gram is defined as a collection of rules of the shape:

$$a_1; \dots a_m; not\ b_1; \dots not\ b_n \leftarrow c_1, \dots c_r, not\ d_1, \dots not\ d_s \quad (1)$$

We call *head* (resp. *body*) to the left (resp. right) hand side of the arrow in (1). The comma and the semicolon are alternative representations of conjunction $\wedge$ and disjunction $\vee$, respectively. When $m = n = 0$ we usually write $\bot \leftarrow B$ instead of $\leftarrow B$, whereas when $r = s = 0$ we directly write H instead of $H \leftarrow$ or $H \leftarrow \top$.

Sometimes, it will be convenient to think about program rules as classical propositional formulas, where $\leftarrow$ and *not* are respectively understood as material implication and classical negation. In this way, the usual expression $I \models R$ denotes that interpretation I satisfies rule R (seen as a classical formula), whereas $I \models \Pi$ means that I is a model of the program Π (seen as a classical theory).

The *reduct* of a program Π w.r.t. some set of atoms I, written Π^I, is defined as the result of replacing in Π any default literal *not* p by $\top$, if $p \notin I$, or by $\bot$ otherwise.

Definition 1 (Stable model) *A set of atoms $I \subseteq \Sigma$ is a* stable model *of a logic program Π iff I is a minimal model of Π^I.* □

Strong equivalence in classical logic

We can think about the definition of stable models as a try-and-error procedure which handles (propositional) interpretations for two different purposes. On the one hand, we start from some arbitrary interpretation I^a, we can call the initial *assumption*, used to compute the reduct Π^{I^a}. On the other hand, in a second step, we deal with minimal models of Π^{I^a} which, in principle, *need not to have any connection* with I^a. Each minimal model I^p can be seen as the set of propositions we can *prove* by deductive closure using the rules in Π^{I^a}. When the proved atoms coincide with the initial assumption, $I^p = I^a$, a stable model is obtained.

In order to capture this behavior, we reify all the atoms $p \in \Sigma$ to become arguments of two unary predicates, $assumed(p)$ and $proved(p)$, that respectively talk about I^a and I^p. Sort variable X will be used for ranging over any propositional symbol in Σ. When considering the models of any reified formula F, we will implictly assume that they actually correspond to $F \wedge$ UNA, where UNA stands for the unique names assumption for sort Σ. This allows us identifying any Herbrand model M of this type of formulas with a pair[2] (I^p, I^a) so that $M[assumed] = I^a$ and $M[proved] = I^p$. Expression $M \models F$ represents again satisfaction of reified formulas – ambiguity with respect to $I \models F$ is cleared by the shape of structures and formulas.

Given this simple framework, we provide two encodings: the first one is a *completely straightforward* translation to capture stable models, whereas the second one is a stronger translation to characterize strong equivalence.

[2]The superscripts p and a, which stand here for *proved* and *assumed*, respectively correspond to the worlds *here* and *there* in HT or to the sets of *positive* and *non-negative* atoms in L_3.

Definition 2 (First translation) *For any logic program rule R like (1), we define the classical formula $\dot{R}$ as the material implication:*

$$\Big(\bigwedge_{i=1}^{r} proved(c_i)\Big) \wedge \Big(\bigwedge_{i=1}^{s} \neg assumed(d_i)\Big) \supset \Big(\bigvee_{i=1}^{n} proved(a_i)\Big) \vee \Big(\bigvee_{i=1}^{m} \neg assumed(b_i)\Big) \quad (2)$$

Given a logic program Π, the formula $\dot{\Pi}$ stands for the conjunction of all the $\dot{R}$, for each rule $R \in \Pi$. □

Intuitively, to obtain the minimal models I^p of Π^{I^a} we can use an ordering relation among pairs $(I^p, I^a) \preceq (J^p, J^a)$ that holds when both $I^a = J^a$ is fixed and $I^p \subseteq J^p$. The corresponding models $\preceq$-minimization have a simple syntactic counterpart[3]: predicate circumscription CIRC$[\dot{\Pi}; proved]$.

After circumscription captures the minimal models, we must further require $I^p = I^a$, that is, we want pairs of shape (I, I) where what we assumed results to be exactly what we proved. These pairs of shape (I, I) will be called *total*. Clearly, forcing models to be total corresponds to including the formula:

$$\forall X.\ \big(proved(X) \equiv assumed(X)\big) \quad (3)$$

The intuitions above are not new. In fact, they were used in Theorem 5.2 in (Lin & Shoham 1992) which, adapted[4] to our current presentation, states the following result:

Proposition 1 *Let Σ be a propositional signature. A set of atoms $I \subseteq \Sigma$ is a stable model of a logic program Π iff $M = (I, I)$ satisfies the formula:*

$$\text{CIRC}[\dot{\Pi}; proved] \wedge (3)$$

□

In order to capture strong equivalence of two programs, it seems that we should not only compare the final selected models, but also the set of non-minimal ones involved in the minimization. For instance, it is easy to see that, due to monotonicity of classical logic, the following proposition trivially applies:

Proposition 2 *Let Π_1 and Π_2 be two logic programs such that $\models \dot{\Pi}_1 \equiv \dot{\Pi}_2$. Then Π_1 and Π_2 are strongly equivalent.* □

Unfortunately, the opposite does not necessarily hold: Π_1 and Π_2 can be strongly equivalent while $\dot{\Pi}_1$ and $\dot{\Pi}_2$ have different models. This is because encoding in Definition 2 allows some models which are actually irrelevant for strong equivalence, as we will show next.

[3]See Section 2.5 in (Lifschitz 1993).

[4]In (Lin & Shoham 1992) they used a duplicated signature (atoms p and p') instead of reification and, therefore, they actually applied parallel circumscription. This result seems to have been first presented in (Lin 1991).

Definition 3 (Subtotal model) *For any reified theory T, a model (I^p, I^a) of T is called* subtotal *iff (I^a, I^a) is also model of T and $(I^p, I^a) \preceq (I^a, I^a)$.* □

Let $\mathrm{SUBT}(T)$ represent the set of subtotal models of T (note that total models are also included). It is clear that any model $M \notin \mathrm{SUBT}(\dot\Pi)$ is irrelevant for selecting the total $\preceq$-minimal models, i.e., for obtaining the stable models of Π. The next theorem shows that the coincidence of subtotal models is a necessary condition for strong equivalence. The proof (available at (Cabalar 2002)) constitutes a direct rephrasing of that for the main theorem in (Lifschitz, Pearce, & Valverde 2000).

Theorem 1 *Two logic programs Π_1 and Π_2 are strongly equivalent iff* $\mathrm{SUBT}(\dot\Pi_1) = \mathrm{SUBT}(\dot\Pi_2)$ □

Theorem 1 points out that the $\dot\Pi$ encoding is still too weak for a full characterization of strong equivalence. We show next how, using a more restrictive translation (that is, adding more formulas) it is possible to obtain theories for which all their models are subtotal. To understand how to do this, consider the example program $\Pi_0 = \{p \leftarrow q\}$ where:

$$\dot\Pi_0 \stackrel{\mathrm{def}}{=} proved(q) \supset proved(p)$$

This formula has 12 models: it restricts the extent of $proved$ to 3 cases ($\emptyset$, $\{p\}$ and $\{p, q\}$) leaving free, in each case, the 4 possibilities for $assumed$.

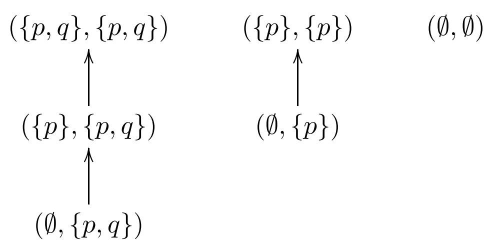

Figure 1: Subtotal models of $\dot\Pi_0$.

Figure 1 shows the 6 subtotal models of $\dot\Pi_0$, representing the $\preceq$-ordering relationships among them. Notice how subtotal models always satisfy $I^p \subseteq I^a$, that is, we can require:

$$\forall X. \big(proved(X) \supset assumed(X)\big) \tag{4}$$

Unfortunatley, the addition of this axiom is still not enough to rule out all the irrelevant models. For instance, $\dot\Pi_0 \wedge (4)$ has still one non-subtotal model: $(\emptyset, \{q\})$. This model, however, has the particularity that its assumed atoms $I^a = \{q\}$ do not satisfy the original program rule: $I^a \not\models p \leftarrow q$. As it is well-known, any stable model I of a program Π, is also a classical model: $I \models \Pi$. So, instead of starting from any arbitrary initial assumption I^a, we can begin requiring $I^a \models \Pi$. This can be easily incorporated into the encoding as follows. For each logic program rule R like (1), we define $\ddot R$ as:

$$\big(\bigwedge_{i=1}^{r} assumed(c_i)\big) \wedge \big(\bigwedge_{i=1}^{s} \neg assumed(d_i)\big) \supset \big(\bigvee_{i=1}^{n} assumed(a_i)\big) \vee \big(\bigvee_{i=1}^{m} \neg assumed(b_i)\big) \tag{5}$$

Again, $\ddot\Pi$ stands for the conjunction of $\ddot R$ for all $R \in \Pi$.

Definition 4 (Second translation) *For any logic program Π we define the formula $\Pi^* \stackrel{\mathrm{def}}{=} \dot\Pi \wedge \ddot\Pi \wedge (4)$.* □

The proof for the following theorem uses well-known properties of circumscription (see (Lifschitz 1993)) to show that the additional formulas *do not affect* to the final set of stable models.

Theorem 2 *For any logic program Π:*
$\mathrm{CIRC}[\Pi^*; proved] \wedge (3) \equiv \mathrm{CIRC}[\dot\Pi; proved] \wedge (3)$ □

But, of course, the real interest of Π^* is that it finally rules out irrelevant models:

Property 1 *Let Π be a logic program. Then, any model $M = (I^p, I^a)$ of Π^* is subtotal.* □

Finally, this property, together with theorem 1, directly implies:

Theorem 3 *Two logic programs Π_1 and Π_2 are strongly equivalent iff* $\models \Pi_1^* \equiv \Pi_2^*$. □

Nested expressions.

The previous section has shown that strong equivalence of logic programs can be reduced to a simple equivalence test in classical logic, providing in this way a (we think) easier alternative to the HT characterization. Unfortunately, the applicability of the classical encoding is limited for rules of shape (1), whereas the HT formalization is still applicable to a more flexible rule syntax like, for instance, rules with nested expressions.

In (Lifschitz, Tang, & Turner 1999) a more general shape for program rules was considered. A *nested expression* is defined as any propositional combination of atoms with 0-ary operators $\bot$, $\top$, unary operator *not* and binary operators ',' and ';'. A logic program is now a set of rules $Head \leftarrow Body$ where $Head$ and $Body$ are nested expressions (notice that the rule conditional $\leftarrow$ is the only operator that cannot be nested). An example of rule could be, for instance:

$$a, b \leftarrow not\ (c; not\ d) \tag{6}$$

Stable models for this kind of programs can be easily described by a simple modification in the definition of program reduct. We define now Π^I as the result of replacing in Π every maximal occurrence[5] of *not* F by $\bot$ if $I \models F$ or by $\top$ otherwise. Note that the previous definition of reduct corresponds to the particular case in which F is an atom.

An interesting result derived from this modified semantics (proposition 7 in (Lifschitz, Tang, & Turner 1999)) is that

[5]That is, any *not* F that is not in the scope of an outer *not* .

any program with nested expressions is strongly equivalent to some (non-nested) program, just consisting of rules like (1). To obtain this non-nested program, the following transformations are defined. Let F, G and H represent nested expressions. By $\alpha \Leftrightarrow \beta$ we mean that we replace some regular occurrence of α by β. Then, we handle the following strongly equivalent transformations:

(i) $F, G \Leftrightarrow G, F$ and $F; G \Leftrightarrow G; F$.
(ii) $(F, G), H \Leftrightarrow F, (G, H)$ and $(F; G); H \Leftrightarrow F; (G; H)$.
(iii) $F, (G; H) \Leftrightarrow (F, G); (F, H)$ and $F; (G, H) \Leftrightarrow (F; G), (F; H)$.
(iv) $\mathit{not}\ (F; G) \Leftrightarrow \mathit{not}\ F, \mathit{not}\ G$ and $\mathit{not}\ (F, G) \Leftrightarrow \mathit{not}\ F; \mathit{not}\ G$.
(v) $\mathit{not\ not\ not}\ F \Leftrightarrow \mathit{not}\ F$.
(vi) $F, \top \Leftrightarrow F$ and $F; \top \Leftrightarrow \top$.
(vii) $F, \bot \Leftrightarrow \bot$ and $F; \bot \Leftrightarrow F$.
(viii) $\mathit{not}\ \top \Leftrightarrow \bot$ and $\mathit{not}\ \bot \Leftrightarrow \top$.
(ix) $(F, G \leftarrow H) \Leftrightarrow (F \leftarrow H), (G \leftarrow H)$.
(x) $(F \leftarrow G; H) \Leftrightarrow (F \leftarrow G), (F \leftarrow H)$.
(xi) $(F \leftarrow G, \mathit{not\ not}\ H) \Leftrightarrow (F; \mathit{not}\ H \leftarrow G)$.
(xii) $(F; \mathit{not\ not}\ G \leftarrow H) \Leftrightarrow (F \leftarrow \mathit{not}\ G, H)$.

For instance, rule (6) can be successively transformed as follows:

$a, b \leftarrow \mathit{not}\ c, \mathit{not\ not}\ d.$ By (iv)

$a \leftarrow \mathit{not}\ c, \mathit{not\ not}\ d,$
$b \leftarrow \mathit{not}\ c, \mathit{not\ not}\ d.$ By (ix)

$a; \mathit{not}\ d \leftarrow \mathit{not}\ c,$
$b; \mathit{not}\ d \leftarrow \mathit{not}\ c.$ By (xi)

This treatment of nested expressions exceeds the applicability of our previous classical logic representation. From a practical point of view, such a limitation is not very important, since we can always unfold nested expressions by applying (i)-(xii). Nevertheless, from a theoretical point of view, this clearly points out that the classical encoding fails as a real semantic characterization for LP connectives.

As shown in (Lifschitz, Pearce, & Valverde 2000), one of the important features of the HT formalization, apart from the result for strong equivalence, is that it preserves the above interpretation of nested expressions. We show next that a similar behavior can be obtained using standard 3-valued logic (L_3). Surprisingly, L_3 provides the same interpretation for nested expressions, but generally differs once free nesting of rule conditionals is allowed.

L_3: Three valued logic.

We will use propositional syntax plus Lukasiewicz's unary operator[6] $\boldsymbol{l}$. Intuitively, a formula $\boldsymbol{l}\,F$ is never unknown and points out that F is valuated to true. In this way, $\neg \boldsymbol{l}\,F$ would mean that "F is not true," i.e., it is either false or unknown.

[6]For instance, see (Bull & Segerberg 1984), pag. 8, where $\boldsymbol{l}$ is denoted as $\Box$.

If F, G are L_3 formulas and p an atom of the signature Σ then:

$$p,\ \neg F,\ F \vee G,\ \top,\ \bot,\ \boldsymbol{l}\,F$$

are also L_3 formulas. Propositional derived operators ($\wedge, \supset, \equiv$) are defined in the usual way.

A three valued interpretation M is a function $M : \Sigma \longrightarrow \{0, 1/2, 1\}$ assigning to each atom $p \in \Sigma$ a truth value $M(p)$ which can be 0 (false), 1/2 (unknown) or 1 (true). We will usually represent M as the pair of sets of atoms (I^p, I^a) respectively containing the *positive* (true) and *consistent* (non-false) atoms where, of course, we require consistence: $I^p \subseteq I^a$. Consequently:

$$M(p) = \begin{cases} 1 & \text{if } p \in I^p \\ 0 & \text{if } p \notin I^a \\ 1/2 & \text{otherwise} \end{cases}$$

Note that we use here the same notation as for the pairs we handled in the reified approach. This is not casual: the negative information of a 3-valued interpretation will be used to represent default negation, whereas the positive information will represent the set of proved atoms.

Definition 5 (**L_3 valuation of a formula**)
We extend the valuation function M to any formula F, $M(F) \in \{0, 1/2, 1\}$, so that:

1) $M(\top) = 1$ and $M(\bot) = 0$
2) $M(\neg F) = 1 - M(F)$
3) $M(F \vee G) = max(M(F), M(G))$
4) $M(\boldsymbol{l}\,F) = \begin{cases} 1 & \text{if } M(F) = 1 \\ 0 & \text{otherwise} \end{cases}$

□

An interpretation M *satisfies* a formula F, written $M \models_3 F$ when $M(F) = 1$. When F is satisfied by *any* interpretation, we call it an L_3-*tautology* and write $\models_3 F$. As usual, an interpretation is a *model* of a theory when it satisfies all its formulas. Maintaining the previous terminology, a 3-valued interpretation M is called *total* iff it has the shape $M = (I, I)$, that is, it contains no unknown atoms. Clearly, when considering total interpretations, the $\boldsymbol{l}$ operator can be simply removed, and L_3 collapses into 2-valued propositional logic.

LP connectives are simply defined among the following derived operators:

$$\begin{aligned} \boldsymbol{m}\,F &\stackrel{\text{def}}{=} \neg \boldsymbol{l}\,\neg F \\ \mathit{not}\ F &\stackrel{\text{def}}{=} \neg \boldsymbol{m}\,F \\ G \leftarrow F &\stackrel{\text{def}}{=} (\boldsymbol{l}\,F \supset \boldsymbol{l}\,G) \wedge (\boldsymbol{m}\,F \supset \boldsymbol{m}\,G) \\ F \leftrightarrow G &\stackrel{\text{def}}{=} (F \leftarrow G) \wedge (G \leftarrow F) \end{aligned}$$

It is easy to check that the derived semantics for each one of these operators corresponds to:

5) $M(\boldsymbol{m}\,F) = \begin{cases} 1 & \text{if } M(F) \neq 0 \\ 0 & \text{otherwise} \end{cases}$

6) $M(\mathit{not}\ F) = \begin{cases} 1 & \text{if } M(F) = 0 \\ 0 & \text{otherwise} \end{cases}$

7) $M(G \leftarrow F) = \begin{cases} 1 & \text{if } M(F) \leq M(G) \\ 0 & \text{otherwise} \end{cases}$

8) $M(F \leftrightarrow G) = \begin{cases} 1 & \text{if } M(F) = M(G) \\ 0 & \text{otherwise} \end{cases}$

Operator $\boldsymbol{m}$ acts is the dual of $\boldsymbol{l}$, ($\boldsymbol{m}\, F$ can be read as "F is consistent") whereas implication $\leftarrow$ is the one proposed by Fitting (Fitting 1985) and Kunen (Kunen 1987). When we represent some program Π inside L_3, we will consider it as a single formula consisting in the conjunction of all the program rules. Note also that when $\models_3 F \leftrightarrow G$, we can apply uniform substitution in L_3 as we would do in classical propositional logic. For instance, $\models_3 (\textit{not } F) \leftrightarrow (\bot \leftarrow F)$ means that we can replace any occurrence of $(\textit{not } F)$ by $(\bot \leftarrow F)$ and vice versa. Let $\circ$ and $\bullet$ be two meta-operators, any of them indistinctly representing $\boldsymbol{l}$ or $\boldsymbol{m}$. Then, the following formulas are also L_3 tautologies:

$$\circ (F \wedge G) \leftrightarrow \circ F \wedge \circ G \tag{7}$$

$$\circ (F \vee G) \leftrightarrow \circ F \vee \circ G \tag{8}$$

$$\bullet \circ F \leftrightarrow \circ F \tag{9}$$

$$\bullet \neg \circ F \leftrightarrow \neg \circ F \tag{10}$$

As $\boldsymbol{m}$ is defined in terms[7] of $\boldsymbol{l}$, this means that we can unfold any L_3 formula until $\boldsymbol{l}$ is exclusively applied to literals. Using these properties, the following lemma can be easily proved:

Lemma 1 *Let R be a program rule like (1), and let the pair $M = (I^p, I^a)$ have the common shape of an L_3-interpretation and a classical interpretation for proved/assumed. Then, $M \models_3 R$ iff $M \models \dot{R} \wedge \ddot{R}$.* □

Besides, by inspection on L_3 semantics, we also have that:

Lemma 2 *For any transformation $\alpha \Leftrightarrow \beta$ in* (i)-(xii): $\models_3 \alpha \leftrightarrow \beta$. □

Theorem 4 *Let Π_1 and Π_2 be two logic programs possibly containing nested expressions. Then Π_1 and Π_2 are strongly equivalent iff: $\models_3 \boldsymbol{l}\,\Pi_1 \equiv \boldsymbol{l}\,\Pi_2$.* □

Notice that we check $\boldsymbol{l}\,\Pi_1 \equiv \boldsymbol{l}\,\Pi_2$ instead of the stronger condition $\Pi_1 \leftrightarrow \Pi_2$. To understand the difference, consider $\Pi_1 = \{a\}$ and $\Pi_2 = \{a \leftarrow \top\}$. The interpretation $M = (\{a\},\{a\})$ is the only model of both programs and so, $\models_3 \boldsymbol{l}\,\Pi_1 \equiv \boldsymbol{l}\,\Pi_2$. However, $\Pi_1 \leftrightarrow \Pi_2$ is not a tautology, since $M' = (\emptyset, \{a\})$ makes $M'(\Pi_1) = 1/2 \neq 1 = M'(\Pi_2)$.

Differences with respect to HT

Theorem 4 shows that HT and L_3 coincide in their interpretations of programs with nested expressions. The next natural question is, do the HT and L_3 interpretations coincide for any arbitrary theory? The answer to this question is negative, as we will show with a pair of counterexamples. Of course, due to theorem 4, these counterexamples cannot be just programs with nested expressions, as defined in the fourth section. We study, for instance, a nested conditional, and the negation of a conditional.

Consider the theory consisting of the singleton formula $(a \leftarrow b) \leftarrow c$. In HT, this theory is equivalent to $(a \leftarrow b, c)$, which seems to be the most intuitive solution, whereas in L_3 it is actually equivalent to $(a; \textit{not } c \leftarrow b)$. Both equivalences hold in classical propositional logic. However, for computing stable models, their behavior is quite different. For instance, the theory $\{b, (a \leftarrow c), (c \leftarrow a), ((a \leftarrow b) \leftarrow c)\}$ would have a unique stable model $\{b\}$ under the HT interpretation whereas, under L_3, an additional stable model $\{a, b, c\}$ is obtained.

The second example shows the most important problem of the L_3 interpretation: once we allow arbitrary theories, we may obtain non-subtotal models, something that does not happen[8] in HT. Let Π be the theory $\{b, \textit{not } (a \leftarrow b)\}$. Its unique stable model is $\{b\}$ both in HT and L_3. However, while the pair $(\{b\},\{b\})$ is the unique HT model[9] of Π, in L_3 there exists a second model $(\{b\},\{a,b\})$ which is not subtotal. In other words, when using L_3 for this general syntax, the set of L_3 models does not fully characterize strong equivalence.

Discussion

The study of strong equivalence is probably one of the most active current topics in research in Logic Programming, as it becomes evident by the increasing amount of new results obtained recently (just to cite three examples (Turner 2001; Pearce, Tompits, & Woltran 2001; de Jongh & Hendriks 2001)).

In (Pearce, Tompits, & Woltran 2001), a classical logic characterization is also provided, which presents several similarities with the approach we present here. The main difference of Pearce et al's method is that it actually relies on a syntactic translation from HT into classical logic. This translation informally consists in a duplication of the atoms in the signature so that an atom p denotes our *proved* whereas an atom p' would denote *assumed*. In this paper, our initial motivation for using classical logic was to improve the presentation and the understanding. In this way, we have directly started from non-nested programs, trying to capture the definition of stable models in a way as direct as possible. As a result, our characterization does not provide an interpretation of nested connectives. In order to deal with them, we could apply a previous step, using transformations (i)-(xii). Pearce et al's encoding starts from HT logic, and so, deals with nested expressions (in the same way as HT does). Besides, the transformation presented in (Pearce, Tompits, & Woltran 2001) has the additional advantage of being linear, while (i)-(xii) are not polynomial in the general case. Despite of these two advantages of Pearce et al's approach, it must be noticed that none of the two classical encodings can actually be considered a full-semantics for nested logic programs, since *in both cases*, a previous syntactic transformation is required. Therefore, translation to classical logic is very interesting for practical purposes, but is limited from a purely semantic point of view.

[7] Of course, we could have equally chosen the dual operator $\boldsymbol{m}$ as the basic one.

[8] See for instance Fact 1 in (Lifschitz, Pearce, & Valverde 2000).

[9] In fact, the expression $\textit{not } (a \leftarrow b)$ is HT-equivalent to the pair of constraints $(\bot \leftarrow \textit{not } b)$ and $(\bot \leftarrow a)$.

Another similarity between our classical encoding with respect to (Pearce, Tompits, & Woltran 2001) is, not only how to decide strong equivalence, but how to obtain stable models. In our case, we simply used to that purpose the result presented by Lin and Shoham in (Lin & Shoham 1992) and then included slight variations that we proved to be sound. In (Pearce, Tompits, & Woltran 2001), a quantified boolean formula is used instead:

$$\phi' \wedge \neg\exists V((V < V') \wedge \tau_{HT}[\phi]) \qquad (11)$$

where V is the set of atoms, ϕ is the original program, ϕ' results from replacing any atom p by p' and finally $\tau_{HT}[\phi]$ is Pearce et al's translation from HT to classical logic. On the other hand, Lin and Shoham's result involving circumscription can be formulated[10] as:

$$(V = V') \wedge \mathcal{C}[\phi] \wedge \neg\exists V((V < V') \wedge \mathcal{C}[\phi]) \qquad (12)$$

where $\mathcal{C}[\phi]$ simply replaces each *not* p by $\neg p'$. Notice how, at least structurally, (12) is very similar to (11).

As for the L_3 encoding, it must also be noticed that other logical characterizations have been obtained apart from HT. In (de Jongh & Hendriks 2001), for instance, they use instead another logic, KC, and show that this is, in fact, the weakest intermediate logic (between intuitionistic and classical) that allows capturing strong equivalence of logic programs with nested expressions. An interesting open question is how logic KC deals with nested conditionals since, as we have shown, this is the case where HT and L_3 diverge.

Acknowledgements I want to thank Vladimir Lifschitz for his discussions and comments about a preliminary draft of the L_3 encoding, and to the anonymous referees for drawing my attention to part of the related work cited in this paper. This research is partially supported by the Government of Spain, grant TIC2001-0393.

References

Bull, R., and Segerberg, K. 1984. Basic modal logic. In Gabbay, D., and Guenthner, F., eds., *Handbook of Philosophical Logic*, volume 2. D. Reidel Publishing Company. 1–88.

Cabalar, P. 2002. Alternative characterizations for strong equivalence of logic programs. In *In Proc. of the Ninth Int'l Workshop on Non-monotonic Reasoning (NMR'2002)*.

D.H.J. de Jongh, A. Hendriks. 2001. Characterization of strongly equivalent logic programs in intermediate logics. Unpublished draft. `http://turing.wins.uva.nl/~lhendrik/` .

Fitting, M. 1985. A kripke-kleene semantics for logic programs. *Journal of Logic Programming* 2(4):295–312.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R. A., and Bowen, K. A., eds., *Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 2)*. Cambridge, MA: MIT Press. 1070–1080.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 365–385.

Inoue, K., and Sakama, C. 1998. Negation as failure in the head. *Journal of Logic Programming* 35(1):39–78.

Kunen, K. 1987. Negation in logic programming. *Journal of Logic Programming* 4(4):289–308.

Lifschitz, V.; Pearce, D.; and Valverde, A. 2000. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*. (to appear).

Lifschitz, V.; Tang, L. R.; and Turner, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence* 25:369–389.

Lifschitz, V. 1993. Circumscription. In D.M. Gabbay, C. H., and Robinson, J., eds., *Handbook of Logic in AI and Logic Programming*, volume 3. Oxford University Press. 298–352.

Lifschitz, V. 1994. Minimal belief and negation as failure. *Artificial Intelligence* 70:53–72.

Lin, F., and Shoham, Y. 1992. A logic of knowledge and justified assumptions. *Artificial Intelligence* 57:271–289.

Lin, F. 1991. *A Study of Nonmonotonic Reasoning*. Ph.D. Dissertation, Stanford.

Pearce, D.; Tompits, H.; and Woltran, S. 2001. Encodings of equilibrium logic and logic programs with nested expressions. In Bradzil, P., and Jorge, A., eds., *Lecture Notes in Artificial Intelligence*, volume 2258. Springer Verlag. 306–320.

Pearce, D. 1997. A new logical characterisation of stable models and answer sets. In *Non monotonic extensions of logic programming. Proc. NMELP'96. (LNAI 1216)*. Springer-Verlag.

Turner, H. 2001. Strong equivalence for logic programs and default theories (made easy). In *In Proc. of the Sixth Int'l Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, 81–92.

[10] As described for instance in (Lifschitz 1993), propositional circumscription is nothing else but a quantified boolean formula.