Partially justifying an implementation of tabling in an extension of the propositional Horn fragment of Bedwyr v1.4*

Quentin Heath
Parsifal, LIX, École polytechnique

October 29, 2014

Abstract

Proof search in Bedwyr uses tabling to deal with long or infinite proofs. The proof-theoretic version of tabling is the cut rule, where a result can be assumed in a computation, provided it can be proved separately. While this is all there is to tabling in the event that only previously (dis)proved atoms are met again, dealing with a new occurrence of an atom *within* the proof or the refutation of another of its occurrences is more complicated. Firstly, this induces a loop in the proof and, if no care is taken when using the cut rule, in the computation; secondly, loops from distinct atoms can interfere. These problems seem to be direct consequences of the fact that the use of the cut rule implies the use of both sides of sequents, while Bedwyr purposely handles single-formula sequents, and thus actually has no real support for hypothesis (resp. conclusions) when proving (resp. disproving). We will see how a carefully chosen (co-)invariant can be used to prove the correctness of Bedwyr's loop handling, how a rudimentary notion of context (hypothesis or conclusions) has to be added to ensure that this doesn't break the regular tabling, and how this context can be extracted as a history of the proof.

Contents

*Support for this work has been partially obtained from Inria (through the ADT Grant "BATT").

Introduction

The Bedwyr system [Bae+07] is an extension of logic programming which deals symmetrically with finite success and finite failure. This is done by incorporating in the sequent calculus inference rules for definitions that allow arbitrary fixpoints to be explored. As those rules can generate divergence in proof-search, tabling is used to detect some loops over least (resp. greatest) fixpoints, which are treated as failures (resp. successes). A side effect of tabling is that, once an atom is tabled as proved or disproved, whether by using a loop rule or not, it can be used as a fact later in the computation. We will start by justifying both those uses for the table with cuts and, in the first case, a carefully chosen (co-)induction (co-)invariant. We will then extract from this a set of new rules to be included to *tabling calculus*, a variant of sequent calculus.

The following is to be seen more as a precise technical description of parts of an existing system, than as a theoretical work that can justify a posteriori an implementation. As such, it assumes Bedwyr is known by the reader, makes use of the syntax of Bedwyr definition files, offers only a limited view of the possibilities of tabling with respect to fixpoints exploration, and fails to prove most of what it states.

1 Shortcuts and notations

In all of the following, mutual recursion is not considered. Indeed, mutually recursive predicates can always be rewritten as a single predicate:

```
Define
  inductive p : nat -> nat -> prop,
  inductive q : string -> prop
by
  p 0 1   := q "bar" ;
  q "foo" := p 2 3.

% the following is equivalent to the previous definition bloc
Kind t type.
Type p' nat -> nat -> t.
Type q' string -> t.

Define inductive r : t -> prop by
  r (p' 0 1)   := r (q' "bar") ;
  r (q' "foo") := r (p' 2 3).
```

Moreover, most of the time, only one of the inductive and the co-inductive cases will be discussed, the other one having dual properties and limitations.

We will have the following notation conventions:

- lower-case roman letters are predicates (p, q), atoms (a, b) or literals (l)
- upper-case roman letters are formulae (A, B)
- upper-case greek letters are sets of literals (Δ) or sets of extended literals (Γ, Θ – cf. section 2.3)
- Δ, l means $\Delta \cup \{l\}$
- $A \wedge \Theta$ means $A \wedge (\bigwedge_{\theta \in \Theta} \theta)$

- operators switch between their binary form (with neutral element) and their n-ary form at will
- the extended formula $A_{|\varnothing}$ can be written A
- if $a = p\,\boldsymbol{t}$ and $p\,\boldsymbol{x} \triangleq \mathcal{B}\,p\,\boldsymbol{x}$, then $\widetilde{\mathcal{B}}\,a$ actually means $(\mathcal{B}\,p)\,\boldsymbol{t}$ while $\widetilde{\mathcal{B}}\,(a^{\perp})$ means $((\mathcal{B}\,p)\,\boldsymbol{t})^{\perp}$
- if $l = a^{\perp}$, then $l^{\perp}$ means a
- $\doteq$ and $\neq$ are connectives, corresponding to the Bedwyr unifier and to inference rules

2 Loop handling

As we handle the least and greatest fixpoints of a definition, we want this definition to be monotonous. This is the case if a predicate only appears positively in its definition; hence we will assume definitions are well stratified.

We call the first occurrence of an atom the *root occurrence*, and those that appear within its proof, *looping occurrences*. The *depth* of a loop is the maximum relative depth of its looping occurrences, i.e. the maximum number of definition unfoldings needed to reach them from the root occurrence.

We start with the simplest loop possible, then introduce additional looping occurrences, root occurrences of other atoms, and looping occurrences of the latter. Each time, we present features to add to sequent calculus to support the new framework. Last, we show a set of inference rule for this extended calculus, that handle completely loop detection.

2.1 Rationale – simple loop of depth 1

The intuition is that, for an co-inductively defined predicate, the looping occurrence succeeds. If by doing so we prove the root occurrence, then we evaluated the greatest fixpoint; on the other hand, if the root occurrence still fails, this was not a fixpoint, but since the definition is monotonous, having the looping occurrence fail would also have made the root occurrence fail, which means that in any case the value computed for the root occurrence is the correct value for the atom.

For instance, given the following predicate:

```
Define coinductive p : nat -> prop by
  p 0 := p 0 ;          % true
  p 1 := p 1 /\ false. % false
```

the proof of the atom `p 0` only involves the simplest kind of loop and an obvious success, while the proof of `p 1` involves a successful loop inside of a failing disjunction, and thus fails despite of the loop.

We could see this as iterating the definition on an initial maximal value, which is known to eventually reach the greatest fixpoint. Instead, we choose to build a pre-fixpoint, as it has a direct translation to the implementation in Bedwyr.

Listing 1: Single multiloop (predicate)

```
Define coinductive p : nat -> prop by
  p 0 := p 1 /\ p 2 ;
  p 1 := p 1 /\ p 0 ;
  p 2 := false.
```

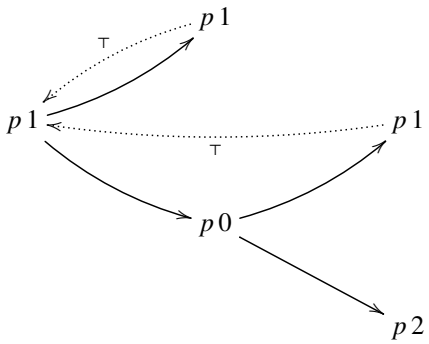

Figure 1: Single multiloop (computation)

2.2 Basic unfolding rule – single multiloop

Here, *single* means that we still consider a single root occurrence of a single atom, and *multiloop* means that we can have any finite number of looping occurrences of this atom of any depth.

For instance, considering the predicate given in listing 1, a request on `p 1` will bring about the following computation (fig. 1): we try to prove `p 1` by unfolding it, we find the conjunction of `p 1` (which succeeds, being a looping occurrence) and `p 0`, we unfold the latter and find the conjunction of `p 1` (which succeeds again) and `p 2` (which fails, thereby disproving `p 0` then `p 1`).

Consider now the general case of some co-inductive predicate p defined by $\mathcal{B}$:

$$p\,\boldsymbol{x} \overset{\nu}{=} \mathcal{B}\,p\,\boldsymbol{x}$$

Assume that the proof of some atom $p\,\boldsymbol{t}$ involves at least one looping occurrence of this atom, and that the depth of the loop is n. Until we make loop-detection a primitive of the logic, what we are computing is not $p\,\boldsymbol{t}$ but $S_0\,\boldsymbol{t}$ where S_d are predicates defined as shown in fig. 2:

- at depth d, $0 < d \leq n$, we use the value of T_d
- when on a looping occurrence, we succeed: $T_d\,\boldsymbol{x} = (x \doteq \boldsymbol{t}) \vee S_d\,\boldsymbol{x}$
- when not on a looping occurrence:
  - at depth d, $0 \leq d < n$, we unfold $\mathcal{B}$ once: $S_d = \mathcal{B}\,T_{d+1}$
  - at depth n, as there are no more looping occurrences, we use the actual predicate: $S_n = p$

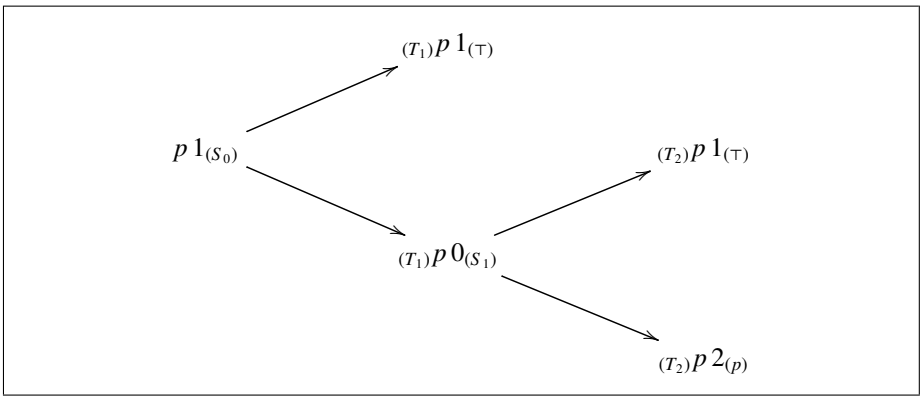

Figure 2: Single multiloop (annotated computation)

This S_0 mimics the computations Bedwyr avoids (looping occurrences) or does (everything else) when $p\,\boldsymbol{t}$ is requested. To show that $S_0\,\boldsymbol{t} \equiv p\,\boldsymbol{t}$, we use simple cuts:

$$\frac{p\,\boldsymbol{x} \vdash S\,\boldsymbol{x} \qquad S\,\boldsymbol{t} \vdash}{p\,\boldsymbol{t} \vdash}\ \text{failure} \qquad\qquad \frac{\vdash T\,\boldsymbol{t} \quad T\,\boldsymbol{x} \vdash p\,\boldsymbol{x}}{\vdash p\,\boldsymbol{t}}\ \text{success}$$

To choose the cut formulae, let us use $\mathcal{A} \preceq \mathcal{B}$ as syntactic sugar for $\forall \boldsymbol{x}\, \mathcal{A}\,\boldsymbol{x} \supset \mathcal{B}\,\boldsymbol{x}$. $S_n \preceq T_n$, so as $\mathcal{B}$ is monotonous, $S_n = \mathcal{B}\,S_n \preceq \mathcal{B}\,T_n = S_{n-1}$. It follows that $\forall d\ (S_{d+1} \preceq S_d)$, which together with the definition of T_d, gives us the three facts

$$\begin{cases} \vdash S_0 \succeq p \\ \vdash S_0 \succeq S_1 \\ \vdash T_1\,\boldsymbol{t} \end{cases}$$

The first fact gives S_0 as cut formula for the failure rule, which becomes

$$\frac{\overline{\vdash p \preceq S_0} \quad S_0\,\boldsymbol{t} \vdash}{p\,\boldsymbol{t} \vdash}\ \text{failure}$$

For the success rule, we need to add to the logic a post-fixpoint rule:

$$\frac{\vdash T \preceq \mathcal{B}\,T}{\vdash T \preceq p}\ p = \nu\mathcal{B}$$

and, with T_1 as cut formula[1] and the two remaining facts, we get

$$\dfrac{\overline{\vdash T_1\,\boldsymbol{t}} \qquad \dfrac{\dfrac{\dfrac{\vdash S_0\,\boldsymbol{t}}{x \doteq \boldsymbol{t} \vdash S_0}\ \doteq_L \quad \overline{\vdash S_1 \preceq S_0}}{\vdash T_1 \preceq S_0}\ \vee_L}{\vdash T_1 \preceq p}\ p = \nu\mathcal{B}}{\vdash p\,\boldsymbol{t}}\ \text{success}$$

hence the two new rules

$$\frac{S_0\,\boldsymbol{t} \vdash}{p\,\boldsymbol{t} \vdash}\ \nu\text{-failure} \qquad\qquad \frac{\vdash S_0\,\boldsymbol{t}}{\vdash p\,\boldsymbol{t}}\ \nu\text{-success}$$

[1] … and co-invariant. S_0 could be used instead, with a barely longer success derivation.

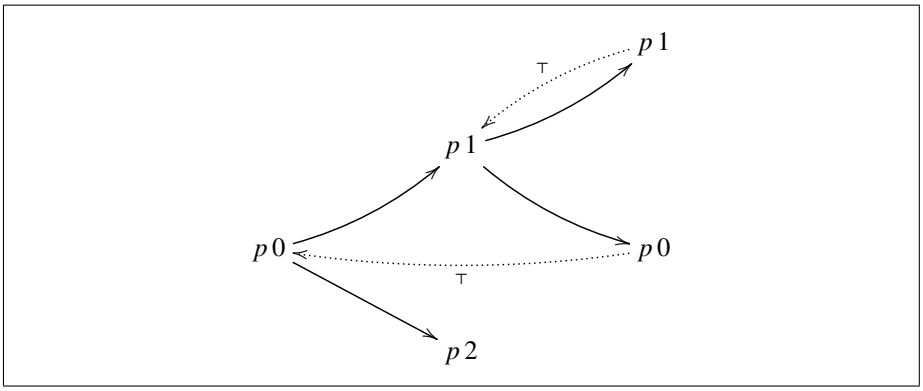

Figure 3: Single multiloop (conundrum)

which we could rewrite

$$\frac{\mathcal{B}\,p\,\boldsymbol{t} \vdash}{p\,\boldsymbol{t} \vdash}\ \nu\text{-failure} \qquad\qquad \frac{(p\,\boldsymbol{t}) \vdash \mathcal{B}\,p\,\boldsymbol{t}}{\vdash p\,\boldsymbol{t}}\ \nu\text{-success}$$

provided that the parenthesised literal is only involved in init rules.

The use of a post-fixpoint rule together with a cut is also found in μLJ [Bae08], and the resulting least-fixpoint rule was already described in *LINC* [Tiu04].

For the inductive case, similar rules μ-FAILURE and μ-SUCCESS can be proved with dual predicates S_d and T_d. These rules justify the computation used by the historical implementation of Bedwyr (pre-1.4), where (root occurrences of) atoms were tabled in a "working" table until proved or disproved, and (looping occurrences of) atoms already present in the working table succeeded (resp. failed) for co-inductive (resp. inductive) predicates.

2.3 Local context – included atoms

Unfortunately, the previous inference rules fail to account for atoms appearing between a looping occurrence and the corresponding root occurrence. If, given listing 1, we query `p 0` instead of `p 1`, we unfold it, we find the conjunction of `p 1` and `p 2`, we unfold the former, we find the conjunction of `p 1` (which succeeds, being a looping occurrence) and `p 0` (which succeeds for the same reason), thereby proving `p 1`, then `p 2` fails, thereby disproving `p 0` (fig. 3).

We already know that the S_0 corresponding to this computation is such that $S_0\,0 = p\,0 = \bot$ but we now see that $S_0\,1 = \top$ while $p\,1 = \bot$. Not only this means that the results computed within a loop must not be kept, it also prevents us to use the previous rules on the root occurrence of an atom which is itself within a loop. The first implementations of Bedwyr dealt by invalidating all results within loops (i.e. between, but not including, a root occurrence and its looping occurrences)[2], which could have a serious performance impact.

In order to only invalidate wrong intermediate results and to keep the others, we need a way to express the fact that, during the computation for $p\,0$, we proved $p\,0 \vdash p\,1$

[2]Actually, until version 1.3, only inductive results within loops were invalidated and forgotten, while buggy co-inductive results were kept. Version 1.4 introduced the correction along with the improvement described in section 2.4. Therefore no version of Bedwyr actually corresponds to the present description.

Listing 2: Multicut

```
Define coinductive p : nat -> prop by
  p 0 := p 1 ;
  p 1 := p 2 ;
  p 2 := p 0.

Define test : nat -> prop by
  test 0 := p 0 /\ p 2 /\ p 1 /\ p 0 ;
  test 1 := p 0 /\ p 1 /\ p 2 /\ p 0.
```

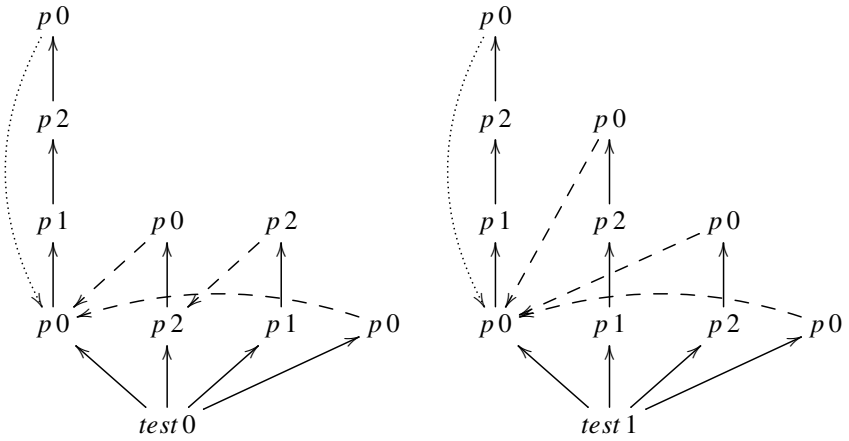

Figure 4: Multicut

and not plain $\vdash p\,1$. We will thus add a notion of "local context" to our calculus, corresponding to the hypotheses under which a result is valid. The range of these hypotheses will be restricted to the set of tablable literals, and will usually be noted Δ. A formula (resp. literal) together with a local context is an *extended formula* (resp. *extended literal*), and is usually noted $A_{|\Delta}$ (resp. $l_{|\Delta}$). Although a local context can be described as the left-hand side of an implication ($\vdash A_{|\Delta} \cong \Delta \vdash A$), we use this notation to distinguish the notion from a regular logical implication, as the computational meaning is quite different.

2.4 Local table – interleaved multiloops

Even though we didn't describe yet what kind of cut rule is used to avoid computation on *secondary occurrences* of atoms (i.e. occurrences that are neither root or looping), we will use listing 2 as a motivating example as to why we bother to keep those valid intermediate results.

Here, all p-atoms would be part of one another's loop, so the first to be called will be the only one using a looping rule, while the computations started by the other root atoms will stop prematurely on a secondary occurrence of a previously queried atom (fig. 4). Even if this prevents us from duplicating the whole loop for each of the

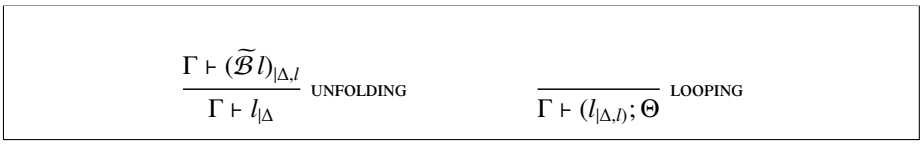

Figure 5: Unfolding rules

three following root atoms, the amount of duplicate unfoldings caused by forgetting intermediate atoms still goes from linear (best case, atoms are queried in the opposite order of their appearance in the loop) to quadratic (worst case, atoms are queried in the same order). For queries exploring a strongly connected graph, this can have quite an impact. For instance, the fix Bedwyr 1.4 brought to this reduced the time needed to check Peterson's algorithm [Pet81] by a factor of several thousand, although such a change cannot be expected in most cases.

We can see that what we want to prove by unfolding the loop is actually the set of literals $\{p\,0; p\,1; p\,2\}$, which suggests that we need to add the notion of "local table", usually noted Θ or $\{l_{i|\Delta_i}\}_i$, to our calculus. This is a set of extended literals, corresponding to a list of tablable facts that appear as intermediate results in a derivation: a proof of $\vdash A_{|\Delta}; \{l_{i|\Delta_i}\}_i$ is morally a poof of $\vdash (\Delta \supset A) \wedge \bigwedge_i (\Delta_i \supset l_i)$ which focuses on the first conjunct.

The way the local table is used is presented in section 3.

2.5 Looping rules – general case

We now extend the basic unfolding rules presented in section 2.2 to account for the local table (which is modified by the unfolding rules) and the local context (which is created by unfolding rules and used by looping rules). This gives us fig. 5, where we use literals to present rules for induction and co-induction at the same time.

The usual restrictions apply: unfolding is always allowed, looping only works with a negated atom and induction or an atom and co-induction.

These inference rules could be proved complete with respect to a given fragment of the logic, by means of more complex $(S_d)_d$ and $(T_d)_{d>0}$ providing us (co-)invariants accounting for all sorts of literals. Unfortunately the present method is to local, and not intuitive enough when dealing with multiple predicates. We refer to another presentation of the problem in an ulterior technical report from the author, where (co-)invariants are built in a different way.

3 Memoization

Let us now take care of secondary occurrences. In the case of a branching rule, we want the proof of the first premise to provide some (valid) intermediate results, and the proof of the second premise to use them. By allowing the permutation of the conjuncts, we build this new $\wedge_R$ rule:

$$\dfrac{\Gamma \vdash A \wedge \Theta_1 \qquad \dfrac{\dfrac{}{\Gamma \wedge A \wedge \Theta_1 \vdash \Theta_1} \qquad \Gamma \wedge \Theta_1 \vdash B \wedge \Theta_2}{\Gamma \wedge A \wedge \Theta_1 \vdash B \wedge \Theta_1 \wedge \Theta_2}\ \wedge_R, \text{WEAKENING}}{\Gamma \vdash A \wedge B \wedge \Theta_1 \wedge \Theta_2}\ \text{CUT}$$

or, if we stay focused,

$$\frac{\Gamma \vdash A; \Theta_1 \qquad \Gamma \cup \Theta_1 \vdash B; \Theta_2}{\Gamma \vdash A \wedge B; \Theta_1 \cup \Theta_2} \wedge'_R$$

This way, the proof gets a more sequential structure where the local tables remind us of the state of the memory during the computation, instead of a pure tree structure. A weakening rule was used to remove the (extended) formula A from the left-hand side of the second premise, so that it only contains extended literals.

To apply this to non branching rules, a transformation is involved. For instance, the set of rules

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_{R1} \qquad\qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_{R2}$$

doesn't have the sought-after linear structure, and thus the premise of the second rule lacks any kind of information about what may have been proved during a previous failed computation. For instance, under the negation-as-failure assumption, if we assume that the prover always tries the conjuncts in a fixed order, then the rules should be rewritten

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_{R1} \qquad\qquad \frac{\Gamma \vdash A^{\perp} \qquad \Gamma \vdash B}{\Gamma \vdash A \vee B} \vee'_{R2}$$

and then local table Θ can be added as before.

Two remarks have to be made about these rules. First, they leave the local context untouched; this is why it is left out here. Then, they are the rules that build the left-hand side of the sequents, so an init rule has to be added at the same time in the logic:

$$\frac{}{\Gamma, l \vdash l} \text{ TABLE ACCESS}$$

The computational interpretation of this initial rule is a membership test on the table, and it appears in the proof over all secondary occurrences, and only there.

4 Tabling calculus

We present here the whole set of inference rules for *tabling calculus*. The notations used are:

$$\begin{cases} \Gamma \vdash G; \Theta & \text{tabling-aware sequent} & \Gamma \cap \theta = \varnothing & \\ \Gamma & \text{input table} & \{l_{i|\Delta_i}\}_{i \in \mathcal{I}} & \text{set of extended literals} \\ G & \text{goal} & F_{|\Delta} & \text{extended formula} \\ \Theta & \text{output table} & \{l_{j|\Delta_j}\}_{j \in \mathcal{J}} & \text{set of extended literals} \\ \Delta & \text{hypotheses} & \{l_k\}_{k \in \mathcal{K}} & \text{set of literals} \end{cases}$$

The hypotheses of an extended formula are atoms of co-inductive predicates and negated atoms of inductive predicates. Most of the inference rules break down the goal to an extended literal; axioms and unfolding rules then apply by comparing this extended literal to the input and output tables, or by comparing its head to its hypotheses.

A few remarks:

$$\frac{}{\Gamma \vdash l_{|\Delta,l};\{\}}\ \text{L} \qquad \frac{}{\Gamma, a_{|\Delta} \vdash a_{|\Delta};\{\}}\ \text{P} \qquad \frac{}{\Gamma, a^{\perp}_{|\Delta} \vdash a^{\perp}_{|\Delta};\{\}}\ \text{D}$$

$l = (p\,\boldsymbol{t})$ if p is co-inductive, $l = (p\,\boldsymbol{t})^{\perp}$ if p is inductive

(a) Tabling axioms

$$\frac{\Gamma \vdash A^{\perp}_{|\Delta};\Theta}{\Gamma \vdash (A \wedge B)^{\perp}_{|\Delta};\Theta}\ \wedge_{L1} \qquad \frac{\Gamma \vdash A_{|\Delta};\Theta_1 \qquad \Gamma \uplus \Theta_1 \vdash B^{\perp}_{|\Delta};\Theta_2}{\Gamma \vdash (A \wedge B)^{\perp}_{|\Delta};\Theta_1 \uplus \Theta_2}\ \wedge_{L2}$$

$$\frac{\Gamma \vdash A_{|\Delta};\Theta_1 \qquad \Gamma \uplus \Theta_1 \vdash B_{|\Delta};\Theta_2}{\Gamma \vdash (A \wedge B)_{|\Delta};\Theta_1 \uplus \Theta_2}\ \wedge_{R} \qquad \frac{}{\Gamma \vdash \top_{|\Delta};\{\}}\ \top$$

(b) Conjunction rules

$$\frac{\Gamma \vdash A_{|\Delta};\Theta}{\Gamma \vdash (A \vee B)_{|\Delta};\Theta}\ \vee_{R1} \qquad \frac{\Gamma \vdash A^{\perp}_{|\Delta};\Theta_1 \qquad \Gamma \uplus \Theta_1 \vdash B_{|\Delta};\Theta_2}{\Gamma \vdash (A \vee B)_{|\Delta};\Theta_1 \uplus \Theta_2}\ \vee_{R2}$$

$$\frac{\Gamma \vdash A^{\perp}_{|\Delta};\Theta_1 \qquad \Gamma \uplus \Theta_1 \vdash B^{\perp}_{|\Delta};\Theta_2}{\Gamma \vdash (A \vee B)^{\perp}_{|\Delta};\Theta_1 \uplus \Theta_2}\ \vee_{L} \qquad \frac{}{\Gamma \vdash \bot^{\perp}_{|\Delta};\{\}}\ \bot$$

(c) Disjunction rules

$$\frac{\Gamma \vdash l_{|\Delta_1};\Theta}{\Gamma \vdash l_{|\Delta_2};\Theta}\ \text{w}\ (\Delta_1 \subseteq \Delta_2)$$

(d) Structural rule

$$\frac{\Gamma \vdash (\widetilde{\mathcal{B}}\, l_j)_{|\Delta'_j};\{l_{i|\Delta'_i}\}_{i\neq j}}{\Gamma \vdash l_{j|\Delta_j};\{l_{i|\Delta_i}\}_i}\ \nu/\mu^{\perp}\ (\Delta_i = \Delta'_i \setminus \{l_j\},\ l_j \in \Delta'_i \text{ ONLY IF } \Delta_j \subseteq \Delta_i)$$

$$\frac{\Gamma \vdash (\widetilde{\mathcal{B}}\, l_j)_{|\Delta'_j};\{l_{i|\Delta'_i}\}_{i\neq j}}{\Gamma \vdash l_{j|\Delta_j};\{l_{i|\Delta_i}\}_{i\in\mathcal{I}}}\ \nu^{\perp}/\mu\ (\Delta_i = \Delta'_i \setminus \{l^{\perp}_j\},\ l^{\perp}_j \in \Delta'_i \text{ ONLY IF } i \notin \mathcal{I} - \{j\})$$

$l_j = a$ for the success rules, $l_j = a^{\perp}$ for the failures rules

(e) Unfolding rules

Figure 6: Tabling calculus inference rules

fig. 6a the guard for the L rule is superfluous, as a local context (whether Δ or that of an extended literal from Γ or Θ) can only contain such literals

fig. 6b and fig. 6c those are completely dual, which makes one wonder about the intuitionist trait of the logic; the answer might be that the $\vee'_{R2}$ chosen in section 3 relies on the negation-as-failure assumption, which looks like a weak form (or meta-form) of the law of excluded middle

fig. 6d this weakening rule is to be used before an unfolding rule or a P or D axiom and only there; it is written as a separate rule to simplify the guards of those others, but when we write that an unfolding rule is just above a root occurrence, we really mean that a weakening can appear in between

fig. 6e the first guard means that we can assume a dependency on l only if we already explicitly depend on the literals l itself depends on; the second guard means that we remove results which were just invalidated because of their dependencies

5 Example

Let us now switch to mutually recursive co-inductive predicates to ease proof display, and choose a more complex example of interleaved multiloops: fig. 7. Here, $\underline{a}$ is a root occurrence, $\overline{a}$ is a looping occurrence and a is a secondary occurrence.

fig. 8 shows that, at any point in the proof, tables show the sets (or list, depending on the chosen representation) of literals involved in tabling rules:

- the local table corresponds to (co-)induction occurring as ancestors
- the input table corresponds to tabling rules occurring before in a postfix traversal
- the output table corresponds to tabling rules occurring after in a prefix traversal

It is possible to extract these information to show the history of the proof. This has been implemented in recent versions of Bedwyr (see the `#export` command), although the output is raw XML that is not used by external tools yet.

Listing 3: Complex interleaved multiloops

```
Define
  coinductive a : nat -> prop, coinductive b : nat -> prop,
  coinductive c : nat -> prop, coinductive d : nat -> prop,
  coinductive e : nat -> prop
by
  a := b /\ c /\ d /\ e /\ true ;
  b := b ;
  c := a /\ b /\ c /\ d /\ e ;
  d := a /\ b /\ c /\ d /\ e ;
  e := e.
```

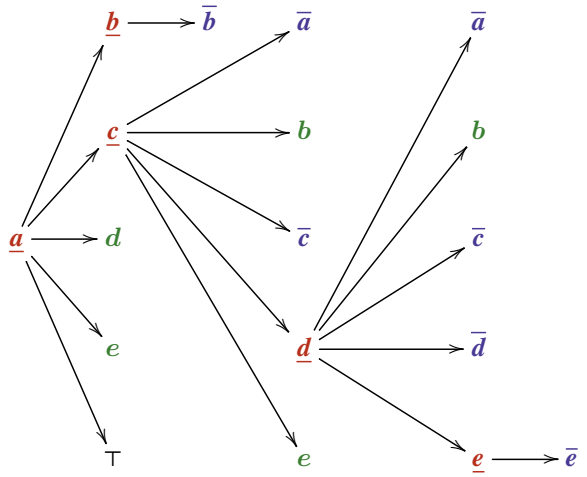

Figure 7: Complex interleaved multiloops

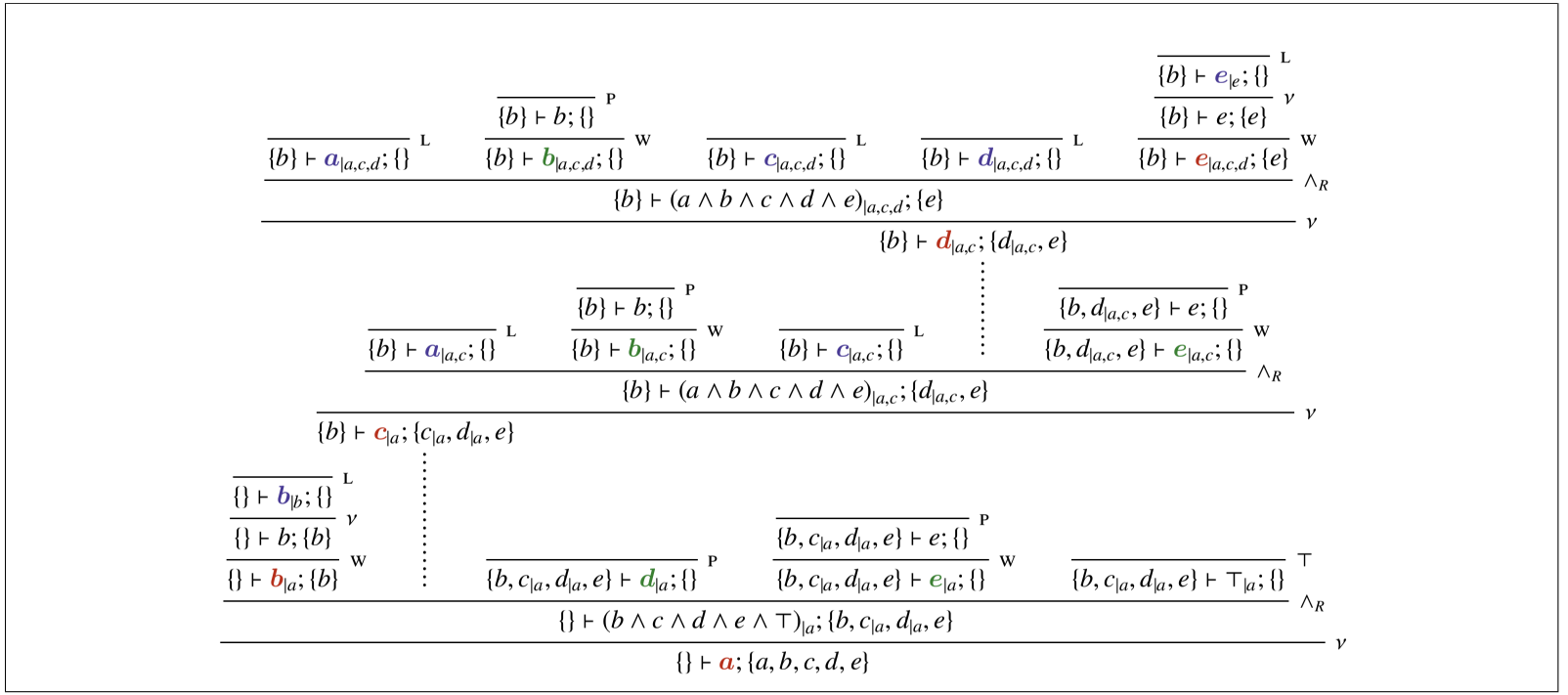

Figure 8: Complex success derivation (root occurrences,looping occurrences,secondary occurrences)

Conclusion

We have presented a way to build (co-)invariants that can be used with (co-)induction rules to prove the behaviour of Bedwyr in decreasingly simple settings. In the same time, we have introduced additions to sequent calculus which culminate in rules for *tabling calculus*.

These additions can be extracted as a history of the proof at any point in its computation.

As we haven't showed soundness or relative completeness of this calculus, we refer to another presentation of the problem in another unpublished technical report from the author. There, (co-)invariants are built in a different way, that doesn't need to augment sequent calculus with a tabling lexicon to be justified; instead, the notion of derivation itself is augmented with *back-arcs*. The present report is kept for the sake of historical completeness, and as only presentation of *tabling calculus*.

References

[Bae+07] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. "The Bedwyr System for Model Checking over Syntactic Expressions". In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Bremen, Germany: Springer-Verlag, 2007, pp. 391–397. ISBN: 978-3-540-73594-6. DOI: 10.1007/978-3-540-73595-3_28.

[Bae08] David Baelde. "A linear approach to the proof-theory of least and greatest fixed points". PhD thesis. Ecole Polytechnique, Dec. 2008. URL: http://www.lix.polytechnique.fr/~dbaelde/thesis/.

[Pet81] Gary L. Peterson. "Myths About the Mutual Exclusion Problem." In: *Inf. Process. Lett.* 12.3 (1981), pp. 115–116. URL: http://dblp.uni-trier.de/db/journals/ipl/ipl12.html#Peterson81.

[Tiu04] Alwen Tiu. "A Logical Framework for Reasoning about Logical Specifications". PhD thesis. Pennsylvania State University, May 2004. URL: http://www.lix.polytechnique.fr/Labo/Alwen.Tiu/etd.pdf.