



Paraconsistency

Schlichtkrull, Anders; Villadsen, Jørgen

Published in:
Archive of Formal Proofs

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

[Link back to DTU Orbit](#)

Citation (APA):
Schlichtkrull, A., & Villadsen, J. (2016). Paraconsistency. *Archive of Formal Proofs*, 1-27. <https://www.isa-afp.org/entries/Paraconsistency.shtml>

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Paraconsistency

Anders Schlichtkrull & Jørgen Villadsen, DTU Compute, Denmark

8 December 2016

Abstract

Paraconsistency is about handling inconsistency in a coherent way. In classical and intuitionistic logic everything follows from an inconsistent theory. A paraconsistent logic avoids the explosion. Quite a few applications in computer science and engineering are discussed in the Intelligent Systems Reference Library Volume 110: Towards Paraconsistent Engineering (Springer 2016). We formalize a paraconsistent many-valued logic that we motivated and described in a special issue on logical approaches to paraconsistency (Journal of Applied Non-Classical Logics 2005). We limit ourselves to the propositional fragment of the higher-order logic. The logic is based on so-called key equalities and has a countably infinite number of truth values. We prove theorems in the logic using the definition of validity. We verify truth tables and also counterexamples for non-theorems. We prove meta-theorems about the logic and finally we investigate a case study.

Contents

Preface	1
On Paraconsistency	2
Syntax and Semantics	2
Truth Tables	4
Basic Theorems	7
Further Non-Theorems	9
Further Meta-Theorems	15
Case Study	23
Acknowledgements	26
References	27

Preface

The present formalization in Isabelle essentially follows our extended abstract [1]. The Stanford Encyclopedia of Philosophy has a comprehensive overview of logical approaches to paraconsistency [2]. We have elsewhere explained the rationale for our paraconsistent many-valued logic and considered applications in multi-agent systems and natural language semantics [3, 4, 5, 6].

It is a revised and extended version of our formalization <https://github.com/logic-tools/mvl> that accompany our chapter in a book on partiality published by Cambridge Scholars Press. The GitHub link provides more information. We are grateful to the editors — Henning Christiansen, M. Dolores Jiménez López, Roussanka Loukanova and Larry Moss — for the opportunity to contribute to the book.

On Paraconsistency

Paraconsistency concerns inference systems that do not explode given a contradiction.

The Internet Encyclopedia of Philosophy has a survey article on paraconsistent logic.

The following Isabelle theory formalizes a specific paraconsistent many-valued logic.

```
theory Paraconsistency imports Main begin
```

The details about our logic are in our article in a special issue on logical approaches to paraconsistency in the Journal of Applied Non-Classical Logics (Volume 15, Number 1, 2005).

Syntax and Semantics

Syntax of Propositional Logic

Only the primed operators return indeterminate truth values.

```
type_synonym id = string
```

```
datatype fm = Pro id | Truth | Neg' fm | Con' fm fm | Eql fm fm | Eql' fm fm
```

```
abbreviation Falsity :: fm where Falsity  $\equiv$  Neg' Truth
```

```
abbreviation Dis' :: fm  $\Rightarrow$  fm  $\Rightarrow$  fm where Dis' p q  $\equiv$  Neg' (Con' (Neg' p) (Neg' q))
```

```
abbreviation Imp :: fm  $\Rightarrow$  fm  $\Rightarrow$  fm where Imp p q  $\equiv$  Eql p (Con' p q)
```

```
abbreviation Imp' :: fm  $\Rightarrow$  fm  $\Rightarrow$  fm where Imp' p q  $\equiv$  Eql' p (Con' p q)
```

```
abbreviation Box :: fm  $\Rightarrow$  fm where Box p  $\equiv$  Eql p Truth
```

```
abbreviation Neg :: fm  $\Rightarrow$  fm where Neg p  $\equiv$  Box (Neg' p)
```

```
abbreviation Con :: fm  $\Rightarrow$  fm  $\Rightarrow$  fm where Con p q  $\equiv$  Box (Con' p q)
```

```
abbreviation Dis :: fm  $\Rightarrow$  fm  $\Rightarrow$  fm where Dis p q  $\equiv$  Box (Dis' p q)
```

```
abbreviation Cla :: fm  $\Rightarrow$  fm where Cla p  $\equiv$  Dis (Box p) (Eql p Falsity)
```

```
abbreviation Nab :: fm  $\Rightarrow$  fm where Nab p  $\equiv$  Neg (Cla p)
```

Semantics of Propositional Logic

There is a countably infinite number of indeterminate truth values.

```
datatype tv = Det bool | Indet nat
```

```
abbreviation (input) eval_neg :: tv  $\Rightarrow$  tv
```

```
where
```

```
  eval_neg x  $\equiv$ 
```

```
  (
```

```
    case x of
```

```
      Det False  $\Rightarrow$  Det True |
```

```
      Det True  $\Rightarrow$  Det False |
```

```
      Indet n  $\Rightarrow$  Indet n
```

```

)

fun eval :: (id  $\Rightarrow$  tv)  $\Rightarrow$  fm  $\Rightarrow$  tv
where
  eval i (Pro s) = i s |
  eval i Truth = Det True |
  eval i (Neg' p) = eval_neg (eval i p) |
  eval i (Con' p q) =
    (
      if eval i p = eval i q then eval i p else
      if eval i p = Det True then eval i q else
      if eval i q = Det True then eval i p else Det False
    ) |
  eval i (Eq1 p q) =
    (
      if eval i p = eval i q then Det True else Det False
    ) |
  eval i (Eq1' p q) =
    (
      if eval i p = eval i q then Det True else
      (
        case (eval i p, eval i q) of
          (Det True, _)  $\Rightarrow$  eval i q |
          (_, Det True)  $\Rightarrow$  eval i p |
          (Det False, _)  $\Rightarrow$  eval_neg (eval i q) |
          (_, Det False)  $\Rightarrow$  eval_neg (eval i p) |
          _  $\Rightarrow$  Det False
        )
      )
    )

lemma eval_equality_simplify: eval i (Eq1 p q) = Det (eval i p = eval i q)
by simp

theorem eval_equality:
  eval i (Eq1' p q) =
    (
      if eval i p = eval i q then Det True else
      if eval i p = Det True then eval i q else
      if eval i q = Det True then eval i p else
      if eval i p = Det False then eval i (Neg' q) else
      if eval i q = Det False then eval i (Neg' p) else
      Det False
    )
by (cases eval i p; cases eval i q) simp_all

theorem eval_negation:
  eval i (Neg' p) =
    (
      if eval i p = Det False then Det True else
      if eval i p = Det True then Det False else
      eval i p
    )
by (cases eval i p) simp_all

corollary eval i (Cla p) = eval i (Box (Dis' p (Neg' p)))
using eval_negation
by simp

lemma double_negation: eval i p = eval i (Neg' (Neg' p))
using eval_negation

```

by simp

Validity and Consistency

Validity gives the set of theorems and the logic has at least a theorem and a non-theorem.

definition valid :: fm \Rightarrow bool

where

valid p $\equiv \forall i. \text{eval } i \text{ p} = \text{Det True}$

proposition valid Truth and \neg valid Falsity

unfolding valid_def

by simp_all

Truth Tables

String Functions

The following functions support arbitrary unary and binary truth tables.

definition tv_pair_row :: tv list \Rightarrow tv \Rightarrow (tv * tv) list

where

tv_pair_row tvs tv = map ($\lambda x. (tv, x)$) tvs

definition tv_pair_table :: tv list \Rightarrow (tv * tv) list list

where

tv_pair_table tvs \equiv map (tv_pair_row tvs) tvs

definition map_row :: (tv \Rightarrow tv \Rightarrow tv) \Rightarrow (tv * tv) list \Rightarrow tv list

where

map_row f tvsvs = map ($\lambda(x, y). f \ x \ y$) tvsvs

definition map_table :: (tv \Rightarrow tv \Rightarrow tv) \Rightarrow (tv * tv) list list \Rightarrow tv list list

where

map_table f tvsvss = map (map_row f) tvsvss

definition unary_truth_table :: fm \Rightarrow tv list \Rightarrow tv list

where

unary_truth_table p tvs =
map ($\lambda x. \text{eval } ((\lambda s. \text{undefined})(''p'' := x)) \text{ p}$) tvs

definition binary_truth_table :: fm \Rightarrow tv list \Rightarrow tv list list

where

binary_truth_table p tvs =
map_table ($\lambda x \ y. \text{eval } ((\lambda s. \text{undefined})(''p'' := x, ''q'' := y)) \text{ p}$) (tv_pair_table tvs)

fun string_of_nat :: nat \Rightarrow string

where

string_of_nat n = (if n < 10 then [char_of_nat (48 + n)] else
string_of_nat (n div 10) @ [char_of_nat (48 + (n mod 10))])

fun string_tv :: tv \Rightarrow string

where

string_tv (Det True) = ''*' |
string_tv (Det False) = ''o'' |
string_tv (Indet n) = string_of_nat n

definition appends :: string list \Rightarrow string

```

where
  appends strs = foldr append strs []

definition appends_nl :: string list ⇒ string
where
  appends_nl strs = '' $\leftrightarrow$ '' @
    foldr ( $\lambda$ s s'. s @ '' $\leftrightarrow$ '' @ s') (butlast strs) (last strs) @ '' $\leftrightarrow$ ''

definition string_table :: tv list list ⇒ string list list
where
  string_table tvss = map (map string_tv) tvss

definition string_table_string :: string list list ⇒ string
where
  string_table_string strss = appends_nl (map appends strss)

definition unary :: fm ⇒ tv list ⇒ string
where
  unary p tvs = appends_nl (map string_tv (unary_truth_table p tvs))

definition binary :: fm ⇒ tv list ⇒ string
where
  binary p tvs = string_table_string (string_table (binary_truth_table p tvs))

```

Main Truth Tables

The omitted Cla (for Classic) is discussed later; Nab (for Nabla) is simply the negation of it.

proposition

```

unary (Box (Pro ''p'')) [Det True, Det False, Indet 1] = ''
*
o
o
'',
  by code_simp

```

proposition

```

binary (Con' (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
*o12
oooo
1o1o
2oo2
'',
  by code_simp

```

proposition

```

binary (Dis' (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
****
*o12
*11*
*2*2
'',
  by code_simp

```

proposition

```

unary (Neg' (Pro ''p'')) [Det True, Det False, Indet 1] = ''
o
*
1
'',

```

by code_simp

proposition

```
binary (Eq1' (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
*o12
o*12
11*o
22o*
''
```

by code_simp

proposition

```
binary (Imp' (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
*o12
****
*1*1
*22*
''
```

by code_simp

proposition

```
unary (Neg (Pro ''p'')) [Det True, Det False, Indet 1] = ''
o
*
o
''
```

by code_simp

proposition

```
binary (Eq1 (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
*ooo
o*oo
oo*o
ooo*
''
```

by code_simp

proposition

```
binary (Imp (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
*ooo
****
*o*o
*oo*
''
```

by code_simp

proposition

```
unary (Nab (Pro ''p'')) [Det True, Det False, Indet 1] = ''
o
o
*
''
```

by code_simp

proposition

```
binary (Con (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
*ooo
oooo
oooo
oooo
```

```

'',
  by code_simp

proposition
  binary (Dis (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
  ****
  *ooo
  *oo*
  *o*o
'',
  by code_simp

```

Basic Theorems

Selected Theorems and Non-Theorems

Many of the following theorems and non-theorems use assumptions and meta-variables.

```

proposition valid (Cla (Box p)) and ¬ valid (Nab (Box p))
  unfolding valid_def
  by simp_all

```

```

proposition valid (Cla (Cla p)) and ¬ valid (Nab (Nab p))
  unfolding valid_def
  by simp_all

```

```

proposition valid (Cla (Nab p)) and ¬ valid (Nab (Cla p))
  unfolding valid_def
  by simp_all

```

```

proposition valid (Box p) ↔ valid (Box (Box p))
  unfolding valid_def
  by simp

```

```

proposition valid (Neg p) ↔ valid (Neg' p)
  unfolding valid_def
  by simp

```

```

proposition valid (Con p q) ↔ valid (Con' p q)
  unfolding valid_def
  by simp

```

```

proposition valid (Dis p q) ↔ valid (Dis' p q)
  unfolding valid_def
  by simp

```

```

proposition valid (Eq1 p q) ↔ valid (Eq1' p q)
  unfolding valid_def
  using eval.simps tv.inject eval_equality eval_negation
  by (metis (full_types))

```

```

proposition valid (Imp p q) ↔ valid (Imp' p q)
  unfolding valid_def
  using eval.simps tv.inject eval_equality eval_negation
  by (metis (full_types))

```

```

proposition ¬ valid (Pro ''p'')
  unfolding valid_def
  by auto

```



```

proposition  $\neg$  valid (Neg' (Pro ''p''))
proof -
  have eval ( $\lambda$ s. Det True) (Neg' (Pro ''p'')) = Det False
    by simp
  then show ?thesis
    unfolding valid_def
    using tv.inject
    by metis
qed

```

```

proposition assumes valid p shows  $\neg$  valid (Neg' p)
  using assms
  unfolding valid_def
  by simp

```

```

proposition assumes valid (Neg' p) shows  $\neg$  valid p
  using assms
  unfolding valid_def
  by force

```

```

proposition valid (Neg' (Neg' p))  $\longleftrightarrow$  valid p
  unfolding valid_def
  using double_negation
  by simp

```

```

theorem conjunction: valid (Con' p q)  $\longleftrightarrow$  valid p  $\wedge$  valid q
  unfolding valid_def
  by auto

```

```

corollary assumes valid (Con' p q) shows valid p and valid q
  using assms conjunction
  by simp_all

```

```

proposition assumes valid p and valid (Imp p q) shows valid q
  using assms eval.simps tv.inject
  unfolding valid_def
  by (metis (full_types))

```

```

proposition assumes valid p and valid (Imp' p q) shows valid q
  using assms eval.simps tv.inject eval_equality
  unfolding valid_def
  by (metis (full_types))

```

Key Equalities

The key equalities are part of the motivation for the semantic clauses.

```

proposition valid (Eq1 p (Neg' (Neg' p)))
  unfolding valid_def
  using double_negation
  by simp

```

```

proposition valid (Eq1 Truth (Neg' Falsity))
  unfolding valid_def
  by simp

```

```

proposition valid (Eq1 Falsity (Neg' Truth))
  unfolding valid_def
  by simp

```

```

proposition valid (Eq1 p (Con' p p))
  unfolding valid_def
  by simp

proposition valid (Eq1 p (Con' Truth p))
  unfolding valid_def
  by simp

proposition valid (Eq1 p (Con' p Truth))
  unfolding valid_def
  by simp

proposition valid (Eq1 Truth (Eq1' p p))
  unfolding valid_def
  by simp

proposition valid (Eq1 p (Eq1' Truth p))
  unfolding valid_def
  by simp

proposition valid (Eq1 p (Eq1' p Truth))
  unfolding valid_def
proof
  fix i
  show eval i (Eq1 p (Eq1' p Truth)) = Det True
    by (cases eval i p) simp_all
qed

proposition valid (Eq1 (Neg' p) (Eq1' Falsity p))
  unfolding valid_def
proof
  fix i
  show eval i (Eq1 (Neg' p) (Eq1' (Neg' Truth) p)) = Det True
    by (cases eval i p) simp_all
qed

proposition valid (Eq1 (Neg' p) (Eq1' p Falsity))
  unfolding valid_def
  using eval.simps eval_equality eval_negation
  by metis

```

Further Non-Theorems

Smaller Domains and Paraconsistency

Validity is relativized to a set of indeterminate truth values (called a domain).

```

definition domain :: nat set  $\Rightarrow$  tv set
where
  domain U  $\equiv$  {Det True, Det False}  $\cup$  Indet ' U

theorem universal_domain: domain {n. True} = {x. True}
proof -
  have  $\forall x. x = \text{Det True} \vee x = \text{Det False} \vee x \in \text{range Indet}$ 
    using range_eqI tv.exhaust tv.inject
    by metis
  then show ?thesis
    unfolding domain_def

```

```

    by blast
qed

definition valid_in :: nat set  $\Rightarrow$  fm  $\Rightarrow$  bool
where
  valid_in U p  $\equiv \forall i. \text{range } i \subseteq \text{domain } U \longrightarrow \text{eval } i \text{ p} = \text{Det True}$ 

abbreviation valid_boole :: fm  $\Rightarrow$  bool where valid_boole p  $\equiv$  valid_in {} p

proposition valid p  $\longleftrightarrow$  valid_in {n. True} p
  unfolding valid_def valid_in_def
  using universal_domain
  by simp

theorem valid_valid_in: assumes valid p shows valid_in U p
  using assms
  unfolding valid_in_def valid_def
  by simp

theorem transfer: assumes  $\neg$  valid_in U p shows  $\neg$  valid p
  using assms valid_valid_in
  by blast

proposition valid_in U (Neg' (Neg' p))  $\longleftrightarrow$  valid_in U p
  unfolding valid_in_def
  using double_negation
  by simp

theorem conjunction_in: valid_in U (Con' p q)  $\longleftrightarrow$  valid_in U p  $\wedge$  valid_in U q
  unfolding valid_in_def
  by auto

corollary assumes valid_in U (Con' p q) shows valid_in U p and valid_in U q
  using assms conjunction_in
  by simp_all

proposition assumes valid_in U p and valid_in U (Imp p q) shows valid_in U q
  using assms eval.simps tv.inject
  unfolding valid_in_def
  by (metis (full_types))

proposition assumes valid_in U p and valid_in U (Imp' p q) shows valid_in U q
  using assms eval.simps tv.inject eval_equality
  unfolding valid_in_def
  by (metis (full_types))

abbreviation (input) Explosion :: fm  $\Rightarrow$  fm  $\Rightarrow$  fm
where
  Explosion p q  $\equiv$  Imp' (Con' p (Neg' p)) q

proposition valid_boole (Explosion (Pro ''p'') (Pro ''q''))
  unfolding valid_in_def
proof (rule; rule)
  fix i :: id  $\Rightarrow$  tv
  assume range i  $\subseteq$  domain {}
  then have
    i ''p''  $\in$  {Det True, Det False}
    i ''q''  $\in$  {Det True, Det False}
  unfolding domain_def
  by auto

```

```

then show eval i (Explosion (Pro ''p'') (Pro ''q'')) = Det True
  by (cases i ''p''; cases i ''q'') simp_all
qed

```

```

lemma explosion_counterexample: ¬ valid_in {1} (Explosion (Pro ''p'') (Pro ''q''))
proof -
  let ?i = (λs. Indet 1)(''q'' := Det False)
  have range ?i ⊆ domain {1}
    unfolding domain_def
    by (simp add: image_subset_iff)
  moreover have eval ?i (Explosion (Pro ''p'') (Pro ''q'')) = Indet 1
    by simp
  moreover have Indet 1 ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid_in_def
    by metis
qed

```

```

theorem explosion_not_valid: ¬ valid (Explosion (Pro ''p'') (Pro ''q''))
  using explosion_counterexample transfer
  by simp

```

```

proposition ¬ valid (Imp (Con' (Pro ''p'') (Neg' (Pro ''p'')))) (Pro ''q'')
  using explosion_counterexample transfer eval.simps tv.simps
  unfolding valid_in_def
  — by smt OK

```

```

proof -
  assume *: ¬ (∀i. range i ⊆ domain U → eval i p = Det True) ⇒ ¬ valid p for U p
  assume ¬ (∀i. range i ⊆ domain {1} →
    eval i (Explosion (Pro ''p'') (Pro ''q'')) = Det True)
  then obtain i where
    **: range i ⊆ domain {1} ∧
      eval i (Explosion (Pro ''p'') (Pro ''q'')) ≠ Det True
  by blast
  then have eval i (Con' (Pro ''p'') (Neg' (Pro ''p''))) ≠
    eval i (Con' (Con' (Pro ''p'') (Neg' (Pro ''p'')))) (Pro ''q'')
  by force
  then show ?thesis
    using * **
    by force
qed

```

Example: Contraposition

Contraposition is not valid.

```

abbreviation (input) Contraposition :: fm ⇒ fm ⇒ fm
where

```

```

  Contraposition p q ≡ Eql' (Imp' p q) (Imp' (Neg' q) (Neg' p))

```

```

proposition valid_boole (Contraposition (Pro ''p'') (Pro ''q''))
  unfolding valid_in_def

```

```

proof (rule; rule)

```

```

  fix i :: id ⇒ tv

```

```

  assume range i ⊆ domain {}

```

```

  then have

```

```

    i ''p'' ∈ {Det True, Det False}

```

```

    i ''q'' ∈ {Det True, Det False}

```

```

  unfolding domain_def

```

```

    by auto
  then show eval i (Contraposition (Pro ''p'') (Pro ''q'')) = Det True
    by (cases i ''p''; cases i ''q'') simp_all
qed

```

```

proposition valid_in {1} (Contraposition (Pro ''p'') (Pro ''q''))
  unfolding valid_in_def
proof (rule; rule)
  fix i :: id  $\Rightarrow$  tv
  assume range i  $\subseteq$  domain {1}
  then have
    i ''p''  $\in$  {Det True, Det False, Indet 1}
    i ''q''  $\in$  {Det True, Det False, Indet 1}
    unfolding domain_def
    by auto
  then show eval i (Contraposition (Pro ''p'') (Pro ''q'')) = Det True
    by (cases i ''p''; cases i ''q'') simp_all
qed

```

```

lemma contraposition_counterexample:  $\neg$  valid_in {1, 2} (Contraposition (Pro ''p'') (Pro ''q''))
proof -
  let ?i = ( $\lambda$ s. Indet 1)(''q'' := Indet 2)
  have range ?i  $\subseteq$  domain {1, 2}
    unfolding domain_def
    by (simp add: image_subset_iff)
  moreover have eval ?i (Contraposition (Pro ''p'') (Pro ''q'')) = Det False
    by simp
  moreover have Det False  $\neq$  Det True
    by simp
  ultimately show ?thesis
    unfolding valid_in_def
    by metis
qed

```

```

theorem contraposition_not_valid:  $\neg$  valid (Contraposition (Pro ''p'') (Pro ''q''))
  using contraposition_counterexample transfer
  by simp

```

More Than Four Truth Values Needed

Cla3 is valid for two indeterminate truth values but not for three indeterminate truth values.

```

lemma ranges: assumes range i  $\subseteq$  domain U shows eval i p  $\in$  domain U
  using assms
  unfolding domain_def
  by (induct p) auto

```

```

proposition
  unary (Cla (Pro ''p'')) [Det True, Det False, Indet 1] = ''
  *
  *
  o
  ''
  by code_simp

```

```

proposition valid_boole (Cla p)
  unfolding valid_in_def
proof (rule; rule)
  fix i :: id  $\Rightarrow$  tv
  assume range i  $\subseteq$  domain {}

```

```

then have
  eval i p ∈ {Det True, Det False}
  using ranges[of i {}]
  unfolding domain_def
  by auto
then show eval i (Cla p) = Det True
  by (cases eval i p) simp_all
qed

```

proposition \neg valid_in {1} (Cla (Pro ''p''))

```

proof -
  let ?i =  $\lambda$ s. Indet 1
  have range ?i  $\subseteq$  domain {1}
    unfolding domain_def
    by (simp add: image_subset_iff)
  moreover have eval ?i (Cla (Pro ''p'')) = Det False
    by simp
  moreover have Det False  $\neq$  Det True
    by simp
  ultimately show ?thesis
    unfolding valid_in_def
    by metis

```

qed

abbreviation (input) Cla2 :: fm \Rightarrow fm \Rightarrow fm

where

Cla2 p q \equiv Dis (Dis (Cla p) (Cla q)) (Eq1 p q)

proposition

```

binary (Cla2 (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
****
****
***o
**o*
,,
  by code_simp

```

proposition valid_boole (Cla2 p q)

unfolding valid_in_def

proof (rule; rule)

fix i :: id \Rightarrow tv

assume range: range i \subseteq domain {}

then have

eval i p ∈ {Det True, Det False}

eval i q ∈ {Det True, Det False}

using ranges[of i {}]

unfolding domain_def

by auto

then show eval i (Cla2 p q) = Det True

by (cases eval i p; cases eval i q) simp_all

qed

proposition valid_in {1} (Cla2 p q)

unfolding valid_in_def

proof (rule; rule)

fix i :: id \Rightarrow tv

assume range: range i \subseteq domain {1}

then have

eval i p ∈ {Det True, Det False, Indet 1}

eval i q ∈ {Det True, Det False, Indet 1}

```

    using ranges[of i {1}]
    unfolding domain_def
    by auto
  then show eval i (Cla2 p q) = Det True
    by (cases eval i p; cases eval i q) simp_all
qed

```

proposition \neg valid_in {1, 2} (Cla2 (Pro ''p'') (Pro ''q''))

```

proof -
  let ?i = ( $\lambda$ s. Indet 1)(''q'' := Indet 2)
  have range ?i  $\subseteq$  domain {1, 2}
    unfolding domain_def
    by (simp add: image_subset_iff)
  moreover have eval ?i (Cla2 (Pro ''p'') (Pro ''q'')) = Det False
    by simp
  moreover have Det False  $\neq$  Det True
    by simp
  ultimately show ?thesis
    unfolding valid_in_def
    by metis
qed

```

abbreviation (input) Cla3 :: fm \Rightarrow fm \Rightarrow fm \Rightarrow fm

where

Cla3 p q r \equiv Dis (Dis (Cla p) (Dis (Cla q) (Cla r))) (Dis (Eq1 p q) (Dis (Eq1 p r) (Eq1 q r)))

proposition valid_boole (Cla3 p q r)

unfolding valid_in_def

proof (rule; rule)

fix i :: id \Rightarrow tv

assume range i \subseteq domain {}

then have

eval i p \in {Det True, Det False}

eval i q \in {Det True, Det False}

eval i r \in {Det True, Det False}

using ranges[of i {}]

unfolding domain_def

by auto

then show eval i (Cla3 p q r) = Det True

by (cases eval i p; cases eval i q; cases eval i r) simp_all

qed

proposition valid_in {1} (Cla3 p q r)

unfolding valid_in_def

proof (rule; rule)

fix i :: id \Rightarrow tv

assume range i \subseteq domain {1}

then have

eval i p \in {Det True, Det False, Indet 1}

eval i q \in {Det True, Det False, Indet 1}

eval i r \in {Det True, Det False, Indet 1}

using ranges[of i {1}]

unfolding domain_def

by auto

then show eval i (Cla3 p q r) = Det True

by (cases eval i p; cases eval i q; cases eval i r) simp_all

qed

proposition valid_in {1, 2} (Cla3 p q r)

unfolding valid_in_def

```

proof (rule; rule)
  fix i :: id  $\Rightarrow$  tv
  assume range i  $\subseteq$  domain {1, 2}
  then have
    eval i p  $\in$  {Det True, Det False, Indet 1, Indet 2}
    eval i q  $\in$  {Det True, Det False, Indet 1, Indet 2}
    eval i r  $\in$  {Det True, Det False, Indet 1, Indet 2}
  using ranges[of i {1, 2}]
  unfolding domain_def
  by auto
  then show eval i (Cla3 p q r) = Det True
  by (cases eval i p; cases eval i q; cases eval i r) auto
qed

```

```

proposition  $\neg$  valid_in {1, 2, 3} (Cla3 (Pro ''p'') (Pro ''q'') (Pro ''r''))
proof -
  let ?i = ( $\lambda$ s. Indet 1)(''q'' := Indet 2, ''r'' := Indet 3)
  have range ?i  $\subseteq$  domain {1, 2, 3}
  unfolding domain_def
  by (simp add: image_subset_iff)
  moreover have eval ?i (Cla3 (Pro ''p'') (Pro ''q'') (Pro ''r'')) = Det False
  by simp
  moreover have Det False  $\neq$  Det True
  by simp
  ultimately show ?thesis
  unfolding valid_in_def
  by metis
qed

```

Further Meta-Theorems

Fundamental Definitions and Lemmas

The function `props` collects the set of propositional symbols occurring in a formula.

```

fun props :: fm  $\Rightarrow$  id set
where
  props Truth = {} |
  props (Pro s) = {s} |
  props (Neg' p) = props p |
  props (Con' p q) = props p  $\cup$  props q |
  props (Eq1 p q) = props p  $\cup$  props q |
  props (Eq1' p q) = props p  $\cup$  props q

```

```

lemma relevant_props: assumes  $\forall$ s  $\in$  props p. i1 s = i2 s shows eval i1 p = eval i2 p
  using assms
  by (induct p) (simp_all, metis)

```

```

fun change_tv :: (nat  $\Rightarrow$  nat)  $\Rightarrow$  tv  $\Rightarrow$  tv
where
  change_tv f (Det b) = Det b |
  change_tv f (Indet n) = Indet (f n)

```

```

lemma change_tv_injection: assumes inj f shows inj (change_tv f)
proof -
  have change_tv f tv1 = change_tv f tv2  $\implies$  tv1 = tv2 for tv1 tv2
  using assms
  by (cases tv1; cases tv2) (simp_all add: inj_eq)
  then show ?thesis

```


by (simp add: injI)
qed

definition

change_int :: (nat \Rightarrow nat) \Rightarrow (id \Rightarrow tv) \Rightarrow (id \Rightarrow tv)
where
change_int f i \equiv λ s. change_tv f (i s)

lemma eval_change: assumes inj f shows eval (change_int f i) p = change_tv f (eval i p)
proof (induct p)

fix p
assume eval (change_int f i) p = change_tv f (eval i p)
then have eval_neg (eval (change_int f i) p) = eval_neg (change_tv f (eval i p))
by simp
then have eval_neg (eval (change_int f i) p) = change_tv f (eval_neg (eval i p))
by (cases eval i p) (simp_all add: case_bool_if)
then show eval (change_int f i) (Neg' p) = change_tv f (eval i (Neg' p))
by simp

next

fix p1 p2
assume ih1: eval (change_int f i) p1 = change_tv f (eval i p1)
assume ih2: eval (change_int f i) p2 = change_tv f (eval i p2)
show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
proof (cases eval i p1 = eval i p2)
assume a: eval i p1 = eval i p2
then have yes: eval i (Con' p1 p2) = eval i p1
by auto
from a have change_tv f (eval i p1) = change_tv f (eval i p2)
by auto
then have eval (change_int f i) p1 = eval (change_int f i) p2
using ih1 ih2
by auto
then have eval (change_int f i) (Con' p1 p2) = eval (change_int f i) p1
by auto
then show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
using yes ih1
by auto

next

assume a': eval i p1 \neq eval i p2
from a' have b': eval (change_int f i) p1 \neq eval (change_int f i) p2
using assms ih1 ih2 change_tv_injection the_inv_f_f
by metis
show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
proof (cases eval i p1 = Det True)
assume a: eval i p1 = Det True
from a a' have eval i (Con' p1 p2) = eval i p2
by auto
then have c: change_tv f (eval i (Con' p1 p2)) = change_tv f (eval i p2)
by auto
from a have b: eval (change_int f i) p1 = Det True
using ih1
by auto
from b b' have eval (change_int f i) (Con' p1 p2) = eval (change_int f i) p2
by auto
then show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
using c ih2
by auto

next

assume a'': eval i p1 \neq Det True
from a'' have b'': eval (change_int f i) p1 \neq Det True

```

    using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
    by metis
show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
proof (cases eval i p2 = Det True)
  assume a: eval i p2 = Det True
  from a a' a'' have eval i (Con' p1 p2) = eval i p1
    by auto
  then have c: change_tv f (eval i (Con' p1 p2)) = change_tv f (eval i p1)
    by auto
  from a have b: eval (change_int f i) p2 = Det True
    using ih2
    by auto
  from b b' b'' have eval (change_int f i) (Con' p1 p2) = eval (change_int f i) p1
    by auto
  then show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
    using c ih1
    by auto
next
  assume a''' : eval i p2 ≠ Det True
  from a' a'' a''' have eval i (Con' p1 p2) = Det False
    by auto
  then have c: change_tv f (eval i (Con' p1 p2)) = Det False
    by auto
  from a''' have b''' : eval (change_int f i) p2 ≠ Det True
    using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
    by metis
  from b' b'' b''' have eval (change_int f i) (Con' p1 p2) = Det False
    by auto
  then show eval (change_int f i) (Con' p1 p2) = change_tv f (eval i (Con' p1 p2))
    using c
    by auto
qed
qed
qed
next
  fix p1 p2
  assume ih1: eval (change_int f i) p1 = change_tv f (eval i p1)
  assume ih2: eval (change_int f i) p2 = change_tv f (eval i p2)
  have Det (eval (change_int f i) p1 = eval (change_int f i) p2) =
    Det (change_tv f (eval i p1) = change_tv f (eval i p2))
    using ih1 ih2
    by simp
  also have ... = Det ((eval i p1) = (eval i p2))
    using assms change_tv_injection
    by (simp add: inj_eq)
  also have ... = change_tv f (Det (eval i p1 = eval i p2))
    by simp
  finally show eval (change_int f i) (Eq1 p1 p2) = change_tv f (eval i (Eq1 p1 p2))
    by simp
next
  fix p1 p2
  assume ih1: eval (change_int f i) p1 = change_tv f (eval i p1)
  assume ih2: eval (change_int f i) p2 = change_tv f (eval i p2)
  show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
  proof (cases eval i p1 = eval i p2)
    assume a: eval i p1 = eval i p2
    then have yes: eval i (Eq1' p1 p2) = Det True
      by auto
    from a have change_tv f (eval i p1) = change_tv f (eval i p2)
      by auto

```

```

then have eval (change_int f i) p1 = eval (change_int f i) p2
  using ih1 ih2
  by auto
then have eval (change_int f i) (Eq1' p1 p2) = Det True
  by auto
then show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
  using yes ih1
  by auto
next
assume a': eval i p1 ≠ eval i p2
show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
proof (cases eval i p1 = Det True)
  assume a: eval i p1 = Det True
  from a a' have yes: eval i (Eq1' p1 p2) = eval i p2
    by auto
  from a have change_tv f (eval i p1) = Det True
    by auto
  then have b: eval (change_int f i) p1 = Det True
    using ih1
    by auto
  from a' have b': eval (change_int f i) p1 ≠ eval (change_int f i) p2
    using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
    by metis
  from b b' have eval (change_int f i) (Eq1' p1 p2) = eval (change_int f i) p2
    by auto
  then show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
    using ih2 yes
    by auto
next
assume a'': eval i p1 ≠ Det True
show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
proof (cases eval i p2 = Det True)
  assume a: eval i p2 = Det True
  from a a' a'' have yes: eval i (Eq1' p1 p2) = eval i p1
    using eval_equality[of i p1 p2]
    by auto
  from a have change_tv f (eval i p2) = Det True
    by auto
  then have b: eval (change_int f i) p2 = Det True
    using ih2
    by auto
  from a' have b': eval (change_int f i) p1 ≠ eval (change_int f i) p2
    using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
    by metis
  from a'' have b'': eval (change_int f i) p1 ≠ Det True
    using b b'
    by auto
  from b b' b'' have eval (change_int f i) (Eq1' p1 p2) = eval (change_int f i) p1
    using eval_equality[of change_int f i p1 p2]
    by auto
  then show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
    using ih1 yes
    by auto
next
assume a''': eval i p2 ≠ Det True
show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
proof (cases eval i p1 = Det False)
  assume a: eval i p1 = Det False
  from a a' a'' a''' have yes: eval i (Eq1' p1 p2) = eval i (Neg' p2)
    using eval_equality[of i p1 p2]

```

```

    by auto
  from a have change_tv f (eval i p1) = Det False
    by auto
  then have b: eval (change_int f i) p1 = Det False
    using ih1
    by auto
  from a' have b': eval (change_int f i) p1 ≠ eval (change_int f i) p2
    using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
    by metis
  from a'' have b'': eval (change_int f i) p1 ≠ Det True
    using b b'
    by auto
  from a''' have b''': eval (change_int f i) p2 ≠ Det True
    using b b' b''
    by (metis assms change_tv.simps(1) change_tv_injection inj_eq ih2)
  from b b' b'' b'''
  have eval (change_int f i) (Eq1' p1 p2) = eval (change_int f i) (Neg' p2)
    using eval_equality[of change_int f i p1 p2]
    by auto
  then show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
    using ih2 yes a a' a''' b b' b''' eval_negation
    by metis
next
assume a'''': eval i p1 ≠ Det False
show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
proof (cases eval i p2 = Det False)
  assume a: eval i p2 = Det False
  from a a' a'' a''' a'''' have yes: eval i (Eq1' p1 p2) = eval i (Neg' p1)
    using eval_equality[of i p1 p2]
    by auto
  from a have change_tv f (eval i p2) = Det False
    by auto
  then have b: eval (change_int f i) p2 = Det False
    using ih2
    by auto
  from a' have b': eval (change_int f i) p1 ≠ eval (change_int f i) p2
    using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
    by metis
  from a'' have b'': eval (change_int f i) p1 ≠ Det True
    using change_tv.elims ih1 tv.simps(4)
    by auto
  from a''' have b''': eval (change_int f i) p2 ≠ Det True
    using b b' b''
    by (metis assms change_tv.simps(1) change_tv_injection inj_eq ih2)
  from a'''' have b'''': eval (change_int f i) p1 ≠ Det False
    using b b'
    by auto
  from b b' b'' b''' b''''
  have eval (change_int f i) (Eq1' p1 p2) = eval (change_int f i) (Neg' p1)
    using eval_equality[of change_int f i p1 p2]
    by auto
  then show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
    using ih1 yes a a' a''' a'''' b b' b'' b''' b'''' eval_negation a'' b''
    by metis
next
assume a''''': eval i p2 ≠ Det False
from a' a'' a''' a'''' a''''' have yes: eval i (Eq1' p1 p2) = Det False
  using eval_equality[of i p1 p2]
  by auto
from a'''''' have change_tv f (eval i p2) ≠ Det False

```

```

    using change_tv_injection inj_eq assms change_tv.simps
  by metis
then have b: eval (change_int f i) p2 ≠ Det False
  using ih2
  by auto
from a' have b': eval (change_int f i) p1 ≠ eval (change_int f i) p2
  using assms ih1 ih2 change_tv_injection the_inv_f_f change_tv.simps
  by metis
from a'' have b'': eval (change_int f i) p1 ≠ Det True
  using change_tv.elims ih1 tv.simps(4)
  by auto
from a''' have b''': eval (change_int f i) p2 ≠ Det True
  using b b' b''
  by (metis assms change_tv.simps(1) change_tv_injection the_inv_f_f ih2)
from a'''' have b'''': eval (change_int f i) p1 ≠ Det False
  by (metis a'' change_tv.simps(2) ih1 string_tv.cases tv.distinct(1))
from b b' b'' b''' b'''' have eval (change_int f i) (Eq1' p1 p2) = Det False
  using eval_equality[of change_int f i p1 p2]
  by auto
then show eval (change_int f i) (Eq1' p1 p2) = change_tv f (eval i (Eq1' p1 p2))
  using ih1 yes a' a''' a'''' b b' b'' b''' b'''' a'' b''
  by auto
qed
qed
qed
qed
qed
qed (simp_all add: change_int_def)

```

Only a Finite Number of Truth Values Needed

Theorem `valid_in_valid` is a kind of the reverse of `valid_valid_in` (or its transfer variant).

abbreviation `is_indet :: tv ⇒ bool`

where

```
is_indet tv ≡ (case tv of Det _ ⇒ False | Indet _ ⇒ True)
```

abbreviation `get_indet :: tv ⇒ nat`

where

```
get_indet tv ≡ (case tv of Det _ ⇒ undefined | Indet n ⇒ n)
```

theorem `valid_in_valid`: assumes `card U ≥ card (props p)` and `valid_in U p` shows `valid p`
 proof -

have `finite U ⇒ card (props p) ≤ card U ⇒ valid_in U p ⇒ valid p` for `U p`

proof -

assume `assms: finite U card (props p) ≤ card U valid_in U p`

show `valid p`

unfolding `valid_def`

proof

fix `i`

obtain `f` where `f_p: (change_int f i) ' (props p) ⊆ (domain U) ∧ inj f`

proof -

have `finite U ⇒ card (props p) ≤ card U ⇒`

`∃f. change_int f i ' props p ⊆ domain U ∧ inj f` for `U p`

proof -

assume `assms: finite U card (props p) ≤ card U`

show `?thesis`

proof -

let `?X = (get_indet ' ((i ' props p) ∩ {tv. is_indet tv}))`

have `d: finite (props p)`

```

by (induct p) auto
then have cx: card ?X ≤ card U
  using assms surj_card_le Int_lower1 card_image_le finite_Int finite_imageI le_trans
  by metis
have f: finite ?X
  using d
  by simp
obtain f where f_p: (∀n ∈ ?X. f n ∈ U) ∧ (inj f)
proof -
  have finite X ⇒ finite Y ⇒ card X ≤ card Y ⇒ ∃f. (∀n ∈ X. f n ∈ Y) ∧ inj f
    for X Y :: nat set
  proof -
    assume assms: finite X finite Y card X ≤ card Y
    show ?thesis
    proof -
      from assms obtain Z where xyz: Z ⊆ Y ∧ card Z = card X
        by (metis card_image card_le_inj)
      then obtain f where bij_betw f X Z
        by (metis assms(1) assms(2) finite_same_card_bij infinite_super)
      then have f_p: (∀n ∈ X. f n ∈ Y) ∧ inj_on f X
        using bij_betwE bij_betw_imp_inj_on xyz
        by blast
      obtain f' where f': f' = (λn. if n ∈ X then f n else n + Suc (Max Y + n))
        by simp
      have inj f'
        unfolding f' inj_on_def
        using assms(2) f_p le_add2 trans_le_add2 not_less_eq_eq
        by (simp, metis Max_ge add.commute inj_on_eq_iff)
      moreover have (∀n ∈ X. f' n ∈ Y)
        unfolding f'
        using f_p
        by auto
      ultimately show ?thesis
        by metis
    qed
  qed
then show (∧f. (∀n ∈ get_indet ' (i ' props p ∩ {tv. is_indet tv}). f n ∈ U)
  ∧ inj f ⇒ thesis) ⇒ thesis
  using assms cx f
  unfolding inj_on_def
  by metis
qed
have (change_int f i) ' (props p) ⊆ (domain U)
proof
  fix x
  assume x ∈ change_int f i ' props p
  then obtain s where s_p: s ∈ props p ∧ change_int f i s = x
    by auto
  then have change_int f i s ∈ {Det True, Det False} ∪ Indet ' U
  proof (cases change_int f i s ∈ {Det True, Det False})
    case True
    then show ?thesis
      by auto
  next
    case False
    then obtain n' where change_int f i s = Indet n'
      by (cases change_int f i s) simp_all
    then have p: change_tv f (i s) = Indet n'
      by (simp add: change_int_def)
    moreover have n' ∈ U

```

```

proof -
  obtain n'' where f n'' = n'
    using calculation change_tv.elims
    by blast
  moreover have s ∈ props p ∧ i s = (Indet n'')
    using p calculation change_tv.simps change_tv_injection the_inv_f_f f_p s_p
    by metis
  then have (Indet n'') ∈ i ' props p
    using image_iff
    by metis
  then have (Indet n'') ∈ i ' props p ∧ is_indet (Indet n'') ∧
    get_indet (Indet n'') = n''
    by auto
  then have n'' ∈ ?X
    using Int_Collect image_iff
    by metis
  ultimately show ?thesis
    using f_p
    by auto
qed
ultimately have change_tv f (i s) ∈ Indet ' U
  by auto
then have change_int f i s ∈ Indet ' U
  unfolding change_int_def
  by auto
then show ?thesis
  by auto
qed
then show x ∈ domain U
  unfolding domain_def
  using s_p
  by simp
qed
then have (change_int f i) ' (props p) ⊆ (domain U) ∧ (inj f)
  unfolding domain_def
  using f_p
  by simp
then show ?thesis
  using f_p
  by metis
qed
qed
then show (∧f. change_int f i ' props p ⊆ domain U ∧ inj f ⇒ thesis) ⇒ thesis
  using assms
  by metis
qed
obtain i2 where i2: i2 = (λs. if s ∈ props p then (change_int f i) s else Det True)
  by simp
then have i2_p: ∀s ∈ props p. i2 s = (change_int f i) s
  ∀s ∈ - props p. i2 s = Det True
  by auto
then have range i2 ⊆ (domain U)
  using i2 f_p
  unfolding domain_def
  by auto
then have eval i2 p = Det True
  using assms
  unfolding valid_in_def
  by auto
then have eval (change_int f i) p = Det True

```

```

    using relevant_props[of p i2 change_int f i] i2_p
  by auto
then have change_tv f (eval i p) = Det True
  using eval_change f_p
  by auto
then show eval i p = Det True
  by (cases eval i p) simp_all
qed
qed
then show ?thesis
  using assms subsetI sup_bot.comm_neutral image_is_empty subsetCE UnCI valid_in_def
  Un_insert_left card.empty card.infinite finite.intros(1)
  unfolding domain_def
  by metis
qed

theorem reduce: valid p  $\longleftrightarrow$  valid_in {1..card (props p)} p
  using valid_in_valid transfer
  by force

```

Case Study

Abbreviations

Entailment takes a list of assumptions.

abbreviation (input) Entail :: fm list \Rightarrow fm \Rightarrow fm

where

```
Entail l p  $\equiv$  Imp (if l = [] then Truth else fold Con' (butlast l) (last l)) p
```

theorem entailment_not_chain:

```
 $\neg$  valid (Eq1 (Entail [Pro ''p'', Pro ''q''] (Pro ''r''))
  (Box ((Imp' (Pro ''p'') (Imp' (Pro ''q'') (Pro ''r''))))))
```

proof -

```
let ?i = ( $\lambda$ s. Indet 1)(''r'' := Det False)
```

```
have eval ?i (Eq1 (Entail [Pro ''p'', Pro ''q''] (Pro ''r''))
```

```
(Box ((Imp' (Pro ''p'') (Imp' (Pro ''q'') (Pro ''r'')))))) = Det False
```

```
by simp
```

```
moreover have Det False  $\neq$  Det True
```

```
by simp
```

```
ultimately show ?thesis
```

```
unfolding valid_def
```

```
by metis
```

qed

abbreviation (input) B0 :: fm where B0 \equiv Con' (Con' (Pro ''p'') (Pro ''q'')) (Neg' (Pro ''r''))

abbreviation (input) B1 :: fm where B1 \equiv Imp' (Con' (Pro ''p'') (Pro ''q'')) (Pro ''r'')

abbreviation (input) B2 :: fm where B2 \equiv Imp' (Pro ''r'') (Pro ''s'')

abbreviation (input) B3 :: fm where B3 \equiv Imp' (Neg' (Pro ''s'')) (Neg' (Pro ''r''))

Results

The paraconsistent logic is usable in contrast to classical logic.

theorem classical_logic_is_not_usable: valid_boole (Entail [B0, B1] p)


```

unfolding valid_in_def
proof (rule; rule)
  fix i :: id  $\Rightarrow$  tv
  assume range i  $\subseteq$  domain {}
  then have
    i ''p''  $\in$  {Det True, Det False}
    i ''q''  $\in$  {Det True, Det False}
    i ''r''  $\in$  {Det True, Det False}
  unfolding domain_def
  by auto
  then show eval i (Entail [B0, B1] p) = Det True
  by (cases i ''p''; cases i ''q''; cases i ''r'') simp_all
qed

```

```

corollary valid_boole (Entail [B0, B1] (Pro ''r''))
  by (rule classical_logic_is_not_usable)

```

```

corollary valid_boole (Entail [B0, B1] (Neg' (Pro ''r'')))
  by (rule classical_logic_is_not_usable)

```

```

proposition  $\neg$  valid (Entail [B0, B1] (Pro ''r''))
proof -
  let ?i = ( $\lambda$ s. Indet 1)(''r'' := Det False)
  have eval ?i (Entail [B0, B1] (Pro ''r'')) = Det False
  by simp
  moreover have Det False  $\neq$  Det True
  by simp
  ultimately show ?thesis
  unfolding valid_def
  by metis
qed

```

```

proposition valid_boole (Entail [B0, Box B1] p)
  unfolding valid_in_def
proof (rule; rule)
  fix i :: id  $\Rightarrow$  tv
  assume range i  $\subseteq$  domain {}
  then have
    i ''p''  $\in$  {Det True, Det False}
    i ''q''  $\in$  {Det True, Det False}
    i ''r''  $\in$  {Det True, Det False}
  unfolding domain_def
  by auto
  then show eval i (Entail [B0, Box B1] p) = Det True
  by (cases i ''p''; cases i ''q''; cases i ''r'') simp_all
qed

```

```

proposition  $\neg$  valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''p'')))
proof -
  let ?i = ( $\lambda$ s. Indet 1)(''p'' := Det True)
  have eval ?i (Entail [B0, Box B1, Box B2] (Neg' (Pro ''p''))) = Det False
  by simp
  moreover have Det False  $\neq$  Det True
  by simp
  ultimately show ?thesis
  unfolding valid_def
  by metis
qed

```

```

proposition  $\neg$  valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''q'')))

```

```

proof -
  let ?i = (λs. Indet 1)(''q'' := Det True)
  have eval ?i (Entail [B0, Box B1, Box B2] (Neg' (Pro ''q''))) = Det False
    by simp
  moreover have Det False ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid_def
    by metis
qed

```

```

proposition ¬ valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''s'')))
proof -
  let ?i = (λs. Indet 1)(''s'' := Det True)
  have eval ?i (Entail [B0, Box B1, Box B2] (Neg' (Pro ''s''))) = Det False
    by simp
  moreover have Det False ≠ Det True
    by simp
  ultimately show ?thesis
    unfolding valid_def
    by metis
qed

```

```

proposition valid (Entail [B0, Box B1, Box B2] (Pro ''r''))
proof -
  have {1..card (props (Entail [B0, Box B1, Box B2] (Pro ''r'')))} = {1, 2, 3, 4}
    by code_simp
  moreover have valid_in {1, 2, 3, 4} (Entail [B0, Box B1, Box B2] (Pro ''r''))
    unfolding valid_in_def
  proof (rule; rule)
    fix i :: id ⇒ tv
    assume range i ⊆ domain {1, 2, 3, 4}
    then have icase:
      i ''p'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''q'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''r'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''s'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
    unfolding domain_def
    by auto
  show eval i (Entail [B0, Box B1, Box B2] (Pro ''r'')) = Det True
    using icase
    by (cases i ''p''; cases i ''q''; cases i ''r''; cases i ''s'') simp_all
  qed
  ultimately show ?thesis
    using reduce
    by simp
qed

```

```

proposition valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''r'')))
proof -
  have {1..card (props (Entail [B0, Box B1, Box B2] (Neg' (Pro ''r''))))} = {1, 2, 3, 4}
    by code_simp
  moreover have valid_in {1, 2, 3, 4} (Entail [B0, Box B1, Box B2] (Neg' (Pro ''r'')))
    unfolding valid_in_def
  proof (rule; rule)
    fix i :: id ⇒ tv
    assume range i ⊆ domain {1, 2, 3, 4}
    then have icase:
      i ''p'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''q'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}

```

```

    i ''r'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
    i ''s'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
    unfolding domain_def
    by auto
  show eval i (Entail [B0, Box B1, Box B2] (Neg' (Pro ''r''))) = Det True
    using icase
    by (cases i ''p''; cases i ''q''; cases i ''r''; cases i ''s'') simp_all
qed
ultimately show ?thesis
  using reduce
  by simp
qed

proposition valid (Entail [B0, Box B1, Box B2] (Pro ''s''))
proof -
  have {1..card (props (Entail [B0, Box B1, Box B2] (Pro ''s'')))} = {1, 2, 3, 4}
    by code_simp
  moreover have valid_in {1, 2, 3, 4} (Entail [B0, Box B1, Box B2] (Pro ''s''))
    unfolding valid_in_def
  proof (rule; rule)
    fix i :: id ⇒ tv
    assume range i ⊆ domain {1, 2, 3, 4}
    then have icase:
      i ''p'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''q'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''r'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''s'' ∈ {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
    unfolding domain_def
    by auto
  show eval i (Entail [B0, Box B1, Box B2] (Pro ''s'')) = Det True
    using icase
    by (cases i ''p''; cases i ''q''; cases i ''r''; cases i ''s'') simp_all
qed
ultimately show ?thesis
  using reduce
  by simp
qed

```

Acknowledgements

Thanks to the Isabelle developers for making a superb system and for always being willing to help.

end — Paraconsistency file

References

- [1] A. S. Jensen and J. Villadsen. *Paraconsistent Computational Logic*. In P. Blackburn, K. F. Jørgensen, N. Jones, and E. Palmgren, editors, 8th Scandinavian Logic Symposium: Abstracts, pages 59–61, Roskilde University, 2012.
- [2] G. Priest, K. Tanaka and Z. Weber. *Paraconsistent Logic*. In E. N. Zalta et al., editors, Stanford Encyclopedia of Philosophy, Online Entry <http://plato.stanford.edu/entries/logic-paraconsistent/> Spring Edition, 2015.
- [3] J. Villadsen. *Supra-logic: Using Transfinite Type Theory with Type Variables for Paraconsistency*. Logical Approaches to Paraconsistency, Journal of Applied Non-Classical Logics, 15(1):45–58, 2005.
- [4] J. Villadsen. *Infinite-Valued Propositional Type Theory for Semantics*. In J.-Y. Béziau and A. Costa-Leite, editors, Dimensions of Logical Concepts, pages 277–297, Unicamp Coleç. CLE 54, 2009.
- [5] J. Villadsen. *Nabla: A Linguistic System Based on Type Theory*. Foundations of Communication and Cognition (New Series), LIT Verlag, 2010.
- [6] J. Villadsen. *Multi-dimensional Type Theory: Rules, Categories and Combinators for Syntax and Semantics*. In P. Blache, H. Christiansen, V. Dahl, D. Duchier, and J. Villadsen, editors, Constraints and Language, pages 167–189, Cambridge Scholars Press, 2014.