Loose notes and conclusions on a fixed point operator for rSM and rWFS

– DRAFT VERSION –

Luís Rodrigues Soares
March 27th, 2006

Introductory notes

This document includes a set of conclusions and steps taken in the direction of defining a fixed point operator for rSMs and rWFS. It is an account of the past 4 months of work and includes all ideas and intuitions tested so far. Its tone is rather informal and I'll be introducing all necessary concepts as they are required.

Part I

Four Months of Counter-Factual Programs

I start by describing the last four months of definitions based on generating counter-factual programs and calculating the stable models of the CFPs along with the original program.

1 The first definition[1]

We were somewhere in December when the first stable idea for a fixed point operator came. Professor Moniz Pereira talked to me and Alexandre of the idea of Counter-Factual Programs (CFPs) and how they were used to help prove a certain literal in a RAA chain. This version of the definition of CFPs was unindexed in the default literals and throughout the rest of the document I'll refer to it as the unindexed version:

Definition 1 (Unindexed Counter-Factual Program P_{cf}) *Let P be a normal logic program with l rules in the form $r_k = H \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$ ($m, n \geq 0$ and $1 \leq k \leq l$) and X a literal such that*

$$\exists_{r_k \in P} : X \in \{B_1, \ldots, B_n\}$$

The counter-factual program $P_{cf}(X)$ regarding X and rule r_k in program P is the following transformation of P:

1. *Replace all non-negated literals A in all rules with the auxiliary literal A_ι;*
2. *Remove from r_k all default literals $\sim X$;*
3. *Add to $P_{cf}(X_\iota)$ the rule $X \leftarrow X_\iota$*

[1] Cova do Vapor version

The index used (ι, iota) is unique for this counter-factual program and literal X and is called the counter-factual index of $P_{cf}(X)$ and it identifies the literal we are trying to conclude.

The idea was that in order to prove a certain literal X by *reductio ad absurdum* (RAA) one would have to use a couterfactual program where we would assume $\sim X$. For example, consider a direct odd loop:

$$\begin{aligned} P = \{ & a \leftarrow x \\ & x \leftarrow y \\ & y \leftarrow \sim a \} \end{aligned}$$

The CFP for a would be:

$$\begin{aligned} P_{cf}(a) = \{ & a \leftarrow a_1 \\ & a_1 \leftarrow x_1 \\ & x_1 \leftarrow y_1 \\ & y_1 \leftarrow \} \end{aligned}$$

In this program, we assume $\sim a$ as being true and try to see if a is a model of the program. We changed all literals to their indexed versions so we can later remove them from the program, i.e., the indexed literals are only necessary to create the RAA chain between $\sim a$ and a. The only thing provable from the CFP is actually a, because all indexed literals are not to be considered.

So, now we have:

$$\begin{aligned} & \Gamma_P^r(I) = \\ = & \Gamma_{P_{cf}(a) \cup P}(I) = \\ = & \{a_1, x_1, y_1, a\} = \\ = & \{a\} \end{aligned}$$

So the idea is that we should be able to use thew CFPs of the atoms in an interpretation I, along with the original one, and if I was a stable model

(SM) of $P \cup P_{cf}s$, then I would be a rSM. We could then remove the indexed literals and only the unindexed ones were used in the result. Problems started precisely because when we are considering several literals in an interpretation they may interfere with each other's CFPs. For the odd loop of 3, $I = \{a, b\}$ is not a SM of $P \cup P_{cf}(a) \cup P_{cf}(b)$:

$$\begin{aligned} P = \{ & a \leftarrow\sim b \\ & b \leftarrow\sim c \\ & c \leftarrow\sim a\} \end{aligned}$$

the CFPs for a and b would be:

$$\begin{aligned} P_{cf}(a) = \{ & a \leftarrow a_1 \\ & a_1 \leftarrow\sim b \\ & b_1 \leftarrow\sim c \\ & c_1 \leftarrow\} \end{aligned} \qquad \begin{aligned} P_{cf}(b) = \{ & b \leftarrow b_2 \\ & a_2 \leftarrow \\ & b_2 \leftarrow\sim c \\ & c_2 \leftarrow\sim a\} \end{aligned}$$

However,

$$\begin{aligned} & \Gamma_{P_{cf}(a) \cup P_{cf}(b) \cup P}(I) = \\ = & \{b, b_1, c_1, a_2, b_2\} = \\ = & \{b\} \end{aligned}$$

From this observation, we came up with the idea that a_1 should be able to derive its truth value from a_2, sort of like an auxiliary assumption, in order to give context to the fact that we were using several CFPs together. Simmetrically, b_2 could also depend on b_1. Auxiliary assumptions undergone several versions, as they were more or less permissive and more or less *hacked*.

Definition 2 (Permissive Counter-Factual Auxiliary Assumptions operator α_{cf}) *Let P be a normal logic program, $I = \{I_1, \ldots, I_j\}$ a set of literals such that $1 \leq i \leq j$ and B a set of counter-factual programs obtained from the generation of a counter-factual program for each literal in I.*

For each counter-factual program $P_{cf}(X)$ regarding a literal X that contains a rule in the form:

$$H_\iota \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$$

such that

$$\exists_{P_{cf}(H)} : the\ counter\ factual\ index\ for\ P_{cf}(H)\ is\ \kappa$$

introduce the rule $H_\iota \leftarrow H_\kappa$

Roughly speaking, this would give us the new result for $P_{cf}(a)$ and $P_{cf}(b)$:

$$\begin{aligned} P_cf(a) = \{ & a \leftarrow a_1 \\ & a_1 \leftarrow\sim b \\ & b_1 \leftarrow\sim c \\ & c_1 \leftarrow \\ & a_1 \leftarrow a_2\} \end{aligned} \qquad \begin{aligned} P_cf(b) = \{ & b \leftarrow b_2 \\ & a_2 \leftarrow \\ & b_2 \leftarrow\sim c \\ & c_2 \leftarrow\sim a \\ & b_2 \leftarrow b_1\} \end{aligned}$$

now giving us the intended result

$$\begin{aligned} & \Gamma_{P_{cf}(a)\cup P_{cf}(b)\cup P}(I) = \\ = & \{b, b_1, c_1, a_1, a_2, b_2\} = \\ = & \{a, b\} \end{aligned}$$

However, if $I = \{a, b, c\}$, which is not a revised stable model, our CFPs would be

$$
\begin{aligned}
P_cf\,(a) = \{a &\leftarrow a_1 \\
a_1 &\leftarrow\sim b \\
b_1 &\leftarrow\sim c \\
c_1 &\leftarrow \\
a_1 &\leftarrow a_3 \\
a_1 &\leftarrow a_2\}
\end{aligned}
\qquad
\begin{aligned}
P_cf\,(b) = \{b &\leftarrow b_2 \\
a_2 &\leftarrow \\
b_2 &\leftarrow\sim c \\
c_2 &\leftarrow\sim a \\
b_2 &\leftarrow b_1 \\
b_2 &\leftarrow b_3\}
\end{aligned}
\qquad
\begin{aligned}
P_cf\,(c) = \{c &\leftarrow c_3 \\
a_3 &\leftarrow\sim b \\
b_3 &\leftarrow \\
c_3 &\leftarrow\sim a \\
c_3 &\leftarrow c_1 \\
c_3 &\leftarrow c_2\}
\end{aligned}
$$

thus allowing $\Gamma_{P_{cf}(a)\cup P_{cf}(b)\cup P_{cf}(c)\cup P}\,(I) = \{a, b, c\}$.

So apparently, our auxiliary assumptions have to be more restrictive. Our next definition of the auxiliary assumptions added some defaults to each added rule; more specifically we would add to each auxiliary rule the defaults present in the original rule in program P, regarding the literal we were considering. For $I = \{a, b\}$, $P_{cf}\,(a)$ would become:

$$
\begin{aligned}
P_cf\,(a) = \{a &\leftarrow a_1 \\
a_1 &\leftarrow\sim b \\
b_1 &\leftarrow\sim c \\
c_1 &\leftarrow \\
a_1 &\leftarrow a_2, \sim c\}
\end{aligned}
$$

because as a_2 was present in $P_{cf}\,(b)$, it would have to respect the original defaults of the rule for head b in the original program[2].

This is the definition I brought with me to the first exam of KRR, before professor Moniz Pereira went to Indonesia. It became clear, during the exam, that the definition was very dificult to explain, and the intuitions were not very clear. Afterwards, I noticed that all the ideas I had presented still didn't work in some programs, which eventually took me to the next definition.

[2]The definition is very unintuitive and there is no point in presenting it here because it added no interesting notion to our work.

2 Afternoons at Praia da Torre

It was with the following program that I eventually came up with the most promising definition so far. The program that originated this definition was:

$$\begin{aligned} P = \{ & a \leftarrow\sim b \\ & b \leftarrow\sim c, x \\ & c \leftarrow\sim a \\ & x \leftarrow\sim x, a\} \end{aligned}$$

I'll skip all the steps I eventually went through, just to enumerate all the tricks I added in this definition:

Direct OLONs All direct single OLONs ($a \leftarrow\sim a$) were solved before anuthing else was done. This was because we already knew how these loops were handled, what their result was. Basically, if all our problems were like this, we would simply add a rule to the transformation which stated somthing like "remove $\sim X$ from the body of all rules with head X"

Several CF programs for the same CF assumption If several $\sim X$ were present in the program, we would be creating a CFP for each on of them. This was because we viewed each $\sim X$ as one possible counterfactual path to X, so we would treat each path differently (we'll see later that this idea was indeed wrong).

Facts Additionally, we would be not create CFPs for literal which were facts in the program. Although it was a trick, which if it wasn't present some programs would be giving more results than they should give, I still maintain that CFPs should not be created in this condition.

Auxiliary assumptions Finally, and probably most importantly, the auxiliary assumptions were now a lot simpler: a_1 could simply depend on b_2 if the rule for a_1 originally depended on $\sim b$, an idea of professor Moniz Pereira.

This is where the most promissing definition of Γ^r was born. After some comments on the second exam of KRR I divided it in the following definitions :

Definition 3 (Counter-Factual Program P_{cf}) *Let P be a normal logic program with l rules in the form $r_k = H \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$ $(m, n \geq 0$ and $1 \leq k \leq l)$ and X a literal such that*

$$\begin{aligned} \nexists_{r_k \in P} : & r_k = X \leftarrow \\ & \wedge \\ \exists_{r_k \in P} : & X \in \{B_1, \ldots, B_n\} \end{aligned}$$

The counter-factual program $P_{cf}(X_\iota)$ regarding X and rule r_k in program P is the following transformation of P:

1. *Replace all non-negated literals A in all rules with the auxiliary literal A_ι;*
2. *Remove from r_k the default literal $\sim X$;*
3. *Add to $P_{cf}(X_\iota)$ the rule $X \leftarrow X_\iota$*

The index used (ι, iota) is unique for this counter-factual program and this particular rule and is called the counter-factual index of $P_{cf}(X_\iota)$. It identifies both the rule used to try to conclude X and the literal we are trying to conclude. Hence the notation $P_{cf}(X_\iota)$.

Definition 4 (Counter-Factual Program Set β_{cf}) *Let P be a normal logic program with rules in the form $r_k = H \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$ $(1 \leq k \leq l)$ and $I = \{I_1, \ldots, I_j\}$ a set of literals such that $1 \leq i \leq j$. The set of all counter-factual programs of P regarding all literals in I and all rules in P is the set*

$$\beta_{cf}^{I}(P) = \bigcup_{\iota} P_{cf}(X_\iota)$$

with ι being the program's counter-factual index, incremented as a new rule or literal is considered.

Definition 5 (Elegant Counter-Factual Auxiliary Assumptions operator α_{cf}) *Let P be a normal logic program, $I = \{I_1, \ldots, I_j\}$ a set of literals such that*

$1 \leq i \leq j$ *and* B *the counter-factual program set obtained from the application of* $\beta_{cf}^{I}(P)$.

For each counter-factual program $P_{cf}(X_\iota)$ *regarding a literal* X_ι *that contains a rule in the form:*

$$H_\iota \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$$

such that

$$\exists_{Y \in \{B_1, \ldots, B_n\}} : P_{cf}(Y) \in B \wedge Y \neq X$$

introduce a copy of the rule regarding H_ι *such that* $\sim Y$ *is replaced with* H_κ, κ *being the counter-factual index identifying* $P_{cf}(Y)$. *Add this rule to* $P_{cf}(X_\iota)$.

The application of $\alpha_{cf}^{I}(B)$ *results in a program set with auxiliary assumptions added to each rule in* B.

Definition 6 (Γ^r operator) *Let* P *be a normal logic program and* I *an interpretation. The revised program* P^r *is the following sequence of operations:*

1. *Remove from all rules in* P *the default literal* $\sim X$ *such that* X *belongs to the head of the rule;*
2. *Create the counter-factual program set* $B^r = \beta_{cf}^{I}(P_r)$ *from interpretation* I *and the revised program* P^r*;*
3. *Apply the counter-factual auxiliary assumptions operator* $\alpha_{cf}^{I}(B^r)$ *on* B^r*;*
4. *Let* $P^r = P^r \cup \alpha_{cf}^{B^r}(I)$.

On P^r *apply the* Γ *operator and remove all counter-factual auxiliary literals* X_ι *from the resulting set.*

I eventually wrote several pages based on this definition and tried all the examples we had imagined so far and all was working well. Even though there were some tricks in between the results were promissing. I hand a hunch, however, that our auxiliary assumptions could be too permissive, and I was right. Consider program

$$\begin{aligned} P = \{ & y \leftarrow\sim b \\ & b \leftarrow\sim c \\ & c \leftarrow\sim a\} \end{aligned}$$

whose only $rSM = \{y, c\}$. Let's create the CFPs for b and c, the only literals which appear in the head of a rule and negated in the body of other rules:

$$\begin{aligned} P_c f\,(b) = \{ & b \leftarrow b_1 \\ & y_1 \leftarrow \\ & b_1 \leftarrow\sim c \\ & c_1 \leftarrow\sim a \\ & b_1 \leftarrow c_2\} \end{aligned} \qquad \begin{aligned} P_c f\,(c) = \{ & c \leftarrow c_2 \\ & y_2 \leftarrow b_2 \\ & b_2 \leftarrow \\ & c_2 \leftarrow\sim a \\ & y_2 \leftarrow b_1\} \end{aligned}$$

The problem is that our CFPs allow us to prove that $\Gamma^r_P(\{b, c\}) = \{b, c\}$, even though not only c wasn't in an OLON, as b wasn't part of any kind of model (SM or rSM). c is provable from he main program, as a is not part of the model. And b comes out from the auxiliary assumption $b_1 \leftarrow c_2$. So the problem was, again, with the auxiliary assumptions...

3 Houston, we have a problem...

...was the subject of an email I sent to professor Moniz Pereira regarding this subject. The result, along with some discussion with Alexandre Pinto was a somewhat different approach to the definition which had the following main differences:

- Instead of generating only some CFPs, we generated all possible CFPs;
- All literals in our CFPs' rules, including the default ones, were indexed;

- The stable models of $P \cup P_{cf}s$ were the rSM of P. This involved trying to find out the right auxiliary literals of our transformation, in order to give the intended SM;
- The auxiliary assumptions were now defined differently: if a_1 depended on $\sim b_1$, we would add the dependency for b's counterfactual program, $a_1 \leftarrow\sim b_1, \sim b_2$;
- Additionally, the counterfactual assumption of the CFP would be replaced by an odd loop, in order to switch on or off the odd loop.

Only to illustrate the definition, in the odd loop of 3 the CFPs would become:

$$
\begin{array}{lll}
P_cf\,(a) = \{a \leftarrow a_1 & P_cf\,(b) = \{b \leftarrow b_2 & P_cf\,(c) = \{c \leftarrow c_3 \\
\quad a_1 \leftarrow\sim b_1, \sim b_2 & \quad a_2 \leftarrow b^- & \quad a_3 \leftarrow\sim b_3 \\
\quad b_1 \leftarrow\sim c_1 & \quad b_2 \leftarrow\sim c_2, \sim c_3 & \quad b_3 \leftarrow c^- \\
\quad c_1 \leftarrow a^- & \quad c_2 \leftarrow\sim a_2 & \quad c_3 \leftarrow\sim a_3, \sim a_1 \\
\quad a^- \leftarrow\sim a^+ & \quad b^- \leftarrow\sim b^+ & \quad c^- \leftarrow\sim c^+ \\
\quad a^+ \leftarrow\sim a^-\} & \quad b^+ \leftarrow\sim b^-\} & \quad c^+ \leftarrow\sim c^-\}
\end{array}
$$

We can see the main differences: the auxiliary assumptions are now embebbed in the main rules of the program, all literals have indexes and the odd loop can be switched on or off by considering wither b^- or b^+ as being part of the model. All examples appeared to be working well until the following program:

$$
\begin{array}{l}
P = \{b \leftarrow\sim a, c \\
\quad c \leftarrow a \\
\quad a \leftarrow\sim b\}
\end{array}
$$

which has $rSM_1 = \{a, c\}$ and $rSM_2 = \{b\}$.

$$
\begin{aligned}
P_{cf}(a) = \{ & a \leftarrow a_1 \\
& b_1 \leftarrow a^-, c_1 \\
& c_1 \leftarrow a_1 \\
& a_1 \leftarrow\sim b_1 \\
& a^- \leftarrow a^+ \\
& a^+ \leftarrow a^- \}
\end{aligned}
$$

$$
\begin{aligned}
P_{cf}(b) = \{ & b \leftarrow b_2 \\
& b_2 \leftarrow\sim a_2, c_2, \sim a_1 \\
& c_2 \leftarrow a_2 \\
& a_2 \leftarrow b^- \\
& b^- \leftarrow b^+ \\
& b^+ \leftarrow b^- \}
\end{aligned}
$$

$$
\begin{aligned}
P_{cf}(c) = \{ & c \leftarrow c_3 \\
& b_3 \leftarrow\sim a_3, c_3 \\
& c_3 \leftarrow a_3 \\
& a_3 \leftarrow\sim b_3 \\
& c^- \leftarrow c^+ \\
& c^+ \leftarrow c^- \}
\end{aligned}
$$

With our transformation, b would never make it to the model because it would end up depending on both a_2 and $\sim a_2$. The problem was that, even though the chain of support for b demands a to be true, in order to get to $\sim b$, the chain also demanded $\sim a$ to be true. However, $\sim a$ would eventually become true as soon as we proved that there was in fact a chain of support between b and $\sim b$ and hence all the literals in the chain would have to be false.

The solution we came up with was to replace all default literals $\sim X_1$ in the body of all rules by an extended literal X_1^*, being that $X_1^* \leftarrow\sim X_1$ and $X_1^* \leftarrow\sim X$. This allowed us to, while having a_2 in the stable model,

therefore cutting one of the rules for a_2^*, keep the other one, because a was not part of the model.

It was an interesting idea, and a necessary one, but it was incomplete. With our well known odd loop of 3, we couldn't get the models we wanted. A solution for this was discussed, which involved counting the number of negations we went through as we were descending through the dependency graph for each literal (the problem was here...), but these ideas still weren't enough[3].

Here's why: not only did the auxiliary assumptions kept some odd loops active in the transformed program, as they prevented some things from being proved. Let's look at the odd loop of 3. Our CFPs are[4]:

$$
\begin{array}{lll}
P_cf\,(a) = \{a \leftarrow a_1 & P_cf\,(b) = \{b \leftarrow b_2 & P_cf\,(c) = \{c \leftarrow c_3 \\
\quad a_1 \leftarrow b_1^*, \sim b_2 & \quad a_2 \leftarrow b^- & \quad a_3 \leftarrow\sim b_3 \\
\quad b_1 \leftarrow\sim c_1 & \quad b_2 \leftarrow\sim c_2^*, \sim c_3 & \quad b_3 \leftarrow c^- \\
\quad c_1 \leftarrow a^- & \quad c_2 \leftarrow\sim a_2 & \quad c_3 \leftarrow\sim a_3^*, \sim a_1 \\
\quad b_1^* \leftarrow\sim b_1 & \quad c_2^* \leftarrow\sim c_2 & \quad a_3^* \leftarrow\sim a_3 \\
\quad b_1^* \leftarrow\sim b & \quad c_2^* \leftarrow\sim c & \quad a_3^* \leftarrow\sim a \\
\quad a^- \leftarrow\sim a^+ & \quad b^- \leftarrow\sim b^+ & \quad c^- \leftarrow\sim c^+ \\
\quad a^+ \leftarrow\sim a^-\} & \quad b^+ \leftarrow\sim b^-\} & \quad c^+ \leftarrow\sim c^-\}
\end{array}
$$

Now, if $I = \{a, b\}$, a_1 must be part of the interpretation, because the only way to prove a is through its CFP. However, if a_1 is part of the interpretation, c_3 will be false, thus allowing b_2 to become true. But b_2 can't be, or else the rule for a_1 will be cut. Even if we only generated the CFPs for the literals present in the model, the problem still maintained: b_2 would depend solely on c_2^* which, in order to become false, would demand both c and c_2 to be true. But c can't be true, particularly if we are testing $I = \{a, b\}$...

The Houston definition was not ready yet...

[3] I never wrote any formal definition of these ideas, since the Houston problem, because all the problems appeared during testing of programs, no definition ever succeded in all programs

[4] We are now only generating the star rules for the even transitions over negation, starting from the literal we are trying to prove. this was the very last changing I discussed with Professor Moniz Pereira.

4 Some initial conclusions

More or less at the same time I was realizing all the previous problems, I started writing the several things I thought that contributed to the failure of all the previous definitions, as well as all the things I liked/disliked. Here's a brief account on them:

How to generate CFPs Two methods were tested to geenrate CFPs. The first one, used until the Houston problems, used no indexes on the default literals. From the Houston definition onward the definition started using indexes on the default negated literals. From these two methods, I prefer the approach of the Houston definition, i.e., generate indexes on all literals. This is because it gives a better image of the reasoning we are trying to apply: our CFP is a test program, to see if, by assuming a certain $\sim X$ we eventually get to X. "get to" is a way of saying that there has to be a chain of support from the CF assumption to the literal we are trying to prove. It's easier to see this chain if all literals are numbered. The rest of the method to generate CFPs is equal on all approaches. The CF assumption is removed from the CFP and a rule is added to the CFP which connects it to the original program. This is a garantee that this CFP only allows iits CF assumption to be concluded.

When to generate CFPs The idea of generating CFPs is uniquelly to allow some literal to be proven by RAA reasoning. On some instances of the definition we generated alll possible CFPs possible, i.e., for all literals present in the program. On other instances, we only generated CFPs for the literals present in the interpretation. This could be constrained even further by demanding the literals for which we generated CFPs had to appear at least once in the head of some rule, and at least once negated in the body of some rule. These two conditions are essential conditions for any literal to be provable by RAA. At least these two conditions have to be satisfied.

Auxiliary assumptions How should we connect the several CFPs? Well, I really have no conclusion here. I can say that the last approach was the one more intuitive for me, and the one that appeared to work the best. I believe that, ideally, the connection between the CFPs should be implied from the original program, i.e., we shouldn't have to add any rules to the CFPs. However, so far, no perfect way for doing this was defined...

Tricks On the several tricks we've tested, I believe on the idea of not generating CFPs for facts. The reason is simple: we want to generate as

few CFPs as possible. So if something is a fact it is already proved. Additionally, I also believe that all programs can be simplified by solving all the direct OLONs, although, as professor Moniz Pereira said, this is only an optimization, it should not be mandatory in the definition. The trick of generating several CFPs, for each CF assumption was obviously wrong. When we assume some $\sim A$, all $\sim A$s of the program are assumed, we can't assume some and don't assume others.

Using Stable Models on the transformed program The use of stable models to see if a certain literal is provable by RAA is an idea I don't find very intuitive, precisely because it doesn't reflect the original idea that an atom is provable by RAA if there is a chain of support between the atom and its default negation. And this was probably the thing that was annoying me the most in all our previous attempts.

This concludes the first part of this document. All the definitions we have tested so far, why they have failed and where they have failed.

Part II

An alternative definition

Here I describe the last idea I had for a fixed point operator for revised stable models, not based on CFPs.

5 Origin of the idea

About the same time I wrote the previous text, Alexandre talked with me about an idea he once had for an operator, but which he never explored. It basically consisted in inverting some rules of the program that reflected the conditions the program was stating. For example in our odd loop of 3, we can see that the main condition for having a is not to have b and not to have c. One of the rules that states this is already present in the program ($a \leftarrow\sim b$), so apparently, we could add $a \leftarrow\sim c$ to the program. Now, if we proceed simmetrically to the other literals, our odd loop of 3 would become:

$$\begin{aligned} P = \{a &\leftarrow\sim b & b &\leftarrow\sim a \\ b &\leftarrow\sim c & c &\leftarrow\sim b \\ c &\leftarrow\sim a & a &\leftarrow\sim c\} \end{aligned}$$

With this program, operator Γ is able to return all three revised stable models– $\{a, b\}, \{b, c\}, \{c, a\}$. The idea was interesting but the seemed to be no relevant principle behind it so I didn't thought of it for much longer. The principle I was missing came to me a couple of days later, when it was starting to become obvious to me that the ideas we had previously for our definition would not work. I then then considered the following: if we could get rid of the OLONs and ICONs, the Γ operator would have no problem in calculating the models (revised or not). In other words, it would be handy if we could transform OLONs and ICONs into ELONs. How could we do that? For example, for the following program:

$$P = \{a \leftarrow b$$
$$b \leftarrow c$$
$$c \leftarrow\sim a\}$$

the only thing we needed was to invert the last rule, and this would generate an ELON between a and c:

$$P = \{a \leftarrow b$$
$$b \leftarrow c$$
$$c \leftarrow\sim a$$
$$a \leftarrow\sim c\}$$

Now, the only stable model of P is a, which happens to be a revised stable model. Curiosly, this was very close to the idea Alexandre discussed with me, although it was seen from a different pont of view. I showed him some preliminary results, looking promissing and we went on trying the idea on several other programs. Initially it was a rather *naïve* approach, we would invert every rule in the program, which had no justification possible. Obviously after a few programs the idea stopped working, and I put it to rest for a few days. I would return to it once more, only to try to understand why it didn't worked.

6 Häagen-Dazs at Belém

The Häagen-Dazs shop near Torre de Belém, in Lisbon, is a great place to have breakfast and ideas. It was there, on a Saturday morning, that I returned to my last suggestion of an operator for rSMs and rWFS, using the previous intuitions and some new ideas I explore next.

I eventually returned to approach I had discussed with Alexandre because I thought that there had to be some reason for it to work in some programs. I then did the oposite exercise (using the previous program): if I wanted to prove[5] a, what should be the body? Well, if we look at the loop, we can see that the only safe condition for a to be true is for c to be false. Why

[5] I'm not sure if the notion of proof is the correct one, formally speaking...

is the the only safe condition? Because, being the head of the last rule in the OLON's chain, if a is to be true (a RSM), then the the head of the rule that depends on $\sim a$ must be false[6]. Now if c if false, so will b, and so will a, by the first rule of the program. However, the last rule garantees, that a is part of the revised stable model.

So what intuition have I followed: if an atom is in an OLON, then we can add the last rule of the OLON inverted. This rule solves the OLON, by transforming it into an ELON. There was, however, a catch: how do we know an atom is in an OLON, i.e., how do we know that we are able to use RAA reasoning in order to prove a certain literal? According to the definition of RSM[1]:

> (...) RAA reasoning takes place when $\sim a$ implies, directly or indirectly, a; and this happens only when tehre are OLONs in the program. In fact, that is the essential nature of an OLON: an atom depending on its own negation.

This notion of *dependency* can be grasped in many ways. Let's use here a sort of dependency graph, generated from the RAA assumption, for the previous program:

We can see that, just it was stated in the original definition, $\sim a$ implies, directly a. What about other programs? For example:

[6] In the case of this program, b and c will also be false but the safest condition is to use only the last rule in the chain. We'll see later why this is

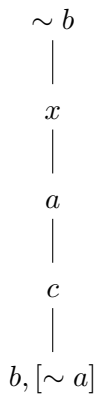
$$P = \{b \leftarrow\sim a, c$$
$$c \leftarrow a$$
$$a \leftarrow x$$
$$x \leftarrow\sim b\}$$

which has $rSM_1 = \{a, c\}$ and $rSM_2 = \{b\}$. To begin with, we can see that a is not in an OLON. It's easy to see this if we draw the chain of support, starting with $\sim a$.

$$\sim a$$
$$|$$
$$b, [c]$$
$$|$$
$$\sim x$$
$$|$$
$$\sim a$$

I put c in brackets to say that c is a precondition to move on to the next branch of the tree. As we have reached an atom which is already present in the dependency graph, we stop the generation of branches. Additionally, we can see that by assuming $\sim a$ we won't reach a, so a is not in an OLON. Well, if a is not in an OLON, it must be proven by the classical notion of support, or it doesn't belong to any model (revised or otherwise). Either way, it is not important to our generation of rules.

Moving to the other literals present in the program, c never appears negated in the body of any rule, so it can never be proven by RAA reasoning. Let's turn our attention now to b.



$\sim b$ leads to b, meaning it is in an odd loop. It has a precondition that is $\sim a$ must be true. Aditionally, the head of the last rule in the chain must be false ($\sim x$). So if we test any interpretation which includes b, we must add the rule $b \leftarrow\sim x, \sim a$, where $\sim a$ is the precondition in brackets and $\sim x$ is the tail of the chain of support.

So, our original program remains the same for all interpretations we test except for $I = \{b\}$. In this case we add the rule $b \leftarrow\sim x, \sim a$ to our program, which allows us to conclude that I is a revised stable model of P.

However, this idea of drawing the chain of support for each literal X, starting with $\sim X$ to see if we reach X is not very elegant, from the mathematical point of view, so I put the intuition behind this in an operator. I called it Δ operator (*delta*, which starts with "d", as the word "dependency"). Intuitivelly it works somewhat like this:

Definition 7 (Δ operator – Intuitive definition) *Let P be a NLP with rules r in the form $r = H \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$, and I and interpretation such that $I = \{L_1, \ldots, L_j\}$.*

1. *Δ operator is to be applied to every literal L_j in I such that L_j appears negated at least once in the tail of at least one rule of P, and L_j appears at least once in the head of at least one rule of P.*
2. *This being the case, which is a necessary condition for the existence of an OLON, Δ operator will generate an ordered set for each literal L_j, starting preciselly with each literal's default negation $\sim L_j$.*
3. *For each of the ordered sets generated, if the ending element of the set is the negation of the starting element of the set, we generate a*

rule with the first two elements of the set, which will be something like $first\_element \leftarrow second\_element$.

4. *We then add the generated rules to the original program and can go on with the calculation of* Γ *operator.*

The main issues in this definition are:

- The actual working of the Δ operator, i.e., specifically how do we go on generating each of the ordered sets;
- After generating the ordered sets, how do we *transform* the ordered sets into rules.

So let's start with the the first point. The idea is that Δ operator starts with a certain $\sim X$, something that can only exist in the tail of a rule, and will eventually reach an X, something in the head of a rule. So, Δ operator will search for literals in the tails of rules and will return their heads, which is something like

$$\Delta\left(\{X\}\right) = \{H\}, \exists_{r \in P} : r = H \leftarrow X.$$

Now several things can happen when we search for literals in the tails of rules. One of them is that we don't have X in the tail of a rule, but we do have $\sim X$. How should Δ operator work in this case? Well, it's easy to see that if we have a certain literal X we are trying to prove, all rules that have $\sim X$ in the tail will have their heads false:

$$\Delta\left(\{X\}\right) = \{\sim H\}, \exists_{r \in P} : r = H \leftarrow\sim X.$$

Something else that can happen is that we are actually looking the a negated literal. In this case, our rules would be the simetrical to the ones above:

$$\Delta\left(\{\sim X\}\right) = \{H\}, \exists_{r \in P} : r = H \leftarrow\sim X;$$
$$\Delta\left(\{\sim X\}\right) = \{\sim H\}, \exists_{r \in P} : r = H \leftarrow X.$$

These four rules pretty much define the simplest case which is the existence of one (only one...) rule in the program with precisely one literal in the body of the rule. But this case is rather unreal as most of the time we have rules with several literals in the body (conjunctions), and in some cases we also have several rules for the same literal (disjunctions). Now for the case of conjunctions, the idea is pretty simple: we just state what are the conditions we must obey in order to have the head of the rule. In other words, if $r = H \leftarrow W, X, Y, Z$ and we are calculating $\Delta(\{X\})$, the result is H with the condition that $\{W, Y, Z\}$ must hold. We are not interested in going down the trees of W,Y and Z because the operator is only concerned with showing if X depends on some $\sim X$. Therefore, $\Delta(\{X, [W, Y, Z]\}) = \Delta(\{X\})$. If it does depend, then we are also concerned in knowing what other conditions must hold. So the conjunction rule would be something like:

$$\Delta(\{X\}) = \{H, [W_1, \ldots, W_n]\}, \exists_{r \in P} : r = H \leftarrow X, W_1, \ldots, W_n.$$

Now this is the real general case, which we can extend to include the previous four rules:

$$\begin{aligned}
\Delta(\{X\}) &= \{H, [W_1, \ldots, W_n]\}, \exists_{r \in P} : r = H \leftarrow X, W_1, \ldots, W_n. \\
\Delta(\{X\}) &= \{\sim H, [W_1, \ldots, W_n]\}, \exists_{r \in P} : r = H \leftarrow\sim X, W_1, \ldots, W_n. \\
\Delta(\{\sim X\}) &= \{H, [W_1, \ldots, W_n]\}, \exists_{r \in P} : r = H \leftarrow\sim X, W_1, \ldots, W_n. \\
\Delta(\{\sim X\}) &= \{\sim H, [W_1, \ldots, W_n]\}, \exists_{r \in P} : r = H \leftarrow X, W_1, \ldots, W_n.
\end{aligned}$$

The only case missing to explore is when we have several rules sharing one same head tail X. What this case means is that by assuming X we can prove several (possibly) different heads. So the idea is that we generate a copy of the ordered set as it is so far, and continue with both sets from that point:

$$\begin{aligned}
\Delta(\{X\}) = \{H_1, \ldots, H_n\}, \exists_{r_1, \ldots, r_n \in P} : \\
r_1 = H_1 \leftarrow Body_1, \ldots, r_n = H_n \leftarrow Body_n; \\
\Delta(\{X_1, \ldots, X_n\}) = \text{ create a copy of the ordered set} \\
\text{as it is so far, and continue with the two instances separated.}
\end{aligned}$$

We now have an idea of all the rules we can use when generating our dependency ordered sets. Now what rules dictate the generation of the rule

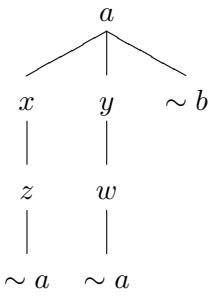
that transforms the OLON into an ELON? Basically, we have to look at the generated ordered set to see the conditions that have to be respected. For example, $\Delta(\{\sim b\})$ in the previous program would generate the following ordered set:

$$\{\{\sim b\}, \{x\}, \{a\}, \{c\}, \{b, [\sim a]\}\}$$

From this set the rule we should generate the rule $b \leftarrow\sim x, \sim a$ which correspond the the idea we defined above, i.e., $\sim a$ is the precondition in brackets and $\sim x$ is the tail of the chain of support. However, more spicificity has to be added to these explanation. Consider a program with two RAA chains of support for the same literal over a conjunction:

$$\begin{aligned} P = \{&a \leftarrow x, y, \sim b \\ &x \leftarrow z \\ &z \leftarrow\sim a \\ &y \leftarrow w \\ &w \leftarrow\sim a\} \end{aligned}$$

This program would have the following dependency graph



and the application of Δ Operator would provide us with the following result:

$$\{\{\sim a\}, \{w\}, \{y\}, \{a, [x, \sim b]\}\}$$
$$\{\{\sim a\}, \{z\}, \{x\}, \{a, [y, \sim b]\}\}$$

suggesting the rules we should add to the program would be:

$$a \leftarrow\sim w, x, \sim b$$
$$a \leftarrow\sim z, y, \sim b$$

which is incorrect because both x and y will never be true in any rSM, as they are part of the RAA chain for each of the sides of the conjunction. The question here that the literals in the brackets are part of another RAA chain, and as such they will eventually be false. $\sim b$, however, is not part of any of the RAA chain so can be added as a constraint as usual. The other ones will be added negated:

$$a \leftarrow\sim w, \sim x, \sim b$$
$$a \leftarrow\sim z, \sim y, \sim b$$

This allows us to define how to generate a rule, given an ordered set:

- The head of the rule is the atom in the default literal in the first element of the ordered set;
- We add to the body of the rule, the negation of the second element of ordered set, if this atom is different from the atom in the first element[7];
- Furthermore, we add all literals in brackets to the body of the rule in their original form, if they are not part of an RAA chain of the same literal, or negated, if they are part of the RAA chain of a literal.

We can now provide with a primary formal definition of the Δ Operator:

Definition 8 (Δ operator) *Let P be a NLP with rules r in the form $r = H \leftarrow A_1, \ldots, A_m, \sim B_1, \ldots, \sim B_n$, and I an interpretation such that $I = \{L_1, \ldots, L_j\}$.*

[7]This allows it to work with the simplest OLON, $X \leftarrow\sim X$

For all $L_j \in I$ such that

$$\begin{aligned}\exists_{r_k,r_i} :& r_k = H_k \leftarrow A_{k_1},\ldots,A_{k_m},\sim B_{k_1},\ldots,\sim B_{k_n} \wedge \\ & r_i = H_i \leftarrow A_{i_1},\ldots,A_{i_m},\sim B_{i_1},\ldots,\sim B_{i_n} \wedge \\ & L_j \in B_i \wedge L_j \in H_k (\textit{with the possibility of } i = k).\end{aligned}$$

build the L_j's dependency ordered set using the following rules:

$$\begin{aligned}\Delta(\{X\}) = \{H,[W_1,\ldots,W_n]\}, \exists_{r\in P} : r = H \leftarrow X, W_1,\ldots,W_n;\\ \Delta(\{X\}) = \{\sim H,[W_1,\ldots,W_n]\}, \exists_{r\in P} : r = H \leftarrow\sim X, W_1,\ldots,W_n;\\ \Delta(\{\sim X\}) = \{H,[W_1,\ldots,W_n]\}, \exists_{r\in P} : r = H \leftarrow\sim X, W_1,\ldots,W_n;\\ \Delta(\{\sim X\}) = \{\sim H,[W_1,\ldots,W_n]\}, \exists_{r\in P} : r = H \leftarrow X, W_1,\ldots,W_n;\\ \Delta(\{X,[W,Y,Z]\}) = \Delta(\{X\});\\ \Delta(\{X\}) = \{H_1,\ldots,H_n\}, \exists_{r_1,\ldots,r_n\in P} :\\ r_1 = H_1 \leftarrow Body_1,\ldots,r_n = H_n \leftarrow Body_n;\\ \Delta(\{X_1,\ldots,X_n\}) = \textit{ create a copy of the ordered set}\\ \textit{as it is so far, and continue with the two instances separated.}\end{aligned}$$

Generate the rules regarding each dependency ordered set using the following rules:

- *The head of the rule is the atom in the default literal in the first element of the ordered set;*
- *We add to the body of the rule, the negation of the second element of ordered set, if this atom is different from the atom in the first element*[8]*;*
- *Furthermore, we add all literals in brackets to the body of the rule in their original form, if they are not part of an RAA chain of the same literal, or negated, if they are part of the RAA chain of a literal.*

Add the generated rules to P.

[8]This allows it to work with the simplest OLON, $X \leftarrow\sim X$

So, this operator generates ordered sets, which start with the default literal we are assuming to test our RAA chain. Everytime we reach a point where we have a conjunction, we simply put the head of that conjunction in brackets. We don't need to go down that conjunction because this operator's purpose is not to prove a certain literal – its objective is simply to show the literals we have to go through to reach a certain X starting with $\sim X$. If, at any point, we reach a disjunction, i.e., two rules, with the same literal in the tail, we generate two derivation trees, which are equal until the disjunction point, and continue from that point onward.

I'll now show how this behaves in some programs, showing the result of the Δ operator. These programs are generally enhanced versions of the programs available in the previous tests set Alexandre gave me.

$$\begin{aligned} P_1 = \{ & b \leftarrow y \\ & y \leftarrow\sim a, c \\ & c \leftarrow a \\ & a \leftarrow x \\ & x \leftarrow\sim b\} \end{aligned}$$

This is an enhanced version os a previous program. a and b are in an ELON. Furthermore, b is also in an OLON through c, a and x, although having a precondition of $\sim a$.

$$\Delta\left(\{\sim a\}\right) = \{\{\sim a\}, \{y, [c]\}, \{b\}, \{\sim x\}, \{\sim a\}\}$$

Δ Operator stops because it reaches a literal it already passed by. In this case, we started from $\sim a$ and ended up in $\sim a$, so $\sim a$ is not in an OLON. No rule is to be generated here.

$$\Delta\left(\{\sim b\}\right) = \{\{\sim b\}, \{x\}, \{a\}, \{c\}, \{y, [\sim a]\}, \{b\}\}$$

We reached b starting from $\sim b$, which means b is in an OLON. It also has a precondition of $\sim a$ and as $\sim a$ is not part of any other RAA chain, it will be added without modifications. The generated rule is $b \leftarrow\sim x, \sim a$ and is only added when b is aprt of the interpretation.

We can now see that the intended rSMs can be calculated as SMs with Γ Operator:

$$\Gamma(\{a, c, x\}) = \{a, c, x\}$$
$$\Gamma(\{b, y\}) = \{b, y\}$$

$$\begin{aligned} P_2 = \{ & a \leftarrow x \\ & x \leftarrow\sim a, y \\ & y \leftarrow\sim b \\ & b \leftarrow z \\ & z \leftarrow\sim b, w \\ & w \leftarrow\sim a\} \end{aligned}$$

Both a and b are in OLONs through x and z respectively. however, both of them have preconditions ($\sim b$ and $\sim a$ respectively).

$$\begin{aligned} \Delta(\{\sim b\}) = & \{\{\sim b\}, \{y\}, \{x, [\sim a]\}, \{a\}, \{\sim x\}\} \\ & \{\{\sim b\}, \{y\}, \{x, [\sim a]\}, \{a\}, \{\sim w\}, \{\sim z\}, \{\sim b\}\} \\ & \{\{\sim b\}, \{z, [w]\}, \{b\}\} \end{aligned}$$

$$\begin{aligned} \Delta(\{\sim a\}) = & \{\{\sim a\}, \{w\}, \{z, [\sim b]\}, \{b\}, \{\sim z\}\} \\ & \{\{\sim a\}, \{w\}, \{z, [\sim b]\}, \{b\}, \{\sim y\}, \{\sim x\}, \{\sim a\}\} \\ & \{\{\sim a\}, \{x, [y]\}, \{a\}\} \end{aligned}$$

The first two dependency ordered sets don't lead to b. The last one leads and the to add if $b \leftarrow\sim z, w$. Simmetrically for a, the rule to add is $a \leftarrow\sim x, y$.

The rSMs are:

$$\Gamma(\{a, y\}) = \{a, y\}$$
$$\Gamma(\{b, w\}) = \{b, w\}$$

$$P_4 = \{a \leftarrow x$$
$$x \leftarrow y$$
$$y \leftarrow\sim a$$
$$y \leftarrow\sim b\}$$

a is in an OLON but can be proven in two diferent ways.

$$\Delta\left(\{\sim a\}\right) = \{\{\sim a\}, \{y\}, \{x\}, \{a\}\}$$

The rule to add is $a \leftarrow\sim y$. Note that the added rule clearly prevents a from being proven by RAA if y is in the model. This allows for a way of seeing if something was proven by RAA or using classical support.

The only rSM is:

$$\Gamma\left(\{a, x, y\}\right) = \{a, x, y\}$$

$$P_5 = \{a \leftarrow x$$
$$x \leftarrow y, \sim a$$
$$y \leftarrow\sim a\}$$

a is in two RAA chains.

$$\Delta\left(\{\sim a\}\right) = \{\{\sim a\}, \{y\}, \{x, [\sim a]\}, \{a\}\}$$
$$\{\{\sim a\}, \{x, [y]\}, \{a\}\}$$

Note that $\sim a$ is actually provable by RAA through a conjunction, and both sides of the conjunction end up in a. So the rules to add are $a \leftarrow\sim y$ (where

we don't include $\sim a$ as that is the literal we came from in the first place) and $a \leftarrow\!\sim x, \sim y$ (where y is added negated because it is in another RAA chain for $\sim a$).

a is the only rSM.

$$\Gamma(\{a\}) = \{a\}$$

$$\begin{aligned} P_6 = \{&a \leftarrow x \\ &x \leftarrow y, \sim x \\ &y \leftarrow\!\sim a\} \end{aligned}$$

Both a and x are in OLONs with preconditions which deny each other.

$$\begin{aligned} \Delta(\{\sim x\}) =& \{\{\sim x\}, \{x, [y]\}\} \\ \Delta(\{\sim a\}) =& \{\{\sim a\}, \{y\}, \{x, [\sim x]\}, \{a\}\} \end{aligned}$$

The generated rules are $x \leftarrow y$ and $a \leftarrow\!\sim y, \sim x$. a is the only rSM, even though x is also in a RAA chain.

$$\Gamma(\{a\}) = \{a\}$$

$$\begin{aligned} P_7 = \{&a \leftarrow\!\sim b \\ &b \leftarrow\!\sim c, x \\ &c \leftarrow\!\sim a \\ &x \leftarrow a, z \\ &z \leftarrow\!\sim x\} \end{aligned}$$

This is the OLON of 3, with a few extra rules.

$$
\begin{aligned}
\Delta\left(\{\sim a\}\right) =& \{\{\sim a\},\{c\},\{\sim b\},\{a\}\} \\
\Delta\left(\{\sim b\}\right) =& \{\{\sim b\},\{a\},\{\sim c\},\{b,[x]\}\} \\
& \{\{\sim b\},\{a\},\{x,[z]\},\{\sim z\}\} \\
& \{\{\sim a\},\{c\},\{\sim b\},\{a\}\} \\
\Delta\left(\{\sim c\}\right) =& \{\{\sim a\},\{c\},\{\sim b\},\{a\}\} \\
& \{\{\sim a\},\{c\},\{\sim b\},\{a\}\} \\
& \{\{\sim a\},\{c\},\{\sim b\},\{a\}\} \\
\Delta\left(\{\sim x\}\right) =& \{\{\sim a\},\{c\},\{\sim b\},\{a\}\}
\end{aligned}
$$

I show the generated rules side by side with the program, to be easier to check the several rSMs:

$$
\begin{aligned}
P_7 = \{ & a \leftarrow\sim b \\
& b \leftarrow\sim c, x \\
& c \leftarrow\sim a \\
& x \leftarrow a, z \\
& z \leftarrow\sim x\}
\end{aligned}
\qquad\qquad
\begin{aligned}
Rules_a &= \{a \leftarrow\sim c\} \\
Rules_b &= \{b \leftarrow\sim a, \sim x \\
& \qquad\quad b \leftarrow\sim a, z, c\} \\
Rules_c &= \{c \leftarrow\sim b, x\} \\
Rules_x &= \{x \leftarrow\sim z, a\}
\end{aligned}
$$

$$
\Gamma\left(\{a\}\right) = \{a\}
$$

$$
\begin{aligned}
P_8 = \{ & a \leftarrow x, y \\
& x \leftarrow z \\
& z \leftarrow\sim a \\
& y \leftarrow w \\
& w \leftarrow\sim a\}
\end{aligned}
$$

$$
\begin{aligned}
P_9 = \{ & a \leftarrow\sim a, \sim b \\
& b \leftarrow\sim b, d \\
& d \leftarrow\sim a\}
\end{aligned}
$$

$$P_{10} = \{a \leftarrow x$$
$$x \leftarrow\sim a$$
$$b \leftarrow\sim a$$
$$c \leftarrow\sim b$$
$$d \leftarrow\sim c\}$$

$$P_{11} = \{a \leftarrow\sim b$$
$$b \leftarrow a\}$$

$$P_{12} = \{y \leftarrow\sim a$$
$$a \leftarrow\sim b$$
$$b \leftarrow\sim c\}$$

$$P_{13} = \{a \leftarrow\sim x, \sim y$$
$$x \leftarrow\sim z$$
$$z \leftarrow\sim a$$
$$y \leftarrow\sim w$$
$$w \leftarrow\sim a\}$$

References

[1] A. M. Pinto. Explorations in revised stable models – a new semantics for logic programs. Master's thesis, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal, 2005.