

Abstract Completeness

Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel

August 16, 2018

Abstract

This is a formalization of an abstract property of possibly infinite derivation trees (modeled by a codatatype), that represents the core of a Beth–Hintikka-style proof of the first-order logic completeness theorem and is independent of the concrete syntax or inference rules. This work is described in detail in a publication by the authors [2].

The abstract proof can be instantiated for a wide range of Gentzen and tableau systems as well as various flavors of FOL—e.g., with or without predicates, equality, or sorts. Here, we give only a toy example instantiation with classical propositional logic. A more serious instance—many-sorted FOL with equality—is described elsewhere [1].

References

- [1] J. C. Blanchette and A. Popescu. Mechanizing the metatheory of sledgehammer. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *FroCoS 2013*, volume 8152 of *LNCS*, pages 245–260. Springer, 2013.
- [2] J. C. Blanchette, A. Popescu, and D. Traytel. Unified classical logic completeness: A coinductive pearl. In S. Demri, D. Kapur, and C. Weidenbach, editors, *IJCAR 2014*, LNCS. Springer, 2014.

Contents

1	General Tree Concepts	2
2	Rule Systems	2
3	A Fair Enumeration of the Rules	3
4	Persistent rules	7
5	Code generation	9
6	Toy instantiation: Propositional Logic	10

1 General Tree Concepts

codatatype 'a tree = Node (root: 'a) (cont: 'a tree fset)

inductive tfinite **where**

tfinite: ($\bigwedge t'. t' \in \text{cont } t \implies \text{tfinite } t'$) $\implies \text{tfinite } t$

coinductive ipath **where**

ipath: $\llbracket \text{root } t = \text{shd steps}; t' \in \text{cont } t; \text{ipath } t' (\text{stl steps}) \rrbracket \implies \text{ipath } t \text{ steps}$

primcorec konig **where**

shd (konig t) = root t

| stl (konig t) = konig (SOME t'. t' \in cont t $\wedge \neg$ tfinite t')

lemma Konig: $\neg \text{tfinite } t \implies \text{ipath } t (\text{konig } t)$

by (coinduction arbitrary: t) (metis (lifting) tfinite.simps konig.simps someI-ex)

2 Rule Systems

type-synonym ('state, 'rule) step = 'state \times 'rule **type-synonym** ('state, 'rule)

dtree = ('state, 'rule) step tree

locale RuleSystem-Defs =

fixes eff :: 'rule \Rightarrow 'state \Rightarrow 'state fset \Rightarrow bool

and rules :: 'rule stream

begin

abbreviation R \equiv sset rules

lemma countable-R: countable R **by** (metis countableI-type countable-image sset-range)

lemma NE-R: $R \neq \{\}$ **by** (metis UNIV-witness all-not-in-conv empty-is-image sset-range)

definition enabled r s $\equiv \exists sl. \text{eff } r s sl$

definition pickEff r s \equiv if enabled r s then (SOME sl. eff r s sl) else the None

lemma pickEff: enabled r s $\implies \text{eff } r s (\text{pickEff } r s)$

by (metis enabled-def pickEff-def tfl-some)

abbreviation effStep step $\equiv \text{eff } (\text{snd step}) (\text{fst step})$

abbreviation enabledAtStep r step $\equiv \text{enabled } r (\text{fst step})$

abbreviation takenAtStep r step $\equiv \text{snd step} = r$

Saturation is a very strong notion of fairness: If a rule is enabled at some point, it will eventually be taken.

definition saturated r $\equiv \text{alw } (\text{holds } (\text{enabledAtStep } r) \text{ impl ev } (\text{holds } (\text{takenAtStep } r)))$

definition Saturated steps $\equiv \forall r \in R. \text{saturated } r \text{ steps}$

coinductive wf where

wf: $\llbracket \text{snd } (\text{root } t) \in R; \text{effStep } (\text{root } t) (\text{fimage } (\text{fst } o \text{ root}) (\text{cont } t));$
 $\bigwedge t'. t' \in \text{cont } t \implies \text{wf } t' \rrbracket \implies \text{wf } t$

coinductive epath where

epath: $\llbracket \text{snd } (\text{shd } \text{steps}) \in R; \text{fst } (\text{shd } (\text{stl } \text{steps})) \in \text{sl}; \text{effStep } (\text{shd } \text{steps}) \text{ sl};$
 $\text{epath } (\text{stl } \text{steps}) \rrbracket \implies \text{epath } \text{steps}$

lemma wf-ipath-epath:

assumes *wf t ipath t steps*

shows *epath steps*

proof –

have $*$: $\bigwedge t \text{ st. ipath } t \text{ st} \implies \text{root } t = \text{shd } \text{st}$ **by** (*auto elim: ipath.cases*)

show *?thesis* **using** *assms*

proof (*coinduction arbitrary: t steps*)

case *epath*

then show *?case* **by** (*cases rule: wf.cases[case-product ipath.cases]*) (*metis **

o-apply fimageI)

qed

qed

definition *fair rs* $\equiv \text{sset } rs \subseteq R \wedge (\forall r \in R. \text{alw } (\text{ev } (\text{holds } ((=) r))) rs)$

lemma *fair-stl*: *fair rs* $\implies \text{fair } (\text{stl } rs)$

unfolding *fair-def* **by** (*metis alw.simps set-mp stl-sset subsetI*)

lemma *sdrop-fair*: *fair rs* $\implies \text{fair } (\text{sdrop } m rs)$

using *alw-sdrop* **unfolding** *fair-def* **by** (*metis alw.coinduct alw-nxt fair-def fair-stl*)

3 A Fair Enumeration of the Rules

definition *fenum* $\equiv \text{flat } (\text{smap } (\lambda n. \text{stake } n \text{ rules}) (\text{fromN } 1))$

lemma *sset-fenum*: *sset fenum* = *R*

unfolding *fenum-def* **by** (*subst sset-flat*)

(*auto simp: stream.set-map in-set-conv-nth sset-range[of rules]*,

metis atLeast-Suc-greaterThan greaterThan-0 lessI range-eqI stake-nth)

lemma *fair-fenum*: *fair fenum*

proof –

{ **fix** *r* **assume** $r \in R$

then obtain *m* **where** $r: r = \text{rules} !! m$ **unfolding** *sset-range* **by** *blast*

{ **fix** $n :: \text{nat}$ **and** *rs* **let** $?fenum = \lambda n. \text{flat } (\text{smap } (\lambda n. \text{stake } n \text{ rules}) (\text{fromN } n))$

assume $n > 0$

hence $\text{alw } (\text{ev } (\text{holds } ((=) r))) (rs @- ?fenum n)$

proof (*coinduction arbitrary: n rs*)

case *alw*

```

show ?case
proof (rule exI[of - rs @- ?fenum n], safe)
  show  $\exists n' rs'. stl (rs @- ?fenum n) = rs' @- ?fenum n' \wedge n' > 0$ 
  proof(cases rs)
    case Nil thus ?thesis unfolding alw by (intro exI) auto
  qed (auto simp: alw intro: exI[of - n])
next
  show ev (holds ((=) r)) (rs @- flat (smap ( $\lambda n$ . stake n rules) (fromN
n)))
    using alw r unfolding ev-holds-sset
    by (cases m < n) (force simp: stream.set-map in-set-conv-nth)+
  qed
}
}
thus fair fenum unfolding fair-def sset-fenum
by (metis fenum-def alw-shift le-less zero-less-one)
qed

```

definition trim rs s = sdrop-while (λr . Not (enabled r s)) rs

primcorec mkTree **where**

```

  root (mkTree rs s) = (s, (shd (trim rs s)))
| cont (mkTree rs s) = fimage (mkTree (stl (trim rs s))) (pickEff (shd (trim rs s))
s)

```

lemma mkTree-unfold[code]: mkTree rs s =
(case trim rs s of SCons r s' \Rightarrow Node (s, r) (fimage (mkTree s') (pickEff r s)))
by (subst mkTree.ctr) (simp split: stream.splits)

end

locale RuleSystem = RuleSystem-Defs eff rules

for eff :: 'rule \Rightarrow 'state \Rightarrow 'state fset \Rightarrow bool **and** rules :: 'rule stream +
fixes S :: 'state set

assumes eff-S: $\bigwedge s r sl s'. \llbracket s \in S; r \in R; \text{eff } r s sl; s' \in | sl \rrbracket \Longrightarrow s' \in S$

and enabled-R: $\bigwedge s. s \in S \Longrightarrow \exists r \in R. \exists sl. \text{eff } r s sl$

begin

definition minWait rs s \equiv LEAST n. enabled (shd (sdrop n rs)) s

lemma trim-alt:

assumes s: s \in S **and** rs: fair rs

shows trim rs s = sdrop (minWait rs s) rs

proof (unfold trim-def minWait-def sdrop-simps, rule sdrop-while-sdrop-LEAST[unfolded
o-def])

from enabled-R[OF s] **obtain** r sl **where** r: r \in R **and** sl: eff r s sl **by** blast

from bspec[OF conjunct2[OF rs[unfolded fair-def]] r] **obtain** m **where** r = rs
!! m

by atomize-elim (erule alw.cases, auto simp only: ev-holds-sset sset-range)

with r sl **show** $\exists n$. enabled (rs !! n) s **unfolding** enabled-def **by** auto

qed

lemma *minWait-ex*:

assumes $s: s \in S$ and $rs: \text{fair } rs$

shows $\exists n. \text{enabled } (\text{shd } (\text{sdrop } n \text{ } rs)) \text{ } s$

proof –

obtain r where $r: r \in R$ and $e: \text{enabled } r \text{ } s$ using *enabled-R s unfolding enabled-def* by *blast*

then obtain n where $\text{shd } (\text{sdrop } n \text{ } rs) = r$ using *sdrop-fair[OF rs]*

by (*metis (full-types) alw-nxt holds.simps sdrop.simps(1) fair-def sdrop-wait*)

thus *?thesis* using $r \ e$ by *auto*

qed

lemma assumes $s \in S$ and *fair rs*

shows *trim-in-R*: $\text{shd } (\text{trim } rs \text{ } s) \in R$

and *trim-enabled*: $\text{enabled } (\text{shd } (\text{trim } rs \text{ } s)) \text{ } s$

and *trim-fair*: *fair (trim rs s)*

unfolding *trim-alt[OF assms] minWait-def*

using *LeastI-ex[OF minWait-ex[OF assms]] sdrop-fair[OF assms(2)]*

conjunct1[OF assms(2)[unfolded fair-def]] by *simp-all (metis set-mp snth-sset)*

lemma *minWait-least*: $\llbracket \text{enabled } (\text{shd } (\text{sdrop } n \text{ } rs)) \text{ } s \rrbracket \implies \text{minWait } rs \text{ } s \leq n$

unfolding *minWait-def* by (*intro Least-le conjI*)

lemma *in-cont-mkTree*:

assumes $s: s \in S$ and $rs: \text{fair } rs$ and $t': t' \in \text{cont } (\text{mkTree } rs \text{ } s)$

shows $\exists sl' \ s'. \ s' \in S \wedge \text{eff } (\text{shd } (\text{trim } rs \text{ } s)) \text{ } s \ sl' \wedge$

$s' \in \text{cont } (\text{mkTree } (stl (\text{trim } rs \text{ } s)) \text{ } s')$

proof –

define sl' where $sl' = \text{pickEff } (\text{shd } (\text{trim } rs \text{ } s)) \text{ } s$

obtain s' where $s': s' \in \text{cont } (\text{mkTree } (stl (\text{trim } rs \text{ } s)) \text{ } s')$

using t' unfolding *sl'-def* by *auto*

moreover have $1: \text{enabled } (\text{shd } (\text{trim } rs \text{ } s)) \text{ } s$ using *trim-enabled[OF s rs]*.

moreover with *trim-in-R pickEff eff-S s rs s'[unfolded sl'-def]* have $s' \in S$ by *blast*

ultimately show *?thesis* unfolding *sl'-def* using *pickEff* by *blast*

qed

lemma *ipath-mkTree-sdrop*:

assumes $s: s \in S$ and $rs: \text{fair } rs$ and $i: \text{ipath } (\text{mkTree } rs \text{ } s) \text{ } steps$

shows $\exists n \ s'. \ s' \in S \wedge \text{ipath } (\text{mkTree } (\text{sdrop } n \text{ } rs) \text{ } s') \text{ } (\text{sdrop } m \text{ } steps)$

using $s \ rs \ i$ proof (*induct m arbitrary: steps*)

case (*Suc m*)

then obtain $n \ s'$ where $s': s' \in S$

and $ip: \text{ipath } (\text{mkTree } (\text{sdrop } n \text{ } rs) \text{ } s') \text{ } (\text{sdrop } m \text{ } steps)$ (*is ipath ?t -*) by *blast*

from ip obtain t' where $r: \text{root } ?t = \text{shd } (\text{sdrop } m \text{ } steps)$ and $t': t' \in \text{cont } ?t$

and $i: \text{ipath } t' \text{ } (\text{sdrop } (\text{Suc } m) \text{ } steps)$ by (*cases, simp*)

from *in-cont-mkTree[OF s' sdop-fair[OF Suc.prem(2)] t']* obtain $sl'' \ s''$ where $e: \text{eff } (\text{shd } (\text{trim } (\text{sdrop } n \text{ } rs) \text{ } s')) \text{ } s' \ sl''$ and

s'' : $s'' \in | \in | sl''$ and t' -def: $t' = mkTree (stl (trim (sdrop n rs) s')) s''$ by blast
have $shd (trim (sdrop n rs) s') \in R$ by (metis *sdrop-fair Suc.prem*(2) *trim-in-R s'*)
thus ?case using $i s'' e s'$ **unfolding** *sdrop-stl t'-def sdrop-add add.commute*[of n]
trim-alt[OF s' *sdrop-fair*[OF *Suc.prem*(2)]]
by (intro *exI*[of - *minWait (sdrop n rs) s' + Suc n exI*[of - s'']]) (simp *add: eff-S*)
qed (auto intro!: *exI*[of - 0] *exI*[of - s])

lemma *wf-mkTree*:

assumes $s: s \in S$ and *fair rs*
shows $wf (mkTree rs s)$
using *assms proof* (coinduction arbitrary: *rs s*)
case ($wf rs s$) **let** $?t = mkTree rs s$
have $snd (root ?t) \in R$ using *trim-in-R*[OF wf] by simp
moreover **have** $fst \circ root \circ mkTree (stl (trim rs s)) = id$ by auto
hence *effStep (root ?t) (fimage (fst \circ root) (cont ?t))*
using *trim-enabled*[OF wf] by (simp *add: pickEff*)
ultimately show ?case using *fair-stl*[OF *trim-fair*[OF wf]] *in-cont-mkTree*[OF wf]
by (auto intro!: *exI*[of - *stl (trim rs s)*])
qed

definition *pos rs r* \equiv LEAST n . $shd (sdrop n rs) = r$

lemma *pos*: $\llbracket fair rs; r \in R \rrbracket \implies shd (sdrop (pos rs r) rs) = r$
unfolding *pos-def*
by (rule *LeastI-ex*) (metis (full-types) *alw.cases fair-def holds.simps sdrop-wait*)

lemma *pos-least*: $shd (sdrop n rs) = r \implies pos rs r \leq n$
unfolding *pos-def* by (metis (full-types) *Least-le*)

lemma *minWait-le-pos*: $\llbracket fair rs; r \in R; enabled r s \rrbracket \implies minWait rs s \leq pos rs r$
by (auto simp *del: sdrop-simps intro: minWait-least simp: pos*)

lemma *stake-pos-minWait*:

assumes *rs: fair rs* and $m: minWait rs s < pos rs r$ and $r: r \in R$ and $s: s \in S$
shows $pos (stl (trim rs s)) r = pos rs r - Suc (minWait rs s)$
proof –
have $pos rs r - Suc (minWait rs s) + minWait rs s = pos rs r - Suc 0$ using m by auto
moreover **have** $shd (stl (sdrop (pos rs r - Suc 0) rs)) = shd (sdrop (pos rs r) rs)$
by (metis *Suc-pred gr-implies-not0 m neq0-conv sdrop.simps*(2) *sdrop-stl*)
ultimately have $pos (stl (trim rs s)) r \leq pos rs r - Suc (minWait rs s)$
using *pos*[OF $rs r$] by (auto simp: *add.commute trim-alt*[OF $s rs$] intro: *pos-least*)

moreover
have $pos\ rs\ r \leq pos\ (stl\ (trim\ rs\ s))\ r + Suc\ (minWait\ rs\ s)$
using $pos[OF\ sdrop-fair[OF\ fair-stl[OF\ rs]]\ r, of\ minWait\ rs\ s]$
by $(auto\ simp: trim-alt[OF\ s\ rs]\ add.commute\ intro: pos-least)$
hence $pos\ rs\ r - Suc\ (minWait\ rs\ s) \leq pos\ (stl\ (trim\ rs\ s))\ r$ **by** *linarith*
ultimately show *?thesis* **by** *simp*
qed

lemma *ipath-mkTree-ev*:
assumes $s: s \in S$ **and** $rs: fair\ rs$
and $i: ipath\ (mkTree\ rs\ s)\ steps$ **and** $r: r \in R$
and $alw: alw\ (holds\ (enabledAtStep\ r))\ steps$
shows $ev\ (holds\ (takenAtStep\ r))\ steps$
using $s\ rs\ i\ alw$ **proof** $(induction\ pos\ rs\ r\ arbitrary: rs\ s\ steps\ rule: less-induct)$
case $(less\ rs\ s\ steps)$ **note** $s = \langle s \in S \rangle$ **and** $trim-def' = trim-alt[OF\ s\ \langle fair\ rs \rangle]$
let $?t = mkTree\ rs\ s$
from $less(4,3)\ s\ in-cont-mkTree$ **obtain** $t' :: ('state, 'rule)\ step\ tree$ **and** s'
where
 $rt: root\ ?t = shd\ steps$ **and** $i: ipath\ (mkTree\ (stl\ (trim\ rs\ s))\ s')\ (stl\ steps)$ **and**
 $s': s' \in S$ **by** *cases fast*
show *?case*
proof $(cases\ pos\ rs\ r = minWait\ rs\ s)$
case *True*
with $pos[OF\ less.prem(2)\ r]\ rt[symmetric]$ **show** *?thesis* **by** $(auto\ simp: trim-def'\ ev.base)$
next
case *False*
have $e: enabled\ r\ s$ **using** $less.prem(4)\ rt$ **by** $(subst\ (asm)\ alw-nxt, cases\ steps)\ auto$
with $False\ r\ less.prem(2)$ **have** $2: minWait\ rs\ s < pos\ rs\ r$ **using** $minWait-le-pos$
by *force*
let $?m1 = pos\ rs\ r - Suc\ (minWait\ rs\ s)$
have $Suc\ ?m1 \leq pos\ rs\ r$ **using** 2 **by** *auto*
moreover **have** $?m1 = pos\ (stl\ (trim\ rs\ s))\ r$ **using** $e\ \langle fair\ rs \rangle\ 2\ r\ s$
by $(auto\ intro: stake-pos-minWait[symmetric])$
moreover **have** $fair\ (stl\ (trim\ rs\ s))\ alw\ (holds\ (enabledAtStep\ r))\ (stl\ steps)$
using $less.prem$ **by** $(metis\ fair-stl\ trim-fair, metis\ alw.simps)$
ultimately show *?thesis* **by** $(auto\ intro: ev.step[OF\ less.hyps[OF\ -\ s' - i]])$
qed
qed

4 Persistent rules

definition

$per\ r \equiv$
 $\forall s\ r1\ sl'\ s'. s \in S \wedge enabled\ r\ s \wedge r1 \in R - \{r\} \wedge eff\ r1\ s\ sl' \wedge s' \in S \longrightarrow$
 $enabled\ r\ s'$

lemma *per-alw*:

```

assumes  $p$ : per  $r$  and  $e$ : epath steps  $\wedge$  fst (shd steps)  $\in S$ 
shows alw (holds (enabledAtStep  $r$ ) impl
  (holds (takenAtStep  $r$ ) or next (holds (enabledAtStep  $r$ )))) steps
using  $e$  proof coinduct
  case (alw steps)
  moreover
  { let  $?s = \text{fst}$  (shd steps) let  $?r1 = \text{snd}$  (shd steps)
    let  $?s' = \text{fst}$  (shd (stl steps))
    assume  $?s \in S$  and enabled  $r$   $?s$  and  $?r1 \neq r$ 
    moreover have  $?r1 \in R$  using alw by (auto elim: epath.cases)
    moreover obtain  $sl'$  where eff  $?r1$   $?s$   $sl' \wedge ?s' \in |sl'|$  using alw by (auto
  elim: epath.cases)
    ultimately have enabled  $r$   $?s'$  using  $p$  unfolding per-def by blast
  }
  ultimately show  $?case$  by (auto intro: eff-S elim: epath.cases)
qed

```

end — context *RuleSystem*

```

locale PersistentRuleSystem = RuleSystem eff rules  $S$ 
for eff :: 'rule  $\Rightarrow$  'state  $\Rightarrow$  'state fset  $\Rightarrow$  bool and rules :: 'rule stream and  $S +$ 
assumes per:  $\bigwedge r. r \in R \implies \text{per } r$ 
begin

```

lemma *ipath-mkTree-saturated*:

```

  assumes  $s$ :  $s \in S$  and  $rs$ : fair  $rs$ 
  and  $i$ : ipath (mkTree  $rs$   $s$ ) steps and  $r$ :  $r \in R$ 
  shows saturated  $r$  steps
unfolding saturated-def using  $s$   $rs$   $i$  proof (coinduction arbitrary: rs s steps)
  case (alw rs s steps)
  show  $?case$ 
  proof (intro exI[of - steps], safe)
    assume holds (enabledAtStep  $r$ ) steps
    hence alw (holds (enabledAtStep  $r$ ) steps  $\vee$  ev (holds (takenAtStep  $r$ ) steps)
      by (rule variance[OF - per-alw[OF per[OF r]]])
      (metis wf-ipath-epath wf-mkTree alw mkTree.simps(1) ipath.simps fst-conv)
    thus ev (holds (takenAtStep  $r$ ) steps) by (metis ipath-mkTree-ev[OF alw r])
  next
    from alw show  $\exists rs' s'$  steps'.
      stl steps = steps'  $\wedge s' \in S \wedge$  fair  $rs' \wedge$  ipath (mkTree  $rs'$   $s'$ ) steps'
      using ipath-mkTree-sdrop[where  $m=1$ , simplified] trim-in-R sdrop-fair by
fast
  qed
qed

```

theorem *ipath-mkTree-Saturated*:

```

  assumes  $s \in S$  and fair  $rs$  and ipath (mkTree  $rs$   $s$ ) steps
  shows Saturated steps
  unfolding Saturated-def using ipath-mkTree-saturated[OF assms] by blast

```


theorem *epath-completeness-Saturated*:
assumes $s \in S$
shows
 $(\exists t. \text{fst}(\text{root } t) = s \wedge \text{wf } t \wedge \text{tfinite } t) \vee$
 $(\exists \text{steps}. \text{fst}(\text{shd } \text{steps}) = s \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps})$ (**is** $?A \vee ?B$)
proof –
{ **assume** $\neg ?A$
with *assms* **have** $\neg \text{tfinite}(\text{mkTree } \text{fenum } s)$ **using** *wf-mkTree fair-fenum* **by**
auto
then obtain *steps* **where** $\text{ipath}(\text{mkTree } \text{fenum } s) \text{steps}$ **using** *Konig* **by** *blast*
with *assms* **have** $\text{fst}(\text{shd } \text{steps}) = s \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps}$
by (*metis wf-ipath-epath ipath.simps ipath-mkTree-Saturated*
wf-mkTree fair-fenum mkTree.simps(1) fst-conv)
hence $?B$ **by** *blast*
}
thus *?thesis* **by** *blast*
qed

end — context *PersistentRuleSystem*

5 Code generation

locale *RuleSystem-Code* =
fixes $\text{eff}' :: 'rule \Rightarrow 'state \Rightarrow 'state \text{fset option}$
and $\text{rules} :: 'rule \text{stream}$ — countably many rules
begin

definition $\text{eff } r \ s \ \text{sl} \equiv \text{eff}' \ r \ s = \text{Some } \text{sl}$

end

definition [*code del*]: $\text{effG } \text{eff}' \ r \ s \ \text{sl} \equiv \text{RuleSystem-Code.eff } \text{eff}' \ r \ s \ \text{sl}$

sublocale *RuleSystem-Code* < *RuleSystem-Defs*
where $\text{eff} = \text{effG } \text{eff}'$ **and** $\text{rules} = \text{rules}$.

context *RuleSystem-Code*
begin

lemma *enabled-eff'*: $\text{enabled } r \ s \longleftrightarrow \text{eff}' \ r \ s \neq \text{None}$
unfolding *enabled-def effG-def eff-def* **by** *auto*

lemma *pickEff-the[code]*: $\text{pickEff } r \ s = \text{the}(\text{eff}' \ r \ s)$
unfolding *pickEff-def enabled-def effG-def eff-def* **by** *auto*

lemmas [*code-unfold*] = *trim-def enabled-eff' pickEff-the*

```

setup Locale-Code.open-block
interpretation i: RuleSystem-Code eff' rules for eff' and rules .
declare [[lc-delete RuleSystem-Defs.mkTree (effG ?eff')]]
declare [[lc-delete RuleSystem-Defs.trim]]
declare [[lc-delete RuleSystem-Defs.enabled]]
declare [[lc-delete RuleSystem-Defs.pickEff]]
declare [[lc-add RuleSystem-Defs.mkTree (effG ?eff') i.mkTree-unfold]]
setup Locale-Code.close-block

```

code-printing

```

constant the  $\rightarrow$  (Haskell) fromJust
| constant Option.is-none  $\rightarrow$  (Haskell) isNothing

```

```

export-code mkTree-effG-uu in Haskell module-name Tree

```

6 Toy instantiation: Propositional Logic

```

datatype fmla = Atom nat | Neg fmla | Conj fmla fmla

```

primrec *max-depth* where

```

max-depth (Atom -) = 0
| max-depth (Neg  $\varphi$ ) = Suc (max-depth  $\varphi$ )
| max-depth (Conj  $\varphi$   $\psi$ ) = Suc (max (max-depth  $\varphi$ ) (max-depth  $\psi$ ))

```

lemma *max-depth-0*: *max-depth* φ = 0 = ($\exists n. \varphi = \text{Atom } n$)

by (*cases* φ) *auto*

lemma *max-depth-Suc*: *max-depth* φ = *Suc* n = ($(\exists \psi. \varphi = \text{Neg } \psi \wedge \text{max-depth } \psi = n) \vee$

$(\exists \psi1 \psi2. \varphi = \text{Conj } \psi1 \psi2 \wedge \text{max} (\text{max-depth } \psi1) (\text{max-depth } \psi2) = n)$)

by (*cases* φ) *auto*

abbreviation *atoms* \equiv *smap Atom nats*

abbreviation *depth1* \equiv

sinterleave (smap Neg atoms) (smap (case-prod Conj) (sproduct atoms atoms))

abbreviation *sinterleaves* \equiv *fold sinterleave*

fun *extendLevel* where *extendLevel* (*belowN*, N) =

```

(let Next = sinterleaves
  (map (smap (case-prod Conj)) [sproduct belowN  $N$ , sproduct  $N$  belowN, sproduct
   $N$   $N$ ])
  (smap Neg  $N$ )
  in (sinterleave belowN  $N$ , Next))

```

lemma *extendLevel-step*:

```

[[sset belowN = { $\varphi. \text{max-depth } \varphi < n$ };
  sset  $N$  = { $\varphi. \text{max-depth } \varphi = n$ }; st = (belowN,  $N$ )]]  $\implies$ 

```

$\exists \text{ belowNext Next. extendLevel st} = (\text{belowNext}, \text{Next}) \wedge$
 $\text{sset belowNext} = \{\varphi. \text{max-depth } \varphi < \text{Suc } n\} \wedge \text{sset Next} = \{\varphi. \text{max-depth } \varphi$
 $= \text{Suc } n\}$
by (*auto simp: sset-sinterleave sset-sproduct stream.set-map*
image-iff max-depth-Suc)

lemma *sset-atoms*: $\text{sset atoms} = \{\varphi. \text{max-depth } \varphi < 1\}$
by (*auto simp: stream.set-map max-depth-0*)

lemma *sset-depth1*: $\text{sset depth1} = \{\varphi. \text{max-depth } \varphi = 1\}$
by (*auto simp: sset-sinterleave sset-sproduct stream.set-map*
max-depth-Suc max-depth-0 max-def image-iff)

lemma *extendLevel-Nsteps*:
 $\llbracket \text{sset belowN} = \{\varphi. \text{max-depth } \varphi < n\}; \text{sset } N = \{\varphi. \text{max-depth } \varphi = n\} \rrbracket \implies$
 $\exists \text{ belowNext Next. (extendLevel } \hat{\hat{m}}) (\text{belowN}, N) = (\text{belowNext}, \text{Next}) \wedge$
 $\text{sset belowNext} = \{\varphi. \text{max-depth } \varphi < n + m\} \wedge \text{sset Next} = \{\varphi. \text{max-depth } \varphi =$
 $n + m\}$
proof (*induction m arbitrary: belowN N n*)
case (*Suc m*)
then obtain *belowNext Next where* $(\text{extendLevel } \hat{\hat{m}}) (\text{belowN}, N) = (\text{belowNext},$
 $\text{Next})$
 $\text{sset belowNext} = \{\varphi. \text{max-depth } \varphi < n + m\} \text{sset Next} = \{\varphi. \text{max-depth } \varphi =$
 $n + m\}$
by *blast*
thus *?case unfolding funpow.simps o-apply add-Suc-right*
by (*intro extendLevel-step[of belowNext - Next]*)
qed *simp*

corollary *extendLevel*:
 $\exists \text{ belowNext Next. (extendLevel } \hat{\hat{m}}) (\text{atoms}, \text{depth1}) = (\text{belowNext}, \text{Next}) \wedge$
 $\text{sset belowNext} = \{\varphi. \text{max-depth } \varphi < 1 + m\} \wedge \text{sset Next} = \{\varphi. \text{max-depth } \varphi =$
 $1 + m\}$
by (*rule extendLevel-Nsteps (auto simp: sset-atoms sset-depth1)*)

definition *fmlas* = *sinterleave atoms (smerge (smap snd (siterate extendLevel*
(atoms, depth1)))))

lemma *fmlas-UNIV*: $\text{sset fmlas} = (\text{UNIV} :: \text{fmla set})$
proof (*intro equalityI subsetI UNIV-I*)
fix φ
show $\varphi \in \text{sset fmlas}$
proof (*cases max-depth } \varphi*)
case 0 **thus** *?thesis unfolding fmlas-def sset-sinterleave stream.set-map*
by (*intro UnI1 (auto simp: max-depth-0)*)
next
case (*Suc m*) **thus** *?thesis using extendLevel[of m]*
unfolding fmlas-def sset-smerge sset-siterate sset-sinterleave stream.set-map

by (intro UnI2) (auto, metis (mono-tags) mem-Collect-eq)
qed
qed

datatype rule = Idle | Ax nat | NegL fmla | NegR fmla | ConjL fmla fmla | ConjR fmla fmla

abbreviation mkRules f \equiv smap f fmlas

abbreviation mkRulePairs f \equiv smap (case-prod f) (sproduct fmlas fmlas)

definition rules where

rules = Idle ##
sinterleaves [mkRules NegL, mkRules NegR, mkRulePairs ConjL, mkRulePairs ConjR]
(smap Ax nats)

lemma rules-UNIV: sset rules = (UNIV :: rule set)

unfolding rules-def by (auto simp: sset-sinterleave sset-sproduct stream.set-map fmlas-UNIV image-iff) (metis rule.exhaust)

type-synonym state = fmla fset * fmla fset

fun eff' :: rule \Rightarrow state \Rightarrow state fset option where

eff' Idle (Γ , Δ) = Some {(| Γ , Δ)|}
| eff' (Ax n) (Γ , Δ) =
(if Atom n \in | Γ \wedge Atom n \in | Δ then Some {||} else None)
| eff' (NegL φ) (Γ , Δ) =
(if Neg φ \in | Γ then Some {(| Γ |-| { | Neg φ |}, finsert φ Δ)|} else None)
| eff' (NegR φ) (Γ , Δ) =
(if Neg φ \in | Δ then Some {(|finsert φ Γ , Δ |-| { | Neg φ |})|} else None)
| eff' (ConjL φ ψ) (Γ , Δ) =
(if Conj φ ψ \in | Γ
then Some {(|finsert φ (finsert ψ (Γ |-| { | Conj φ ψ |})), Δ)|}
else None)
| eff' (ConjR φ ψ) (Γ , Δ) =
(if Conj φ ψ \in | Δ
then Some {(| Γ , finsert φ (Δ |-| { | Conj φ ψ |})), (Γ , finsert ψ (Δ |-| { | Conj
 φ ψ |})))|}
else None)

abbreviation Disj φ ψ \equiv Neg (Conj (Neg φ) (Neg ψ))

abbreviation Imp φ ψ \equiv Disj (Neg φ) ψ

abbreviation Iff φ ψ \equiv Conj (Imp φ ψ) (Imp ψ φ)

definition thm1 \equiv ({|Conj (Atom 0) (Neg (Atom 0))|}, {||})

declare Stream.smember-code [code del]

lemma [code]: Stream.smember x (y ## s) = (x = y \vee Stream.smember x s)

unfolding *Stream.smember-def* **by** *auto*

interpretation *RuleSystem* $\lambda r s ss. \text{eff}' r s = \text{Some } ss \text{ rules UNIV}$
by *unfold-locales* (*auto simp: rules-UNIV intro: exI[of - Idle]*)

interpretation *PersistentRuleSystem* $\lambda r s ss. \text{eff}' r s = \text{Some } ss \text{ rules UNIV}$

proof (*unfold-locales, unfold enabled-def per-def rules-UNIV, clarsimp*)

fix $r \Gamma \Delta ss r' \Gamma' \Delta' ss'$

assume $r' \neq r \text{eff}' r (\Gamma, \Delta) = \text{Some } ss \text{eff}' r' (\Gamma, \Delta) = \text{Some } ss' (\Gamma', \Delta') \mid \in \mid$
 ss'

then show $\exists sl. \text{eff}' r (\Gamma', \Delta') = \text{Some } sl$

by (*cases r r' rule: rule.exhaust[case-product rule.exhaust]*) (*auto split: if-splits*)

qed

definition $\text{rho} \equiv i.\text{fenum rules}$

definition $\text{propTree} \equiv i.\text{mkTree eff}' \text{rho}$

export-code *propTree thm1* **in** *Haskell* **module-name** *PropInstance*