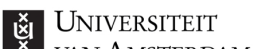
eFLINT - A DSL for Testing Normative Specifications

L. Thomas van Binsbergen
Centrum Wiskunde & Informatica

22 November, 2019

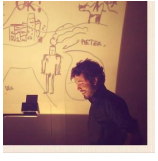


People

UvA and more

Robert van Doesburg

Tom van Engers



Giovanni Sileno

Lu-Chi Liu

CWI

Marc Stevens

Thomas van Binsbergen

Tijs van der Storm

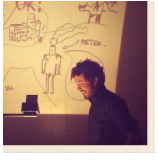
People

Policy-CAD

UvA and more

Robert van Doesburg

Tom van Engers



Giovanni Sileno

Lu-Chi Liu

CWI

Thomas van Binsbergen

Marc Stevens

Tijs van der Storm

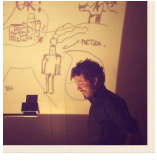
People

SSPDDP

UvA and more

Robert van Doesburg

Tom van Engers



Giovanni Sileno

Lu-Chi Liu

CWI

Marc Stevens

Thomas van Binsbergen

Tijs van der Storm

Norms - Philosophy/Sociology

Normative sentences are "ought-to" types of statements

Norms - Philosophy/Sociology

Normative sentences are "ought-to" types of statements

Examples: legal norms - social norms

Norms - Philosophy/Sociology

Normative sentences are "ought-to" types of statements

Examples: legal norms - social norms

As a resident of The Netherlands, you must have health insurance

Norms - Philosophy/Sociology

Normative sentences are "ought-to" types of statements

Examples: legal norms - social norms

As a resident of The Netherlands, you must have health insurance
CWI's SWAT team has lunch together at noon

Norms - Philosophy/Sociology

Normative sentences are "ought-to" types of statements

Examples: legal norms - social norms

As a resident of The Netherlands, you must have health insurance
CWI's SWAT team has lunch together at noon
A player cannot score from an offside position

Norms - Philosophy/Sociology

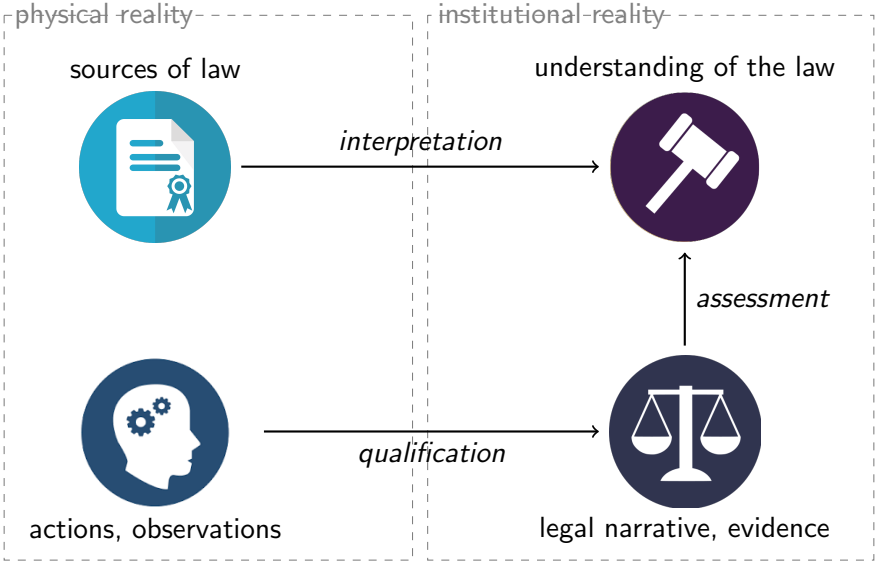
Normative sentences are "ought-to" types of statements

Examples: legal norms - social norms

As a resident of The Netherlands, you must have health insurance
CWI's SWAT team has lunch together at noon
A player cannot score from an offside position

Deontic	**Potestative**
duties, obligations	powers, actions
permissions	liabilities

Analyzing legal cases



Interpreting normative sources

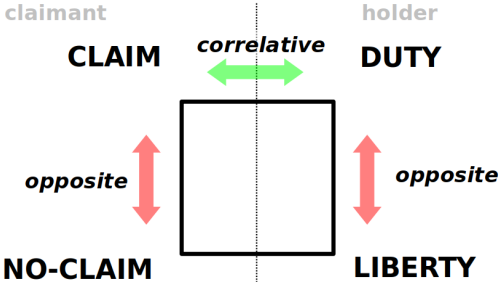
What does the result of interpretation look like?

Interpreting normative sources

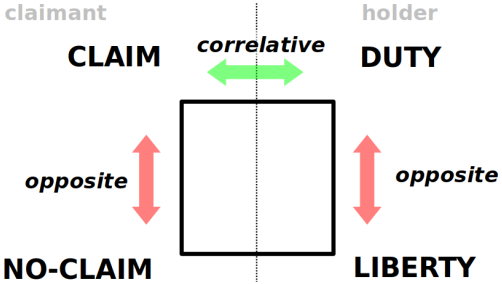
What does the result of interpretation look like?

How do we write down an interpretation formally?

Hohfeld's fundamental legal conceptions

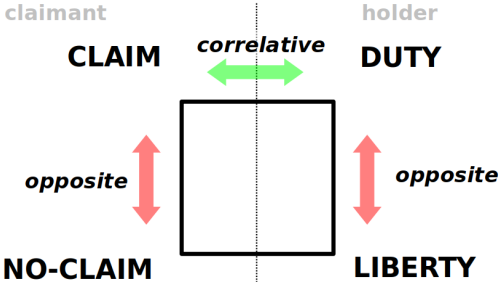


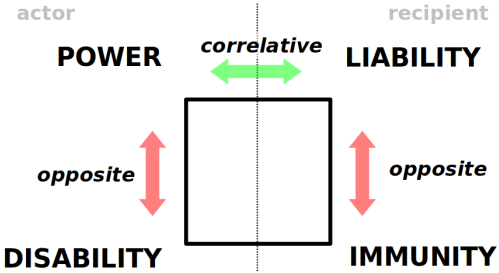
Hohfeld's fundamental legal conceptions



fundamental relation: **duty-claim**
between *duty holder* and *claimant*

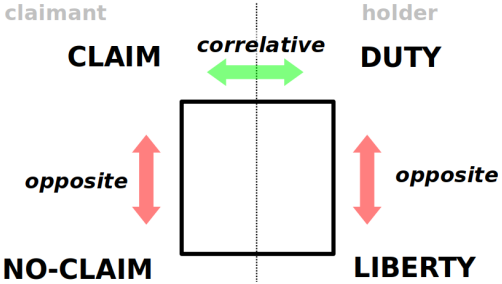
Hohfeld's fundamental legal conceptions



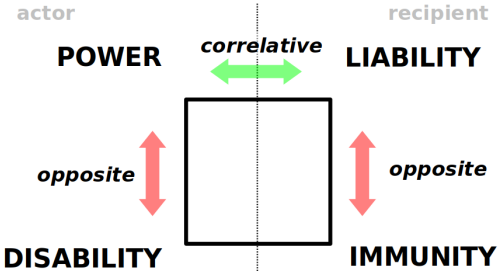


fundamental relation: **duty-claim**
between *duty holder* and *claimant*

Hohfeld's fundamental legal conceptions

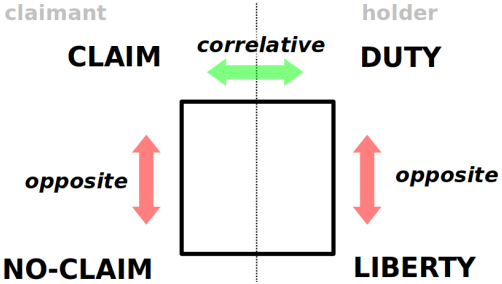


fundamental relation: **duty-claim**
between *duty holder* and *claimant*

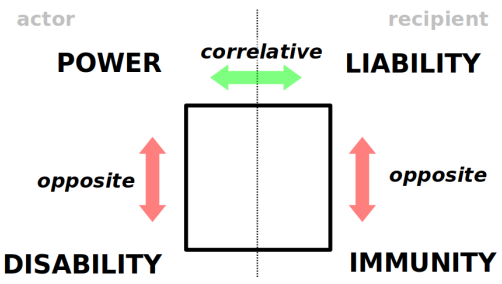


fundamental relation: **power-liability**
between *actor* and *recipient*

Hohfeld's fundamental legal conceptions



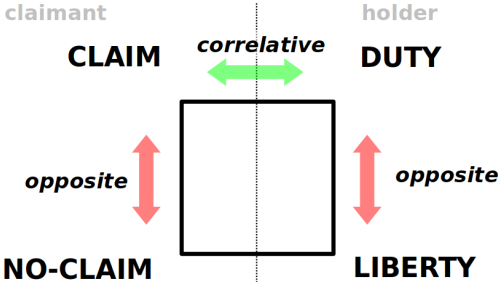
fundamental relation: **duty-claim**
between *duty holder* and *claimant*



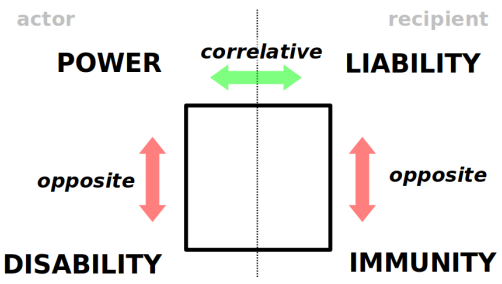
fundamental relation: **power-liability**
between *actor* and *recipient*

What does the result of interpretation look like?

Hohfeld's fundamental legal conceptions



fundamental relation: **duty-claim**
between *duty holder* and *claimant*



fundamental relation: **power-liability**
between *actor* and *recipient*

What does the result of interpretation look like?

How do we write down an interpretation formally?

Formal Language for the Interpretation of Norms (FLINT)

Robert van Doesburg / Tijs van der Storm / eFLINT

Commonalities

- Judgements characterize the relevant sub-set of the world
  - certain facts are postulated (to hold true or false)
  - other facts are derived (from other judgements)
- Transition rules determine the availability of actions and their effects

Formal Language for the Interpretation of Norms (FLINT)

Robert van Doesburg / Tijs van der Storm / eFLINT

Commonalities

- Judgements characterize the relevant sub-set of the world
  - certain facts are postulated (to hold true or false)
  - other facts are derived (from other judgements)
- Transition rules determine the availability of actions and their effects

Challenges

- Language design: appeal, scope, fit-for-purpose ...
- Policy design: consistency, composition, qualification ...
- Policy analysis: exploration, testing, verification, reasoning, planning ...
- System compliance: testing, verification, “by construction” ...

Language design - overview

1. World: values, types, expressions
2. Norms: duties, acts, transitions
3. Pragmatics: refinement, scripts, testing

Facts

Fact-type declarations associate a type with a fact identifier:

```
1 Fact citizen
2 Fact candidate       Identified by Atom
3 Fact administrator   Identified by Atom
4 Fact voter           Identified by citizen
5 Fact winner          Identified by candidate
6 Fact vote            Identified by (voter * candidate)
```

Type expressions

Types are essentially record-types:

$$
\begin{array}{lllll}
x \in & \texttt{vars} & ::= & \ldots \\
s \in & \texttt{atoms} & ::= & \ldots \\
i \in & \mathbb{Z} & ::= & \ldots \\
\tau \in & \texttt{types} & ::= & \texttt{atoms} \\
 & & | & \mathit{atom\_set}(s_1, \ldots, s_n) \\
 & & | & \mathbb{Z} \\
 & & | & \mathit{int\_set}(i_1, \ldots, i_n) \\
 & & | & \mathit{fields}(x_1, \ldots, x_n)
\end{array}
$$

- Field names are variables (possibly decorated fact identifiers)

Instances

```
1  Alice
2  7
3
4  Alice:citizen
5  Chloe:candidate
6  Admin:administrator
7
8  (Alice:citizen):voter
9
10 ((Alice:citizen):voter, Chloe:candidate):vote
```

example instances

Instances

```
1  Alice
2  7
3
4  Alice:citizen
5  Chloe:candidate
6  Admin:administrator
7
8  (Alice:citizen):voter
9
10 ((Alice:citizen):voter, Chloe:candidate):vote
```

example instances

The state of the world at any particular moment is a set of instances σ, containing those instances that *hold true* at that moment

Expressions

- Expressions evaluate to atoms, integers, Booleans or instances of fact-types

```
1  citizen
2  citizen(Alice)
3
4  voter(citizen(Alice))
5  voter(Alice)
6  voter(citizen = citizen(Alice))
7
8  vote(voter(Alice), Chloe)
9  vote(voter = voter(Alice), candidate = Chloe)
10 vote(candidate = Chloe, voter = voter(Alice))
11
12 vote(voter = voter(Alice))
13 vote(candidate = candidate, voter = voter(Alice))
14 vote()
```

variables and constructors

Operators

```
1  Holds(voter(Alice))
2
3  vote[voter]
4  vote[candidate]
5
6  vote[candidate] When Holds(vote)
7  vote[candidate] When vote
```

operators

Quantifiers and aggregators

Quantifiers bind variables to all instances of the variable's type:

```
1 (Exists candidate : vote(voter(Alice), candidate))
2
3 (Forall citizen : vote(voter(citizen), Chloe))
```

Foreach can only be used in combination with an *aggregator*:

```
1 Count(Foreach vote : vote When Holds(vote) && vote[candidate] = candidate)
```

Derived facts

Derivation expression as a predicate (type-components are bound):

```
1 Fact has voted Identified by voter
2   Holds when (Exists candidate : vote(voter,candidate))
```

```
1 Predicate vote concluded When (Exists candidate : winner(candidate))
2 Predicate voters done  When (Forall citizen : !voter() || has voted(voter()))
```

Derived facts

Derivation expression as a predicate (type-components are bound):

```
1 Fact has voted Identified by voter
2   Holds when (Exists candidate : vote(voter,candidate))
```

```
1 Predicate vote concluded When (Exists candidate : winner(candidate))
2 Predicate voters done  When (Forall citizen : !voter() || has voted(voter()))
```

Derivation expression computes the set of instances that hold true:

```
1 Fact number of votes Identified by Int
2   Derived from Count(Foreach vote : vote When Holds(vote))
```

Derived facts

Derivation expression as a predicate (type-components are bound):

```
1 Fact has voted Identified by voter
2   Holds when (Exists candidate : vote(voter,candidate))
```

```
1 Predicate vote concluded When (Exists candidate : winner(candidate))
2 Predicate voters done  When (Forall citizen : !voter() || has voted(voter()))
```

Derivation expression computes the set of instances that hold true:

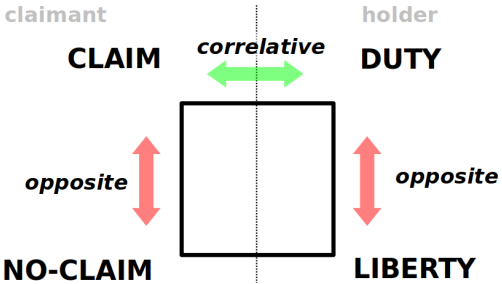
```
1 Fact number of votes Identified by Int
2   Derived from Count(Foreach vote : vote When Holds(vote))
```

- Derived facts cannot be postulated

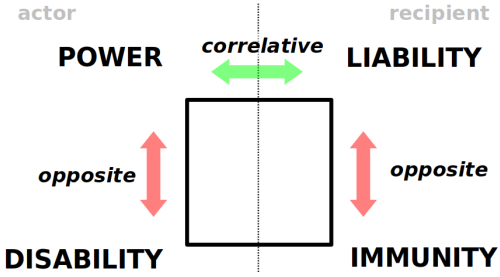
Language design - overview

1. World: values, types, expressions
2. Norms: duties, acts, transitions
3. Pragmatics: refinement, scripts, testing

Recall Hohfeld's conceptions



fundamental relation: **duty-claim**
between *duty holder* and *claimant*



fundamental relation: **power-liability**
between *actor* and *recipient*

How do we write down an interpretation formally?

Duties

A duty indicate that its holder ought to perform some action:

```
1 Duty cast vote duty Holder voter Claimant administrator
```

- A duty-type declaration is a fact-type declaration with a record-type

Acts

Actions modify the world by adding or removing instances from σ:

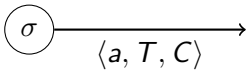
```
1  Act cast vote
2    Actor voter
3    Recipient administrator
4    Related to candidate
5    Conditioned by voter && !has voted()
6    Creates vote()
7    Terminates cast vote duty()
```

- An act-type declaration is a fact-type declaration with a record-type

Transitions

A transition is a state σ, an instance a of an act, and the sets T and C of instances terminated and created by the act, if and only if:

1. a holds true in σ
2. the pre-condition of a holds in σ
3. T is the result of evaluating the terminating post-conditions of a in σ
4. C is the result of evaluating the creating post-conditions of a in σ

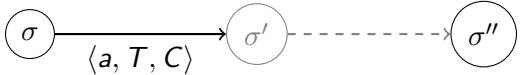


More acts

Derived facts may have to be recomputed after an action is performed:

```
1 Act cast vote
2   Actor voter
3   Recipient administrator
4   Related to candidate
5   Conditioned by voter && !has voted()
6   Creates vote()
7   Terminates cast vote duty()
```

- σ' may be incomplete and inconsistent w.r.t. derivation expressions



Completing the example (1)

```
1  Act enable vote
2    Actor administrator
3    Recipient citizen
4    Conditioned by !voter() && !vote concluded()
5    Creates voter(),
6           cast vote duty(voter = voter()),
7           (Foreach candidate : cast vote(voter = voter()))
```

Completing the example (2)

Placeholders can be introduced for types:

```
Placeholder other candidate For candidate
```

```
Act declare winner
  Actor administrator
  Recipient candidate
  Conditioned by
      !vote concluded()
   && voters done()
   && (Forall other candidate :
         Count(Foreach vote : vote[voter]
                 When vote && vote[candidate] == other candidate) <
         Count(Foreach vote : vote[voter]
                 When vote && vote[candidate] == candidate)
       When other candidate != candidate)
  Creates winner(candidate)
```

Language design - overview

1. World: values, types, expressions
2. Norms: duties, acts, transitions
3. Pragmatics: refinement, scripts, testing

Refinement

A refinement of a policy description replaces all simple, infinite types with finite types:

```
1 Fact citizen       Identified by [John, Frank, Peter, Chloe, Hannah]
2 Fact candidate     Identified by [Mary, David]
3 Fact administrator Identified by Admin
```

and also identifies an initial state (implicit Foreach):

```
1 administrator.
2 citizen.
3 candidate.
4 declare winner(Admin, candidate).
5 enable vote(Admin, citizen).
```

- A refinement enables exploring the reachable states manually

Scripts

- Scripts are basic programs for stepping through reachability graphs

Scripts

- Scripts are basic programs for stepping through reachability graphs

Action call !<EXPR>. evaluates to an enabled act and executes it (or fails)

Scripts

- Scripts are basic programs for stepping through reachability graphs

Action call !<EXPR>. evaluates to an enabled act and executes it (or fails)

Query ?<EXPR>. fails if expression does not evaluate to True in the current state

Scripts - positive test

```
1 !enable vote(citizen = John).
2 !enable vote(citizen = Frank).
3 !enable vote(citizen = Peter).
4 !cast vote(voter = voter(John), candidate = Mary).
5 !cast vote(voter = voter(Frank), candidate = Mary).
6 !cast vote(voter = voter(Peter), candidate = David).
7 !declare winner().
8 ?winner(Mary).
9 ?(Forall candidate : !winner() When candidate != Mary).
```

Scripts - negative test

```
1 !enable vote(citizen = John).
2 !enable vote(citizen = Frank).
3 !enable vote(citizen = Peter).
4 !cast vote(voter = voter(Frank), candidate = Mary).
5 !cast vote(voter = voter(Peter), candidate = David).
6 !enable vote(citizen = Hannah).
7 !cast vote(voter = voter(Hannah), candidate = David).
8 !declare winner().
```

Scripts - negative test 2

```
1 !enable vote(citizen = John).
2 !enable vote(citizen = Frank).
3 !enable vote(citizen = Peter).
4 !cast vote(voter = voter(Frank), candidate = Mary).
5 !cast vote(voter = voter(Peter), candidate = David).
6 !enable vote(citizen = Hannah).
7 !cast vote(voter = voter(Hannah), candidate = David).
8 !cast vote(voter = voter(John), candidate = Mary).
9 !declare winner().
```

Language design - overview

1. World: values, types, expressions
2. Norms: duties, acts, transitions
3. Pragmatics: refinement, scripts, testing

Reflection

Curb your enthusiasm Thomas...

Challenges

- Language design: appeal, scope, fit-for-purpose ...
- Policy design: consistency, composition, qualification ...
- Policy analysis: exploration, testing, verification, reasoning, planning ...
- System compliance: testing, verification, "by construction" ...

```
public void register_voter(String name) {
  registered_voters.add(name);
  ScriptGen.add_actor(name, "citizen"); // qualification
  ScriptGen.enable_vote(name);
}

public void register_candidate(String name) {
  vote_count.put(name,0);
  ScriptGen.add_actor(name, "candidate"); // qualification
}

public void vote(String voter, String candidate) {
  vote_count.put(candidate, vote_count.getOrDefault(candidate,0) + 1);
  ScriptGen.cast_vote(voter, candidate); // qualification
}

public String find_winner() { // returns null if there is no winner
  String winner = null;
  int winning_votes = 0;
  for (Entry<String,Integer> entry : vote_count.entrySet()) {
    if(entry.getValue() > winning_votes) {
      winner = entry.getKey();
      winning_votes = entry.getValue();
    }
  }
  if (winner != null) { // qualification
    ScriptGen.declare_winner(winner);
  }
  return winner;
}
```

eFLINT - A DSL for Testing Normative Specifications

L. Thomas van Binsbergen
Centrum Wiskunde & Informatica

22 November, 2019