

Paraconsistency

Anders Schlichtkrull & Jørgen Villadsen, DTU Compute, Denmark

17 March 2025

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	8
Further Meta-Theorems	11
Case Study	12
Acknowledgements	13
Notation	14
Injections From Sets to Sets	14
Extension of Paraconsistency Theory	14
Logics of Equal Cardinality Are Equal	15
Conversions Between Nats and Strings	16
Derived Formula Constructors	16
Pigeon Hole Formula	17
Validity Is the Intersection of the Finite Logics	19

Logics of Different Cardinalities Are Different	19
Finite Logics Are Different from Infinite Logics	19
References	21

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 [4, 5, 6, 7].

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)
  <proof>

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
    )
  <proof>

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
    )
  <proof>

corollary eval i (Cla p) = eval i (Box (Dis' p (Neg' p)))
  <proof>

lemma double_negation: eval i p = eval i (Neg' (Neg' p))
  <proof>

```

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

<proof>

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 $\equiv \text{map } (\lambda x. (tv, x)) \text{ tvs}$

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

where

tv_pair_table tvs $\equiv \text{map } (\text{tv_pair_row tvs}) \text{ tvs}$

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

where

map_row f tvtvs $\equiv \text{map } (\lambda (x, y). f \ x \ y) \text{ tvtvs}$

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

where

map_table f tvtvss $\equiv \text{map } (\text{map_row } f) \text{ tvtvss}$

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

where

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

definition digit_of_nat :: nat $\Rightarrow$ char

where

digit_of_nat n $\equiv$
(if n = 1 then (CHR ''1'') else if n = 2 then (CHR ''2'') else if n = 3 then (CHR ''3'') else
if n = 4 then (CHR ''4'') else if n = 5 then (CHR ''5'') else if n = 6 then (CHR ''6'') else
if n = 7 then (CHR ''7'') else if n = 8 then (CHR ''8'') else if n = 9 then (CHR ''9'') else
(CHR ''0''))

fun string_of_nat :: nat $\Rightarrow$ string

where

string_of_nat n =
(if n < 10 then [digit_of_nat n] else string_of_nat (n div 10) @ [digit_of_nat (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  $\equiv$  foldr append strs []

definition appends_nl :: string list  $\Rightarrow$  string
where
  appends_nl strs  $\equiv$  '' $\hookleftarrow$ '' @ foldr ( $\lambda$ s s'. s @ '' $\hookleftarrow$ '' @ s') (butlast strs) (last strs) @ '' $\hookleftarrow$ ''

definition string_table :: tv list list  $\Rightarrow$  string list list
where
  string_table tvss  $\equiv$  map (map string_tv) tvss

definition string_table_string :: string list list  $\Rightarrow$  string
where
  string_table_string strss  $\equiv$  appends_nl (map appends strss)

definition unary :: fm  $\Rightarrow$  tv list  $\Rightarrow$  string
where
  unary p tvs  $\equiv$  appends_nl (map string_tv (unary_truth_table p tvs))

definition binary :: fm  $\Rightarrow$  tv list  $\Rightarrow$  string
where
  binary p tvs  $\equiv$  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
'',
<proof>

```

proposition

```

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

```

proposition

```

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

```

proposition

```

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

```

```

*
1
'',
<proof>

```

proposition

```

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

```

proposition

```

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

```

proposition

```

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

```

proposition

```

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

```

proposition

```

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

```

proposition

```

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

```

proposition

```

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

```

```

oooo
oooo
oooo
,,
<proof>

```

proposition

```

binary (Dis (Pro ''p'') (Pro ''q'')) [Det True, Det False, Indet 1, Indet 2] = ''
****
*ooo
*oo*
*o*o
,,
<proof>

```

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 $\neg$ valid (Nab (Box p))
 <proof>

proposition valid (Cla (Cla p)) and $\neg$ valid (Nab (Nab p))
 <proof>

proposition valid (Cla (Nab p)) and $\neg$ valid (Nab (Cla p))
 <proof>

proposition valid (Box p) $\longleftrightarrow$ valid (Box (Box p))
 <proof>

proposition valid (Neg p) $\longleftrightarrow$ valid (Neg' p)
 <proof>

proposition valid (Con p q) $\longleftrightarrow$ valid (Con' p q)
 <proof>

proposition valid (Dis p q) $\longleftrightarrow$ valid (Dis' p q)
 <proof>

proposition valid (Eq1 p q) $\longleftrightarrow$ valid (Eq1' p q)
 <proof>

proposition valid (Imp p q) $\longleftrightarrow$ valid (Imp' p q)
 <proof>

proposition $\neg$ valid (Pro ''p'')
 <proof>

proposition $\neg$ valid (Neg' (Pro ''p''))
 <proof>

proposition assumes valid p shows $\neg$ valid (Neg' p)
 <proof>

proposition assumes valid (Neg' p) shows $\neg$ valid p
 <proof>

proposition valid (Neg' (Neg' p)) $\longleftrightarrow$ valid p
<proof>

theorem conjunction: valid (Con' p q) $\longleftrightarrow$ valid p $\wedge$ valid q
<proof>

corollary assumes valid (Con' p q) shows valid p and valid q
<proof>

proposition assumes valid p and valid (Imp p q) shows valid q
<proof>

proposition assumes valid p and valid (Imp' p q) shows valid q
<proof>

Key Equalities

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

proposition valid (Eq1 p (Neg' (Neg' p)))
<proof>

proposition valid (Eq1 Truth (Neg' Falsity))
<proof>

proposition valid (Eq1 Falsity (Neg' Truth))
<proof>

proposition valid (Eq1 p (Con' p p))
<proof>

proposition valid (Eq1 p (Con' Truth p))
<proof>

proposition valid (Eq1 p (Con' p Truth))
<proof>

proposition valid (Eq1 Truth (Eq1' p p))
<proof>

proposition valid (Eq1 p (Eq1' Truth p))
<proof>

proposition valid (Eq1 p (Eq1' p Truth))
<proof>

proposition valid (Eq1 (Neg' p) (Eq1' Falsity p))
<proof>

proposition valid (Eq1 (Neg' p) (Eq1' p Falsity))
<proof>

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>

definition valid_in :: nat set  $\Rightarrow$  fm  $\Rightarrow$  bool
where
  valid_in U p  $\equiv$   $\forall i.$  range i  $\subseteq$  domain U  $\longrightarrow$  eval i p = 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
  <proof>

theorem valid_valid_in: assumes valid p shows valid_in U p
  <proof>

theorem transfer: assumes  $\neg$  valid_in U p shows  $\neg$  valid p
  <proof>

proposition valid_in U (Neg' (Neg' p))  $\longleftrightarrow$  valid_in U p
  <proof>

theorem conjunction_in: valid_in U (Con' p q)  $\longleftrightarrow$  valid_in U p  $\wedge$  valid_in U q
  <proof>

corollary assumes valid_in U (Con' p q) shows valid_in U p and valid_in U q
  <proof>

proposition assumes valid_in U p and valid_in U (Imp p q) shows valid_in U q
  <proof>

proposition assumes valid_in U p and valid_in U (Imp' p q) shows valid_in U q
  <proof>

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''))
  <proof>

lemma explosion_counterexample:  $\neg$  valid_in {1} (Explosion (Pro ''p'') (Pro ''q''))
  <proof>

theorem explosion_not_valid:  $\neg$  valid (Explosion (Pro ''p'') (Pro ''q''))
  <proof>

proposition  $\neg$  valid (Imp (Con' (Pro ''p'') (Neg' (Pro ''p'')))) (Pro ''q'')
  <proof>

```

Example: Contraposition

Contraposition is not valid.

```

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

```

proposition valid_boole (Contraposition (Pro ''p'') (Pro ''q''))
 <proof>

proposition valid_in {1} (Contraposition (Pro ''p'') (Pro ''q''))
 <proof>

lemma contraposition_counterexample: $\neg$ valid_in {1, 2} (Contraposition (Pro ''p'') (Pro ''q''))
 <proof>

theorem contraposition_not_valid: $\neg$ valid (Contraposition (Pro ''p'') (Pro ''q''))
 <proof>

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
 <proof>

proposition
 unary (Cla (Pro ''p'')) [Det True, Det False, Indet 1] = ''
 *
 *
 o
 ,,
 <proof>

proposition valid_boole (Cla p)
 <proof>

proposition $\neg$ valid_in {1} (Cla (Pro ''p''))
 <proof>

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*
 ,,
 <proof>

proposition valid_boole (Cla2 p q)
 <proof>

proposition valid_in {1} (Cla2 p q)
 <proof>

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

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)
  ⟨proof⟩

```

```

proposition valid_in {1} (Cla3 p q r)
  ⟨proof⟩

```

```

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

```

```

proposition ¬ valid_in {1, 2, 3} (Cla3 (Pro ''p'') (Pro ''q'') (Pro ''r''))
  ⟨proof⟩

```

Further Meta-Theorems

Fundamental Definitions and Lemmas

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

```

fun props :: fm ⇒ id set
where
  props Truth = {} |
  props (Pro s) = {s} |
  props (Neg' p) = props p |
  props (Con' p q) = props p ∪ props q |
  props (Eq1 p q) = props p ∪ props q |
  props (Eq1' p q) = props p ∪ props q

```

```

lemma relevant_props: assumes ∀ s ∈ props p. i1 s = i2 s shows eval i1 p = eval i2 p
  ⟨proof⟩

```

```

fun change_tv :: (nat ⇒ nat) ⇒ tv ⇒ 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⟩

```

```

definition
  change_int :: (nat ⇒ nat) ⇒ (id ⇒ tv) ⇒ (id ⇒ tv)
where
  change_int f i ≡ λs. 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⟩

```

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 $\geq$ card (props p) and valid_in U p shows valid p
<proof>

theorem reduce: valid p $\longleftrightarrow$ valid_in {1..card (props p)} p
<proof>

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>

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)
<proof>

corollary valid_boole (Entail [B0, B1] (Pro ''r''))
<proof>

corollary valid_boole (Entail [B0, B1] (Neg' (Pro ''r'')))
<proof>

proposition $\neg$ valid (Entail [B0, B1] (Pro ''r''))
<proof>

proposition valid_boole (Entail [B0, Box B1] p)
<proof>

proposition $\neg$ valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''p'')))
<proof>

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

proposition $\neg$ valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''s'')))
<proof>

proposition valid (Entail [B0, Box B1, Box B2] (Pro ''r''))

<proof>

proposition valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''r'')))
<proof>

proposition valid (Entail [B0, Box B1, Box B2] (Pro ''s''))
<proof>

Acknowledgements

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

end — Paraconsistency file

```
theory Paraconsistency_Validity_Infinite imports Paraconsistency
abbrevs
  Truth =  $\top$ 
and
  Falsity =  $\perp$ 
and
  Neg' =  $\neg$ 
and
  Con' =  $\wedge$ 
and
  Eql =  $\Leftrightarrow$ 
and
  Eql' =  $\leftrightarrow$ 
and
  Dis' =  $\vee$ 
and
  Imp =  $\Rightarrow$ 
and
  Imp' =  $\rightarrow$ 
and
  Box =  $\Box$ 
and
  Neg =  $\neg\neg$ 
and
  Con =  $\wedge\wedge$ 
and
  Dis =  $\vee\vee$ 
and
  Cla =  $\Delta$ 
and
  Nab =  $\nabla$ 
and
  CON = [ $\wedge\wedge$ ]
and
  DIS = [ $\vee\vee$ ]
and
  NAB = [ $\nabla$ ]
and
  ExiEql = [ $\exists=$ ]
begin
```

The details about the definitions, lemmas and theorems are described in an article in the Post-proceedings of the 24th International Conference on Types for Proofs and Programs (TYPES 2018).

Notation

```
notation Pro (<⟦_⟧> [39] 39)
notation Truth (<⊤>)
notation Neg' (<¬ _> [40] 40)
notation Con' (infixr <∧> 35)
notation Eq1 (infixr <⇔> 25)
notation Eq1' (infixr <↔> 25)
notation Falsity (<⊥>)
notation Dis' (infixr <∨> 30)
notation Imp (infixr <⇒> 25)
notation Imp' (infixr <→> 25)
notation Box (<□ _> [40] 40)
notation Neg (<¬→ _> [40] 40)
notation Con (infixr <∧∧> 35)
notation Dis (infixr <∨∨> 30)
notation Cla (<Δ _> [40] 40)
notation Nab (<∇ _> [40] 40)
abbreviation DetTrue :: tv (<·>) where · ≡ Det True
abbreviation DetFalse :: tv (<◊>) where ◊ ≡ Det False
notation Indet (<⊥_> [39] 39)
```

Strategy: We define a formula that is valid in the sets $0..<1$, $0..<2$, ..., $0..<n-1$ but is not valid in the set $0..<n$

Injections From Sets to Sets

We define the notion of an injection from a set X to a set Y

```
definition inj_from_to :: ('a ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool where
  inj_from_to f X Y ≡ inj_on f X ∧ f ` X ⊆ Y

lemma bij_betw_inj_from_to: bij_betw f X Y ⇒ inj_from_to f X Y
  <proof>
```

Special lemma for finite cardinality only

```
lemma inj_from_to_if_card:
  assumes card X ≤ card Y
  assumes finite X
  shows ∃f. inj_from_to f X Y
  <proof>
```

Extension of Paraconsistency Theory

The Paraconsistency theory is extended with abbreviation `is_det` and a number of lemmas that are or generalizations of previous lemmas

```
abbreviation is_det :: tv ⇒ bool where is_det tv ≡ ¬ is_indet tv

theorem valid_iff_valid_in:
  assumes card U ≥ card (props p)
  shows valid p ↔ valid_in U p
  <proof>
```

Generalization of `change_tv_injection`

```

lemma change_tv_injection_on:
  assumes inj_on f U
  shows inj_on (change_tv f) (domain U)
  <proof>

```

Similar to change_tv_injection_on

```

lemma change_tv_injection_from_to:
  assumes inj_from_to f U W
  shows inj_from_to (change_tv f) (domain U) (domain W)
  <proof>

```

Similar to eval_change_inj_on

```

lemma change_tv_surj_on:
  assumes f ' U = W
  shows (change_tv f) ' (domain U) = (domain W)
  <proof>

```

Similar to eval_change_inj_on

```

lemma change_tv_bij_betw:
  assumes bij_betw f U W
  shows bij_betw (change_tv f) (domain U) (domain W)
  <proof>

```

Generalization of eval_change

```

lemma eval_change_inj_on:
  assumes inj_on f U
  assumes range i  $\subseteq$  domain U
  shows eval (change_int f i) p = change_tv f (eval i p)
  <proof>

```

Logics of Equal Cardinality Are Equal

We prove that validity in a set depends only on the cardinality of the set

```

lemma inj_from_to_valid_in:
  assumes inj_from_to f W U
  assumes valid_in U p
  shows valid_in W p
  <proof>

```

```

corollary
  assumes inj_from_to f U W
  assumes inj_from_to g W U
  shows valid_in U p  $\longleftrightarrow$  valid_in W p
  <proof>

```

```

lemma bij_betw_valid_in:
  assumes bij_betw f U W
  shows valid_in U p  $\longleftrightarrow$  valid_in W p
  <proof>

```

```

theorem eql_finite_eql_card_valid_in:
  assumes finite U  $\longleftrightarrow$  finite W

```

```

    assumes card U = card W
    shows valid_in U p  $\longleftrightarrow$  valid_in W p
  <proof>

```

corollary

```

    assumes U  $\neq$  {}
    assumes W  $\neq$  {}
    assumes card U = card W
    shows valid_in U p  $\longleftrightarrow$  valid_in W p
  <proof>

```

theorem finite_eql_card_valid_in:

```

    assumes finite U
    assumes finite W
    assumes card U = card W
    shows valid_in U p  $\longleftrightarrow$  valid_in W p
  <proof>

```

theorem infinite_valid_in:

```

    assumes infinite U
    assumes infinite W
    shows valid_in U p  $\longleftrightarrow$  valid_in W p
  <proof>

```

Conversions Between Nats and Strings

definition nat_of_digit :: char $\Rightarrow$ nat where

```

    nat_of_digit c =
      (if c = (CHR ''1'') then 1 else if c = (CHR ''2'') then 2 else if c = (CHR ''3'') then 3 else
       if c = (CHR ''4'') then 4 else if c = (CHR ''5'') then 5 else if c = (CHR ''6'') then 6 else
       if c = (CHR ''7'') then 7 else if c = (CHR ''8'') then 8 else if c = (CHR ''9'') then 9 else 0)

```

proposition range nat_of_digit = {0.. <10 }

<proof>

lemma nat_of_digit_of_nat[simp]: $n < 10 \implies \text{nat_of_digit} (\text{digit_of_nat } n) = n$

<proof>

function nat_of_string :: string $\Rightarrow$ nat

where

```

    nat_of_string n = (if length n  $\leq$  1 then nat_of_digit (last n) else
      (nat_of_string (butlast n)) * 10 + (nat_of_digit (last n)))

```

<proof>

termination

<proof>

lemma nat_of_string_step:

```

    nat_of_string (string_of_nat (m div 10)) * 10 + m mod 10 = nat_of_string (string_of_nat m)

```

<proof>

lemma nat_of_string_of_nat: nat_of_string (string_of_nat n) = n

<proof>

lemma inj string_of_nat

<proof>

Derived Formula Constructors

definition PRO :: id list $\Rightarrow$ fm list where

```

PRO ids  $\equiv$  map Pro ids

definition Pro_nat :: nat  $\Rightarrow$  fm ( $\langle \_ \rangle_1$  [40] 40) where
   $\langle n \rangle_1 \equiv \langle \text{string\_of\_nat } n \rangle$ 

definition PRO_nat :: nat list  $\Rightarrow$  fm list ( $\langle \_ \rangle_{123}$  [40] 40) where
   $\langle ns \rangle_{123} \equiv \text{map Pro\_nat } ns$ 

definition CON :: fm list  $\Rightarrow$  fm ( $\langle [\wedge] \_ \rangle$  [40] 40) where
   $[\wedge] ps \equiv \text{foldr Con } ps \top$ 

definition DIS :: fm list  $\Rightarrow$  fm ( $\langle [\vee] \_ \rangle$  [40] 40) where
   $[\vee] ps \equiv \text{foldr Dis } ps \perp$ 

definition NAB :: fm list  $\Rightarrow$  fm ( $\langle [\nabla] \_ \rangle$  [40] 40) where
   $[\nabla] ps \equiv [\wedge] (\text{map Nab } ps)$ 

definition off_diagonal_product :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\times$  'a) set where
  off_diagonal_product xs ys  $\equiv \{(x,y). (x,y) \in (xs \times ys) \wedge x \neq y\}$ 

definition List_off_diagonal_product :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  ('a  $\times$  'a) list where
  List_off_diagonal_product xs ys  $\equiv \text{filter } (\lambda(x,y). \text{not\_equal } x \ y) (\text{List.product } xs \ ys)$ 

definition ExiEq1 :: fm list  $\Rightarrow$  fm ( $\langle [\exists=] \_ \rangle$  [40] 40) where
   $[\exists=] ps \equiv [\vee] (\text{map } (\lambda(x,y). x \Leftrightarrow y) (\text{List.off\_diagonal\_product } ps \ ps))$ 

lemma cla_false_Imp:
  assumes eval i a = .
  assumes eval i b = o
  shows eval i (a  $\Rightarrow$  b) = o
  <proof>

lemma eval_CON:
  eval i ( $[\wedge] ps$ ) = Det ( $\forall p \in \text{set } ps. \text{eval } i \ p = .$ )
  <proof>

lemma eval_DIS:
  eval i ( $[\vee] ps$ ) = Det ( $\exists p \in \text{set } ps. \text{eval } i \ p = .$ )
  <proof>

lemma eval_Nab: eval i ( $[\nabla] p$ ) = Det (is_indet (eval i p))
  <proof>

lemma eval_NAB:
  eval i ( $[\nabla] ps$ ) = Det ( $\forall p \in \text{set } ps. \text{is\_indet } (\text{eval } i \ p)$ )
  <proof>

lemma eval_ExiEq1:
  eval i ( $[\exists=] ps$ ) =
    Det ( $\exists (p1, p2) \in (\text{off\_diagonal\_product } (\text{set } ps) (\text{set } ps)). \text{eval } i \ p1 = \text{eval } i \ p2$ )
  <proof>

```

Pigeon Hole Formula

```

definition pigeonhole_fm :: nat  $\Rightarrow$  fm where
  pigeonhole_fm n  $\equiv [\nabla] \langle [0..\<n] \rangle_{123} \Rightarrow [\exists=] \langle [0..\<n] \rangle_{123}$ 

definition interp_of_id :: nat  $\Rightarrow$  id  $\Rightarrow$  tv where
  interp_of_id maxi i  $\equiv \text{if } (\text{nat\_of\_string } i) < \text{maxi} \text{ then } [\text{nat\_of\_string } i] \text{ else } .$ 

```

```

lemma interp_of_id_pigeonhole_fm_False: eval (interp_of_id n) (pigeonhole_fm n) = 0
⟨proof⟩

lemma range_interp_of_id: range (interp_of_id n) ⊆ domain {0..theorem not_valid_in_n_pigeonhole_fm: ¬ (valid_in {0..theorem not_valid_pigeonhole_fm: ¬ (valid (pigeonhole_fm n))
⟨proof⟩

lemma cla_imp_I:
  assumes is_det (eval i a)
  assumes is_det (eval i b)
  assumes eval i a = . ⇒ eval i b = .
  shows eval i (a ⇒ b) = .
⟨proof⟩

lemma is_det_NAB: is_det (eval i ([∇] ps))
⟨proof⟩

lemma is_det_ExiEq1: is_det (eval i ([∃=] ps))
⟨proof⟩

lemma pigeonhole_nat:
  assumes finite n
  assumes finite m
  assumes card n > card m
  assumes f ' n ⊆ m
  shows ∃x∈n. ∃y∈n. x ≠ y ∧ f x = f y
⟨proof⟩

lemma pigeonhole_nat_set:
  assumes f ' {0..lemma inj_Pro_nat: ((p1)1) = ((p2)1) ⇒ p1 = p2
⟨proof⟩

lemma eval_true_in_lt_n_pigeonhole_fm:
  assumes m < n
  assumes range i ⊆ domain {0..theorem valid_in_lt_n_pigeonhole_fm:
  assumes m < n
  shows valid_in {0..theorem not_valid_in_pigeonhole_fm_card:
  assumes finite U
  shows ¬ valid_in U (pigeonhole_fm (card U))
⟨proof⟩

theorem not_valid_in_pigeonhole_fm_lt_card:
  assumes finite (U::nat set)

```

```

  assumes inj_from_to f U W
  shows  $\neg$  valid_in W (pigeonhole_fm (card U))
<proof>

```

```

theorem valid_in_pigeonhole_fm_n_gt_card:
  assumes finite U
  assumes card U < n
  shows valid_in U (pigeonhole_fm n)
<proof>

```

Validity Is the Intersection of the Finite Logics

```

lemma valid p  $\longleftrightarrow$  ( $\forall U$ . finite U  $\longrightarrow$  valid_in U p)
<proof>

```

Logics of Different Cardinalities Are Different

```

lemma finite_card_lt_valid_in_not_valid_in:
  assumes finite U
  assumes card U < card W
  shows valid_in U  $\neq$  valid_in W
<proof>

```

```

lemma valid_in_UNIV_p_valid: valid_in UNIV p = valid p
<proof>

```

```

theorem infinite_valid_in_valid:
  assumes infinite U
  shows valid_in U p  $\longleftrightarrow$  valid p
<proof>

```

```

lemma finite_not_finite_valid_in_not_valid_in:
  assumes finite U  $\neq$  finite W
  shows valid_in U  $\neq$  valid_in W
<proof>

```

```

lemma card_not_card_valid_in_not_valid_in:
  assumes card U  $\neq$  card W
  shows valid_in U  $\neq$  valid_in W
<proof>

```

Finite Logics Are Different from Infinite Logics

```

theorem extend: valid  $\neq$  valid_in U if finite U
<proof>

```

```

corollary  $\neg$  ( $\exists n$ .  $\forall p$ . valid p  $\longleftrightarrow$  valid_in {0..n} p)
<proof>

```

```

corollary  $\forall n$ .  $\exists p$ .  $\neg$  (valid p  $\longleftrightarrow$  valid_in {0..n} p)
<proof>

```

```

corollary  $\neg$  ( $\forall p$ . valid p  $\longleftrightarrow$  valid_in {0..n} p)
<proof>

```

```

corollary valid  $\neq$  valid_in {0..n}
<proof>

```

```

proposition valid = valid_in {0..}

```

<proof>

corollary valid = valid_in {n..}
<proof>

corollary $\neg (\exists n\ m. \forall p. \text{valid } p \longleftrightarrow \text{valid_in } \{m..n\} \ p)$
<proof>

end — Paraconsistency_Validity_Infinite 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] A. Schlichtkrull. *New Formalized Results on the Meta-Theory of a Paraconsistent Logic*. In P. Dybjer, J. E. Santo, and L. Pinto, editors, 24th International Conference on Types for Proofs and Programs, pages 5:1–5:15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.
- [4] 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.
- [5] 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.
- [6] J. Villadsen. *Nabla: A Linguistic System Based on Type Theory*. Foundations of Communication and Cognition (New Series), LIT Verlag, 2010.
- [7] 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.