2)$ which is just $P, D \vdash_o \text{enf}(G_1) \wedge \text{enf}(G_2)$, i.e. $P, D \vdash_o \exists\tilde{x}\, \text{enf}'(G_1) \wedge \exists\tilde{x}\, \text{enf}'(G_2)$ which is equivalent to $P, D \vdash_o \exists\tilde{x}\, \text{enf}'(G_1 \wedge G_2)$, which is just $P, D \vdash_o \text{enf}(G_1 \wedge G_2)$.

$\exists x G$: $P, D \vdash_o \exists x G$ iff $P, D \vdash_o G[t/x]$ for some $t \in \mathcal{U}'$.

By the hypothesis this is equivalent to $P, \text{efnf}(D) \vdash_o G[t/x]$, which is just $P, \text{efnf}(D) \vdash_o \exists x G$.

From the hypothesis we may also deduce that the above is equivalent to $P, D \vdash_o \text{enf}(G[t/x])$, which is $P, D \vdash_o \exists\tilde{x}\, \text{enf}'(G[t/x])$, which is equivalent to $P, D \vdash_o \exists\tilde{x}\, \text{enf}'(G)[t/x]$ which in turn is just $P, D \vdash_o \exists x \exists\tilde{x}\, \text{enf}'(G)$, i.e. $P, D \vdash_o \text{enf}(G)$.

$D' \supset G$: $P, D \vdash_o D' \supset G$ iff $P, D, D' \vdash_o G$.

By the hypothesis this is equivalent to $P, \text{efnf}(D), D' \vdash_o G$, which is just $P, \text{efnf}(D) \vdash_o D' \supset G$.

From the hypothesis we may also deduce that the above is equivalent to $P, D, D' \vdash_o \text{enf}(G)$ which by another application of the hypothesis is equivalent to $P, D, \text{efnf}(D') \vdash_o \text{enf}(G)$, which is just $P, D, \text{efnf}(D') \vdash_o \exists\tilde{x}\, \text{enf}'(G)$. This in turn is equivalent to $P, D, \text{efnf}(D') \vdash_o \text{enf}'(G)[\tilde{t}/\tilde{x}]$, which in turn is just $P, D \vdash_o \text{efnf}(D') \supset \text{enf}'(G)[\tilde{t}/\tilde{x}]$, which is equivalent to $P, D \vdash_o \exists\tilde{x}\, \text{efnf}(D') \supset \text{enf}'(G)$, i.e. $P, D \vdash_o \text{enf}(D' \supset G)$.

□

An immediate consequence of propositions 3.2 and 3.3 is that the original program and the program with the disjunctions and existential quantifications removed are operationally equivalent. This is shown below.

THEOREM 3.4 *Let P be a D_{mod} program, and G be a G_{mod} goal. Then*

$$P \vdash_o G \Leftrightarrow \text{normal}(P) \vdash_o G \Leftrightarrow P \vdash_o \text{normal}(G) \Leftrightarrow \text{normal}(P) \vdash_o \text{normal}(G)$$

Proof: Follows immediately from propositions 3.2 and 3.3. □

Thus the theorem above not only ensures that for any D_{mod} program there exists an operationally equivalent D_{normal} program, but also shows how such a program may be derived from the original one. In this way we may think of the version of the program without disjunctions and existential quantifiers as a normal form for the original program.

It is possible to derive some stronger results than those above, in that it is possible to show that the above transformations not only preserve success, but also preserve (finite) failure. In

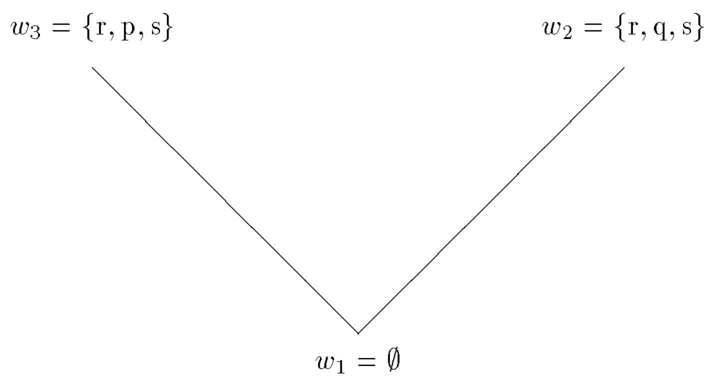

Figure 1: Kripke model in which D_4 is true but D_3 is not

other words, if we use $\vdash_f$ to denote (finite) failure, $P \vdash_f G$ iff $\mathrm{dfnf}(P) \vdash_f G$ iff $P \vdash_f \mathrm{dnf}(G)$ iff $\mathrm{dfnf}(P) \vdash_f \mathrm{dnf}(G)$, and similarly for the existential case. This means that the above results will still hold in the presence of Negation as Failure. However the requisite technicalities are somewhat overwhelming, and we have not included them here for the sake of clarity. More details of this extension (including proofs) may be found in [6].

Note that the results above are expressed purely in terms of $\vdash_o$, and so we think of the transformations given above as preserving operational equivalence. Maher [10] has shown that there are many natural notions of equivalence for logic programs, and in our context, another obvious notion of equivalence is equivalence with respect to $\vdash_I$. Hence, a natural question to ask at this point is whether programs which are operationally equivalent are logically equivalent, i.e. if $\forall G\ D_1 \vdash_o G$ iff $D_2 \vdash_o G$, does it follow that $D_1 \equiv_I D_2$? The answer is negative, as whilst the transformations above preserve operational equivalence, they do not preserve intuitionistic equivalence. A counterexample, due to Dale Miller, is given as follows:

D_1	$\mathrm{dfnf}(D_1)$
$(r \supset (p \vee q)) \supset s$	$((r \supset p) \supset s) \wedge ((r \supset q) \supset s)$

A Kripke model in which $\mathrm{dfnf}(D_1)$ is true but D_1 is not is given in Figure 1.

The reason that D_1 is not true is that $w_1 \models r \supset (p \vee q)$, and so for D_1 to be true in w_1, we require that s be true in w_1, which it is not, and so D_1 is not true in w_1. On the other hand, we have that $w_1 \not\models r \supset p$ and $w_1 \not\models r \supset q$, and so for $\mathrm{dfnf}(D_1)$ to be true in w_1, we do not require that s be true in w_1, and in fact $\mathrm{dfnf}(D_1)$ is trivially true in w_1, as $w_1 \not\models r$. Hence $\mathrm{dfnf}(D_1)$ is true for every world in the model, whereas D_1 is not true in world w_1.

A similar counterexample may be given for the existential case. Consider the program D_2 below.

D_2	$\mathrm{efnf}(D_2)$
$(p \supset \exists x q(x)) \supset r$	$\forall x\, (p \supset q(x)) \supset r$

A Kripke model in which D_2 is true but $\mathrm{efnf}(D_2)$ is not is given in Figure 2.

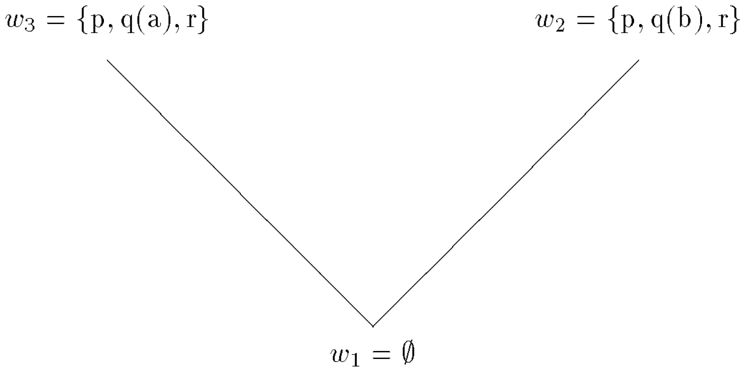

Figure 2: Kripke model in which D_6 is true but D_5 is not

As in the previous case, it is easy to see that both formulae are true in worlds w_2 and w_3, but whilst efnf(D_2) is true at w_1, (as $w_1 \not\models \exists x\ p \supset q(x)$ and so the condition that if $\exists x p \supset q(x)$ is true then r is true is satisfied), D_2 is not true at w_1, as $w_1 \models p \supset \exists x\ q(x)$ but $w_1 \not\models r$.

However, if we augment the rules of intuitionistic logic by the two rules below, we arrive at a logic I' in which it is true that two programs are operationally equivalent iff the programs are logically equivalent.

$$D \supset (G_1 \vee G_2) \vdash_{I'} (D \supset G_1) \vee (D \supset G_2)$$
$$D \supset \exists x G \vdash_{I'} \exists x D \supset G$$

where x is not free in D.

Note that whilst these rules are true in classical logic, this is too strong for our purposes, as there are programs which are classically equivalent which are not operationally equivalent. For example, the two programs below are equivalent in classical logic, but are not operationally equivalent.

$$(q \vee (p \supset p)) \supset r \qquad\qquad ((p \supset q) \vee p) \supset r$$

The problem is that whilst $q \vee (p \supset p)$ and $(p \supset q) \vee p$ are classically equivalent, the first formula succeeds whereas the second fails.

The two extra rules given above are known as the *Independence of Premise* rules. Adding these rules to intuitionistic logic results in an intermediate logic, which may be thought of as a logic of "present choice". For example, given a Kripke model in which $r \supset (p \vee q)$ is true (such as the the former Kripke model above), we know that whenever we find a world in which r is true, then we know that either p or true or q is true, but we do not have to choose in advance which alternative is true. However for $(r \supset p) \vee (r \supset q)$ to be true, we must know in advance which alternative will be true when we find a world in which r is true, and so we need to make a choice in the "present", rather than waiting until we find a world in which r is true and then making the choice. Thus we require that the information in the program be in such a form that we can make an immediate choice between alternatives, which is stronger property than just guaranteeing that a successful choice will exist whenever it is needed.

Another interesting point is that the Independence of Premise rules have usually been studied in conjunction with Markov's Principle [15], which is that $\neg\neg F$ implies F. This latter rule is clearly similar to the identification of the failure of $\neg A$ with the success of A under Negation as Failure, and it seems that this may be a natural setting for investigations of logic programming.

Further discussion on this point is beyond our scope; more on the relationship between operational and logical equivalence and I' may be found in [7, 6].

4 Memoisation

In the light of the above results, we may think of D_{normal} programs as a normal form for D_{mod} programs. We may imagine the transformation occurring at compile time, and so we may think of the above processes as statically converting the program into a more specific form.

Computation in this context takes the form of searching for uniform proofs of goal formulae, but it is interesting to note that a successful search for a uniform proof of a goal G may be thought of as converting G into a more definite statement. Naively, we may think of this as converting G into a D formula. For example, it is well-known that the usual unification methods used to implement the search for a proof of a goal of the form $\exists x G$ will return an answer substitution θ which not only provides a witness, i.e. a term $t \in \mathcal{U}'$ such that $G[t/x]$ succeeds, but where θ is such that *any* instance of $G\theta$ succeeds, so that we may in fact conclude $\forall(G\theta)$. Thus the search process "converts" the $\exists$ quantifier into a number (possibly zero) of $\forall$ quantifiers. In this way we may view the result of a successful computation not as "yes with the answer substitution θ", but as "$\forall(G\theta)$", in the sense that $\forall(G\theta) \vdash_I \exists x G$.

It is easy to see that a similar property holds for disjunction, in that a uniform proof of $G_1 \vee G_2$ will prove one of the disjuncts, and so we may consider the proof search process as determining which of the disjuncts is true, and so we again get that $G_i \vdash_I G_1 \vee G_2$, where i is either 1 or 2. Now if we carry out this conversion procedure for each positive occurrence of $\exists$ and $\vee$ in the goal, then it is easy to see that the resulting formula looks very much like a definite formula.

For example, consider the program $P = \{p(a), q(b)\}$. It is easy to see that the goal $G = \exists x\, p(x) \vee \exists y\, q(y)$ succeeds, i.e. $P \vdash_o G$, as both $p(a)$ and $q(b)$ succeed. Now we may think of either of these as an "answer form" of the original goal, in that not only does the computation process supply witnesses for the existential quantifications, but also determines which of the disjuncts holds. In this way we may think of $p(a)$ and $q(b)$ as *realizers* of G in the sense of Kleene [9], in that either supplies sufficient information "missing" from the statement that $P \vdash_o G$ so that intuitionistic truth is established. We may think of these realizers as the real objects of computation, in that the search process may be considered to operate by establishing that $p(a)$ succeeds, and from that deducing that the success of $p(a)$ implies the success of $\exists x\, p(x) \vee \exists y\, q(y)$. A similar remark applies to $q(b)$. In this way the computation process finds a goal G' which contains no existential quantifiers or disjunctions such that $P \vdash_o G'$ and $G' \vdash_I G$. The analogy should not be pushed too far, but it seems intriguing that the answer forms may be thought of as supplying the information needed to establish the truth of the goal, and in a computational way.

As noted above, this formula G' contains no disjunctions nor existential quantifiers, but it may contain universal quantifiers. For example, consider the program $P = \{\forall x\ p(f(x))\}$. The goal $\exists y\ p(y)$ succeeds with the answer substitution being $y \leftarrow f(x)$, and so the answer form of the goal is $\forall x\ p(f(x))$. A more specific definition of the possible answer forms for a goal may be given by another of the results above, namely that the success of G is equivalent to the success of normal(G), which we may write as

$$\exists(G_1 \vee \ldots \vee G_n)$$

where each of the G_i are G_{normal} goals. Thus any answer form for G must be just $\forall(G_i\theta)$ for some $1 \leq i \leq n$ and some substitution θ, and so we may consider a successful search for a proof of G from P as finding an i and a θ such that

$$P \vdash_o \forall(G_i\theta) \text{ and } \forall(G_i\theta) \vdash_I G$$

where P and $\forall(G_i\theta)$ are considered as a D_{hhf} program and a G_{hhf} goal.

In this way an answer form plays a similar role to a cut formula (in the proof-theoretic sense), in that in order to establish that $P \vdash_o G$, we find a formula $\forall(G_i\theta)$ such that $P \vdash_o \forall(G_i\theta)$ and $\forall(G_i\theta) \vdash_I G$, but in our case we know that the second statement is trivially established, whereas the first may take considerable effort to derive. Thus this is really a special case of the cut rule, in that G defines a finite number of possible choices for G_i, and so we do not need to search for an arbitrary formula, and that one of the two sub-derivations will require no effort to establish.

In order to view an answer form $\forall(G_i\theta)$ as a definite clause, the main difficulty is to accommodate the different classes of formulae which may appear on either side of the $\supset$ connective. For example, $p(a) \supset (q(a) \wedge q(b))$ is a G formula, and hence a possible answer form, but it is not a D formula. However, it is clear that it is not far away from one in some sense, and so it should not be too difficult to convert an answer form into an equivalent D formula. This process is defined formally below.

Definition 4.1 *Let D and G be a D_{normal} program and a G_{normal} goal.*
We define defp(D) *and* defq(G) *as follows:*

$$\begin{array}{ll}
\mathrm{defp}(A) & = A \\
\mathrm{defp}(\forall x D) & = \forall x\ \mathrm{defp}(D) \\
\mathrm{defp}(D_1 \wedge D_2) & = \mathrm{defp}(D_1) \wedge \mathrm{defp}(D_2) \\
\mathrm{defp}(G \supset A) & = \mathrm{defq}(G) \supset A \\
\\
\mathrm{defq}(A) & = A \\
\mathrm{defq}(G_1 \wedge G_2) & = \mathrm{defq}(G_1) \wedge \mathrm{defq}(G_2) \\
\mathrm{defq}(D \supset (G_1 \wedge G_2)) & = \mathrm{defq}(D \supset G_1) \wedge \mathrm{defq}(D \supset G_2) \\
\mathrm{defq}(D \supset (D' \supset G')) & = \mathrm{defq}((D \wedge D') \supset G') \\
\mathrm{defq}(D \supset A) & = \mathrm{defp}(D) \supset A
\end{array}$$

Note that we assume that all bound variables are unique, so that we do not need to worry about variable capture when moving quantifiers around.

An example of this transformation is given below.

$$\begin{array}{cc}
G & \mathrm{defq}(G) \\
p \supset (q \wedge r) & (p \supset q) \wedge (p \supset r)
\end{array}$$

This transformation does not alter the properties of the program. More precisely, we get the following proposition.

Proposition 4.2 *Let D be a D_{normal} program, and let G be a G_{normal} goal. Then*

$$D \equiv_I \mathrm{defp}(D)$$
$$G \equiv_I \mathrm{defq}(G)$$

Proof: We proceed by simultaneous induction on the structure of D and G. The base case occurs when D is an atom and when G is an atom. It is obvious that in this case $D \equiv_I \mathrm{defp}(D)$ and $G \equiv_I \mathrm{defq}(G)$.

Thus the hypothesis is that for all D_{normal} and G_{normal} formulae of no more than a given size, $D \equiv_I \mathrm{defp}(D)$ and $G \equiv_I \mathrm{defq}(G)$. There are several cases to examine.

The cases for D:

$\forall x D$: By the hypothesis, $D \equiv_I \mathrm{defp}(D)$, and so $\forall x D \equiv_I \forall x\ \mathrm{defp}(D)$, which is just $\mathrm{defp}(\forall x D)$.

$D_1 \wedge D_2$: By the hypothesis, $D_1 \wedge D_2 \equiv_I \mathrm{defp}(D_1) \wedge \mathrm{defp}(D_2)$, which is just $\mathrm{defp}(D_1 \wedge D_2)$.

$G' \supset A$: By the hypothesis $G' \equiv_I \mathrm{defq}(G')$, and so $G' \supset A \equiv_I \mathrm{defq}(G') \supset A$, which is just $\mathrm{defp}(Q' \supset A)$.

The cases for G:

$G_1 \wedge G_2$: By the hypothesis, $G_1 \wedge G_2 \equiv_I \mathrm{defq}(G_1) \wedge \mathrm{defq}(G_2)$, which is just $\mathrm{defq}(G_1 \wedge G_2)$.

$D' \supset G'$: There are three cases here, and we use an inductive argument to establish the overall case. The base case occurs when G' is an atom A, and so by the hypothesis, $D' \supset A \equiv_I \mathrm{defp}(D') \supset A$, which is just $\mathrm{defq}(D' \supset A)$.

Hence we will assume, in addition to the main hypothesis, that for all D formulae with less connectives than $D' \supset G'$ and for all G_{normal} formulae of no more than a given size that $D \supset G \equiv_I \mathrm{defq}(D \supset G)$.

There are two sub-cases:

$D \supset (G_1 \wedge G_2)$: It is clear that $D \supset (G_1 \wedge G_2)$ is equivalent to $(D \supset G_1) \wedge (D \supset G_2)$, and by the hypothesis this is equivalent to $\mathrm{defq}(D \supset G_1) \wedge \mathrm{defq}(D \supset G_2)$, which is just $\mathrm{defq}(D \supset G_1 \wedge G_2)$.

$D \supset (D'' \supset G)$: $D \supset (D'' \supset G)$ is clearly equivalent to $(D \wedge D'') \supset G$, and as $D \wedge D''$ must have less connectives than $D \supset (D'' \supset G)$, by the hypothesis we have that this is equivalent to $\mathrm{defq}((D \wedge D'') \supset G)$, which is just $\mathrm{defq}(D \supset (D'' \supset G))$.

Thus we establish the result for any formula of the form $D \supset G$.

□

From this we get the immediate corollary that operational equivalence is preserved under this transformation.

Corollary 4.3 *Let D be a D_{normal} program, and let G be a G_{normal} goal. Then*

$$D \vdash_o G \Leftrightarrow \mathrm{defp}(D) \vdash_o G \Leftrightarrow D \vdash_o \mathrm{defq}(G) \Leftrightarrow \mathrm{defp}(D) \vdash_o \mathrm{defq}(G)$$

Proof: As $D \equiv_I \mathrm{defp}(D)$ and $G \equiv_I \mathrm{defq}(G)$, then

$$D \vdash_I G \Leftrightarrow \mathrm{defp}(D) \vdash_I G \Leftrightarrow D \vdash_I \mathrm{defq}(G) \Leftrightarrow \mathrm{defp}(D) \vdash_I \mathrm{defq}(G)$$

and as $P \vdash_I G'$ iff $P \vdash_o G'$, we are done. □

Next we give the result which establishes that the answer form $\forall(G_i\theta)$ for a goal G may be stored as a part of the program.

PROPOSITION 4.4 *Let D be a D_{normal} program, and G be a G_{normal} goal. Then*

1. defp(D) *is a G_{hhf} formula.*
2. defq(G) *is a D_{hhf} formula.*

Proof: We proceed by simultaneous induction on the structure of D and G. The base case occurs when D and G are both atoms. It is obvious that in this case that both defp(D) and defq(G) are D_{hhf} and G_{hhf} formulae respectively.

Thus we assume that for all D and G formulae of no more than a given size, defp(D) is a D_{hhf} formula and defq(G) is an G_{hhf} formula. We then proceed by cases as follows.

1. There are three cases for D:

$\forall x D'$: defp($\forall x D'$) $= \forall x$ defp(D'), and by the hypothesis, defp(D') is some D_{hhf} formula D, and so defp($\forall x D'$) is just $\forall x D$, and it is clear that $\forall x D$ is a core formula.

$D_1 \wedge D_2$: defp($D_1 \wedge D_2$) = defp(D_1) $\wedge$ defp(D_2), and by the hypothesis, defp(D_1) and defp(D_2) are D_{hhf} formulae D_1' and D_2', and so defp($D_1 \wedge D_2$) is $D_1' \wedge D_2'$, which is obviously a D_{hhf} formula.

$G \supset A$: defp($G \supset A$) = (defq(G)) $\supset A$, and by the hypothesis defq(G) is a G_{hhf} formula G', and so defq($G \supset A$) is $G' \supset A$, which is clearly a D_{hhf} formula.

2. There are two cases for G:

$G_1 \wedge G_2$: defq($G_1 \wedge G_2$) = defq(G_1) $\wedge$ defq(G_2), and by the hypothesis defq(G_1) and defq(G_2) are G_{hhf} formulae G_1' and G_2', and so defq($G_1 \wedge G_2$) $= G_1' \wedge G_2'$, which is clearly a G_{hhf} formula.

$D' \supset G'$: There are three cases here, and as above, we use an inductive argument to establish the overall case. The base case occurs when G' is an atom A, and so defq($D' \supset A$) = defp(D') $\supset A$, and by the hypothesis defp(D') is a D_{hhf} formula D, and so defq($D' \supset A$) $= D \supset A$, which is clearly a D_{hhf} formula.

Hence we will assume, in addition to the main hypothesis, that for all D_{normal} formulae with less connectives than $D' \supset G'$ and for all G_{normal} formula of no more than a given size that defq($D \supset G$) is a core formula.

There are two remaining cases:

$D \supset (G_1 \wedge G_2)$: defq($D \supset (G_1 \wedge G_2)$) = defq($D \supset G_1$) $\wedge$ defq($D \supset G_2$), and by the hypothesis this is just $G_1' \wedge G_2'$ where G_1' and G_2' are G_{hhf} formulae, and obviously so is $G_1' \wedge G_2'$.

$D \supset (D'' \supset G)$: defq($D \supset (D'' \supset G)$) = defq($(D \wedge D'') \supset G$) and by the hypothesis this is a G_{hhf} formula G'.

Thus we establish the result for any formula of the form $D \supset G$.

□

This result shows that we may consider computation as converting G_{mod} formulae into definite clauses of a certain form, which may then be stored in addition to the program. This is done by re-writing G as $\exists(G_1 \vee \ldots \vee G_n)$, and if the goal succeeds, we get an answer form $\forall(G_i\theta)$ and the above proposition establishes that $\forall$(defq(G_i)θ) is a D_{hhf} formula.

It is interesting to note that Horn clauses have a similar property. In this case, the normal form of a goal may be considered to be

$$\exists(A_1 \wedge \ldots \wedge A_n)$$

where the A_i are atoms. Given an answer substitution θ, it is clear that $\forall((A_1 \wedge \ldots \wedge A_n)\theta)$ may be re-written as a conjunction of definite clauses.

In this way we may see the above results as restoring properties lost when generalising from Horn clause programs to D_{mod} programs. Not surprisingly, this required a more sophisticated approach than for Horn clauses, but this more complex approach still retains the essential features of the original class of programs in a natural way.

5 Meta-Programming Possibilities

As discussed in the previous section, we may think of the computation process as searching for information to complete our knowledge of the truth of the goal G, in that G succeeds iff we we can find extra information which allows us to complete the "missing" parts of G. Moreover we saw that the extra information that has been computed enables us to express the success form of G as a definite formula D'. We may then record this information as a larger program $D \cup \{D'\}$ which has the same consequences as D. This larger program may lead to shorter proofs than the smaller one, as we may use previous computations in subsequent proofs, rather than recompute known results. Thus the larger program has the same meaning, but may be more efficient, as we can record the results of previous computations, rather than throwing them away.

This technique is known as *memoisation* [11], and is known to be useful for avoiding redundant computation, as it allows known results to be stored for later use, thus reducing the amount of work that must be done. In our case, this consists of storing consequences of the program, so that the proof search process need not start from scratch each time that a new goal is presented. This is akin to the way that a mathematical theory is built up; one starts from the axioms of a given theory, and derives some basic results. Subsequent proofs then refer to this list of results, rather than resort to the axioms each time, and in so doing often re-discovering known proofs. In this way the reasoning process produces an increasing set of consequences of the axioms, and later proofs refer to this set of consequences, rather than the original axioms alone. Some aspects of memoisation for D_{mod} programs were discussed in [12], where the emphasis was on the memoisation of atomic goals.

We also saw in the previous section how we may think of the computation process as removing disjunctions from goals and replacing existential quantifiers with zero or more universal quantifiers, which is very similar to the processes described in section 3 which exploit the structure of programs to show that disjunctions and existential quantifiers are not necessary. Thus the memoisation process seems to be a natural extension of this process of "tightening up" the program.

However there are some technical difficulties with the answer form of the goal. As shown above, this is of the form $\forall(G_i\theta)$ where G_i is a G_{normal} formula, i.e. G_i is of the form

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

It is clear that $\forall(G_i\theta)$ is very close to a D_{normal} formula. That is, by proposition 4.4 we have that $\text{defq}(G_i\theta)$ is a D_{hhf} formula and so we may memoise $\forall(G_i\theta)$ by adding $\forall(\text{defq}(G_i\theta))$ to the program if this is a D_{normal} formula, and by proposition 4.2 this is equivalent to the original formula.

The reason that $\forall\text{defq}(G_i\theta)$ may not be such a formula is that there may be a subformula $D' \supset G'$ of G_i such that D' contains universal quantifiers. Thus to view memoised goals as clauses we need to allow for the possibility that universal quantifiers may appear in goals.

For example, if we find that the goal $(\forall x p(x)) \supset q$ succeeds, then we cannot store this formula as a clause due to the fact that the premise is not a G_{mod} formula.

The possibility destroys the property of the proof search process that disjunctions are not necessary in any answer form of a goal, as in a goal such as $\forall x$ even$(x) \vee$ odd(x), the disjunction cannot be removed as it was previously. Hence it may seem that there is no appropriate memoisation property for D_{mod} formulae.

Fortunately, this does not seem to be the case. We imagine that the programmer desires the greatest amount of flexibility possible when writing programs, and so whilst any program he or she writes which contains existential quantifiers or disjunctions may be re-written as an operationally equivalent program which contains neither connective, the programmer would presumably wish to use the more general form for writing programs and the more restrictive one for proving properties of programs and so forth, which may be easier in the more restrictive case. In this way there are two ways to view the program: the human view, in which we desire as great a degree of generality and flexibility as possible, and the machine view, in which we desire as great a degree of specific information as possible. The precise level of generality desired in each view is not the issue here; what is important is that the programmer and the machine tend to have two different attitudes towards the same program.

These two different views may be characterised as an object level and a meta-level. The programmer is generally concerned with the data which the program will manipulate, and so thinks of the program as a way to capture and describe the properties of the real objects of his or her concern. On the other hand, the machine sees the same program as an object which will require certain resources, so that the program itself, rather than the objects it manipulates, is the prime concern.

As we envisage memoisation being performed by the system and not by the programmer, a natural way to resolve the above difficulties is to define object level programs as above, but to allow a more general class of programs as meta-level programs, i.e. D_{hhf} formulae.

Whilst we allow some potentially more troublesome programs at the meta-level, memoising the object level programs will not "use" all the possibilities. One example of this is given by theorem 3.4, which states that existential quantification and disjunction are redundant constructs in object level programs. Thus we wish any program which satisfies the appropriate definition to be an object level program, but we do not wish for a corresponding property for meta-level programs. This is due to the fact that meta-level programs are constructed in a known way from object level programs. Hence, any meta-level program will satisfy the above definition, but we do not necessarily wish to consider anything which satisfies the definition as a meta-level program, as we will only consider meta-level programs which arise from the memoisation of object level programs. So in this case the only genuine difference between the two classes of programs will be that universal quantifiers may appear in the bodies of clauses in meta-level programs. This allow us to consider the answer form $\forall(G_i\theta)$ of a G_{mod} goal G as a program clause, and so we may consider that the program at the meta-level is now $D \cup \{\forall \text{defq}((G_i\theta))\}$.

This seems a natural way in which to view memoisation, in that the program has not changed, and so the programmer's view of it should be the same, whereas the machine's view has been extended, in that some of the consequences of the program have been calculated and then reflected back into the program. Hence the internal knowledge contained in the program remains the same; external knowledge about the program has been increased.

For example, consider the program

$$p(f(a)) \supset q$$

The goal $(\forall x p(x)) \supset q$ succeeds, and so the memoised meta-level program is

$$p(f(a)) \supset q$$
$$(\forall x\, p(x)) \supset q$$

Hence a natural consequence of the fact that the answer form of a G_{mod} goal may be stored as a D_{hhf} program rather than a D_{mod} program is that it is necessary to separate the object level from the meta-level. The process of removing disjunctions and existential quantifiers from D_{mod} programs relies heavily on the fact that there are no universal quantifiers occurring positively in goals (i.e. $\forall x\, p(x)$ is not a G_{mod} goal, but $(\forall x\, p(x)) \supset q$ is) and hence we cannot use the same class of programs for both programming and storing memoised goals. This means that memoisation must take place at the meta-level, which seems to be a natural place for it.

The separation into an object and meta-level is of growing importance in logic programming, and seems to correspond quite naturally with programming intuitions. For example, in a relational database system, one may consider the object level to consist of relations containing the data, and the meta-level to contain information such as integrity constraints. This separation into an object level and a meta-level may be seen as a step towards meta-programming.

There are some features of the meta-theory of Horn clauses which require the use of programs as first-class objects, and thus are necessarily higher-order. One such feature is that of program transformation, and has been studied in [5]. However, there are aspects which may be studied in a first-order setting, i.e. by using first-order hereditary Harrop formulae as a meta-language.

An example of such an aspect was given by Gabbay et al. In [4] it is shown how some procedural control rules for Horn clauses may be given a logical explanation in N-Prolog, an extension of Prolog, which is similar to first-order hereditary Harrop formulae. We show how the same explanation will hold for first-order hereditary Harrop formulae. One example of such a control rule is the restriction that a goal may only use certain clauses in the program. As the meta level represents a subset of first-order intuitionistic logic, seemingly non-logical or procedural operations at the object level, such as adding a clause to a program or specifying that only certain clauses are to be used in the computation of a particular goal, may be given a logical interpretation at the meta level.

Meta-theoretic aspects of logic programming have been considered by Bowen and Kowalski [2] and Bowen and Weinberg. The approach by Bowen and Kowalski, extended by Bowen and Weinberg, is centred around a Demo predicate, which takes a program as its first argument and a goal as its second, and Demo(P, G) is true if the derivation of the goal G from the program P is successful. This approach is strictly higher-order, as it involves the use of programs as objects. However, we can achieve something equivalent in hereditary Harrop formulae by the use of implication. [1]

An interesting consequence of the above results is that we may use the techniques described above to achieve a similar function to that of the Demo predicate for a wider class of programs and goals than Horn clauses – namely the object level programs and goals. Now as we know for such programs that $D \vdash_o G$ iff there is an answer form $\forall(G_i\theta)$ such that $D \vdash_o \forall(G_i\theta)$ and $\forall(G_i\theta) \vdash_I G$ (when considered as meta-level programs), if we can memoise the formula $D \supset \forall(G_i\theta)$, then this is effectively representing the Demo predicate in our language. We know that we can indeed memoise this formula, as we know that

$$D \supset \forall(G_i\theta)$$

is intuitionistically equivalent to

[1] Note that for Horn clauses, classical logic and intuitionistic logic coincide, so that the fact that we use intuitionistic logic and Bowen and Kowalski use classical logic makes no difference to this discussion.

$$\text{defp}(D) \supset \forall(\text{defq}(G_i\theta)))$$

and we may clearly write this formula as D_{hhf} clause, as indicated by proposition 4.4.

Thus whilst our conception of object and meta level does not amalgamate the two, as is done in [2], it can achieve the same end.

A further observation that may be made in the light of proposition 4.4, as object level programs may be interpreted as meta-level goals and object level answer forms may be interpreted as meta-level programs, is that we may represent the equivalence of object level programs as meta-level programs. For example, if D_1 and D_2 are object level programs, then the formula $D_1 \supset D_2$ may be re-written as a G_{hhf} formula provided that D_2 may be re-written as a G_{hhf} formula, and by proposition 4.4 we know that this is the case. Similarly, the same formula may be considered as a D_{hhf} formula provided that we may re-write D_1 as a G_{hhf} formula, which we know is possible, and we may then use the techniques suggested by the definition of defp to produce the desired D_{hhf} formula. This means that we may ask $D_1 \supset D_2$ as a goal, and if it succeeds, we may state it as a program. In this way we have not only a method for determining the equivalence of object level programs, but also a way to record the equivalences so derived. Thus we may use our conception of object level and meta-level programs as a calculus for the determining the equivalence of object level programs.

For example, let D_1 be $\exists x p(x) \supset q$ and D_2 be $p(a) \supset q$. Then $D_1 \supset D_2$ is $(\exists x p(x) \supset q) \supset (p(a) \supset q)$, and it should be clear that this is a G_{hhf} formula as D_2 is a G_{hhf} formula. Now D_1 may be re-written as $\forall x(p(x) \supset q)$, which is a G_{hhf} formula, and hence $D_1 \supset D_2$ is equivalent to a D_{hhf} formula.

Another possibility is to represent the information that two programs prove the same goal. We may rewrite the formula $(D_1 \supset G) \supset (D_2 \supset G)$ as a G_{hhf} formula by first re-writing it as $((D_1 \supset G) \wedge D_2) \supset G$, and then re-writing the first occurrence of G as $\exists(G_1 \vee \ldots \vee G_n)$, so that we get $(\exists(D_1 \supset (G_1 \vee \ldots \vee G_n)) \wedge D_2) \supset G$, which is operationally equivalent to $\forall(D_1 \supset (G_1 \vee \ldots G_n)) \wedge D_2 \supset G$. This is turn is operationally equivalent to $\forall((\bigvee_i(D_1 \supset G_i)) \wedge D_2) \supset G$, which is clearly equivalent to $\forall \bigcup_i((D_1 \supset G_i) \supset G)$, and finally we get the G_{hhf} formula

$$\forall(\bigwedge_i((\text{defq}(D_1 \supset G_i) \wedge D_2) \supset G))$$

It is difficult to see how the original formula can be re-written as a D_{hhf} formula due to the possibility that there may be existential quantifiers or disjunctions occurring positively in G. However, it is not hard to see that if $\forall(G_i\theta)$ is an answer form of G, then we may re-write the following formula as a D_{hhf} formula

$$(D_1 \supset \forall(G_i\theta)) \supset (D_2 \supset \forall(G_i\theta))$$

This is done by moving D_2 to the left of the $\supset$ as is done above, and then using the defp technique to derive a D_{hhf} formula.

This is perhaps not surprising, given that we know that the existential quantifier and disjunction are redundant constructs in D_{mod} programs, and so D_{mod} programs may be considered equivalent to D_{normal} programs. These in turn are a sub-class of the meta-level core, i.e. those formulae which are both meta-level programs and meta-level goals, which may be given as

$$M := A \mid \forall x M \mid M_1 \wedge M_2 \mid M_1 \supset M_2$$

Thus we are effectively dealing with programs which are within the core, and hence there is great potential for reflecting information back into the program.

The work of Gabbay et al. had a slightly different motivation, which was to extend the formulae available to the programmer, in a manner similar to the extension provided by hereditary Harrop formulae. It was shown in [4] that this extension allowed for the description of control information, and so the language was used to express some of its own meta-theory. In our approach, all such control information would be encoded at the meta level, thus allowing the clear separation of the (object level) program and the control information, and yet retaining a natural connection between the object and meta-level.

Gabbay and Reyle [4] have shown how the meta-language property of first-order hereditary Harrop formulae may be put to good effect. This revolves around the trick of "naming" clauses, so that we write

$$(D \supset \mathrm{name}) \supset \mathrm{name}$$

rather than just D, and similarly we write

$$(G \supset \mathrm{name}) \supset \mathrm{name}$$

in place of G, where name does not appear anywhere in either D or G. We may use this device to control which clauses are used in the computation and which are not, so that we may specify that the computation for the goal G may only use the clause D_1 and not the clause D_2 by renaming both clauses as

$$\begin{array}{l} (D_1 \supset \mathrm{name}_1) \supset \mathrm{name}_1 \\ (D_2 \supset \mathrm{name}_2) \supset \mathrm{name}_2 \end{array}$$

and the goal G as $(G \supset \mathrm{name}_1) \supset \mathrm{name}_1$, where name_1 and name_2 do not appear anywhere in D_1, D_2 or G. The idea is that due to the fact that the names are new, the computation of G can only gain access to the clause D_1 and not to D_2. The first step is to add $G \supset \mathrm{name}_1$ to the program, so that we get

$$\begin{array}{l} G \supset \mathrm{name}_1 \\ (D_1 \supset \mathrm{name}_1) \supset \mathrm{name}_1 \\ (D_2 \supset \mathrm{name}_2) \supset \mathrm{name}_2 \end{array}$$

and the goal is now name_1. As neither name_1 nor name_2 appear in G, the second clause cannot lead to a success, and so the only way for name_1 to succeed is to use the first clause above, so that the goal is now

$$D_1 \supset \mathrm{name}_1$$

Now the program becomes

$$\begin{array}{l} D_1 \\ G \supset \mathrm{name}_1 \\ (D_1 \supset \mathrm{name}_1) \supset \mathrm{name}_1 \\ (D_2 \supset \mathrm{name}_2) \supset \mathrm{name}_2 \end{array}$$

and the goal is name_1. Now provided that we can prevent name_1 from matching against the third clause above and thus looping, the only possible match is the second clause, so that the goal becomes G. As neither name_1 nor name_2 appear in D_1, D_2 or G, only D_1 can be used in the computation of G, and not D_2.

It is then shown in [4] how a simple loop detection mechanism may be provided, so that the above process may work. However, it is possible to provide the same facility in such a way that the loop detection is not required. If we name the two clauses in the following manner

$$
\begin{array}{l}
(D_1 \supset \text{fire}) \supset \text{load}_1 \\
(D_2 \supset \text{fire}) \supset \text{load}_2
\end{array}
$$

and the goal as $(G \supset \text{fire}) \supset \text{load}_1$, where fire, load_1 and load_2 do not appear anywhere in D_1, D_2 or G, then we may achieve the same effect. The computation may be performed as follows: first we add $G \supset \text{fire}$ to the program, giving us

$$
\begin{array}{l}
G \supset \text{fire} \\
(D_1 \supset \text{fire}) \supset \text{load}_1 \\
(D_2 \supset \text{fire}) \supset \text{load}_2
\end{array}
$$

and the goal load_1. The only clause that this can match is the second one, and so the new program is

$$
\begin{array}{l}
D_1 \\
G \supset \text{fire} \\
(D_1 \supset \text{fire}) \supset \text{load}_1 \\
(D_2 \supset \text{fire}) \supset \text{load}_2
\end{array}
$$

and the new goal is fire. This can only match the second clause, and so the next goal is G, thus allowing G only to use D_1 and not D_2, as none of fire, load_1 and load_2 can match anything in G.

In this way we can get a more straightforward implementation of deterministic exclusion, i.e. using one given clause but not another during computation. A refinement of the above idea can give non-deterministic exclusion, where it does not matter which clause is used, provided that once some clause is chosen, the other clause cannot be used. All we need to do is to use the same name for both clauses, so that D_1 and D_2 become

$$
\begin{array}{l}
(D_1 \supset \text{fire}) \supset \text{load} \\
(D_2 \supset \text{fire}) \supset \text{load}
\end{array}
$$

and G becomes $(G \supset \text{fire}) \supset \text{load}$. The computation proceeds by adding the clause $G \supset \text{fire}$ to the program and the goal is then load. Now there are two clauses it may match, and so the next goal is $D_i \supset \text{fire}$, where $i = 1$ or 2, which involves adding D_i to the program, and the next goal is G. Hence, which clause is added may be decided arbitrarily, but once it is chosen, only that clause and not the other may be used in the computation of G.

If we consider the order of the clauses in the program to be significant (as happens in Prolog systems), then, as shown in [4], we may use this device to arbitrarily re-order clauses in the program. This is done by encoding the desired order of the clauses in a goal so that the clauses are added in the desired order before the goal is asked. For example, if there are three clauses

$$(D_1 \supset \text{fire}_1) \supset \text{load}_1$$
$$(D_2 \supset \text{fire}_2) \supset \text{load}_2$$
$$(D_3 \supset \text{fire}_3) \supset \text{load}_3$$

and the desired order is D_2 followed by D_3 followed by D_1, then this order may be achieved if we use the goal

$$(((((G \supset \text{fire}_2) \supset \text{load}_2) \supset \text{fire}_3) \supset \text{load}_3) \supset \text{fire}_1) \supset \text{load}_1$$

and we assume that clauses are added to the beginning of the program. Thus we first load D_1, then D_3, and finally D_2, thus giving the required order, before the computation of G. Thus we represent the six different object level programs (as the order of the clauses is significant here) by the same meta-level program, but with six different goals, one corresponding to each arrangement of the clauses. This is intuitively attractive, as the meta-level program is the same in each case, but the way it is used in computation varies, which seems to capture our intuitive notion of the difference between the six object level programs, i.e. that they all represent the same declarative information, but vary on the operational details.

Another possible application of this separation into an object level program and a meta-level program is to deductive databases. A deductive database often contains a set of conditions which must be satisfied at all times in order to guarantee that the information represented in the database is consistent and does not get garbled. Such conditions are known as integrity constraints, and are usually checked after each database update to ensure that the changes will not render the database useless or redundant. Hence, the constraints are not themselves part of the database, but conceptually separate, and we consider a change to the object level program, i.e. a database update, to be conceptually very different from a change to the integrity constraints. By considering the integrity constraints as a meta-level program in the above sense, we get a natural representation of a database and its associated information.

The point of these examples is to show that the separation of computation into an object level program and a meta-level program seems to be a natural one, and this difference in level may be naturally studied in the context of hereditary Harrop formulae. This approach may be used to show how some typical programming techniques, such as memoisation and clause re-ordering, may be given an explanation in terms of logic, rather than purely operational notions.

Clearly this approach cannot be a full answer to the problems raised, such as the fact that $\wedge$ is not necessarily commutative, as the order in which the conjuncts are selected may affect termination properties. However, it does show that such properties are not as "extra-logical" as may be initially thought. Fuller and better answers are probable using a higher-order approach, but it is interesting to note that "meta" and "higher-order" need not be synonymous. Rather than merely let the meta-theory of logic programming be expressed in the (usually) informal language used by computer scientists to communicate with one another, we interpolate a third level between the object level and this informal meta-level. As formulae of this new level may be given a computational interpretation, we may think of this as an executable meta-language, in which programming and to some extent meta-programming may both be accommodated.

6 Conclusion

We have seen how existential quantifications and disjunctions may be removed from D_{mod} programs, and how successful G_{mod} goals may be stored as a clause. However, this clause may not be a D_{mod} formula, and hence we require D_{hhf} formulae to make this process work.

Clearly it would be advantageous to have a similar memoisation property for D_{hhf}, and so increase the power of the object level. This will not be as straightforward as in the case for D_{mod} programs. For example, in the clause below there is no way to remove the disjunction.

$$(\forall x p(x) \vee q(x)) \supset r$$

One approach to this problem may be to use infinitary logic, so that the above clause may be considered equivalent to

$$(\bigwedge_{t \in \mathcal{U}} (p(t) \vee q(t))) \supset r$$

and then use techniques similar to those above. Another problem is given by the clause

$$(\forall x \exists y p(x, y)) \supset r$$

The traditional way around this is to use Skolem functions, so that the above is equivalent to

$$(\exists f \forall x p(x, f(x)) \supset r$$

This clearly involves quantifications over functions, and hence is no longer first-order. It is possible that one or both of these approaches may be useful in deriving both a normal form and memoisation property for D_{hhf} programs. Such a property would mean that one may use the same language for both the object level and the meta-level.

7 Acknowledgements

My thanks go to Dale Miller, for many stimulating discussions, and to Don Sannella, for his continual support and critical reading. Comments from Peter Stuckey and some anonymous referees were also very helpful.

This research was carried out whilst the author was a student in the Department of Computer Science at the University of Edinburgh.

References

[1] H. Abramson and M.H. Rogers (eds.), *Meta-Programming in Logic Programming*, MIT Press, 1989.

[2] K.A. Bowen and R.A. Kowalski, Amalgamating Language and Metalanguage in Logic Programming, in *Logic Programming* 153-172, K.L. Clark and S.-A. Tärnlund (eds.), Academic Press, 1982.

[3] K.A. Bowen and T. Weinberg, *A Meta-Level Extension of Prolog*, Technical Report CIS-85-1, Syracuse University, 1985.

[4] D. Gabbay and U. Reyle, N-Prolog: An Extension of Prolog with Hypothetical Implications. I., *Journal of Logic Programming* 1:319-355, 1984.

[5] J. Hannan and D.A. Miller, Uses of Higher-Order Unification for Implementing Program Transformers, *Proceedings of the International Conference and Symposium on Logic Programming*, Seattle, August, 1988.

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

[7] J. Harland, Structural Properties of Logic Programs, Technical Report 90/20, Department of Computer Science, University of Melbourne.

[8] J. Harland, An Intermediate Logic for Logic Programs, Technical Report, Department of Computer Science, University of Melbourne, forthcoming.

[9] S.C. Kleene, *Introduction to Metamathematics*, North-Holland, 1952.

[10] M.J. Maher, Equivalences of Logic Programs, in *Foundations of Deductive Databases and Logic Programming* 627-658, J. Minker (ed.), Morgan Kaufmann, 1988.

[11] D. Michie, Memo functions and machine learning, *Nature* 218:19-22, April, 1968.

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

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

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

[15] A.S. Troelstra, *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, Lecture Notes in Mathematics 344, Springer-Verlag, Berlin, 1973.

A Proof of the Disjunctive Identity

LEMMA A.1 $(G_1 \vee G_2) \supset A \equiv_I (G_1 \supset A) \wedge (G_2 \supset A)$

Proof:

($\Leftarrow$): From figure 3 it is clear that

$$G_i, (G_i \supset A), (G_j \supset A) \vdash_I A$$

when either $i = 1, j = 2$ or $i = 2, j = 1$. Thus we may derive the uppermost sequents in figure 4, which in turn clearly shows the result.

($\Rightarrow$): It is clear that the uppermost sequents of figure 5 may be derived, as it is obvious that

$$\begin{gathered} G_i \vdash_I G_1 \vee G_2 \\ A, G_i \vdash_I A \end{gathered}$$

where $i = 1, 2$. Hence, the result follows.

$\square$

$$\dfrac{G_i, G_j \supset A \longrightarrow G_i \qquad\qquad A, G_i, G_j \supset A \longrightarrow A}{G_i, G_i \supset A, G_j \supset A \longrightarrow A}\ \supset\text{-L}$$

Figure 3: Useful subproof

$$\dfrac{\dfrac{\dfrac{G_1, G_1 \supset A, G_2 \supset A \longrightarrow A \qquad\qquad G_2, G_1 \supset A, G_2 \supset A \longrightarrow A}{G_1 \vee G_2, G_1 \supset A, G_2 \supset A \longrightarrow A}\ \vee\text{-L}}{G_1 \supset A, G_2 \supset A \longrightarrow (G_1 \vee G_2) \supset A}\ \supset\text{-R}}{(G_1 \supset A) \wedge (G_2 \supset A) \longrightarrow (G_1 \vee G_2) \supset A}\ \wedge\text{-L}$$

Figure 4: Proof that $(G_1 \supset A) \wedge (G_2 \supset A) \vdash_I (G_1 \vee G_2) \supset A$

$$\dfrac{\dfrac{\dfrac{G_1 \longrightarrow G_1 \vee G_2 \qquad A, G_1 \longrightarrow A}{G_1, (G_1 \vee G_2) \supset A \longrightarrow A}\ \supset\text{-L}}{(G_1 \vee G_2) \supset A \longrightarrow G_1 \supset A}\ \supset\text{-R} \qquad \dfrac{\dfrac{G_2 \longrightarrow G_1 \vee G_2 \qquad A, G_2 \longrightarrow A}{G_2, (G_1 \vee G_2) \supset A \longrightarrow A}\ \supset\text{-L}}{(G_1 \vee G_2) \supset A \longrightarrow G_2 \supset A}\ \supset\text{-R}}{(G_1 \vee G_2) \supset A \longrightarrow (G_1 \supset A) \wedge (G_2 \supset A)}\ \wedge\text{-R}$$

Figure 5: Proof that $(G_1 \vee G_2) \supset A \vdash_I (G_1 \supset A) \wedge (G_2 \supset A)$