

Paraconsistency

Anders Schlichtkrull & Jørgen Villadsen, DTU Compute, Denmark

26 May 2024

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
Notation	27
Injections From Sets to Sets	28
Extension of Paraconsistency Theory	28
Logics of Equal Cardinality Are Equal	30
Conversions Between Nats and Strings	32
Derived Formula Constructors	33
Pigeon Hole Formula	34
Validity Is the Intersection of the Finite Logics	38

Logics of Different Cardinalities Are Different	38
Finite Logics Are Different from Infinite Logics	39
References	41

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)
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. eval i p = 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  $\equiv$  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  $\equiv$  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  $\equiv$  map (map_row f) tvsvss
```

```
definition unary_truth_table :: fm  $\Rightarrow$  tv list  $\Rightarrow$  tv list
where
  unary_truth_table p tvs  $\equiv$ 
    map ( $\lambda$ x. eval (( $\lambda$ s. undefined)(''p'' := x)) 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. eval (( $\lambda$ s. undefined)(''p'' := x, ''q'' := y)) 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  $\equiv$  '' $\leftarrow$ '' @ foldr ( $\lambda$ s s'. s @ '' $\leftarrow$ '' @ s') (butlast strs) (last strs) @ '' $\leftarrow$ ''

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
'',
  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  $\neg$  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 ⇒ 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 (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  $\Rightarrow$  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 (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  $\in$  {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  $\in$  {Det True, Det False}
    eval i q  $\in$  {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  $\in$  {Det True, Det False, Indet 1}
  eval i q  $\in$  {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  $\implies$  nat)  $\implies$  (id  $\implies$  tv)  $\implies$  (id  $\implies$  tv)
where
  change_int f i  $\equiv$   $\lambda$ 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 ≠ Det True
from a'' have b'': eval (change_int f i) p1 ≠ 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'' 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''' 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''' 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 $\text{card } U \geq \text{card } (\text{props } p)$ 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 ⇒
```

```

     $\exists f. \text{change\_int } f \text{ i ' props } p \subseteq \text{domain } U \wedge \text{inj } f \text{ for } U \text{ p}$ 
proof -
  assume assms: finite U card (props p)  $\leq$  card U
  show ?thesis
  proof -
    let ?X = (get_indet ' ((i ' props p)  $\cap$  {tv. is_indet tv}))
    have d: finite (props p)
      by (induct p) auto
    then have cx: card ?X  $\leq$  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: ( $\forall n \in ?X. f \ n \in U$ )  $\wedge$  (inj f)
    proof -
      have finite X  $\implies$  finite Y  $\implies$  card X  $\leq$  card Y  $\implies$   $\exists f. (\forall n \in X. f \ n \in Y) \wedge \text{inj } f$ 
        for X Y :: nat set
      proof -
        assume assms: finite X finite Y card X  $\leq$  card Y
        show ?thesis
        proof -
          from assms obtain Z where xyz: Z  $\subseteq$  Y  $\wedge$  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: ( $\forall n \in X. f \ n \in Y$ )  $\wedge$  inj_on f X
            using bij_betwE bij_betw_imp_inj_on xyz
            by blast
          obtain f' where f': f' = ( $\lambda n. \text{if } n \in X \text{ then } f \ n \text{ else } n + \text{Suc } (\text{Max } Y + n)$ )
            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 ( $\forall n \in X. f' \ n \in Y$ )
            unfolding f'
            using f_p
            by auto
          ultimately show ?thesis
            by metis
        qed
      qed
    then show ( $\bigwedge f. (\forall n \in \text{get\_indet ' (i ' props } p \cap \{\text{tv. is\_indet tv}\}). f \ n \in U)$ 
       $\wedge$  inj f  $\implies$  thesis)  $\implies$  thesis
      using assms cx f
      unfolding inj_on_def
      by metis
    qed
  have (change_int f i ' (props p)  $\subseteq$  (domain U))
  proof
    fix x
    assume x  $\in$  change_int f i ' props p
    then obtain s where s_p: s  $\in$  props p  $\wedge$  change_int f i s = x
      by auto
    then have change_int f i s  $\in$  {Det True, Det False}  $\cup$  Indet ' U
    proof (cases change_int f i s  $\in$  {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 = (λ 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 = ( $\lambda$ s. 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  $\neq$  Det True
    by simp
  ultimately show ?thesis
    unfolding valid_def
    by metis
qed

```

```

proposition  $\neg$  valid (Entail [B0, Box B1, Box B2] (Neg' (Pro ''s'')))
proof -
  let ?i = ( $\lambda$ s. 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  $\neq$  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  $\Rightarrow$  tv
    assume range i  $\subseteq$  domain {1, 2, 3, 4}
    then have icase:
      i ''p''  $\in$  {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''q''  $\in$  {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''r''  $\in$  {Det True, Det False, Indet 1, Indet 2, Indet 3, Indet 4}
      i ''s''  $\in$  {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

```

theory Paraconsistency_Validity_Infinite imports Paraconsistency
  abbrevs
    Truth = ⊤
  and
    Falsity = ⊥
  and
    Neg' = ¬
  and
    Con' = ∧

```

```

and
Eq1 =  $\Leftrightarrow$ 
and
Eq1' =  $\leftrightarrow$ 
and
Dis' =  $\vee$ 
and
Imp =  $\Rightarrow$ 
and
Imp' =  $\rightarrow$ 
and
Box =  $\square$ 
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
ExiEq1 = [ $\exists =$ ]
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 ( $\langle \_ \rangle$  [39] 39)
notation Truth ( $\top$ )
notation Neg' ( $\neg \_$  [40] 40)
notation Con' (infixr  $\wedge$  35)
notation Eq1 (infixr  $\Leftrightarrow$  25)
notation Eq1' (infixr  $\leftrightarrow$  25)
notation Falsity ( $\perp$ )
notation Dis' (infixr  $\vee$  30)
notation Imp (infixr  $\Rightarrow$  25)
notation Imp' (infixr  $\rightarrow$  25)
notation Box ( $\square \_$  [40] 40)
notation Neg ( $\neg\neg \_$  [40] 40)
notation Con (infixr  $\wedge\wedge$  35)
notation Dis (infixr  $\vee\vee$  30)
notation Cla ( $\Delta \_$  [40] 40)
notation Nab ( $\nabla \_$  [40] 40)
abbreviation DetTrue :: tv ( $\cdot$ ) where  $\cdot \equiv$  Det True
abbreviation DetFalse :: tv ( $\circ$ ) where  $\circ \equiv$  Det False
notation Indet ( $\lfloor \_ \rfloor$  [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  $\Rightarrow$  'b)  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  bool where  
  inj_from_to f X Y  $\equiv$  inj_on f X  $\wedge$  f ` X  $\subseteq$  Y
```

```
lemma bij_betw_inj_from_to: bij_betw f X Y  $\implies$  inj_from_to f X Y  
  unfolding bij_betw_def inj_from_to_def by simp
```

Special lemma for finite cardinality only

```
lemma inj_from_to_if_card:  
  assumes card X  $\leq$  card Y  
  assumes finite X  
  shows  $\exists$ f. inj_from_to f X Y  
  unfolding inj_from_to_def  
  by (metis assms card_image card_le_inj card_subset_eq obtain_subset_with_card_n order_refl)
```

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  $\Rightarrow$  bool where is_det tv  $\equiv$   $\neg$  is_indet tv
```

```
theorem valid_iff_valid_in:  
  assumes card U  $\geq$  card (props p)  
  shows valid p  $\longleftrightarrow$  valid_in U p  
  using assms valid_in_valid valid_valid_in by blast
```

Generalization of `change_tv_injection`

```
lemma change_tv_injection_on:  
  assumes inj_on f U  
  shows inj_on (change_tv f) (domain U)  
proof  
  fix x y  
  assume x  $\in$  domain U y  $\in$  domain U change_tv f x = change_tv f y  
  then show x = y  
    unfolding domain_def using assms inj_onD by (cases x; cases y) auto  
qed
```

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)  
  unfolding inj_from_to_def  
proof  
  show inj_on (change_tv f) (domain U)  
    using assms change_tv_injection_on unfolding inj_from_to_def by blast  
next
```

```

show change_tv f ' domain U  $\subseteq$  domain W
proof
  fix x
  assume x  $\in$  change_tv f ' domain U
  then show x  $\in$  domain W
    unfolding domain_def image_def
    using assms inj_from_to_def[of f U W]
    by (cases x) auto
qed
qed

```

Similar to eval_change_inj_on

```

lemma change_tv_surj_on:
  assumes f ' U = W
  shows (change_tv f) ' (domain U) = (domain W)
proof
  show change_tv f ' domain U  $\subseteq$  domain W
  proof
    fix x
    assume x  $\in$  change_tv f ' domain U
    then show x  $\in$  domain W
    proof
      fix x'
      assume x = change_tv f x' x'  $\in$  domain U
      then show x  $\in$  domain W
        unfolding domain_def using assms by fastforce
    qed
  qed
next
  show domain W  $\subseteq$  change_tv f ' domain U
  proof
    fix x
    assume x  $\in$  domain W
    then show x  $\in$  change_tv f ' domain U
      unfolding domain_def using assms image_iff by fastforce
  qed
qed

```

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)
  using assms change_tv_injection_on change_tv_surj_on unfolding bij_betw_def by simp

```

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 (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) (¬ p) = change_tv f (eval i (¬ 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) (p1 ∧ p2) = change_tv f (eval i (p1 ∧ p2))
  using assms ih1 ih2 change_tv.simps(1) change_tv_injection_on eval.simps(2) eval.simps(4)
  inj_onD ranges by metis
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))
proof -
  have inj_on (change_tv f) (domain U)
    using assms(1) change_tv_injection_on by simp
  then show ?thesis
    using assms(2) ranges by (simp add: inj_on_eq_iff)
qed
also have ... = change_tv f (Det (eval i p1 = eval i p2))
  by simp
finally show eval (change_int f i) (p1 ↔ p2) = change_tv f (eval i (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) (p1 ↔ p2) = change_tv f (eval i (p1 ↔ p2))
  using assms ih1 ih2 inj_on_eq_iff change_tv.simps(1) change_tv_injection_on eval_equality
  eval_negation ranges by smt
qed (simp_all add: change_int_def)

```

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
  unfolding valid_in_def proof (rule, rule)
  fix i :: char list ⇒ tv
  assume a: range i ⊆ domain W
  from assms have valid_p: ∀i. range i ⊆ domain U ⟶ eval i p = .
    unfolding valid_in_def by simp
  have range (change_int f i) ⊆ domain U
  proof
    fix x
    assume x ∈ range (change_int f i)
    then obtain xa where xa: change_int f i xa = x
      by blast
    have inj_from_to (change_tv f) (domain W) (domain U)
      using change_tv_injection_from_to assms by simp
    then have (change_tv f) (i xa) ∈ domain U
      using a by (metis inj_from_to_def image_eqI range_eqI subsetCE)
  
```

```

    then show  $x \in \text{domain } U$ 
      using xa change_int_def by simp
qed
then have eval (change_int f i) p = .
  using valid_p by simp
then have eval (change_int f i) p = .
  by simp
then have change_tv f (eval i p) = .
  using a assms(1) eval_change_inj_on unfolding inj_from_to_def by metis
then show eval i p = .
  using change_tv.elims tv.distinct(1) by fast
qed

```

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
using assms inj_from_to_valid_in by fast

```

lemma bij_betw_valid_in:

```

assumes bij_betw f U W
shows valid_in U p  $\longleftrightarrow$  valid_in W p
using assms inj_from_to_valid_in bij_betw_inv bij_betw_inj_from_to by metis

```

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 (cases finite U)

```

case True
then show ?thesis
  using assms bij_betw_iff_card bij_betw_valid_in by metis

```

next

```

case False
then have ( $\exists f :: \text{nat} \Rightarrow \text{nat}. \text{bij\_betw } f \ U \ \text{UNIV}$ )  $\wedge$  ( $\exists g :: \text{nat} \Rightarrow \text{nat}. \text{bij\_betw } g \ W \ \text{UNIV}$ )
  using assms Schroeder_Bernstein infinite_iff_countable_subset inj_Suc top_greatest by metis
  with bij_betw_valid_in show ?thesis
  by metis

```

qed

corollary

```

assumes  $U \neq \{\}$ 
assumes  $W \neq \{\}$ 
assumes card U = card W
shows valid_in U p  $\longleftrightarrow$  valid_in W p
using assms eql_finite_eql_card_valid_in card_gt_0_iff by metis

```

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
using eql_finite_eql_card_valid_in by (simp add: assms)

```

theorem infinite_valid_in:

```

assumes infinite U
assumes infinite W
shows valid_in U p  $\longleftrightarrow$  valid_in W p
using eql_finite_eql_card_valid_in by (simp add: assms)

```


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

```
show range nat_of_digit  $\subseteq$  {0.. $<10$ }
  unfolding nat_of_digit_def by auto
```

next

```
show {0.. $<10$ }  $\subseteq$  range nat_of_digit
```

proof

```
fix x :: nat
assume a: x  $\in$  {0.. $<10$ }
show x  $\in$  range nat_of_digit
  proof (cases x = 0)
    case True
    then show ?thesis
      unfolding nat_of_digit_def by auto
  next
    case False
    with a show ?thesis
      unfolding nat_of_digit_def by auto
  qed
qed
qed
```

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

```
unfolding digit_of_nat_def nat_of_digit_def
  by simp presburger
```

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))))
```

by simp_all

termination

```
by (relation measure length) simp_all
```

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)
  by simp
```

lemma nat_of_string_of_nat: nat_of_string (string_of_nat n) = n

proof (induct rule: string_of_nat.induct)

case (1 m)

then show ?case

proof (cases m < 10)

case True

then show ?thesis

by simp

next

case False

then have nat_of_string (string_of_nat (m div 10)) = m div 10

using 1 by simp

then have nat_of_string (string_of_nat (m div 10)) * 10 = (m div 10) * 10

by simp

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

```

      (m div 10) * 10 + (m mod 10)
    by simp
  also have ... = m
    by simp
  finally show ?thesis
    using nat_of_string_step by simp
qed
qed

```

```

lemma inj_string_of_nat
  using inj_on_inverseI nat_of_string_of_nat by metis

```

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 ( $[\wedge]$  _ [40] 40) where
   $[\wedge] ps \equiv \text{foldr Con } ps \top$ 

```

```

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

```

```

definition NAB :: fm list  $\Rightarrow$  fm ( $[\nabla]$  _ [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 ( $[\exists=]$  _ [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 =  $\cdot$ 
  assumes eval i b =  $\circ$ 
  shows eval i (a  $\Rightarrow$  b) =  $\circ$ 
  using assms by simp

```

```

lemma eval_CON:
  eval i ( $[\wedge] ps$ ) = Det ( $\forall p \in \text{set } ps. \text{eval } i \ p = \cdot$ )
  unfolding CON_def
  by (induct ps) simp_all

```

```

lemma eval_DIS:
  eval i ( $[\vee] ps$ ) = Det ( $\exists p \in \text{set } ps. \text{eval } i \ p = \cdot$ )
  unfolding DIS_def

```

```

proof (induct ps)
  case Nil
  then show ?case
    by simp

```

```

next

```

```

case Cons
with eval.simps eval_negation foldr.simps list.set_intros o_apply set_ConsD show ?case by smt
qed

```

```

lemma eval_Nab: eval i ( $\nabla$  p) = Det (is_indet (eval i p))
proof (induct p)
case (Pro x)
then show ?case
using string_tv.cases tv.simps(5) tv.simps(6) eval_negation
eval.simps(2) eval.simps(4) eval.simps(5) by smt
next
case (Neg' p)
then show ?case
using eval_negation by fastforce
next
case (Eq1' p1 p2)
then show ?case
using string_tv.cases tv.simps(5) tv.simps(6) eval_negation
eval.simps(2) eval.simps(4) eval.simps(5) by smt
qed auto

```

```

lemma eval_NAB:
eval i ( $\nabla$  ps) = Det ( $\forall p \in \text{set ps. is\_indet (eval i p)$ )
proof (cases  $\forall p \in \text{set ps. is\_indet (eval i p)$ )
case True
then have eval i ( $\nabla$  ps) = .
unfolding NAB_def using eval_CON by fastforce
then show ?thesis
using True by simp
next
case False
then have  $\neg (\forall p \in \text{set ps. eval i (\nabla p) = \cdot)$ 
using eval_Nab by simp
then have  $\neg (\forall p \in \text{set (map Nab ps). eval i p = \cdot)$ 
by simp
then have eval i ( $\nabla$  ps) =  $\circ$ 
unfolding NAB_def using eval_CON[of i (map Nab ps)] by simp
then show ?thesis
using False by simp
qed

```

```

lemma eval_ExistEq:
eval i ( $\exists =$  ps) =
Det ( $\exists (p1, p2) \in (\text{off\_diagonal\_product (set ps) (set ps)}). \text{eval i p1} = \text{eval i p2}$ )
using eval_DIS[of i (map ( $\lambda(x, y). x \Leftrightarrow y$ ) (List_off_diagonal_product ps ps))]
unfolding off_diagonal_product_def ExistEq_def List_off_diagonal_product_def
by auto

```

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  $\equiv$  if (nat_of_string i) < maxi then  $\lfloor \text{nat\_of\_string } i \rfloor$  else  $\cdot$ 

```

```

lemma interp_of_id_pigeonhole_fm_False: eval (interp_of_id n) (pigeonhole_fm n) =  $\circ$ 
proof -
have all_indet:  $\forall p \in \text{set } (\langle [0..<n] \rangle_{123}). \text{is\_indet (eval (interp\_of\_id } n) p)$ 
proof

```

```

fix p
assume a: p ∈ set (([0..<n])123)
show is_indet (eval (interp_of_id n) p)
proof -
  from a have p ∈ Pro_nat ‘ {..1)
  unfolding Pro_nat_def by fast
  then show ?thesis
  unfolding interp_of_id_def Pro_nat_def using nat_of_string_of_nat by fastforce
qed
qed
then have eval (interp_of_id n) ([∇] (([0..<n])123)) = .
  using eval_NAB by simp
moreover
have ∀a b. a ∈ set (map (λn. <n>1) [0..<n]) →
  b ∈ set (map (λn. <n>1) [0..<n]) → a ≠ b →
  eval (interp_of_id n) a = eval (interp_of_id n) b → False
using all_indet in_set_conv_nth length_map nat_of_string_of_nat nth_map tv.inject tv.simps(5)
eval.simps(1)
unfolding interp_of_id_def PRO_def PRO_nat_def Pro_nat_def
by smt
then have ∀(p1, p2)∈off_diagonal_product (set (([0..<n])123)) (set (([0..<n])123)).
  eval (interp_of_id n) p1 ≠ eval (interp_of_id n) p2
unfolding off_diagonal_product_def PRO_nat_def Pro_nat_def by blast
then have ¬ (∃(p1, p2)∈off_diagonal_product (set (([0..<n])123)) (set (([0..<n])123)).
  eval (interp_of_id n) p1 = eval (interp_of_id n) p2)
  by blast
then have eval (interp_of_id n) ([∃=] (([0..<n])123)) = o
  using eval_ExiEq[of interp_of_id n <[0..<n]>123] by simp
ultimately
show ?thesis
  unfolding pigeonhole_fm_def using cla_false_Imp[of interp_of_id n] by blast
qed

lemma range_interp_of_id: range (interp_of_id n) ⊆ domain {0..<n}
  unfolding interp_of_id_def domain_def by (simp add: image_subset_iff)

theorem not_valid_in_n_pigeonhole_fm: ¬ (valid_in {0..<n} (pigeonhole_fm n))
  unfolding valid_in_def using interp_of_id_pigeonhole_fm_False[of n] range_interp_of_id[of n]
  by fastforce

theorem not_valid_pigeonhole_fm: ¬ (valid (pigeonhole_fm n))
  unfolding valid_def using interp_of_id_pigeonhole_fm_False[of n]
  by fastforce

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 -
  have is_det tv = (case tv of Det _ ⇒ True | [_] ⇒ False) for tv
  by (metis (full_types) tv.exhaust tv.simps(5) tv.simps(6))
  then show ?thesis
  using assms
  by (metis (full_types) eval.simps(4) eval.simps(5) tv.exhaust tv.simps(6))
qed

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

```

```

unfolding eval_NAB by auto

lemma is_det_ExiEq1: is_det (eval i (( $\exists$ =) ps))
  using eval_ExiEq1 by auto

lemma pigeonhole_nat:
  assumes finite n
  assumes finite m
  assumes card n > card m
  assumes f ' n  $\subseteq$  m
  shows  $\exists x \in n. \exists y \in n. x \neq y \wedge f x = f y$ 
  using assms not_le inj_on_iff_card_le unfolding inj_on_def
  by metis

lemma pigeonhole_nat_set:
  assumes f ' {0.. $n$ }  $\subseteq$  {0.. $m$ }
  assumes m < (n :: nat)
  shows  $\exists j_1 \in \{0.. $n$ \}. \exists j_2 \in \{0.. $n$ \}. j_1 \neq j_2 \wedge f j_1 = f j_2$ 
  using assms pigeonhole_nat[of {0.. $n$ } {0.. $m$ } f]
  by simp

lemma inj_Pro_nat: ( $\langle p_1 \rangle_1$ ) = ( $\langle p_2 \rangle_1$ )  $\implies$  p1 = p2
  unfolding Pro_nat_def using fm.inject(1) nat_of_string_of_nat
  by metis

lemma eval_true_in_lt_n_pigeonhole_fm:
  assumes m < n
  assumes range i  $\subseteq$  domain {0.. $m$ }
  shows eval i (pigeonhole_fm n) = .
proof -
  {
    assume eval i (( $\nabla$ ) ( $\langle [0.. $n$ ] \rangle_{123}$ )) = .
    then have  $\forall p \in \text{set } (\langle [0.. $n$ ] \rangle_{123}). \text{is\_indet } (\text{eval } i \ p)$ 
      using eval_NAB by auto
    then have *:  $\forall j < n. \text{is\_indet } (\text{eval } i \ (\langle j \rangle_1))$ 
      unfolding PRO_nat_def by auto
    have **:  $\forall j < n. \exists k < m. \text{eval } i \ (\langle j \rangle_1) = (\langle k \rangle)$ 
      proof -
        have  $\forall j < n. \text{is\_indet } (\text{eval } i \ (\langle j \rangle_1)) \implies j < n \implies \exists k < m. \text{eval } i \ (\langle j \rangle_1) = (\langle k \rangle)$  for j
          proof (rule_tac x=get_indet (i (string_of_nat j)) in exI)
            show  $\forall j < n. \text{is\_indet } (\text{eval } i \ (\langle j \rangle_1)) \implies j < n \implies \text{get\_indet } (i \ (\text{string\_of\_nat } j)) < m \wedge$ 
               $\text{eval } i \ (\langle j \rangle_1) = (\langle \text{get\_indet } (i \ (\text{string\_of\_nat } j)) \rangle)$ 
            proof (induct i (string_of_nat j))
              case (Det x)
              then show ?case
                unfolding Pro_nat_def using eval.simps(1) tv.simps(5) by metis
            next
              case (Indet x)
              then show ?case
                proof (subgoal_tac x < m)
                  show  $(\langle x \rangle) = i \ (\text{string\_of\_nat } j) \implies \forall j < n. \text{is\_indet } (\text{eval } i \ (\langle j \rangle_1)) \implies j < n \implies$ 
                     $x < m \implies \text{get\_indet } (i \ (\text{string\_of\_nat } j)) < m \wedge$ 
                     $\text{eval } i \ (\langle j \rangle_1) = (\langle \text{get\_indet } (i \ (\text{string\_of\_nat } j)) \rangle)$ 
                  unfolding Pro_nat_def using eval.simps(1) tv.simps(6) by metis
                next
                  show  $(\langle x \rangle) = i \ (\text{string\_of\_nat } j) \implies \forall j < n. \text{is\_indet } (\text{eval } i \ (\langle j \rangle_1)) \implies j < n \implies x < m$ 
                    using assms(2) atLeast0LessThan unfolding domain_def by fast
                qed
              qed
            qed
          qed
      qed
  }
qed
qed
qed

```

```

    then show ?thesis
      using * by simp
qed
then have  $\forall j < n. \exists k < m. \text{get\_indet} (\text{eval } i \langle (j)_1 \rangle) = k$ 
  by fastforce
then have  $(\lambda j. \text{get\_indet} (\text{eval } i \langle (j)_1 \rangle)) \text{ ' } \{0..<n\} \subseteq \{0..<m\}$ 
  by fastforce
then have  $\exists j_1 \in \{0..<n\}. \exists j_2 \in \{0..<n\}. j_1 \neq j_2 \wedge \text{get\_indet} (\text{eval } i \langle (j_1)_1 \rangle) =$ 
   $\text{get\_indet} (\text{eval } i \langle (j_2)_1 \rangle)$ 
  using assms(1) pigeonhole_nat_set by simp
then have  $\exists j_1 < n. \exists j_2 < n. j_1 \neq j_2 \wedge \text{get\_indet} (\text{eval } i \langle (j_1)_1 \rangle) =$ 
   $\text{get\_indet} (\text{eval } i \langle (j_2)_1 \rangle)$ 
  using atLeastLessThan_iff by blast
then have  $\exists j_1 < n. \exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle$ 
  using ** tv.simps(6) by metis
then have  $\exists (p_1, p_2) \in \text{off\_diagonal\_product} (\text{set} \langle [0..<n]_{123} \rangle) (\text{set} \langle [0..<n]_{123} \rangle).$ 
   $\text{eval } i p_1 = \text{eval } i p_2$ 
proof (rule_tac P= $\lambda j_1. j_1 < n \wedge (\exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle =$ 
   $\text{eval } i \langle (j_2)_1 \rangle)$  in exE)
  show  $\exists j_1 < n. \exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle \implies$ 
     $\exists x < n. \exists j_2 < n. x \neq j_2 \wedge \text{eval } i \langle (x)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle$ 
    by simp
next
  show  $\exists j_1 < n. \exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle \implies$ 
     $j_1 < n \wedge (\exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle) \implies$ 
     $\exists (p_1, p_2) \in \text{off\_diagonal\_product} (\text{set} \langle [0..<n]_{123} \rangle) (\text{set} \langle [0..<n]_{123} \rangle).$ 
     $\text{eval } i p_1 = \text{eval } i p_2$  for j1
proof (rule_tac P= $\lambda j_2. j_2 < n \wedge j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle$  in exE)
  show  $\exists j_1 < n. \exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle \implies$ 
     $j_1 < n \wedge (\exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle) \implies$ 
     $\exists x < n. j_1 \neq x \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (x)_1 \rangle$ 
    by simp
next
  show  $\exists j_1 < n. \exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle \implies$ 
     $j_1 < n \wedge (\exists j_2 < n. j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle) \implies$ 
     $j_2 < n \wedge j_1 \neq j_2 \wedge \text{eval } i \langle (j_1)_1 \rangle = \text{eval } i \langle (j_2)_1 \rangle \implies$ 
     $\exists (p_1, p_2) \in \text{off\_diagonal\_product} (\text{set} \langle [0..<n]_{123} \rangle) (\text{set} \langle [0..<n]_{123} \rangle).$ 
     $\text{eval } i p_1 = \text{eval } i p_2$  for j2
    unfolding off_diagonal_product_def PRO_nat_def using inj_Pro_nat
    by (rule_tac x= $\langle (j_1)_1, (j_2)_1 \rangle$  in bexI) auto
qed
qed
then have  $\text{eval } i ([\exists =] \langle [0..<n]_{123} \rangle) = \cdot$ 
  using eval_ExiEq1 by simp
}
then show ?thesis
  unfolding pigeonhole_fm_def using cla_imp_I is_det_ExiEq1 is_det_NAB by simp
qed

```

```

theorem valid_in_lt_n_pigeonhole_fm:
  assumes  $m < n$ 
  shows  $\text{valid\_in } \{0..<m\} (\text{pigeonhole\_fm } n)$ 
  using assms
  unfolding valid_in_def
  using interp_of_id_pigeonhole_fm_False[of n]
  using range_interp_of_id[of n]
  using eval_true_in_lt_n_pigeonhole_fm
  by simp

```

```

theorem not_valid_in_pigeonhole_fm_card:

```

```

assumes finite U
shows  $\neg$  valid_in U (pigeonhole_fm (card U))
using assms ex_bij_betw_nat_finite not_valid_in_n_pigeonhole_fm bij_betw_valid_in by metis

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 -
  have  $\neg$  valid_in U (pigeonhole_fm (card U))
    using not_valid_in_pigeonhole_fm_card assms by simp
  then show ?thesis
    using assms inj_from_to_valid_in by metis
qed

theorem valid_in_pigeonhole_fm_n_gt_card:
  assumes finite U
  assumes card U < n
  shows valid_in U (pigeonhole_fm n)
  using assms ex_bij_betw_finite_nat bij_betw_valid_in valid_in_lt_n_pigeonhole_fm by metis

```

Validity Is the Intersection of the Finite Logics

```

lemma valid p  $\leftrightarrow$  ( $\forall$ U. finite U  $\longrightarrow$  valid_in U p)
proof
  assume valid p
  then show  $\forall$ U. finite U  $\longrightarrow$  valid_in U p
    using transfer by blast
next
  assume  $\forall$ U. finite U  $\longrightarrow$  valid_in U p
  then have valid_in {1..card (props p)} p
    by simp
  then show valid p
    using reduce by simp
qed

```

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 -
  have finite_W: finite W
    using assms(2) card.infinite by fastforce
  have valid_in U (pigeonhole_fm (card W))
    using valid_in_pigeonhole_fm_n_gt_card assms by simp
  moreover
  have  $\neg$  valid_in W (pigeonhole_fm (card W))
    using not_valid_in_pigeonhole_fm_card assms finite_W by simp
  ultimately show ?thesis
    by fastforce
qed

lemma valid_in_UNIV_p_valid: valid_in UNIV p = valid p
  using universal_domain valid_def valid_in_def by simp

theorem infinite_valid_in_valid:
  assumes infinite U

```

```

shows valid_in U p  $\longleftrightarrow$  valid p
using assms infinite_valid_in[of U UNIV p] valid_in_UNIV_p_valid by simp

lemma finite_not_finite_valid_in_not_valid_in:
  assumes finite U  $\neq$  finite W
  shows valid_in U  $\neq$  valid_in W
proof -
  {
    fix U W :: nat set
    assume inf: infinite U
    assume fin: finite W
    then have valid_in_W_pigeonhole_fm: valid_in W (pigeonhole_fm (Suc (card W)))
      using valid_in_pigeonhole_fm_n_gt_card[of W] by simp
    have  $\neg$  valid (pigeonhole_fm (Suc (card W)))
      using not_valid_pigeonhole_fm by simp
    then have  $\neg$  valid_in U (pigeonhole_fm (Suc (card W)))
      using inf fin infinite_valid_in_valid by simp
    then have valid_in U  $\neq$  valid_in W
      using valid_in_W_pigeonhole_fm by fastforce
  }
  then show ?thesis
    using assms by metis
qed

lemma card_not_card_valid_in_not_valid_in:
  assumes card U  $\neq$  card W
  shows valid_in U  $\neq$  valid_in W
using assms
proof -
  {
    fix U W :: nat set
    assume a: card U < card W
    then have finite W
      using card.infinite gr_implies_not0 by blast
    then have valid_in_W_pigeonhole_fm: valid_in W (pigeonhole_fm (Suc (card W)))
      using valid_in_pigeonhole_fm_n_gt_card[of W] by simp
    have valid_in U  $\neq$  valid_in W
    proof (cases finite U)
      case True
      then show ?thesis
        using a finite_card_lt_valid_in_not_valid_in by simp
    next
      case False
      have  $\neg$  valid (pigeonhole_fm (Suc (card W)))
        using not_valid_pigeonhole_fm by simp
      then have  $\neg$  valid_in U (pigeonhole_fm (Suc (card W)))
        using False infinite_valid_in_valid by simp
      then show ?thesis
        using valid_in_W_pigeonhole_fm by fastforce
    qed
  }
  then show ?thesis
    using assms neqE by metis
qed

```

Finite Logics Are Different from Infinite Logics

```

theorem extend: valid  $\neq$  valid_in U if finite U
  using that not_valid_pigeonhole_fm valid_in_pigeonhole_fm_n_gt_card by fastforce

```



```

corollary  $\neg (\exists n. \forall p. \text{valid } p \longleftrightarrow \text{valid\_in } \{0..n\} p)$ 
  using extend by fast

corollary  $\forall n. \exists p. \neg (\text{valid } p \longleftrightarrow \text{valid\_in } \{0..n\} p)$ 
  using extend by fast

corollary  $\neg (\forall p. \text{valid } p \longleftrightarrow \text{valid\_in } \{0..n\} p)$ 
  using extend by fast

corollary  $\text{valid} \neq \text{valid\_in } \{0..n\}$ 
  using extend by simp

proposition  $\text{valid} = \text{valid\_in } \{0..\}$ 
  unfolding valid_def valid_in_def
  using universal_domain
  by simp

corollary  $\text{valid} = \text{valid\_in } \{n..\}$ 
  using infinite_valid_in[of UNIV {n..}] universal_domain
  unfolding valid_def valid_in_def
  by (simp add: infinite_Ici)

corollary  $\neg (\exists n m. \forall p. \text{valid } p \longleftrightarrow \text{valid\_in } \{m..n\} p)$ 
  using extend by fast

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.