Part III

Appendix

C. Prem, *The Theory of Credit Contracts*,
https://doi.org/10.1007/978-3-658-29362-8

A Software code and documentation

A.1 General concepts

This section introduces general concepts used throughout the algorithm. It is necessary to understand them in order to understand the subsequent documentation.

A.1.1 Information Analysis

First of all we have a number of players `nrOfPlayers`. Note that the number of players is a constant within a game. But since we want to write a piece of software that works for an arbitrary numbers of players, it is a symbol in our code. Each player has a unique `playerKey`.

A single action of a player is a tuple with three components:

`actionName`	can be any description of the action
`actionRepayment`	the repayment associated with this action in the repayment game
`actionMessage`	a potential message sent to the principal (or empty String)

A player's `actionSet` is an ordered list of actions. An action can be addressed unequivocally by its `actionSetKey`, which is its position in the ordered `actionSet`.

An `ability` of a player specifies a upper bound for a player's `actionRepayment`.

Punishments reflect the influence on a borrower's payoff exerted by the lender. A `punishmentRule` consists of two pieces of information:

`punishmentValue`	the value by which a borrower's payoff is reduced by the punishment (the intensity of the punishment)
`punishmentCondition`	a formal condition under which a punishment is to be executed

An ordered list of `punishmentRules` is called a `punishmentFunction`. The order reflects the hierarchy of the rules; the first rule whose `punishmentCondition` yields true is applied.

C. Prem, *The Theory of Credit Contracts*,
https://doi.org/10.1007/978-3-658-29362-8

Sanctions reflect the influence on a borrower's payoff exerted by other borrowers. A `sanctionRule` consists of two pieces of information:

`sanctionValue`	the value by which a borrower's payoff is reduced by the sanction (the intensity of the sanction)
`sanctionCondition`	a formal condition under which a social sanction is to be executed

An ordered list of `sanctionRules` is called a `sanctionFunction`. The order reflects the hierarchy of the rules; the first rule whose `sanctionCondition` yields true is applied.

`Payoff`	a number describing the payoff that a player receives if a certain action combination is played
`Feasibility`	a boolean value represting the information if a certain action combination can be played, i.e. is feasible

A.1.2 Template types

Furthermore we have some sort of abstract compound data types that refer to different arrangement of the more basic data types introduces above. I call these 'abstract' because they are like abstract compound data but the entries of which the compound data type consists is not specified yet.

`xxxSet`	a list of all xxx of a particular player (e.g. actionSet)
`xxxCombination`	one xxx for all players of the game (e.g. actionCombination)
`xxxMatrix`	one xxx for all actionCombinations (e.g. payoffMatrix)
`xxxTos`	one xxx for all abilityCombinations (e.g. feasibilityMatrixTos), read "tabular of shapes"
`xxxKey`	the position of an entry in the list xxx (e.g. actionSetKey, actionCombinationKey)

A.1.3 Sign convention

Repayments, punishments, sanctions, payoffs are denoted from the perspective of a borrower. That is cash flows reducing a borrower's wealth have a negative sign (flowing away from the borrower) and vice versa.

A.1.4 Global variables

These variables are *global* variables. They are specified in the game definition block.

`n` number of players

`actionSet[`*i*`]`
actionSet of player *i*

`punishmentFunction[abilityCombination, actionCombination, playerKey]`
punishment strategy of the lender (loss-flow: borrower → deadweight-loss)

`sanctionFunction[abilityCombination, actionCombination, playerKey]`
rules for social sanctions (loss-flow: borrower → deadweight-loss)

`rewardFunction[abilityCombination, actionCombination, playerKey]`
reward strategy of the lender (transfer-flow: lender → borrower)

`payoffLayers[abilityCombination, actionCombination, playerKey]`
combination of all of the borrower's payoff layers (apart from repayment)

`$Assumptions`
assumptions about the symbols for repayment, punishment, sanctions, rewards

A.1.5 Further Comments

The algorithm is implemented using Mathematica 8.0 syntax only.

The algorithm is realized using the functional programming paradigm. By this approach we avoid introducing many additional variables that would be necessary under the procedural programming paradigm in order to check dimensions and control loops. The functional programming paradigm allows to not bother about dimensions at all, such that the algorithm works for games of arbitrary size (number of players, number of actions) without further modification. Only the depth of arrays becomes relevant from time to time to ensure that the higher-order functions `Map[]` and `MapIndexed[]` work at the right level of a list of lists of lists ....

Notice that some parts of a repayment game naturally can be captured by maps. E.g. we could have functions that assign to each action from the actionSet of a player the repayment amount associated with each action. Likewise for messages associated with a certain action. However, in the following code, I make use of the fact, that each map can be seen as a relation: In technical terms a function can be described as a relation (a set of ordered pairs) that is left-total and right-unique. So instead of defining a repayment function or a message function I represent the same information by defining actions as tuples consisting of (`actionName`, `actionRepayment`, `actionMessage`).

I adopt the following naming conventions:

- Functions commence with `fn...` and procede in upper camel case.
- Functions returning a single atomic boolean value end with `Q` (according to the Mathematica convention).
- Functions that are meant to be applied to individual entries of lists and spring into action via mapping functions like `Map[]` or `MapIndexed[]` are called `fnRobots` (see functional programming paradigm approach, explained above).

Functions are ordered from auxilary functions to combining functions, as Mathematica has to read the auxilary functions first before other functions can call them. Unfortunately this puts the most important functions last.

A.2 Code block: Game construction

A.2.1 Problem Statement

Write a piece of software that helps to define a repayment game (which in turn can be solved by the function `fnSolveGame`). Concretely the program should take as input: the number of players, the actions of each player, payoff layers and global assumptions. The output shall be – for each possible repayment ability-combination – a list of payoff matrices, one for each player, produced from the given rules and assumptions. Furthermore another output should be a feasbility matrix, i.e. the algorithm should automatically know what different ability-combinations can arise and construct the feasibility matrix for each situation.

A.2.2 Brief description of the algorithm

The code is split into several parts.

1. The first part provides auxilary functions working on actions. These function extract components of actions. They are useful to define punishmentConditions or sanctionConditions. They also help to display appropriate headings in TabularForm or MatrixForm of xxxMatrices.
2. The second part constructs an abilitySet for each player. The data type for abilities is simpler that that for actions as they don't have names and messages that come with it.
3. This part deals with xxxKey to xxx transformations. The xxxKeys are important to position entries in xxxMatrices at their correct position (e.g. the payoff under a certain actionCombination).
4. These functions basically measure the size of the game and define templates on which second order functions like Map[] oder MapIndexed[] can operate.
5. Here we construct the payoffMatrix for each player. The function payoffMatrix produces the payoffMatrix after which the function game produces the payoffMatrices of all players and combines them to a payoffMatrixCombination, i.e. the game.
6. The last part finally constructs the feasibilityMatrix that is characteristic for each abilityKeyCombination. What we do here is define a predicate fnFeasibleForPlayerQ which can test whether a certain actionKey is feasible given a certain abilityKey. Next we use this predicate to do something, namely fnDetermineInfeasibleActionKeys collects all infeasible actionKeys of a player for a given abilityKey. Finally, we need to use this list of infeasible actionKeys to find the positions in the feasibilityMatrix that have to be changed to False; the function feasibilityMatrix does this replacement for all players and returns the ready cooked feasibilityMatrix.

A.2.3 Source Code

```
1  (* ================ *)
2  (* DATA DEFINITIONS *)
3
4  (* ------- *)
5  (* PLAYERS *)
6
7  (* nrOfPlayers *)
8  (* DEFINITION        is positive Integer *)
9  (* INTERPRETATION    the total number of players *)
10 (* EXAMPLES          n=2; n=5; *)
11 (* TEMPLATE          fnForN[n_]:= ... n ; *)
12 (* RULES USED        atomic non-distinct: positive Integer *)
13
14 (* playerKey *)
15 (* DEFINITION        is Integer [1, nrOfPlayers] *)
16 (* INTERPRETATION    the number of a certain player; also represents the order in which players move in sequential games *)
17 (* EXAMPLES          playerNrMax=1; playerNrMoritz=2; *)
18 (* TEMPLATE          fnForPlayerNr[playerKey_]:= ... playerKey ; *)
19 (* RULES USED        atomic non-distinct: Integer [1, nrOfPlayers] *)
20
21 (* ------------------- *)
22 (* ACTIONS & ABILITIES *)
23
24 (* actionName *)
25 (* DEFINITION        is String *)
26 (* INTERPRETATION    the name of an action (used as heading in payoffMatrix) *)
27 (* EXAMPLE           actionName1="repay both", actionName2="default-n", actionName3="-2R&n" *)
28 (* TEMPLATE          fnForActionName[actionName_]:= ... actionName; *)
29 (* RULES USED        atomic non-distinct: String *)
30
31 (* actionRepayment *)
32 (* DEFINITION        is negative Real number, or symbol representing a Real number *)
33 (* INTERPRETATION    the repayment associated with a certain action, i.e. the amount by which a certain repayment reduces the payoff of a borrower *)
34 (* EXAMPLES          actionRepayment1=-2R, actionRepayment2=-1R, actionRepayment3=0 *)
35 (* TEMPLATE          fnForActionRepayment[actionRepayment_]:= ... actionRepayment; *)
36 (* RULES USED        atomic non-distinct: Number *)
37
38 (* actionMessage *)
39 (* DEFINITION        is String *)
40 (* INTERPRETATION    a messages associated with a certain action, i.e. a message that is sent to the lender *)
41 (* EXAMPLES          actionMessage1="n", actionMessage2="no", actionMessage3="yes", actionMessage4="" *)
42 (* TEMPLATE          fnForActionMessage[actionMessage_]:= ... actionMessage; *)
43 (* RULES USED        atomic non-distinct: String *)
44
45 (* action *)
46 (* DEFINITION        is list of {actionName,actionRepayment,actionMessage} *)
47 (* INTERPRETATION    an action a player can choose to play in the repayment game *)
48 (* EXAMPLE           action1={"repay both",-2R,""}, action2={"default",0,"yes"} *)
49 (* TEMPLATE          fnForAction[action_]:= ... action [[1]], action [[2]], action [[3]]; *)
50 (* RULES USED        compound: 3 fields, references: actionName is actionName, actionRepayment is actionRepayment, actionMessage is actionMessage *)
51
```

```
52 (* nrOfActions *)
53 (* DEFINITION          is positive Integer *)
54 (* INTERPRETATION     the total number of a player's actions, i.e. the length of his actionSet *)
55 (* EXAMPLES           n=2; n=5; *)
56 (* TEMPLATE           fnForN[n_]:= ... n ; *)
57 (* RULES USED         atomic non-distinct: positive Integer *)
58
59 (* actionSet *)
60 (* DEFINITION          is list of actions *)
61 (* INTERPRETATION     all actions a player can ever play *)
62 (* EXAMPLE            actionSet1 ={{"opt1",0,"n"},{"opt2",0,"y"},{"opt3",-R,"y"},{"opt4",-2R,"y"}} *)
63 (* TEMPLATE           fnForActionSet[ actionSet_ ]:= ... actionSet [[i]] *)
64 (* RULES USED         compound: as many fields as a player has actions, references: action *)
65
66 (* actionSetKey *)
67 (* DEFINITION          is Integer *)
68 (* INTERPRETATION     the position of an action in a player's actionSet *)
69 (* EXAMPLES           actionSetKeyRepay=1; actionSetKeyDefault=2 *)
70 (* TEMPLATE           fnForActionSetKey[ actionSetKey_]:= ... actionSetKey ; *)
71 (* RULES USED         atomic non-distinct: Integer *)
72
73 (* actionSetKeyCombination *)
74 (* DEFINITION          is List of positive Integers *)
75 (* INTERPRETATION     an action combination in the game, the first entry denotes the actionSetKey of the first player etc. *)
76 (* EXAMPLES           actionSetKeyCombinationA={1,1,1}; actionSetKeyCombinationB={1,1,2} *)
77 (* TEMPLATE           fnForActionSetKeyCombination[actionSetKeyCombination_]:= ... actionSetKeyCombination [[...]] ; *)
78 (* RULES USED         compound: nrOfPlayers fields *)
79
80 (* ability *)
81 (* DEFINITION          is negative Real number, or symbol representing a negative Real number *)
82 (* INTERPRETATION     the maximum repayment a player can pay in a certain situation *)
83 (* EXAMPLE            ability1 =-2R, ability2=-1R, ability3=-10, ability4=0 *)
84 (* TEMPLATE           fnForAbility [ ability_ ]:= ... ability ; *)
85 (* RULES USED         atomic non-distinct: Real number *)
86
87 (* abilitySet *)
88 (* DEFINITION          is list of abilities *)
89 (* INTERPRETATION     all relevant abilities that can happen to a player *)
90 (* EXAMPLE            abilitySet1 ={0,-1R,-2R}, abilitySet2={0,-2R} *)
91 (* TEMPLATE           fnForAbilitySet [ abilitySet_ ]:= ... abilitySet [[i]] *)
92 (* RULES USED         compound: as many fields as a player has abilities, references: ability *)
93
94 (* nrOfAbilities *)
95 (* DEFINITION          is positive Integer *)
96 (* INTERPRETATION     the total number of a player's abilities, i.e. the length of his abilitySet *)
97 (* EXAMPLES           n=2; n=5; *)
98 (* TEMPLATE           fnForN[n_]:= ... n ; *)
99 (* RULES USED         atomic non-distinct: positive Integer *)
100
101 (* abilitySetKey *)
102 (* DEFINITION          is Integer *)
103 (* INTERPRETATION     the position of an ability in a player's abilitySet *)
104 (* EXAMPLES           abilitySetKeyRepay=1; abilitySetKeyDefault =2 *)
```

```
105 (* TEMPLATE          fnForAbilitySetKey[ abilitySetKey_ ]:= ... abilitySetKey ; *)
106 (* RULES USED       atomic non-distinct: Integer *)
107
108 (* abilitySetKeyCombination *)
109 (* DEFINITION       is List of positive Integers *)
110 (* INTERPRETATION   an ability combination in the game, the first entry denotes the abilitySetKey of the first player etc. *)
111 (* EXAMPLES         abilitySetKeyCombinationA={1,1,1}; abilitySetKeyCombinationB={1,1,2} *)
112 (* TEMPLATE         fnForAbilitySetKeyCombination[ abilitySetKeyCombination_]:= ... abilitySetKeyCombination [[...]] ; *)
113 (* RULES USED       compound: nrOfPlayers fields *)
114
115
116 (* ---------------------------------- *)
117 (* PUNISHMENTS & SANCTIONS *)
118
119 (* punishmentValue *)
120 (* DEFINITION       is (negative/positive) Real *)
121 (* INTERPRETATION   the amount by which a player's payoff is decreased/increased by the lender *)
122 (* EXAMPLE          punishmentValue1=-p, punishmentValue2=+e *)
123 (* TEMPLATE         fnForPunishmentValue[punishmentValue_]:= ... punishmentValue *)
124 (* RULES USED       atomic non-distinct: Real number *)
125
126 (* sanctionValue *)
127 (* INTERPRETATION   the amount by which a player's payoff is decreased/increased by other borrowers *)
128 (*                  likewise *)
129
130 (* punishmentCondition *)
131 (* DEFINITION       is a predicate (i.e. a function that returns a Boolean) with signature {abilitySetKeyCombination,actionSetKeyCombination,playerKey} --> Boolean *)
132 (* INTERPRETATION   the condition under that decides whether a certain punishment is applicable, i.e. under which circumstances the punishment gets executed *)
133 (* EXAMPLE          Plus@@(fnGetActionRepayment/@actionCombination)<=-n R, Union[fnGetActionMessage/@actionCombination]=={"y"} *)
134 (* TEMPLATE         fnForPunishmentCondition[punishmentCondition_]:= If[punishmentCondition, ...] *)
135 (* RULES USED       the data type here should be 'function', as the information here is captured by a function *)
136
137 (* sanctionCondition *)
138 (*                  likewise *)
139
140 (* punishmentRule *)
141 (* DEFINITION       is a list of {punishmentValue, punishmentRule} *)
142 (* INTERPRETATION   the amount of a punishment and the condition under which the punishment gets executed *)
143 (* EXAMPLE          punishmentRule={0,Union[fnGetActionMessage/@actionCombination]=={"y"}} *)
144 (* TEMPLATE         fnForPunishmentRule[punishmentRule_]:= ... If[ punishmentRule[[2]],punishmentRule[[1]] ] *)
145 (* RULES USED       compound: 2 fields, references: punishmentValue is punishmentValue, punishmentCondition is punishmentCondition *)
146
147 (* sanctionRule *)
148 (*                  likewise *)
149
150 (* punishmentFunction *)
151 (* DEFINITION       an (ordered) list of punishmentRules *)
152 (* INTERPRETATION   specification of the complete punishment regimen, the first punishmentRule whose punishmentCondition yields true is applied *)
153 (* EXAMPLE          punishmentFunction1={{0, Union[fnGetActionMessage/@actionCombination]=={"y"}} (*Rule1*),{+2R+e, Plus@@(fnGetActionRepayment/@actionCombination)<=-n R}}(*Rule2*)} *)
154 (* TEMPLATE         fnForPunishmentFunction[punishmentFunction_]:= ... Piecewise[punishmentFunction]; *)
155 (* RULES USED       compound: arbitrary many fields, reference: punishmentRule *)
156
157 (* sanctionFunction *)
```

```
158 (*                          likewise *)
159
160
161 (* ---------------------------- *)
162 (* PAYOFF & FEASBILITY *)
163
164 (* payoff *)
165 (* DEFINITION          is Real (or a Mathematica expression representing a Real number) *)
166 (* INTERPRETATION      the payoff a player receives if a certain action combination is played *)
167 (* EXAMPLES            payoffCombination111Player1=7; payoffCombination111Player2=-2R-p *)
168 (* TEMPLATE            fnForPayoff[ payoff_ ]:= ... payoff ; *)
169 (* RULES USED          atomic non-distinct: Real number *)
170
171 (*  feasibility  *)
172 (* DEFINITION          is Boolean *)
173 (* INTERPRETATION      true if an action combination is feasible , false otherwise *)
174 (* EXAMPLES            feasibilityCombination111 =True; pfeasibilityCombination112 =False *)
175 (* TEMPLATE            fnForFeasibility [ feasibility_ ]:= ... feasbility ; *)
176 (* RULES USED          atomic non-distinct: Boolean *)
177
178
179 (* ------------------- *)
180 (* TEMPLATE TYPES *)
181
182 (* xxxSet *)
183 (* DEFINITION           is List of arbitrary size with entries xxx (of arbitrary type) *)
184 (* INTERPRETATION       all xxx of a particular player *)
185 (* EXAMPLES             actionSet , repaymentSet, messageSet, abilitySet *)
186 (* TEMPLATE             fnForxxxSet[ xxxSet_ ]:= ... xxxSet[[i]]; *)
187 (* RULES USED           compound: arbitrary many fields *)
188
189 (* xxxCombination *)
190 (* DEFINITION           is a List of length nrOfPlayers (i.e. one entry for each player) with entries xxx (of arbitrary type) *)
191 (* INTERPRETATION       first position =xxx of first player, second position =xxx of second player, etc. *)
192 (* EXAMPLES             actionCombination, actionSetKeyCombination, abilityCombination , payoffCombination, messageCombination, nrOfActionsCombination, payoffMatrixCombination *)
193 (* TEMPLATE             fnForxxxCombination[xxxCombination_]:= ... xxxCombination[[playerKey]];*)
194 (* RULES USED           compound: nrOfPlayer many fields *)
195
196 (* xxxMatrix *)
197 (* DEFINITION          is List of Lists of Lists ... of dimensions nrOfActionsCombination, with content xxx (of arbitrary type) *)
198 (* INTERPRETATION      one entry for each actionCombination, a position is xxx belonging to this actionCombination *)
199 (* EXAMPLES            payoffMatrix , feasibilityMatrix *)
200 (* TEMPLATE            fnForMatrix[ xxxMatrix_]:= ... xxxMatrix[[actionCombination]] ; *)
201 (* RULES USED          compound: the product of nrOfActionsCombination many fields (the dimensions of the game) *)
202
203 (* xxxTos *)
204 (* DEFINITION          is List of Lists of Lists ... of dimensions nrOfAbilitiesCombination , with content xxx (of arbitrary type) *)
205 (* INTERPRETATION      one entry for each abilityCombination (each shape of the game), a position is xxx belonging to this abilityCombination *)
206 (* EXAMPLES            payoffMatrixTos , feasibilityMatrixTos *)
207 (* TEMPLATE            fnForxxxTos[xxxTos]:= ... xxxTos[[ abilityCombination ]] ; *)
208 (* RULES USED          compound: the product of nrOfAbilitesCombination many fields *)
209
210 (* xxxKey *)
```

```
(* DEFINTION          is positive Integer *)
(* INTERPRETATION    the position of an entry in the list xxx *)
(* EXAMPLES          actionSetKey, abilitySetKey *)
(* TEMPLATE          fnForxxxKey[xxxKey]:= ... xxx[[xxxKey]] ; *)
(* RULES USED        atomic non-distinct: positive Integer *)

(* nrOfxxx *)
(* DEFINTION          is positive Integer *)
(* INTERPRETATION    the number of xxx *)
(* EXAMPLES          nrOfPlayers, nrOfActions *)
(* TEMPLATE          fnForNrOfxxx[nrOfxxx]:= ... nrOfxxx; *)
(* RULES USED        atomic non-distinct: positive Integer *)

(* game = payoffMatrixCombination *)
(* DEFINITION        is List of payoffMatrices *)
(* INTERPRETATION    the first payoffMatrix describes the payoffs of the first player ... etc. *)
(* EXAMPLES          game1={{2 3
5 1

,6  2.1
13  2

}; *)
(* TEMPLATE          fnForGame[game_]:=... game [[...]] (*payoffMatrix*); *)
(* RULES USED        compound: nrOfPlayers fields ; xxxMatrix; payoff; *)



(* ========== *)
(* FUNCTIONS *)


(* ------------------1------------------ *)
(* Auxilary functions for actions *)

(* fnGetAction ... *)
(* SIGNATURE         [action_] --> actionName/actionRepayment/actionMessage *)
(* PURPOSE           extract name/repayment/message of an action *)
(* TESTS *)
fnGetActionName[{"repay",-R,"no"}]=="repay";
(* FUNCTION BODIES *)
fnGetActionName[action_]:= action [[1]]; (* select actionName of an action *)
fnGetActionRepayment[action_]:= action [[2]] ;(* select actionRepayment of an action *)
fnGetActionMessage[action_]:= action [[3]] ;(* select actionMessage of an action *)
(* USAGE:
1) these functions can be applied to all lists of actions entrywise by using /@ (e.g. actionSets or actionCombinations are lists of actions).
 2) (fn .../@ actionCombination)[[playerKey]] transforms the whole actionCombination using the function fn ... and then picks out playerNr's part.
          It is equivalent to fn ... @(actionCombination[[playerKey]]) which picks playerNr's action first and then transforms it and is thus less time-consuming.*)

(* ---------------2---------------- *)
(* Construction of abilitySets *)

(* abilitySet *)
```

```
(* SIGNATURE          [playerKey_] --> abilitySet *)
(* PURPOSE            construction of a player's abilitySet, i.e. all relevant abilities considering his repaymentSet *)
(* TESTS *)
(* would interfere with the global variables actionSets[i] *)
(* FUNCTION BODY *)
abilitySet[playerKey_]:= DeleteDuplicates[fnGetActionRepayment/@actionSet[playerKey]]
(* COMMENTS
Since it is a function it should actually be called fnAbilitySet. In analogy to the data type actionSets I nevertheless dropped the fn... prefix. Regard each ability as an additional nature-player with
    actionSet = abilitySet. However, the player does not optimize his behaviour, has no payoffMatrix and is instead steered by chance. *)

(* ------------------3---------------------- *)
(* Key to Action / Ability transformations *)

(* fnKeyToAction / fnKeyToAbility respectively *)
(* SIGNATURE          [playerKey_, actionKey_] --> action *)
(* PURPOSE            get action belonging to a particular actionKey *)
(* TESTS *)
(* would interfere with global variables actionSet[i] *)
(* FUNCTION BODY *)
fnKeyToAction[playerKey_,actionKey_]:= actionSet[playerKey][[actionKey]]; (* actionKey to action *)
fnKeyToAbility[playerKey_, abilityKey_]:= abilitySet[playerKey][[abilityKey]]; (* abilityKey to ability *)

(* fnKeyToActionCombination / fnKeyToAbilityCombination respectively *)
(* SIGNATURE          [actionKeyCombination_] --> actionCombination *)
(* PURPOSE            transform an actionKeyCombination into the corresponding actionCombination *)
(* TESTS *)
(* would interfere with the global variables actionSets[i] *)
(* FUNCTION BODY *)
fnKeyToActionCombination[actionKeyCombination_]:=MapIndexed[fnKeyToAction[Sequence@@#2,#1]&,actionKeyCombination,{1}]; (* actionKeyCombination to actionCombination *)
fnKeyToAbilityCombination[abilityKeyCombination_]:=MapIndexed[fnKeyToAbility[Sequence@@#2,#1]&,abilityKeyCombination,{1}]; (* abilityKeyCombination to abilityCombination *)

(* ---------------4------------------- *)
(* Measuring the size of the game *)

(* nrOfActionsCombination / nrOfAbilitiesCombination respectivels *)
(* SIGNATURE          [] --> nrOfActionsCombination *)
(* PURPOSE            measure the number of actions of each player *)
(* TESTS *)
(* would interfere with the global variables actionSets[i] *)
(* FUNCTION BODY *)
nrOfActionsCombination:=Map[Length[actionSet[#]]&,Range[n],{1}] (* measuring the number of actions of each player, i.e. dimensions of xxxMatrix *)
nrOfAbilitiesCombination:=Map[Length[abilitySet[#]]&,Range[n],{1}]; (* measuring the number of abilities of each player, i.e. dimensions of xxxTos *)

(* emptyMatrix / emptyTos respectively *)
(* SIGNATURE          [] --> xxxMatrix *)
(* PURPOSE            construct a template needed for payoffMatrix construction *)
(* TESTS *)
(* ... *)
(* FUNCTION BODY *)
emptyMatrix:=Array[List,nrOfActionsCombination]; (* one entry for all possible actionCombinations, captures dimensions of xxxMatrix, i.e. size of the game *)
emptyTos:=Array[List,nrOfAbilitiesCombination]; (* one entry for all possible abilityCombinations, captures dimensions of xxxTos, i.e. all possible shapes of the repayment game *)
(* emptyCombination:=Range[n]; So simply write Range[n] to save variable names. *)

```

```
316 (* ----------5---------- *)
317 (* game construction *)
318
319 (* payoff *)
320 (* SIGNATURE           [abilityKeyCombination_, actionKeyCombination_, playerKey_] --> payoff *)
321 (* PURPOSE             calculate a player's payoff in a particular situation *)
322 (* TESTS *)
323 (* depend on global variables punishmentFunction, sanctionFunction, actionSets *)
324 (* FUNCTION BODY *)
325 fnPayoff[abilityKeyCombination_, actionKeyCombination_, playerKey_]:=Module[
326 {abilityCombination, actionCombination, payoff},
327 abilityCombination=fnKeyToAbilityCombination @abilityKeyCombination; (* get from abilityKeys to abilities, since punishmentFunction and sanctionFunction need these *)
328 actionCombination=fnKeyToActionCombination@actionKeyCombination; (* get from actionKeys to actions, since punishmentFunction and sanctionFunction need these *)
329 payoff=fnGetActionRepayment@(actionCombination[[playerKey]])
330 + payoffLayers[abilityCombination, actionCombination, playerKey]; (* calculate payoff *)
331 Return[payoff]; (* return calculated payoff *)
332 ];
333
334 (* payoffMatrix *)
335 (* SIGNATURE           [abilityKeyCombination_, playerKey_] --> payoffMatrix *)
336 (* PURPOSE             robot that constructs the payoff-matrix for a certain player in a certain shape (i.e. for a certain abilityCombination) *)
337 (* TESTS *)
338 (* ... *)
339 (* FUNCTION BODY *)
340 payoffMatrix[abilityKeyCombination_, playerKey_]:=Map[fnPayoff[abilityKeyCombination,#,playerKey]&,emptyMatrix,{n}];
341 (* COMMENTS
342 Since it is a function it should actually be called fnPayoffMatrix. In analogy to the data type payoffMatrix I nevertheless dropped the fn... prefix. *)
343
344 (* game *)
345 (* SIGNATURE           [abilityKeyCombination_] --> game (= payoffMatrixCombination) *)
346 (* PURPOSE             constructs the payoff-matrix for each player in a certain shape (i.e. for a certain abilityCombination) *)
347 (* TESTS *)
348 (* ... *)
349 (* FUNCTION BODY *)
350 game[abilityKeyCombination_]:=Map[payoffMatrix[abilityKeyCombination,#]&,Range[n],{1}]
351 (* COMMENTS
352 Since it is a function it should actually be called fnGame. In analogy to the data type game I nevertheless dropped the fn... prefix. *)
353
354
355 (* ----------6---------- *)
356 (* feasibilityMatrix construction *)
357
358 (* fnFeasibleForPlayerQ *)
359 (* SIGNATURE           [abilityKey_, actionKey_, playerKey_] --> feasibility *)
360 (* PURPOSE             test if the combination repaymentKey and abilityKey is possible for a certain player *)
361 (* TEST *)
362 (* ... *)
363 (* FUNCTION BODY *)
364 fnFeasibleForPlayerQ[abilityKey_, actionKey_, playerKey_]:=Refine[abilitySet[playerKey][[abilityKey]]<=fnGetActionRepayment@(actionSet[playerKey][[actionKey]])];
365
366 (* fnDetermineInfeasibleActionKeys *)
367 (* SIGNATURE           [abilityKey_, playerKey_] --> List of actionKeys *)
368 (* PURPOSE             get all infeasible actionKeys for a certain abilityKey *)
```

```
(* TESTS *)
(* ... *)
(* FUNCTION BODY *)
fnDetermineInfeasibleActionKeys[abilityKey_,playerKey_]:=Module[
{evaluateRobot, infeasibleActionKeys},
(* robot that checks actionKey for feasibility and denotes infeasible Keys *)
evaluateRobot[actionKey_]:=If[fnFeasibleForPlayerQ[abilityKey,actionKey,playerKey],True,actionKey];
(* apply robot to all existing actionKeys *)
infeasibleActionKeys=Map[evaluateRobot[#]&,Range[Length[actionSet[playerKey]]]];
(* delete True entries for the resulting list *)
infeasibleActionKeys=DeleteCases[infeasibleActionKeys,True];
(* return resulting list of infeasible actionKeys *)
Return[infeasibleActionKeys];
];

(* feasibilityMatrix *)
(* SIGNATURE [abilityKeyCombination_] --> feasibilityMatrix *)
(* PURPOSE construct the feasibilityMatrix under a certain abilityCombination, i.e. in a certain shape of the repayment game *)
(* TESTS *)
(* ... *)
(* FUNCTION BODY *)
feasibilityMatrix[abilityKeyCombination_]:=Module[
{feasibilityMatrix,evaluateRobot,fnCancelRobot},

(* STEP 1: Start out with a booleanMatrix with only True entries *)
feasibilityMatrix=ConstantArray[True,nrOfActionsCombination]; (* Template with True everywhere *)

(* STEP 2: *)
(* robot to cancel positions that are infeasible for a player *)
fnCancelRobot[playerKey_]:=Module[
{infeasibleActionKeys, infeasiblePositions},
(* Step A: Find infeasible actions for the player *)
infeasibleActionKeys=fnDetermineInfeasibleActionKeys[abilityKeyCombination[[playerKey]],playerKey];
(* Step B: Transform into infeasible positions *)
infeasiblePositions=Sequence@@ReplacePart[ConstantArray[All,n],playerKey->infeasibleActionKeys];
(* Step C: Replace infeasible positions *)
feasibilityMatrix[[infeasiblePositions]]=False;
];

(* STEP 3: apply fnCancelRobot for each playerKey *)
Map[fnCancelRobot[#]&,Range[n]]; (* excecute fnCancelRobot for every player *)
Return[feasibilityMatrix]; (* return feasibilityMatrix *)
];
(* COMMENTS
1) Since it is a function it should actually be called fnFeasibilityMatrix. In analogy to the data type feasibilityMatrices I nevertheless dropped the fn ... prefix.
2) By actionwise replacement we save computation effort and time. Instead of calling the fnFeasibleForPlayerQ[] function for every entry in the xxxMatrix i.e. length[actionSet]*length[actionSet]*... times,
   we need to call it only length[actionSet]+length[actionSet]+... times *)
```

A.3 Code block: Game solution

A.3.1 Problem statement

Write a software program that can solve discrete games and calculated the Nash-equilibria. More precisely is shall be able to compute pure Nash equilibria in a discrete, non-cooperatively, simultaneously played game. The function must take as input the payoff matrices of the players. In addition it should take a matrix (same dimensions as one payoff matrix) with boolean entries, telling which action combinations are feasible. The Nash-Equilibria shall only be computed based on feasible action combinations. The whole code block here shall be absolutely self-contained and independent from any global variable or functions used in other parts of the code. This makes it reusable outside the world of repayment games. Therefore, a "new" information analysis is to be performed.

A.3.2 Information Analysis

First of all we have a number of players `nrOfPlayers`. Each player has a unique `playerNr`.[116] Notice the number of player is a constant within a game; since we want to write a piece of software that works for arbitrary numbers of players it is in fact a variable in our code.

Each player has of course a set of `actions`, which are addressed by a (within a player's action set unique) `actionNr`. The choice of actions of all players is captured by an `actionCombination`. If we consider a certain action combination, we can have different information about it.

`Payoff`	a number describing the payoff a player receives if a certain action combination is played
`Feasibility`	a boolean value represting the information if a certain action combination can be played, i.e. is feasible
`Best Response`	a boolean value representing the information if a certain action combination is a best response for a certain player
`Nash Equilibrium`	a boolean value telling wether a certain action combination represents a non-cooperative Nash equilibrium
`Solution`	a string with three possible values, one marking an action combination as infeasible, one for marking a combination as Nash Equilibrium, and one for all other cases.

116 Unfortunately, this information is called `playerKey` in the other code blocks. But the term 'Key' is better as it is clearly an ordinal and avoids confusion with cardinal numbers.

xxxMatrix represents a compound data structure. This should be seen as being a new 'primitive' data type (although it is of course not a primitve). It simply is a matrix of adequate dimensions, i.e. the dimensions of the game. It does not specify what are the entries are, how they must be interpretated. It is basically a template which can be filled with different information. We will fill a game matrix with concrete atomic data types.

`Payoff-Matrix`	a game matrix of payoffs, telling the payoff a player receives for all action combinations
`Feasibility-Matrix`	a game matrix of boolean values, telling the feasibility of all action combinations
`BestResponse-Matrix`	a game matrix of boolean values, telling if an action combination is a best response for a certain player for all action combinations
`NashEq-Matrix`	a game matrix of boolean values, telling if an action combination is a non-cooperative Nash equilibrium for all action combinations
`Solution-Matrix`	a game matrix of solution strings, telling feasibility and equilibrium properties for each action combination

`Game` is a data type that combines the payoff-Matrices of all players.

A.3.3 Brief description of the algorithm

The difficult task is to transform the Payoff-Matrix of each player into a BestResponse-Matrix (fnMarkBestResponse) whilst excluding infeasible actionCombinations. Once that is done we easily get the Equilibrium actionCombinations by combining these BestResponse-Matrices (fnMarkNashEq). Also the final presentation of the game is easy, we have to check if an actionCombination is feasible, an equilibrium or not and create a xxxMatrix with the respective string at the position of each actionCombination. (fnSolveGame).

To determine best responses of a player we divide the task into four steps.

1. `fnDropFalse` drops infeasible entries in the `payoffMatrix` as these cannot be best responses (technically this is done be replacing such entries by the symbol `Null`).
2. `fnGetReachablePositions` determines the `actionCombinations` a player can reach given his opponents' actions.

3. `fnCalculateMax` calculates the maximal payoff within all reachable entries (ignoring infeasible entries, using global assumptions).
4. `fnMarkEqualToValue` marks the entries that deliver maximal payoff with `True` and all others with `False`, i.e. it basically saves the result of the best response analysis.

The function `fnMarkBestResponses` finally combines all these auxilary functions to transform a whole `payoffMatrix` into a `bestResponseMatrix`.

Notice that information is 'lost' in the course of the algorithm. In a first step concrete `payoffMatrices` are transformed into `booleanMatrices` that just tell if a certain `actionCombination` is a best response for a player. In a second step several `bestResponseMatrices` are condensed into a `nashEqMatrix` that just tells if a certain `actionCombination` is a Nash equilibrium or not.

The main feature of this algorithm is that is operates on whole sets of feasible matrix positions. The reason we do it this way is to save computation time. If on the contrary we checked each position in the payoffMatrix separately whether it is maximal this would mean to calculate the same maximum several times.

Nevertheless, I believe that further improvements should be possible, also with respect to highly optimized internal functions introduced in newer Mathematica versions. Some suggestions are:

- Make `feasibilityMatrix` an optional argument in the functions `fnSolveGame[]` and `fnSolveGameTraceable[]` and set default value to the constant array filled with `True` everywhere (use build-in `Array[]` function).
- This algorithm works quite 'graphically', `fnSolveGame[]` for example returns a whole `xxxMatrix`, although there maybe only one equilibrium. Also the function `fnMarkBestResponse[]` returns a whole `xxxMatrix` although we could aswell only return a list of the best response `actionCombinations` – which would be more memory efficient.
- The function `fnSolveGame[]` could be made more efficient. Currently it first calculates all players best response matrices and then combines them. But positions that are not a best response for the first player cannot be an equilibrium, so we can save the calculation effort of checking whether they are a best response for any other player. (The function `fnSolveGameTraceable[]`, however, needs to calculate all best responses for all players.)

A.3.4 Conventions

Unfortunately commonly used naming conventions cannot be used together with Mathematica symbolic language. Therefore I adopt the following conventions:

Data definition names use upper camel case, starting with a small type to avoid conflicts the Mathematica expressions.

Constants are typeset in ALL-CAPS.

Variables commence with v… and proceed in upper camel case.

Functions commence with fn… and proceed in upper camel case.

Functions returning a single boolean value end with Q (according to the Mathematica convention).

Functions transforming a list of lists into a list with only boolean leaves commence with fnMark… and then indicate how `True` is to be interpreted.

Functions are ordered from auxilary functions to combining functions, as Mathematica has to read the auxilary functions first before other functions can call them. Unfortunately this puts the most important functions last.

A.3.5 Source code

```
415 (* ========= *)
416 (* CONSTANTS *)
417
418 INF;  (* sympol for impossible action-combinations, without value *)
419 EQU;  (* symbol for Nash equilibrium action-combinations, without value *)
420 ELS;  (* symbol for Not Nash equilibrium action-combinations, without value *)
421
422 (* ================ *)
423 (* DATA DEFINITIONS *)
424
425 (* nrOfPlayers *)
426 (* DEFINITION        is positive Integer *)
427 (* INTERPRETATION    the total number of players *)
428 (* EXAMPLES          n=2; n=5; *)
429 (* TEMPLATE          fnForN[n_]:= ... n ; *)
430 (* RULES USED        atomic non-distinct: positive Integer *)
431
432 (* playerNr *)
433 (* DEFINITION        is Integer[1,nrOfPlayers] *)
434 (* INTERPRETATION    the number of a certain player *)
435 (* EXAMPLES          playerNrMax=1; playerNrMoritz=2; *)
436 (* TEMPLATE          fnForPlayerNr[playerNr_]:= ... playerNr ; *)
437 (* RULES USED        atomic non-distinct: Integer[1,nrOfPlayers] *)
438
439 (* actionNr *)
440 (* DEFINITION        is Integer[1,nrOfPlayers] *)
441 (* INTERPRETATION    the number of an action a specified player can play *)
442 (* EXAMPLES          actionRepay=1; actionDefault=2 *)
443 (* TEMPLATE          fnForAction[action_]:= ... action ; *)
444 (* RULES USED        atomic non-distinct: Integer[1,nrOfPlayers] *)
445
446 (* actionCombination *)
447 (* DEFINITION        is List of positive Integers *)
448 (* INTERPRETATION    an action combination in the game, the first entry denotes the actionNr of the first player etc. *)
449 (* EXAMPLES          actionCombinationA={1,1,1}; actionCombinationB={1,1,2} *)
450 (* TEMPLATE          fnForActionCombination[actionCombination_]:= ... actionCombination [[...]] ; *)
451 (* RULES USED        compound: nrOfPlayers fields *)
452
453 (* payoff *)
454 (* DEFINITION        is Real (or a Mathematica expression representing a Real number) *)
455 (* INTERPRETATION    the payoff a player receives if a certain action combination is played *)
456 (* EXAMPLES          payoffCombination111Player1=7; payoffCombination111Player2=-2R-p *)
457 (* TEMPLATE          fnForPayoff[payoff_]:= ... payoff ; *)
458 (* RULES USED        atomic non-distinct: Real number *)
459
460 (* feasibility *)
461 (* DEFINITION        is Boolean *)
462 (* INTERPRETATION    true if action combination is feasible, false otherwise *)
463 (* EXAMPLES          feasibilityCombination111 =True; pfeasibilityCombination112 =False *)
464 (* TEMPLATE          fnForFeasibility [ feasibility_ ]:= ... feasbility ; *)
465 (* RULES USED        atomic non-distinct: Boolean *)
```

```
(* bestResponse *)
(* DEFINITION          is Boolean *)
(* INTERPRETATION      true if action combination is a best response for a player (to be specified), false otherwise *)
(* EXAMPLES            bestResponseCombination111Player1=True; bestResponseCombination111Player2=False *)
(* TEMPLATE            fnForBestResponse[ bestresponse_ ]:= ... bestResponse ; *)
(* RULES USED          atomic non-distinct: Boolean *)

(* nashEq *)
(* DEFINITION          is Boolean *)
(* INTERPRETATION      true if action combination is a NashEquilibrium, false otherwise *)
(* EXAMPLES            nashEqCombination111=True; nashEqCombination112=False *)
(* TEMPLATE            fnForNashEq[nashEq_]:= ... nashEq ; *)
(* RULES USED          atomic non-distinct: Boolean *)

(* solution *)
(* DEFINITION          is one of INF,EQU,ELS *)
(* INTERPRETATION      the result of the evaluation of a certain action combination: infeasible, equilibrium, else *)
(* EXAMPLES            solutionCombination111=INF; nashequilibriumCombination112=EQU *)
(* TEMPLATE            fnForSolution[ solution_ ]:= Which[ solution==INF, ... , solution==EQU, ... , solution==ELS, ...] ; *)
(* RULES USED          one of: 3 cases; atomic distinct: INF; atomic distinct: EQU; atomic distinct: ELS *)

(* xxxMatrix *)
(* DEFINITION          is List of Lists of Lists ... of the dimensions of the game with content xxx (of arbitrary type) *)
(* INTERPRETATION      placeholder for entries of the data types defined above *)
(* EXAMPLES            payoffMatrix=payoffsOfPlayer1; feasibilityMatrix =feasibleActionCombinations *)
(* TEMPLATE            fnForGameMatrix[ xxxMatrix_]:= ... xxxMatrix [[...]](* entry*) ; *)
(* RULES USED          compound: many fields (according to the dimensions of the game) *)

(* game *)
(* DEFINITION          is List of payoffMatrices *)
(* INTERPRETATION      the first payoffMatrix describes the payoffs of the first player ... etc. *)
(* EXAMPLES            game1 = {{{2, 3}, {5, 1}}, {{6, 2.1}, {13, 2}}}; *)
(* TEMPLATE            fnForGame[game_]:=... game [[...]] (*payoffMatrix*); *)
(* RULES USED          compound: nrOfPlayers fields ; xxxMatrix; payoff; *)


(* ========== *)
(* FUNCTIONS *)

(* fnDropFalse *)
(* SIGNATURE           [payoffMatrix_, feasibilityMatrix_ ] --> payoffMatrix *)
(* PURPOSE             replace entries of infeasible action combinations in payoffMatrix by Null *)
(* TESTS *)
fnDropFalse[{{1, 2, 3}, {4, 5, 6}}, {{True, True, False}, {True, False, True}}] === {{1, 2, Null}, {4, Null, 6}};
fnDropFalse[{{-5, 2}, {7.1, Pi}}, {{True, True}, {True, True}}] === {{-5, 2}, {7.1, Pi}};
fnDropFalse[{{-5, 2}, {7.1, Pi}}, {{False, False}, {False, False}}] === {{Null, Null}, {Null, Null}};
fnDropFalse[{{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}, {{{True, False}, {True, True}}, {{True, True}, {True, True}}}] === {{{1, Null}, {3, 4}}, {{5, 6}, {7, 8}}};
(* FUNCTION BODY *)
fnDropFalse[xxxMatrix_, feasibilityMatrix_ ]:=Module[
 {fnReplaceRobot,vReplacedMatrix},
fnReplaceRobot[ value_, position_ ]:= If[ feasibilityMatrix [[Sequence@@position]]==False,Null,value]; (* function that decides upon replacement *)
```

```
519 vReplacedMatrix=MapIndexed[ fnReplaceRobot[#1,#2]&,xxxMatrix,{ArrayDepth[feasibilityMatrix]}];
520 (* fnReplaceRobot[] applied to entries of payoffMatrix *)
521 Return[vReplacedMatrix]; (* return resulting matrix *) ];
522
523
524 (* fnGetReachablePositions *)
525 (* SIGNATURE [actionCombination_,playerNr_] --> List *)
526 (* PURPOSE determine which positions a player can reach under given actions of his opponents *)
527 (* TESTS *)
528 fnGetReachablePositions[{1,1},1]=={All,1};
529 fnGetReachablePositions[{1,2,5,3,2},4]=={1,2,5,All,2};
530 (* FUNCTION BODY *)
531 fnGetReachablePositions[actionCombination_,playerNr_]:=ReplacePart[actionCombination,playerNr->All];
532
533
534 (* fnCalculateMax *)
535 (* SIGNATURE [ listOfPayoffs_ ] --> Real (or Mathematica symbol representing a Real number) *)
536 (* PURPOSE determine the maximum in a list of (maybe symbolic) payoffs *)
537 (* TESTS *)
538 fnCalculateMax[{2,Pi,5,12.3}]==12.3;
539 fnCalculateMax[{Null,1,5,1,Null}]===5;
540 Assuming[e>R,fnCalculateMax[{R,e,Null}]===e];
541 (* FUNCTION BODY *)
542 fnCalculateMax[ listOfPayoffs_ ]:=Module[
543 {vList,vMax},
544 vList=listOfPayoffs ; (* Initialization *)
545 vList=DeleteCases[vList,Null]; (* Delete entries that are Null *)
546 vMax=Max[vList]; (* Calculate Maximum *)
547 vMax=Refine[vMax]; (* Refine the result using global assumptions *)
548 Return[vMax];];
549
550
551 (* fnMarkEqualToValue *)
552 (* SIGNATURE [ listOfValues_,value_] --> [listOfBoolean] *)
553 (* PURPOSE transform a list of values into a list of boolean entries, with true at positions that coincide with value_ *)
554 (* TESTS *)
555 fnMarkEquafToValue[{3,4,5},4]=={False,True,False};
556 fnMarkEquafToValue[{1,2,1,1,3},1]=={True,False,True,True,False};
557 fnMarkEquafToValue[{1,Null,1,1,3},1]=={True,False,True,True,False};
558 (* FUNCTION BODY *)
559 fnMarkEquafToValue[listOfValues_,value_]:=Module[
560 {fnReplaceRobot,vReplacedList},
561 (* function that decides upon replacement *)
562 fnReplaceRobot[entry_]:=Which[
563 entry===Null, False,
564 Refine[entry==value], True,
565 True, False];
566 (* fnReplaceRobot[] applied to the entries of listOfValues *)
567 vReplacedList=Map[fnReplaceRobot[#]&,listOfValues];
568 (* return resulting list *)
569 Return[vReplacedList]; ];
570
571
```

```
(* fnMarkBestResponse *)
(* SIGNATURE          [payoffMatrix_, feasibilityMatrix_, playerNr_] --> bestResponseMatrix *)
(* PURPOSE            transform a payoffMatrix into booleanMatrix with True if a position is a best response *)
(* TESTS *)
fnMarkBestResponse[{{2,3},{4,2}},{{True,False},{True,True}},1] === {{False,False},{True,True}};
fnMarkBestResponse[{{1,2,3},{4,5,6}},{{True,True,False},{True,False,True}},2] === {{False,True,False},{False,False,True}};
fnMarkBestResponse[{{{1,2},{3,4}},{{5,6},{7,8}}},{{{True,True},{True,True}},{{True,False},{True,True}}},1] ===
 {{{False,True},{False,False}},{{True,False},{True,True}}};
(* FUNCTION BODY *)
fnMarkBestResponse[payoffMatrix_, feasibilityMatrix_, playerNr_]:=Module[
{vPayoffMatrix,fnReplaceRobot,fnReplaceRobotSmart,vResult},
(* Preparation of the payoff Matrix *)
vPayoffMatrix=fnDropFalse[payoffMatrix, feasibilityMatrix ]; (* Drop entries of infeasible action combinations *)
(* Robot function to mark best responses in a certain situation *)
fnReplaceRobot[actionCombination_]:=Module[
{vSelectorSequence,vReachableEntries,vMax},
vSelectorSequence=Sequence@@fnGetReachablePositions[actionCombination,playerNr]; (* get reachable positions *)
vReachableEntries=vPayoffMatrix[[vSelectorSequence]]; (* get list of reachable entries *)
vMax=fnCalculateMax[vReachableEntries]; (* calculate maximum *)
vReachableEntries=fnMarkEqualToValue[vReachableEntries,vMax]; (* replace list of reachable entries by boolean values with True for maximum *)
vPayoffMatrix[[vSelectorSequence]]=vReachableEntries; (* substitute reachable entries boolean version into vPayoffMatrix *)];
(* Making the robot smart by applying it only to new situations for a player *)
fnReplaceRobotSmart[actionCombination_]:=If[actionCombination[[playerNr]]==1,fnReplaceRobot[actionCombination]];
(* Applying the robot to the payoffMatrix *)
MapIndexed[fnReplaceRobotSmart[#2]&,vPayoffMatrix,{ArrayDepth[vPayoffMatrix]}];
(* Return vPayoffMatrix *)
Return[vPayoffMatrix];];


(* fnMarkNashEq *)
(* SIGNATURE          [game_, feasibilityMatrix_] --> nashEqMatrix *)
(* PURPOSE            transform a game into a booleanMatrix with True if a position is a Nash equilibrium *)
(* TESTS *)
fnMarkNashEq[{{{2,3},{4,2}},{{5,7},{2,3}}},{{True,False},{True,True}}] === {{False,False},{False,True}};
(* FUNCTION BODY *)
fnMarkNashEq[game_, feasibilityMatrix_]:=Module[
{vNrOfPlayers,vBestResponseMatrices,vNashEqMatrix},
(* Determine number of players in the game *)
vNrOfPlayers=Length[game];
(* produce vBestResponseMatrices by applying fnMarkBestResponse to all payoffMatrices of the game *)
vBestResponseMatrices=Map[fnMarkBestResponse[game[[#]],feasibilityMatrix,#]&,Range[vNrOfPlayers]];
(* Combine bestResponses to NashEquilibria *)
vNashEqMatrix=MapThread[And,vBestResponseMatrices,vNrOfPlayers];
(* Return vNashEqMatrix *)
Return[vNashEqMatrix];];


(* fnSolveGame *)
(* SIGNATURE          [game_, feasibilityMatrix_] --> solutionMatrix *)
(* PURPOSE            mark the actionCombinations in a xxxMatrix with one of INF,EQU,ELS *)
(* TESTS *)
fnSolveGame[{{{2,3},{4,2}},{{5,7},{2,3}}},{{True,False},{True,True}}] === {{ELS,INF},{ELS,EQU}};
(* FUNCTION BODY *)
```

```
fnSolveGame[game_, feasibilityMatrix_ ]:=Module[
{vNashEqMatrix,fnRobot,vSolutionMatrix},
(* Find Nash Equilibria *)
vNashEqMatrix=fnMarkNashEq[game,feasibilityMatrix];
(* Robot that treats each position accordingly *)
fnRobot[actionCombination_]:=Which[
! feasibilityMatrix [[Sequence@@actionCombination]], INF,
vNashEqMatrix[[Sequence@@actionCombination]],          EQU,
True,                                                                                    ELS];
(* Apply fnRobot to each actionCombination *)
vSolutionMatrix=MapIndexed[fnRobot[#2]&,game[[1]],{ArrayDepth[game[[1]]]}];
(* Return vSolutionMatrix *)
Return[vSolutionMatrix];];


(* fnSolveGameTraceable *)
(* SIGNATURE          [game_, feasibilityMatrix_ ] --> xxxMatrix *)
(* PURPOSE            produce a solution that is traceable by humans, entries xxx are payoffCombinations with best responses underlined *)
(* TESTS *)
(* FUNCTION BODY *)
fnSolveGameTraceable[game_, feasibilityMatrix_ ] :=Module[
{vNrOfPlayers,vBestResponseMatrices,fnUnderlineRobot,fnUnderlineRobots,vUnderlinedGame,vResult},
(* Determine number of players in the game *)
vNrOfPlayers=Length[game];
(* produce vBestResponseMatrices by applying fnMarkBestResponse to all payoffMatrices of the game *)
vBestResponseMatrices=Map[fnMarkBestResponse[game[[#]],feasibilityMatrix,#]&,Range[vNrOfPlayers]];
(* Robot that underlines best responses in the payoffMatrices *)
fnUnderlineRobot[booleanMatrix_, value_, position_ ]:=If[booleanMatrix[[Sequence@@position]],Style[value,Underlined], Style[value]];
(* clone the robot apply them to a whole payoffMatrix *)
fnUnderlineRobots[payoffMatrix_,bestResponseMatrix_]:=MapIndexed[fnUnderlineRobot[bestResponseMatrix,#1,#2]&,payoffMatrix,{ArrayDepth[payoffMatrix]}];
(* apply the cloned robots to the whole game *)
vUnderlinedGame=Map[fnUnderlineRobots[game[[#]],vBestResponseMatrices[[#]]]&,Range[vNrOfPlayers]];
(* combine the payoffMatrices to one Matrix with entries payoffCombinations *)
vResult=MapThread[",vUnderlinedGame,ArrayDepth[vUnderlinedGame[[1]]]];
(* Drop infeasible actionCombinations in the Matrix *)
vResult=fnDropFalse[vResult, feasibilityMatrix]/. Null->INF;
(* Return result *)
Return[vResult];
];
```

A.4 Code block: Game analysis

A.4.1 Problem statement

These functions shall take the `solutionMatrix` of a certain `abilityCombination` and calculate information about the equilibria found in the `solutionMatrix`. To perform these calculation they will have to refer to the global variables described on page p. 203 and combine this information with the information from the `solutionMatrix`. We want to calculate:

1. Number of equilibria
2. To build the map of equilibria: equilibrium actionNameCombinations
3. To build the map of outcomes: equilibrium payoffCombinations
4. To build the map of repayment: equilibrium sum of repayments
5. To built the map of punishment load: equilibrium total punishment load (punishment and sanctions of all borrowers)

In addition we want to have a method to display what the punishment strategy resulting from our `punishmentFunction` is. So there shall be a function that returns a `xxxTos` with entries the punishment applied in each combination. Or more general: a function that returns a `xxxTos` with entries the `payoffLayerCombinations` for a `payoffLayer` of our choice.

The methods should not use any data types other than defined in the game construction block.

A.4.2 Brief description of the algorithm

The methods defined in this code block are rather self-explanatory.

Maybe the code could be made more elegant by using more abstract 2nd order extractor functions, because the functions here are so similar with lots of redundant code. For example, write a function that extracts equilibrium Combinations and then functions that turn combinations to repaymentCombination, or combination to punishmentCombination, etc.

A.4.3 Source code

```
665 (* ========== *)
666 (* FUNCTIONS *)
667
668 (* fnGetEqActions *)
669 (* SIGNATURE          [ solutionMatrix_ ] --> actionNameCombinationsList *)
670 (* PURPOSE            get the equilibrium actionCombinations of a game, to build the map of equilibria *)
671 (* TESTS *)
672 (* ... depend on global variable actionSet[ i ] ... *)
673 (* fnGetEqActions[ {{EQU,ELS,ELS,ELS,INF,INF},{ELS,ELS,ELS,ELS,INF,INF},{ELS,ELS,ELS,ELS,INF,INF},{ELS,ELS,ELS,ELS,EQU,INF,INF},{INF,INF,INF,INF,INF,INF},{INF,INF,INF,INF,INF,INF}}]==="0R-n|0
        R-n\nIR-y|IR-y" *)
674 (* FUNCTION BODY *)
675 fnGetEqActions[ solutionMatrix_ ]:=Module[
676  {result},
677 (* Extraction of equilibrium actionKeyCombinations *)
678  result =Position[ solutionMatrix ,EQU];
679 (* Get from actionKeys to actions *)
680  result =Map[fnKeyToActionCombination,result];
681 (* Get form actions to actionNames *)
682  result =Map[fnGetActionName,result,{2}];
683 (* Formatting *)
684  result =fnFormatCombinationsList[ result ];
685 (* return result *)
686 Return[result];
687  ];
688
689
690 (* fnGetLenderPayoffPerPlayer *)
691 (* SIGNATURE          [abilityKeyCombination_,actionKeyCombination_,playerKey_] --> lenderPayoff *)
692 (* PURPOSE            calculate a lenders payoff from one borrower in a particular situation *)
693 (* TESTS *)
694 (* depend on global variables punishmentFunction, sanctionFunction, actionSets *)
695 (* FUNCTION BODY *)
696  fnLenderPayoffPerPlayer [abilityKeyCombination_,actionKeyCombination_,playerKey_]:=Module[
697  {abilityCombination, actionCombination, payoff},
698  abilityCombination=fnKeyToAbilityCombination @abilityKeyCombination; (* get from abilityKeys to abilities, since punishmentFunction and sanctionFunction need these *)
699 actionCombination=fnKeyToActionCombination@actionKeyCombination; (* get from actionKeys to actions, since punishmentFunction and sanctionFunction need these *)
700 payoff=-fnGetActionRepayment@(actionCombination[[playerKey]])
701 -rewardFunction[abilityCombination,actionCombination,playerKey]; (* calculate payoff *)
702 Return[payoff]; (* return calculated payoff *)
703  ];
704
705
706 (* fnGetLenderPayoffTotal *)
707 (* SIGNATURE          [abilityKeyCombination_,actionKeyCombination_] --> lenderPayoff *)
708 (* PURPOSE            calculate a lenders total payoff in a particular situation *)
709 (* TESTS *)
710 (* depend on global variables punishmentFunction, sanctionFunction, actionSets *)
711 (* FUNCTION BODY *)
712  fnLenderPayoffTotal [abilityKeyCombination_,actionKeyCombination_]:=Plus@@Map[fnLenderPayoffPerPlayer[abilityKeyCombination,actionKeyCombination,#]&,Range[n],{1}];
713
714 (* fnGetEqLenderPayoffTotal *)
```

```
(* SIGNATURE          [abilityKeyCombination_, solutionMatrix_ ] --> lenderPayoffList *)
(* PURPOSE            calculate lenderPayoff in equilibrium, to build the map of lenderPayoff *)
(* TESTS *)
(* depends on global variables *)
(* FUNCTION BODY *)
fnGetEqLenderPayoffTotal[abilityKeyCombination_, solutionMatrix_ ]:=Module[
{eqActionKeyCombinationsList, lenderPayoffTotal },
(* Extraction of equilibrium actionKeyCombinations *)
eqActionKeyCombinationsList=Position[ solutionMatrix ,EQU];
(* get lenderPayoffTotal from all equilibrium actionKeyCombinations *)
 lenderPayoffTotal =Map[fnLenderPayoffTotal[abilityKeyCombination,#]&,eqActionKeyCombinationsList,{1}];
(* return result *)
Return[ lenderPayoffTotal ];
];


(* fnGetEqPayoffs *)
(* SIGNATURE          [abilityKeyCombination_, solutionMatrix_ ] --> payoffCombinationsList *)
(* PURPOSE            get the equilibrium payoff for each borrower, to calculate expected return (meta-game relevant) *)
(* TESTS *)
(* depends on global variables *)
(* FUNCTION BODY *)
fnGetEqPayoffs[abilityKeyCombination_, solutionMatrix_ ]:=Module[
{eqActionKeyCombinationsList,robotGetPayoffCombination,payoffCombinationsList},
(* Extraction of equilibrium actionKeyCombinations *)
eqActionKeyCombinationsList=Position[ solutionMatrix ,EQU];
(* robot to get payoffCombination from actionKeyCombination *)
robotGetPayoffCombination[actionKeyCombination_]:=Map[fnPayoff[abilityKeyCombination,actionKeyCombination,#]&,Range[n],{1}];
(* apply robot *)
 payoffCombinationsList=Map[robotGetPayoffCombination[#]&,eqActionKeyCombinationsList,{1}];
(* return result *)
Return[payoffCombinationsList];
];


(* fnGetLossSum *)
(* SIGNATURE          [abilityKeyCombination_,actionKeyCombination_] --> deadweightList *)
(* PURPOSE            get the deadweight-loss in a certain situation *)
(* TESTS *)
(* FUNCTION BODY *)
fnGetLossSum[abilityKeyCombination_,actionKeyCombination_]:=Module[
{abilityCombination, actionCombination, loss, lossCombination, lossSum},
 abilityCombination=fnKeyToAbilityCombination @abilityKeyCombination; (* get from abilityKeys to abilities, since punishmentFunction and sanctionFunction need these *)
actionCombination=fnKeyToActionCombination@actionKeyCombination; (* get from actionKeys to actions, since punishmentFunction and sanctionFunction need these *)

 loss[playerKey_]:= punishmentFunction[ abilityCombination ,actionCombination,playerKey]
+ sanctionFunction[abilityCombination ,actionCombination,playerKey]; (* calculate loss of a player *)
lossCombination=Map[loss[#]&,Range[n],1]; (* calculate loss for all players *)
lossSum=Plus@@lossCombination; (* calculate the total loss, i.e. the sum of the loss of each player *)

Return[lossSum]; (* return calculated payoff *)
];

```

```
768
769 (* fnGetEqLoss *)
770 (* SIGNATURE          [abilityKeyCombination_, solutionMatrix_] --> lenderPayoffList *)
771 (* PURPOSE            calculate lenderPayoff in equilibrium, to build the map of lenderPayoff *)
772 (* TESTS *)
773 (* depends on global variables *)
774 (* FUNCTION BODY *)
775 fnGetEqLoss[abilityKeyCombination_, solutionMatrix_]:=Module[
776 {eqActionKeyCombinationsList,loss},
777 (* Extraction of equilibrium actionKeyCombinations *)
778 eqActionKeyCombinationsList=Position[solutionMatrix,EQU];
779 (* get lenderPayoffTotal from all equilibrium actionKeyCombinations *)
780 loss=Map[fnGetLossSum[abilityKeyCombination,#]&,eqActionKeyCombinationsList,{1}];
781 (* return result *)
782 Return[loss];
783 ];
784
785
786 (* fnGetPayoffLayerCombinationsTos *)
787 (* SIGNATURE          xxxFunction --> xxxTos *)
788 (* PURPOSE            builds payoffLayersTos from the input layerFunction, used to illustrate the punishment strategy defined by punishmentFunction *)
789 (* TESTS *)
790 (* ... *)
791 (* FUNCTION BODY *)
792 fnGetPayoffLayerCombinationsTos[layerFunction_]:=Module[
793 {layerFunctionWithKeys,layerCombination, layerMatrix, layerMatrixFeasible,layerTos},
794 (* transform actionKeyCombination to actionCombination as layerFunction needs it that way *)
795 layerFunctionWithKeys[abilityKeyCombination_,actionKeyCombination_,playerKey_]:=layerFunction[fnKeyToAbilityCombination@abilityKeyCombination,fnKeyToActionCombination@actionKeyCombination,playerKey];
796 (* Build layerCombinations from individual layer entries per player *)
797 layerCombination[abilityKeyCombination_,actionKeyCombination_]:=Map[layerFunctionWithKeys[abilityKeyCombination,actionKeyCombination,#]&,Range[n],1];
798 (* Build layerMatrices from layerCombinations *)
799 layerMatrix[abilityKeyCombination_]:=Map[layerCombination[abilityKeyCombination,#]&,emptyMatrix,{n}];
800 (* Drop infeasible entries from layerMatrix *)
801 layerMatrixFeasible[abilityKeyCombination_]:=fnDropFalse[layerMatrix[abilityKeyCombination],feasibilityMatrix[abilityKeyCombination]];
802 (* Build layerTos from layerMatrices *)
803 layerTos=Map[layerMatrixFeasible[#]&,emptyTos,{n}];
804 Return[layerTos];
805 ];
```

A.5 Code block: Game presentation

A.5.1 Problem statement

Write a method that take an `xxxMatrix` of arbitrary dimension and present it nicely and easy to read. In particular the output should have headings, such that each entry can be easily attributed to the respective `actionCombination`. Likewise, write a method to display a tabular of shape, i.e. one `xxxMatrix` for every `abilityCombination`.

Furthermore, include any helpful functions to generate a nice graphical output in this code block here.

The functions here shall only refer to the global variables `n`, `actionSet[i]`, `abilitySet[i]`, and use only methods from the game construction block. This way the code block remains independent of the blocks game solution and game analysis.

A.5.2 Brief description of the algorithm

The algorithm to display a `xxxMatrix` or a `xxxTos` proceeds in two steps. Step1: Construct Headings. This part collects the necessary data for the headings and formats it. Step2: Display `xxxMatrix/xxxTos`. This part actually formats and displays a given `xxxMatrix/xxxTos`.

A.5.3 Source code

```
(* ========== *)
(* CONSTANTS *)

cCombinationEntrySeparator = "|"; (* Separates the entries of xxxCombination *)
cCombinationSeparator = "\n"; (* Separates the entries of xxxCombinationsList *)

 cPlayerActionSeparator =":"; (* Separates playerKey and action in the headings of an xxxMatrix *)
 cPlayerAbilitySeparator = ":"; (* Separates playerKey and ability in the headings of an xxxTos *)

cHeadlineStyle={Bold,Underlined,Larger};
cActionHeadingsStyle = {Smaller, Bold}; (* Style of actionHeadings *)
 cAbilityHeadingsStyle = {Bold}; (* Style of abilityHeadings *)

cMatrixStyle={Bold,Smaller}; (* Style of xxxMatrix entries *)
cTosStyle={Italic}; (* Style of xxxTos entries *)

 cInfString = "*"; (* String to mark infeasible actionCombinations in an xxxMatrix *)
cEquString = "\[SixPointedStar]"; (* String to mark equilibrium actionCombinations in an xxxMatrix *)
cElsString = "-"; (* String to mark all other actionCombinations in an xxxMatrix *)


(* ========== *)
(* ADDITIONAL DATA TYPES *)

(* actionHeading *)
(* DEFINITION          list of {playerKey,actionName} *)
(* INTERPRETATION      the heading for a column/row in an xxxMatrix; denoting the player who "owns" this dimension and the column/rows actionName *)
(* EXAMPLES            actionHeading1={1,"repay"}, actionHeading2={5,"2R-n"} *)
(* TEMPLATE            fnForActionHeading[actionHeading_]:= ... actionHeading[[1]], actionHeading[[2]]; *)
(* RULES USED          compound: 2 fields ; references : playerKey, actionName *)

(* abilityHeading *)
(* DEFINITION          list of {playerKey, ability } *)
(* INTERPRETATION      the heading for a column/row in an xxxTos ,; denoting the player who "owns" this dimension and the column/rows ability *)
(* EXAMPLES            abilityHeading1={1,2R}, abilityHeading2={5,0} *)
(* TEMPLATE            fnForAbilityHeading[ abilityHeading_ ]:= ... abilityHeading [[1]], abilityHeading [[2]]; *)
(* RULES USED          compound: 2 fields ; references : playerKey, ability *)

(* xxxCombinationsList *)
(* DEFINITION          a list of xxxCombinations *)
(* INTERPRETATION      a list of xxxCombinations *)
(* EXAMPLES            CombinationsList1={ { { "A:pl1","A:pl2","A:pl3" }, { "B:pl1","B:pl2","B:pl3" } } *)
(* TEMPLATE            fnFor:xxxCombinationsList[ xxxCombinationsList_]:= ... xxxCombinationsList[[item ,playerKey]]; *)
(* RULES USED          compound: arbitrary many fields ; references : xxxCombinations *)


(* ========== *)
(* FUNCTIONS *)

(* ---------------------------------- *)
(* Format section headings *)
```

```
858
859 headline[ text_ ]:= Style["\n"<>text,cHeadlineStyle]; (* a simple tag to print headlines *)
860
861 (* ---------------------------------- *)
862 (* Construct tabular headings *)
863
864 (* fnGetActionNameSet *)
865 (* SIGNATURE          actionSet --> actionNameSet *)
866 (* PURPOSE            construct action headings for presentation *)
867 (* TESTS *)
868 fnGetActionNameSet[{{"opt1", 0, "n"}, {"opt2", 0, "y"}, {"opt3", -R, "y"}, {"opt4", -2 R, "y"}}] === {"opt1", "opt2", "opt3", "opt4"};
869 (* FUNCTION BODY *)
870 fnGetActionNameSet[actionSet_] := fnGetActionName /@ actionSet;
871
872 (* fnGetActionHeadings *)
873 (* SIGNATURE          [actionNameSet_,playerKey_] --> actionHeadingSet *)
874 (* PURPOSE            construct actionNameHeadingSet, i.e. actionNameSet with player denoted; this function determines how they are formated *)
875 (* TESTS *)
876 (* fnGetActionHeadings[fnGetActionNameSet[actionSet [1]],1]; *)
877 (* FUNCTION BODY *)
878 fnGetActionHeadings[actionNameSet_, playerKey_] := Map[(ToString["P"] <> ToString[playerKey] <> cPlayerActionSeparator <> ToString[#]) &, actionNameSet, {1}];
879
880 (* fnGetAbilityHeadings *)
881 (* SIGNATURE          [ abilitySet_ , playerKey_] --> abilityHeadingSet *)
882 (* PURPOSE            construct abilityHeadingSet for a player; this function determines how they are formated *)
883 (* TESTS *)
884 (* fnGetAbilityHeadings[ abilitySet [1],1]; *)
885 (* FUNCTION BODY *)
886 fnGetAbilityHeadings[ abilitySet_ , playerKey_] := Map[(ToString["A"] <> ToString[playerKey] <> cPlayerAbilitySeparator <> ToString[#]) &, abilitySet, {1}];
887
888 (* actionHeadingsCombination *)
889 (* SIGNATURE          [ ] --> actionHeadingsCombinarion *)
890 (* PURPOSE            combine the actionNameSets of each player *)
891 (* TESTS *)
892 (* actionHeadingsCombination; *)
893 (* FUNCTION BODY *)
894 actionHeadingsCombination := Module[
895 {vHeadings},
896 (* Step 1: Construct actionHeadings for all players *)
897 vHeadings = Map[fnGetActionHeadings[fnGetActionNameSet[actionSet[#]], #] &, Range[n], {1}];
898 (* Step 2: Format Headings *)
899 vHeadings = Map[Style[#, cActionHeadingsStyle] &, vHeadings, {2}];
900 (* Return result *)
901 Return[vHeadings]; ];
902
903 (* abilityHeadingsCombination *)
904 (* SIGNATURE          [ ] --> abilityHeadingsCombinarion *)
905 (* PURPOSE            combine the abilityName of each player *)
906 (* TESTS *)
907 (* abilityHeadingsCombination; *)
908 (* FUNCTION BODY *)
909 abilityHeadingsCombination := Module[
910 {vHeadings},
```

```
(* Step 1: Construct abilityHeadings for all players *)
vHeadings = Map[fnGetAbilityHeadings[ abilitySet [#], #] &, Range[n], {1}];
(* Step 2: Format Headings *)
vHeadings = Map[Style[#, cAbilityHeadingsStyle ] &, vHeadings, {2}];
(* Return result *)
Return[vHeadings]; ];


(* ------------------------------------ *)
(* Display and format xxxMatrix,xxxTos *)

(* fnDisplayMatrix *)
(* SIGNATURE          xxxMatrix --> Output *)
(* PURPOSE             display xxxMatrices in nice Mathematica Output Form *)
(* TESTS *)
(* ... *)
(* FUNCTION BODY *)
fnDisplayMatrix[xxxMatrix_] := Module[
{xxxMatrixStyled,output},
(* Format entries *)
xxxMatrixStyled=Map[Style[#,cMatrixStyle]&,xxxMatrix,{n}];
(* Plots the tabularOfShapes in TableForm with the following options ... *)
output = TableForm[xxxMatrixStyled,
TableAlignments -> Left, (* how to align entries in each dimension *)
TableDirections -> If[EvenQ[n], Column, Row], (* whether to arrange dimensions as rows or columns *)
TableHeadings -> actionHeadingsCombination, (* how to label table entries *)
TableSpacing -> 1 ]; (* how many spaces to put between entries in each dimension *)
(* Return result *)
Return[output]; ];

(* fnDisplayTos *)
(* SIGNATURE          xxxTos --> Output *)
(* PURPOSE             present all shapes at once *)
(* TESTS *)
(* ... *)
(* FUNCTION BODY *)
fnDisplayTos[xxxTos_] := Module[
{xxxTosStyled,output},
(* Format entries *)
xxxTosStyled=Map[Framed[Style[#,cTosStyle]]&,xxxTos,{n}];
(* Plots the tabularOfShapes in TableForm with the following options ... *)
output = TableForm[xxxTosStyled,
TableAlignments -> Center, (* how to align entries in each dimension *)
TableDirections -> If[EvenQ[n], Column, Row], (* whether to arrange dimensions as rows or columns *)
TableHeadings -> abilityHeadingsCombination, (* how to label table entries *)
TableSpacing -> 3];(* how many spaces to put between entries in each dimension *)
(* Return result *)
Return[output]; ];


(* ------------------------------------------------------------ *)
(* Format xxxCombinations, xxxCombinationsLists , solutionMatrix *)

```

```
964
965 (* fnFormatCombination *)
966 (* SIGNATURE          xxxCombination --> String *)
967 (* PURPOSE            format a xxxCombination using the constant cCombinationEntrySeparator *)
968 (* TESTS *)
969 fnFormatCombination[{"A:pl1", "A:pl2", "A:pl3"}] === "A:pl1|A:pl2|A:pl3";
970 (* FUNCTION BODY *)
971 fnFormatCombination[xxxCombination_] := StringReplace[ToString[xxxCombination], {
972 "," -> cCombinationEntrySeparator, (* separate entries *)
973 "{" -> "", (* delete opening parenthese *)
974 "}" -> "", (* delete closing parenthese *)
975 "␣" -> ""}]; (* delete remanin white spaces *)
976
977 (* fnFormatCombinationsList *)
978 (* SIGNATURE          xxxCombinationsList --> String *)
979 (* PURPOSE            format a list of xxxCombinations using the constants cCombinationEntrySeparator and
980 cCombinationSeparator *)
981 (* TESTS *)
982 fnFormatCombinationsList[{{"A:pl1", "A:pl2", "A:pl3"}, {"B:pl1", "B:pl2", "B:pl3"}, {"C:pl1", "C:pl2", "C:pl3"}}] === "A:pl1|A:pl2|A:pl3\nB:pl1|B:pl2|B:pl3\nC:pl1|C:pl2|C:pl3";
983 (* FUNCTION BODY *)
984 fnFormatCombinationsList[xxxCombinationsList_] := Module[
985 {vFormattedCombiList},
986 (* Step 1: format xxxCombinations using the function fnFormatCombination *)
987 vFormattedCombiList = Map[fnFormatCombination[#] &, xxxCombinationsList, {1}];
988 (* Step 2: format the list of formated xxxCombinations usind the constant cCombinationSeparator *)
989 vFormattedCombiList = ToString[vFormattedCombiList]; (* make everything a string *)
990 vFormattedCombiList = StringReplace[vFormattedCombiList, {
991 "," -> cCombinationSeparator, (* separate entries *)
992 "{" -> "", (* delete opening parenthese *)
993 "}" -> "", (* delete closing parenthese *)
994 "␣" -> ""}]; (* delete remaining white spaces *)
995 (* Return result *)
996 Return[vFormattedCombiList]; ];
997
998 (* fnFormatSolutionMatrix *)
999 (* SIGNATURE          solutionMatrix --> StringMatrix *)
1000 (* PURPOSE            replace the symbols in a solution Matrix by graphically nice strings using the constants cInfString ,cEquString, cElsString *)
1001 (* TESTS *)
1002 fnFormatSolutionMatrix[{{EQU, ELS, INF}, {ELS, ELS, INF}, {INF, INF, INF}}] == {{cEquString, cElsString, cInfString}, {cElsString, cElsString, cInfString}, {cInfString, cInfString, cInfString}};
1003 (* FUNCTION BODY *)
1004 fnFormatSolutionMatrix[ solutionMatrix_ ] := solutionMatrix /. {INF -> cInfString, EQU -> cEquString, ELS -> cElsString};
```

A.6 Code block: Game definition

Now, that all necessary methods are defined and loaded into the Mathematica kernel, we can define the game and use them so solve it.

This piece of code serves as the actual ‘user interface’. Here the game definition is written down and the output of the results is generated. Although these are actually two concepually different blocks code, for convenience I combined it in one Mathematica notebook. Thereby you can change the game definition and observe the new results immediately in the same notebook.

The code that follows gives a small example what a repayment game definition looks like and also shows the corresponding output of the program. Of course the code and the results will change if the user defines own repayment games.

A.6.1 Source code (example)

```
1005 (* ================ *)
1006 (* GAME DEFINITION *)
1007
1008 (* ----Number of Players---- *)
1009
1010 n=2;
1011
1012 (* ----Actionset for each player ---- *)
1013
1014 actionSet[1]={
1015 {"0R-n",0,"n"},
1016 {"0R-y",0,"y"},
1017 {"1R",-R,"y"},
1018 {"2R",-2R,"y"}};
1019
1020 actionSet[2]=actionSet[1];
1021
1022 (* ----Punishment function: borrower to deadweight, controlled by lender ---- *)
1023
1024 punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
1025 {0,                (fnGetActionMessage/@actionCombination)==ConstantArray["y",n]&& Plus@@(fnGetActionRepayment/@actionCombination)<=-n R},(* Rule (1) *)
1026 {0,                (fnGetActionRepayment/@actionCombination)==ConstantArray[0,n]&&fnGetActionMessage[actionCombination[[playerKey]]]=="n"},(* Rule (2a) *)
1027 {-p,               (fnGetActionRepayment/@actionCombination)==ConstantArray[0,n]&&fnGetActionMessage[actionCombination[[playerKey]]]=="y"},(* Rule (2b) *)
1028 {-p,               fnGetActionMessage[actionCombination[[playerKey]]]=="n"&&Plus@@(fnGetActionRepayment/@actionCombination)<=-n R},(* Rule (3a) *)
1029 {0,                Union[Delete[(fnGetActionMessage/@actionCombination),playerKey]]=={"n"}&&Union[Delete[(fnGetActionRepayment/@actionCombination),playerKey]]=={0}&&fnGetActionMessage[actionCombination
          [[playerKey]]]=="y"&&fnGetActionRepayment[actionCombination[[playerKey]]]<=-n R},(* Rule (3a) *)
1030 {-p,               True}(* Rule (3)&(4) *)
1031 }]];
1032
1033 (* ----Transfer function: borrower to lender, controlled by lender ---- *)
1034
1035 rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
1036 {+n R+e, Union[Delete[(fnGetActionMessage/@actionCombination),playerKey]]=={"n"}&&Union[Delete[(fnGetActionRepayment/@actionCombination),playerKey]]=={0}&&fnGetActionMessage[actionCombination[[
          playerKey]]]=="y"&&fnGetActionRepayment[actionCombination[[playerKey]]]<=-n R},(* Rule (3a) *)
1037 {0,                True}(* Rule (3)&(4) *)
1038 }]];
1039
1040 (* ----Sanctions function: borrower to deadweigh, controlled by group member---- *)
1041
1042 sanctionFunction[abilityCombination_,actionCombination_,playerNr_]:=0;
1043
1044 (* ----How the layers (apart from repayment) combine to yield the payoff. This is a game-specific information. ---- *)
1045
1046 payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+punishmentFunction[abilityCombination,actionCombination,playerNr]+rewardFunction[abilityCombination,actionCombination,playerNr]+
          sanctionFunction[abilityCombination,actionCombination,playerNr];
1047
1048 (* ----Assumptions---- *)
1049
1050 $Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
1051
```

```
(* Caution: Use == for comparisons, do not use = instead! Watch out for contradictory assumptions! Do not use ambiguous assumptions like <=! *)
$Assumptions=And[
R > 0, (* sign gross interest *)
p > 2R, (* intensity punishment *)
e > 0]; (* intensity reward *)



(* ============ *)
(* GAME RESULTS *)

(* ---- Game Construction & Solution ---- *)

 solutionMatrixTos=MapIndexed[fnSolveGame[game[#2],feasibilityMatrix[#2]]&,emptyTos,{n}];

(* ---- Main Results ---- *)

 headline["Map_of_equilibria"]
 Map[fnGetEqActions,solutionMatrixTos,{n}];
 fnDisplayTos[%]

 lenderH=headline["Map_of_lender_payoff"] ;
 MapIndexed[fnGetEqLenderPayoffTotal[#2,#1]&,solutionMatrixTos,{n}];
 Map[fnFormatCombinationsList,%,{n}];
 lender=fnDisplayTos[%];

 borrowerH=headline["Map_of_borrower_payoffs\n"] ;
 MapIndexed[fnGetEqPayoffs[#2,#1]&,solutionMatrixTos,{n}];
 Map[fnFormatCombinationsList,%,{n}];
 borrower=fnDisplayTos[%];

 lossH=headline["Map_of_deadweight_loss"];
 MapIndexed[fnGetEqLoss[#2,#1]&,solutionMatrixTos,{n}];
 Map[fnFormatCombinationsList,%,{n}];
 loss=fnDisplayTos[%];

 Grid[{{lenderH,borrowerH,lossH},{lender,borrower,loss}},Frame->All,Spacings->{5,5}] (* Combines results in a grid *)

(* ---- Further Results ---- *)

 headline["Traceable_Solution"]
 MapIndexed[fnSolveGameTraceable[game[#2],feasibilityMatrix[#2]]&,emptyTos,{n}];
 Map[fnFormatSolutionMatrix,%,{n}];
 Map[fnDisplayMatrix,%,{n}];
 fnDisplayTos[%]

 headline["Payoff_layers_(apart_from_repayment)_"]
 layer=payoffLayers ; (* Choose payoff layer here *)
 fnGetPayoffLayerCombinationsTos[layer]/.{Null->cInfString};
 Map[fnFormatCombination[#]&,%,{2n}];
 Map[fnDisplayMatrix[#]&,%,{n}];
 fnDisplayTos[%]
```

```
(*
headline("Traceable Solution – A single shape") (* Useful to check huge outputs *)
MapIndexed[fnSolveGameTraceable[game[#2],feasibilityMatrix[#2]]&,emptyTos,{1}];
%[[2,2]] (* addresses the shape by its abilityKey : make sure the shape exists ! *)
Map[fnFormatSolutionMatrix,%,{1}];
fnDisplayMatrix[%]
*)
```

A.6.2 Software output (example)

Punishment strategy

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	0 | 0	0 | -p	-p | -p	-p | e+2R
P1:0R-y	-p | 0	-p | -p	-p | -p	0 | 0
P1:1R	-p | -p	-p | -p	0 | 0	0 | 0
P1:2R	e+2R | -p	0 | 0	0 | 0	0 | 0

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R-n | 0R-n	0R-n | 0R-n	
A1:-R	0R-n | 0R-n	0R-n | 0R-n 1R | 1R	1R | 1R
A1:-2 R		1R | 1R	1R | 1R

A*i*: ability to repay for borrower *i*
P*i*: action of player *i*
× infeasible actions

Map of lender payoff

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	
A1:-R	0	0 2R	2R
A1:-2 R		2R	2R

Map of borrower payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0 | 0	0 | 0	
A1:-R	0 | 0	0 | 0 -R | -R	-R | -R
A1:-2 R		-R | -R	-R | -R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	
A1:-R	0	0 0	0
A1:-2 R		0	0

Traceable solution

A2:0

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[0,0]	[0,-p]	×	×
	P1:0R-y	[-p,0]	[-p,-p]	×	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-p]	×	×
	P1:0R-y	[-p,0]	[-p,-p]	×	×
	P1:1R	[-p-R,-p]	[-p-R,-p]	×	×
	P1:2R	×	×	×	×
A1:-2R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-p]	×	×
	P1:0R-y	[-p,0]	[-p,-p]	×	×
	P1:1R	[-p-R,-p]	[-p-R,-p]	×	×
	P1:2R	[e,-p]	[-2 R,0]	×	×

A2:-R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[0,0]	[0,-p]	[-p,-p-R]	×
	P1:0R-y	[-p,0]	[-p,-p]	[-p,-p-R]	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-p]	[-p,-p-R]	×
	P1:0R-y	[-p,0]	[-p,-p]	[-p,-p-R]	×
	P1:1R	[-p-R,-p]	[-p-R,-p]	[-R,-R]	×
	P1:2R	×	×	×	×
A1:-2R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-p]	[-p,-p-R]	×
	P1:0R-y	[-p,0]	[-p,-p]	[-p,-p-R]	×
	P1:1R	[-p-R,-p]	[-p-R,-p]	[-R,-R]	×
	P1:2R	[e,-p]	[-2 R,0]	[-2 R,-R]	×

A2:-2R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[0,0]	[0,-p]	[-p,-p-R]	[-p,e]
	P1:0R-y	[-p,0]	[-p,-p]	[-p,-p-R]	[0,-2 R]
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-p]	[-p,-p-R]	[-p,e]
	P1:0R-y	[-p,0]	[-p,-p]	[-p,-p-R]	[0,-2 R]
	P1:1R	[-p-R,-p]	[-p-R,-p]	[-R,-R]	[-R,-2 R]
	P1:2R	×	×	×	×
A1:-2R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-p]	[-p,-p-R]	[-p,e]
	P1:0R-y	[-p,0]	[-p,-p]	[-p,-p-R]	[0,-2 R]
	P1:1R	[-p-R,-p]	[-p-R,-p]	[-R,-R]	[-R,-2 R]
	P1:2R	[e,-p]	[-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

A.7 Potential improvements

Note that the code presented in this dissertation is prototype code not industry code. The following sections therefore show the directions in which further improvements can be made.

A.7.1 Functionality

The flows of value (p. 103) in the repayment game should be implemented in a conceptually cleaner way. A general 'flow-function' should capture origin, target and value of a flow. This would also require a thought-through naming convention. Finally, the individual flows must be assembled to the players' payoff functions.

It would be nice to have a function that calculates avoidable harm (see 11) automatically. It would have to somehow compare the second player's payoff under various actions of the first player – and vice versa. Caution: In a game with three players the person responsible for causing avoidable harm becomes ambiguous.

The scope of application could be extended by including more solution concepts. A consideration could be given to sequentially played games, continuous games, refined game theoretic equilibrium concepts. Maybe even behavioural rules could be implemented to improve the match between theory and reality.

I also have the feeling that with more expertise in informatics, numerics and algorithm theory the performance of the code could be improved with respect to CPU and memory consumption. This quickly becomes an issue when solving 'bigger' games. Especially solving games with more than three players takes a considerable amount of time. So recycling and reusing every calculation is important. In symmetric games, for example, symmetric shapes should be calculated only once and the solution-matrix should simply be transposed for the other cases.

A.7.2 Usability

A pressing issue to make the algorithm accessible to a wider audience of users would be to increase its usability. This includes a user interface for easy input without the need for Mathematica knowledge. It would also include plausibility checks of the user input. For example contradicting assumptions should be detected right away, before executing the algorithm to solve the game. Likewise whether every borrower has a well-defined action set should be checked beforehand. It would also include a nicely formatted output and some export functions. When I

copied the game results to the LaTeX-file of this dissertation, I used Right-Click "Copy as LaTeX" in the Mathematica context menu. After inserting the string into LaTeX each time I performed numerous replacements to produce nicely formatted tables.

For map of equilibria/payoffs/deadweight loss		**For traceable solution**	
Replace …	by …	Replace …	by …
`$\|$`	`\textbar\`	`array`	`tabular`
`\fbox`	`\makecell`	`\fbox{$`	`{`
`$\text{`		`$}`	`}`
`$}`		`$\times$`	`\texttimes\`
`$0`	`0}`	`\text{}`	
`$\backslash $n`	`\\`		
`\text{`	`\thead{`		

Thereafter I had to manually insert some horizontal lines and put everything in an `lstlistings` environment.

Actually, the whole process could be automated using Mathematica's extensive collection of export functions like `ExportString[%,"LaTeXFragment"]` or like the command `Export["file.tex,%,"LaTeX"]`. By using the commands `StringReplace` or `ConversionRules` the necessary replacements could be defined. By `SetDirectory[]` the location where the exported file is saved can be specified, such that the ready prepared file just has to be inserted into LaTeX using the latex command `\input{file}`.

A.7.3 Code structure

The code should be structured better making use of Mathematica packages. A prerequisite to do this is a thorough analysis of which functions can be private and which methods should be public. The defined Mathematica packages could then simply be loaded at the beginning of the code and the user would not need to manually load the individual code blocks (p. 154) into the Mathematica kernel.

A.7.4 Documentation

Finally, the documentation of the code could be improved. It would be particularly helpful to draw some charts that illuminate the dependencies between the functions of the algorithm. Maybe a dataflow-diagram as used in flow-based programming would be adequate.

B Detailed solutions of repayment games

C. Prem, *The Theory of Credit Contracts*,
https://doi.org/10.1007/978-3-658-29362-8

B.1 Individual liability

```
(* GAME DEFINITION – Individual liability *)

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-M, -fnGetActionRepayment[actionCombination[[playerKey]]]<R (* punishment if default ... *)
&& -Plus@@(fnGetActionRepayment/@actionCombination)< 2 R (* ... and peer did not kindly pay for him *)},
{0, True}(* No punishment otherwise *)
}]];

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=0;

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerKey_]:=0;

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+ punishmentFunction[abilityCombination,actionCombination,playerNr] + rewardFunction[abilityCombination,actionCombination,playerNr]+
        sanctionFunction[abilityCombination,actionCombination,playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
M>R]; (* intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

INDIVIDUAL LIABILITY

Medium punishment intensity ($R < M$)

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-M | -M	-M | 0	0 | 0
P1:1R	0 | -M	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 1R	0R | 1R
A1:-R	1R | 0R	1R | 1R	1R | 1R
A1:-2 R	1R | 0R	1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	R	R
A1:-R	R	2R	2R
A1:-2 R	R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2M	-M	-M
A1:-R	-M	0	0
A1:-2 R	-M	0	0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
A1:0	P1:0R	[-M,-M]	×	×
	P1:1R	×	×	×
	P1:2R	×	×	×
A1:-R	P1:0R	[-M,-M]	×	×
	P1:1R	[-R,-M]	×	×
	P1:2R	×	×	×
A1:-2 R	P1:0R	[-M,-M]	×	×
	P1:1R	[-R,-M]	×	×
	P1:2R	[-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
P1:0R	[-M,-M]	[-M,-R]	×
P1:1R	×	×	×
P1:2R	×	×	×
P1:0R	[-M,-M]	[-M,-R]	×
P1:1R	[-R,-M]	[-R,-R]	×
P1:2R	×	×	×
P1:0R	[-M,-M]	[-M,-R]	×
P1:1R	[-R,-M]	[-R,-R]	×
P1:2R	[-2 R,0]	[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
P1:0R	[-M,-M]	[-M,-R]	[0,-2 R]
P1:1R	×	×	×
P1:2R	×	×	×
P1:0R	[-M,-M]	[-M,-R]	[0,-2 R]
P1:1R	[-R,-M]	[-R,-R]	[-R,-2 R]
P1:2R	×	×	×
P1:0R	[-M,-M]	[-M,-R]	[0,-2 R]
P1:1R	[-R,-M]	[-R,-R]	[-R,-2 R]
P1:2R	[-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

B.2 Joint liability – High punishment intensity

```
(* GAME DEFINITION – Joint liability with high punishment intensity *)

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender flow ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-P, Plus@@(fnGetActionRepayment/@actionCombination)>-2R},(* Punishment if group defaults *)
{0, True}(* No punishment otherwise *)
}]];
(* Order of conditions matters! First that yields TRUE is taken *)

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_, actionCombination_,playerKey_] := 0;

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_, actionCombination_,playerKey_] := 0;

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_, actionCombination_,playerNr_] :=
+ punishmentFunction[abilityCombination, actionCombination,playerNr]
+ rewardFunction[abilityCombination, actionCombination, playerNr]
+ sanctionFunction[abilityCombination, actionCombination, playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first . *)
$Assumptions=And[
R>0, (* sign gross interest *)
P>2R]; (* intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

High punishment intensity
$(2R < P)$

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-P | -P	-P | -P	0 | 0
P1:1R	-P | -P	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 0R	0R | 2R
A1:-R	0R | 0R	0R | 0R 1R | 1R	0R | 2R 1R | 1R
A1:-2 R	2R | 0R	1R | 1R 2R | 0R	0R | 2R 1R | 1R 2R | 0R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	2R
A1:-R	0	0 2R	2R 2R
A1:-2 R	2R	2R 2R	2R 2R 2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2P	-2P	0
A1:-R	-2P	-2P 0	0 0
A1:-2 R	0	0 0	0 0 0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	[-P-R,-P]	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	[-P-R,-P]	×	×
P1:2R	[-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	×
P1:1R	[-P-R,-P]	[-R,-R]	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	×
P1:1R	[-P-R,-P]	[-R,-R]	×
P1:2R	[-2 R,0]	[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	[0,-2 R]
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	[0,-2 R]
P1:1R	[-P-R,-P]	[-R,-R]	[-R,-2 R]
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	[0,-2 R]
P1:1R	[-P-R,-P]	[-R,-R]	[-R,-2 R]
P1:2R	[-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

B.3 Joint liability – Adaptive punishment intensity

```
(* GAME DEFINITION – Joint liability with adaptive punishment intensity *)

(* NUMBER OF PLAYERS---- *)
n=2;

(* ACTIONSET for each player---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight flow ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-P,      -Plus@@(fnGetActionRepayment/@actionCombination)==0}, (* Punishment if no repayment *)
{-M,      -Plus@@(fnGetActionRepayment/@actionCombination)<n R}, (* Punishment if partial repayment *)
{0,          True}(*No punishment otherwise*)}]];
(* Order of conditions matters! First that yields TRUE is taken *)

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=0;

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerKey_]:=0;

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=
+punishmentFunction[abilityCombination,actionCombination,playerNr]
+rewardFunction[abilityCombination,actionCombination,playerNr]
+sanctionFunction[abilityCombination,actionCombination,playerNr];

(* ASSUMPTIONS---- *)
$Assumptions=True;(*Hygiene:Clearing list of global assumptions first.*)
$Assumptions=And[
R>0,(* sign gross interest *)
R<M<2R, (* intensity mild punishment *)
P>R+m];(* intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

Note: You can even get rid of the free-riding equilibria by adding the following punishment rule to the punishment function:

```
{-M,      -Plus @@ (fnGetActionRepayment /@ actionCombination) == n R && fnGetActionRepayment[actionCombination[[playerKey]]] == 0}
```

JOINT LIABILITY

Adaptive punishment intensity
($R < M < 2R$ **and** $R + M < P$)

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-P | -P	-M | -M	0 | 0
P1:1R	-M | -M	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 1R	0R | 2R
A1:-R	1R | 0R	1R | 1R	0R | 2R 1R | 1R
A1:-2 R	2R | 0R	1R | 1R 2R | 0R	0R | 2R 1R | 1R 2R | 0R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	R	2R
A1:-R	R	2R	2R 2R
A1:-2 R	2R	2R 2R	2R 2R 2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2P	-2M	0
A1:-R	-2M	0	0 0
A1:-2 R	0	0 0	0 0 0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	[-M-R,-M]	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	[-M-R,-M]	×	×
P1:2R	[-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-M,-M-R]	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-M,-M-R]	×
P1:1R	[-M-R,-M]	[-R,-R]	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-M,-M-R]	×
P1:1R	[-M-R,-M]	[-R,-R]	×
P1:2R	[-2 R,0]	[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-M,-M-R]	[0,-2 R]
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-M,-M-R]	[0,-2 R]
P1:1R	[-M-R,-M]	[-R,-R]	[-R,-2 R]
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-M,-M-R]	[0,-2 R]
P1:1R	[-M-R,-M]	[-R,-R]	[-R,-2 R]
P1:2R	[-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

B.4 Joint liability – High punishment intensity (with social sanctions)

```
(* GAME DEFINITION - Joint liability - high punishment intensity (with social sanctions) *)

(* NUMBER OF PLAYERS ---- *)
n=2;
(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-P, Plus@@(fnGetActionRepayment/@actionCombination)>-2R},(* Punishment if group defaults *)
{0, True}}(* No punishment otherwise *)
]];
(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=0;
(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-Sn, -Plus@@(fnGetActionRepayment/@actionCombination)<n R (* not all loans are repaid *)
&& -Plus@@abilityCombination>=n R (* full repayment is possible *)
&& -fnGetActionRepayment[actionCombination[[playerKey]]]<-abilityCombination[[playerKey]](* player still has resources *)}, (* not-helping condition *)
{-Sf, -Plus@@(fnGetActionRepayment/@actionCombination)>=n R (* all loans are repaid *)
&& -abilityCombination[[playerKey]]>=R (* can repay his loan *)
&& fnGetActionRepayment[actionCombination[[playerKey]]]==0 (* does not repay his loan *)}, (* free-riding condition *)
{0,True}
}]];

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+punishmentFunction[abilityCombination,actionCombination,playerNr]+rewardFunction[abilityCombination,actionCombination,playerNr]+
    sanctionFunction[abilityCombination,actionCombination,playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
Sn>(n-1)R,(* intensity not-helping sanctions *)
Sf>R, (* intensity free-riding sanctions *)
P>nR]; (* intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

(with social sanctions)
High punishment intensity
($2R < P$)

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-P | -P	-P | -P	0 | 0
P1:1R	-P | -P	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 0R	0R | 2R
A1:-R	0R | 0R	1R | 1R	1R | 1R
A1:-2 R	2R | 0R	1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	2R
A1:-R	0	2R	2R
A1:-2 R	2R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2P	-2P	0
A1:-R	-2P	0	0
A1:-2 R	0	0	0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	×	×
P1:1R	[-P-R,-P]	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P-Sn,-P]	×	×
P1:1R	[-P-R-Sn,-P]	×	×
P1:2R	[-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P]	[-P,-P-R]	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P-Sn,-P-Sn]	[-P-Sn,-P-R]	×
P1:1R	[-P-R,-P-Sn]	[-R,-R]	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P-Sn,-P-Sn]	[-P-Sn,-P-R]	×
P1:1R	[-P-R-Sn,-P-Sn]	[-R,-R]	×
P1:2R	[-2 R,-Sf]	[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
P1:0R	[-P,-P-Sn]	[-P,-P-R-Sn]	[0,-2 R]
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P-Sn,-P-Sn]	[-P-Sn,-P-R-Sn]	[-Sf,-2 R]
P1:1R	[-P-R,-P-Sn]	[-R,-R]	[-R,-2 R]
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-P-Sn,-P-Sn]	[-P-Sn,-P-R-Sn]	[-Sf,-2 R]
P1:1R	[-P-R-Sn,-P-Sn]	[-R,-R]	[-R,-2 R]
P1:2R	[-2 R,-Sf]	[-2 R,-R]	[-2 R,-2 R]

B.5 Joint liability – Medium punishment intensity

```
1  (* GAME DEFINITION – Joint liability with medium punishment intensity *)
2
3  (* NUMBER OF PLAYERS ---- *)
4  n=2;
5
6  (* ACTIONSET for each player ---- *)
7   actionSet[1]={
8   {"0R",0,""},
9   {"1R",-R,""},
10  {"2R",-2R,""}};
11  actionSet[2]=actionSet[1];
12
13 (* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender flow ---- *)
14 punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
15  {-M,          Plus@@(fnGetActionRepayment/@actionCombination)>-2R},(* Punishment if default *)
16  {0,             True}(* No punishment otherwise *)
17  }]];
18 (* Order of conditions matters! First that yields TRUE is taken *)
19
20 (* REWARD FUNCTION: lender to borrower flow ---- *)
21 rewardFunction[abilityCombination_, actionCombination_,playerKey_] := 0;
22
23 (* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
24  sanctionFunction[abilityCombination_, actionCombination_,playerKey_] := 0;
25
26 (* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
27  payoffLayers[abilityCombination_, actionCombination_,playerNr_] :=
28 + punishmentFunction[abilityCombination, actionCombination,playerNr]
29 + rewardFunction[abilityCombination, actionCombination, playerNr]
30 + sanctionFunction[abilityCombination, actionCombination, playerNr];
31
32 (* ASSUMPTIONS ---- *)
33 $Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
34 $Assumptions=And[
35 R>0, (* sign gross interest *)
36 R<M<2R]; (* intensity punishment *)
37 (* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

Medium punishment intensity
($R < M < 2R$)

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-M | -M	-M | -M	0 | 0
P1:1R	-M | -M	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 0R	0R | 0R
A1:-R	0R | 0R	0R | 0R 1R | 1R	0R | 0R 1R | 1R
A1:-2 R	0R | 0R	0R | 0R 1R | 1R	0R | 0R 1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	0
A1:-R	0	0 2R	0 2R
A1:-2 R	0	0 2R	0 2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2M	-2M	-2M
A1:-R	-2M	-2M 0	-2M 0
A1:-2 R	-2M	-2M 0	-2M 0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
A1:0	P1:0R [-M,-M]	×	×
	P1:1R ×	×	×
	P1:2R ×	×	×
A1:-R	P2:0R	P2:1R	P2:2R
	P1:0R [-M,-M]	×	×
	P1:1R [-M-R,-M]	×	×
	P1:2R ×	×	×
A1:-2 R	P2:0R	P2:1R	P2:2R
	P1:0R [-M,-M]	×	×
	P1:1R [-M-R,-M]	×	×
	P1:2R [-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
A1:0	P1:0R [-M,-M]	[-M,-M-R]	×
	P1:1R ×	×	×
	P1:2R ×	×	×
A1:-R	P2:0R	P2:1R	P2:2R
	P1:0R [-M,-M]	[-M,-M-R]	×
	P1:1R [-M-R,-M]	[-R,-R]	×
	P1:2R ×	×	×
A1:-2 R	P2:0R	P2:1R	P2:2R
	P1:0R [-M,-M]	[-M,-M-R]	×
	P1:1R [-M-R,-M]	[-R,-R]	×
	P1:2R [-2 R,0]	[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
A1:0	P1:0R [-M,-M]	[-M,-M-R]	[0,-2 R]
	P1:1R ×	×	×
	P1:2R ×	×	×
A1:-R	P2:0R	P2:1R	P2:2R
	P1:0R [-M,-M]	[-M,-M-R]	[0,-2 R]
	P1:1R [-M-R,-M]	[-R,-R]	[-R,-2 R]
	P1:2R ×	×	×
A1:-2 R	P2:0R	P2:1R	P2:2R
	P1:0R [-M,-M]	[-M,-M-R]	[0,-2 R]
	P1:1R [-M-R,-M]	[-R,-R]	[-R,-2 R]
	P1:2R [-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

B.6 Joint liability – Medium punishment (with social sanctions)

```
(* GAME DEFINITION - Joint liability with medium punishment intensity (with social sanctions) *)

(* NUMBER OF PLAYERS ---- *)
n=2;
(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-M, Plus@@(fnGetActionRepayment/@actionCombination)>-2R},(* Punishment if group defaults *)
{0, True}(* No punishment otherwise *)
}]];
(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=0;
(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-Sn, -Plus@@(fnGetActionRepayment/@actionCombination)<nR (* not all loans are repaid *)
&& -Plus@@abilityCombination>=nR (* full repayment is possible *)
&& -fnGetActionRepayment[actionCombination[[playerKey]]]<-abilityCombination[[playerKey]](* player still has resources *)}, (* not-helping condition *)
{-Sf, -Plus@@(fnGetActionRepayment/@actionCombination)>=nR (* all loans are repaid *)
&& -abilityCombination[[playerKey]]>=R (* can repay his loan *)
&& fnGetActionRepayment[actionCombination[[playerKey]]]==0 (* does not repay his loan *)}, (* free-riding condition *)
{0,True}
}]];

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+punishmentFunction[abilityCombination,actionCombination,playerNr]+rewardFunction[abilityCombination,actionCombination,playerNr]+
    sanctionFunction[abilityCombination,actionCombination,playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
Sn>(n-1)R,(* intensity not-helping sanctions *)
Sf>R, (* intensity free-riding sanctions *)
R<M<2R]; (* intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

(with social sanctions)
Medium punishment intensity
($R < M < 2R$)

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-M | -M	-M | -M	0 | 0
P1:1R	-M | -M	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 0R	0R | 2R
A1:-R	0R | 0R	1R | 1R	1R | 1R
A1:-2 R	2R | 0R	1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	2R
A1:-R0	0	2R	2R
A1:-2 R	2R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2M	-2M	0
A1:-R	-2M	0	0
A1:-2 R	0	0	0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
P1:0R	[-M,-M]	×	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-M,-M]	×	×
P1:1R	[-M-R,-M]	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-M-Sn,-M]	×	×
P1:1R	[-M-R-Sn,-M]	×	×
P1:2R	[-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
P1:0R	[-M,-M]	[-M,-M-R]	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-M-Sn,-M-Sn]	[-M-Sn,-M-R]	×
P1:1R	[-M-R,-M-Sn]	[-R,-R]	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-M-Sn,-M-Sn]	[-M-Sn,-M-R]	×
P1:1R	[-M-R-Sn,-M-Sn]	[-R,-R]	×
P1:2R	[-2 R,-Sf]	[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
P1:0R	[-M,-M-Sn]	[-M,-M-R-Sn]	[0,-2 R]
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-M-Sn,-M-Sn]	[-M-Sn,-M-R-Sn]	[-Sf,-2 R]
P1:1R	[-M-R,-M-Sn]	[-R,-R]	[-R,-2 R]
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	[-M-Sn,-M-Sn]	[-M-Sn,-M-R-Sn]	[-Sf,-2 R]
P1:1R	[-M-R-Sn,-M-Sn]	[-R,-R]	[-R,-2 R]
P1:2R	[-2 R,-Sf]	[-2 R,-R]	[-2 R,-2 R]

B.7 Joint liability – Medium punishment (with social sanctions) [n=3]

```
(* GAME DEFINITION - Joint liability with medium punishment intensity (with social sanctions) - three borrowers *)

(* NUMBER OF PLAYERS ---- *)
n=3;
(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];
actionSet[3]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-M, Plus@@(fnGetActionRepayment/@actionCombination)>-n R},(* Punishment if group defaults *)
{0, True}(* No punishment otherwise *)
}]];
(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=0;
(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-Sn, -Plus@@(fnGetActionRepayment/@actionCombination)<n R (* not all loans are repaid *)
&& -Plus@@abilityCombination>=n R (* full repayment is possible *)
&& -fnGetActionRepayment[actionCombination[[playerKey]]]<-abilityCombination[[playerKey]](* player still has resources *)}, (* not-helping condition *)
{-Sf, -Plus@@(fnGetActionRepayment/@actionCombination)>= n R (* all loans are repaid *)
&& -abilityCombination[[playerKey]]>=R (* can repay his loan *)
&& fnGetActionRepayment[actionCombination[[playerKey]]]==0 (* does not repay his loan *)}, (* free-riding condition *)
{0,True}
}]];
(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+ punishmentFunction[abilityCombination,actionCombination,playerNr] + rewardFunction[abilityCombination,actionCombination,playerNr]+
    sanctionFunction[abilityCombination,actionCombination,playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
Sn>(n-1)R,(* intensity not-helping sanctions *)
Sf>R, (* intensity free-riding sanctions *)
R<M<2R]; (* intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY [n=3]

(with social sanctions)
Medium punishment intensity
($R < M < 2R$)

Punishment strategy

P1:0R

	P3:0R	P3:1R	P3:2R
P2:0R	-M | -M | -M	-M | -M | -M	-M | -M | -M
P2:1R	-M | -M | -M	-M | -M | -M	0 | 0 | 0
P2:2R	-M | -M | -M	0 | 0 | 0	0 | 0 | 0

P1:1R

	P3:0R	P3:1R	P3:2R
P2:0R	-M | -M | -M	-M | -M | -M	0 | 0 | 0
P2:1R	-M | -M | -M	0 | 0 | 0	0 | 0 | 0
P2:2R	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0

P1:2R

	P3:0R	P3:1R	P3:2R
P2:0R	-M | -M | -M	0 | 0 | 0	0 | 0 | 0
P2:1R	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
P2:2R	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0

Map of equilibria

	A1:0			**A1:-R**			**A1:-2 R**		
	A3:0	**A3:-R**	**A3:-2 R**	**A3:0**	**A3:-R**	**A3:-2 R**	**A3:0**	**A3:-R**	**A3:-2 R**
A2:0	0R | 0R | 0R	0R | 0R | 0R	0R | 0R | 0R	0R | 0R | 0R	0R | 0R | 0R	1R | 0R | 2R	0R | 0R | 0R	2R | 0R | 1R	1R | 0R | 2R 2R | 0R | 1R
A2:-R	0R | 0R | 0R	0R | 0R | 0R	0R | 1R | 2R	0R | 0R | 0R	1R | 1R | 1R	1R | 1R | 1R	2R | 1R | 0R	1R | 1R | 1R	1R | 1R | 1R
A2:-2 R	0R | 0R | 0R	0R | 2R | 1R	0R | 1R | 2R 0R | 2R | 1R	1R | 2R | 0R	1R | 1R | 1R	1R | 1R | 1R	1R | 2R | 0R 2R | 1R | 0R	1R | 1R | 1R	1R | 1R | 1R

Traceable solution has 3^3 shapes of dimension 3^3 and is too big to draw on DIN A4 paper.

B.8 Individual liability (with social sanctions)

```
(* GAME DEFINITION – Individual liability (with social sanctions) *)

(* NUMBER OF PLAYERS ---- *)
n=2;
(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R",0,""},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-M, -fnGetActionRepayment[actionCombination[[playerKey]]]<R (*punishment if default ... *)},
&&-Plus@@(fnGetActionRepayment/@actionCombination)<2 R (*... and peer did not kindly pay for him *)},
{0, True}(*No punishment otherwise *)
}]];

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=0;

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{-Sn, MemberQ[Delete[abilityCombination,playerKey],0]} (* peer cannot repay *)
&&-abilityCombination[[playerKey]]>R (*borrower has more resources than he needs for his own *)
&&-Plus@@(fnGetActionRepayment/@actionCombination)<n R (*borrower does not help *)
}, (* not-helping condition *)
{0,True}
}]];

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+punishmentFunction[abilityCombination,actionCombination,playerNr]+rewardFunction[abilityCombination,actionCombination,playerNr]+
    sanctionFunction[abilityCombination,actionCombination,playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (*Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (*sign gross interest *)
Sn>(n-1)R,(*intensity not-helping sanctions *)
M>R]; (*intensity punishment *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

INDIVIDUAL LIABILITY

(with social sanctions)
Medium punishment intensity
($R < M$)

Punishment strategy

	P2:0R	P2:1R	P2:2R
P1:0R	-M | -M	-M | 0	0 | 0
P1:1R	0 | -M	0 | 0	0 | 0
P1:2R	0 | 0	0 | 0	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R | 0R	0R | 1R	0R | 2R
A1:-R	1R | 0R	1R | 1R	1R | 1R
A1:-2 R	2R | 0R	1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	R	2R
A1:-R	R	2R	2R
A1:-2 R	2R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2M	-M	0
A1:-R	-M	0	0
A1:-2 R	0	0	0

Traceable solution

A2:0

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×	×
P1:1R	×[-R,-M]	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M-Sn,-M]	×	×
P1:1R	×[-R-Sn,-M]	×	×
P1:2R	×[-2 R,0]	×	×

A2:-R

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×[-M,-R]	×
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×[-M,-R]	×
P1:1R	×[-R,-M]	×[-R,-R]	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×[-M,-R]	×
P1:1R	×[-R,-M]	×[-R,-R]	×
P1:2R	×[-2 R,0]	×[-2 R,-R]	×

A2:-2 R

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M-Sn]	×[-M,-R-Sn]	×[0,-2 R]
P1:1R	×	×	×
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×[-M,-R]	×[0,-2 R]
P1:1R	×[-R,-M]	×[-R,-R]	×[-R,-2 R]
P1:2R	×	×	×

	P2:0R	P2:1R	P2:2R
P1:0R	×[-M,-M]	×[-M,-R]	×[0,-2 R]
P1:1R	×[-R,-M]	×[-R,-R]	×[-R,-2 R]
P1:2R	×[-2 R,0]	×[-2 R,-R]	×[-2 R,-2 R]

B.9 Cross-reporting in [RS13] with original defect

```
(* GAME DEFINITION - cross-reporting in RS13 with original defect *)

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R-n",0,"n"},
{"0R-y",0,"y"},
{"1R",-R,"y"},
{"2R",-2R,"y"}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
(* Rule (1) *)
{0,   (fnGetActionMessage/@actionCombination)==ConstantArray["y",n] (* all say "yes" *)
  && -Plus@@(fnGetActionRepayment/@actionCombination)>=n R (* loan is repaid *)},
(* Rule (2a) *)
{0,   (fnGetActionRepayment/@actionCombination)==ConstantArray[0,n] (* no repayments *)
  && fnGetActionMessage[actionCombination[[playerKey]]]=="n" (* player said "no" *)},
(* Rule (2b) *)
{-P, (fnGetActionRepayment/@actionCombination)==ConstantArray[0,n] (* no repayments *)
  &&fnGetActionMessage[actionCombination[[playerKey]]]=="y"(* player said "yes" *) },
(* Rule (3a) *)
{-P, fnGetActionMessage[actionCombination[[playerKey]]]=="n" (* player said "no" *)
  &&-Plus@@(fnGetActionRepayment/@actionCombination)>=n R (* but loan is repaid *)},
(* Rule (3b) *)
{0, Union[Delete[(fnGetActionMessage/@actionCombination),playerKey]]=={"n"} (* others say no *)
  &&Union[Delete[(fnGetActionRepayment/@actionCombination),playerKey]]=={0} (* others pay zero *)
  &&-fnGetActionRepayment[actionCombination[[playerKey]]]==n R (* player pays all alone *)},
(* Rule (4) *)
{-P, True}
}]];

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
(* Rule (3b) *)
{+n R+e, Union[Delete[(fnGetActionMessage/@actionCombination),playerKey]]=={"n"} (* others say no *)
  &&Union[Delete[(fnGetActionRepayment/@actionCombination),playerKey]]=={0} (* others pay zero *)
  &&-fnGetActionRepayment[actionCombination[[playerKey]]]==n R (* player pays all alone *)},
{0,   True}
}]];

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerNr_]:=0;

(* ASSEMBLING PAYOFF LAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+ punishmentFunction[
    abilityCombination,actionCombination,playerNr] + rewardFunction[abilityCombination,
    actionCombination,playerNr]+ sanctionFunction[abilityCombination,actionCombination,
    playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
P>2R, (* intensity punishment *)
e>0]; (* intensity reward *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

(with cross-reporting as in [RS13])
High punishment intensity
($2R < P, 0 < e$)

Punishment and reward strategy

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	0 | 0	0 | -P	-P | -P	-P | e+2R
P1:0R-y	-P | 0	-P | -P	-P | -P	0 | 0
P1:1R	-P | -P	-P | -P	0 | 0	0 | 0
P1:2R	e+2R | -P	0 | 0	0 | 0	0 | 0

Ai:	ability to repay for borrower i
Pi:	action of player i
×	infeasible actions

Map of equilibria

	A2:0	A2:-R	A2:-2 R
A1:0	0R-n | 0R-n	0R-n | 0R-n	
A1:-R	0R-n | 0R-n	0R-n | 0R-n 1R | 1R	1R | 1R
A1:-2 R		1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	A2:-R	A2:-2 R
A1:0	0	0	
A1:-R	0	0 2R	2R
A1:-2 R		2R	2R

Map of deadweight loss

	A2:0	A2:-R	A2:-2 R
A1:0	0	0	
A1:-R	0	0 0	0
A1:-2 R		0	0

Traceable solution

A2:0

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[0,0]	[0,-P]	×	×
	P1:0R-y	[-P,0]	[-P,-P]	×	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-P]	×	×
	P1:0R-y	[-P,0]	[-P,-P]	×	×
	P1:1R	[-P-R,-P]	[-P-R,-P]	×	×
	P1:2R	×	×	×	×
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-P]	×	×
	P1:0R-y	[-P,0]	[-P,-P]	×	×
	P1:1R	[-P-R,-P]	[-P-R,-P]	×	×
	P1:2R	[e,-P]	[-2 R,0]	×	×

A2:-R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[0,0]	[0,-P]	[-P,-P-R]	×
	P1:0R-y	[-P,0]	[-P,-P]	[-P,-P-R]	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-P]	[-P,-P-R]	×
	P1:0R-y	[-P,0]	[-P,-P]	[-P,-P-R]	×
	P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	×
	P1:2R	×	×	×	×
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-P]	[-P,-P-R]	×
	P1:0R-y	[-P,0]	[-P,-P]	[-P,-P-R]	×
	P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	×
	P1:2R	[e,-P]	[-2 R,0]	[-2 R,-R]	×

A2:-2 R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[0,0]	[0,-P]	[-P,-P-R]	[-P,e]
	P1:0R-y	[-P,0]	[-P,-P]	[-P,-P-R]	[0,-2 R]
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-P]	[-P,-P-R]	[-P,e]
	P1:0R-y	[-P,0]	[-P,-P]	[-P,-P-R]	[0,-2 R]
	P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	[-R,-2 R]
	P1:2R	×	×	×	×
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	[0,0]	[0,-P]	[-P,-P-R]	[-P,e]
	P1:0R-y	[-P,0]	[-P,-P]	[-P,-P-R]	[0,-2 R]
	P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	[-R,-2 R]
	P1:2R	[e,-P]	[-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

B.10 Cross-reporting in [RS13] with corrected defect

```
(* GAME DEFINITION - cross-reporting in RS13 with corrected defect *)

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R-n",0,"n"},
{"0R-y",0,"y"},
{"1R",-R,"y"},
{"2R",-2R,"y"}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
(* Rule (1) *)
{0,   (fnGetActionMessage/@actionCombination)==ConstantArray["y",n] (* all say "yes" *)
  && -Plus@@(fnGetActionRepayment/@actionCombination)>=n R (* loan is repaid *)},
(* Rule (2a) *)
{0,   (fnGetActionRepayment/@actionCombination)==ConstantArray[0,n] (* no repayments *)
  && fnGetActionMessage/@actionCombination== ConstantArray["n",n] (* all say "no" *)},
(* Rule (2b) *) (* CHANGED RULE *)
{-P, (fnGetActionRepayment/@actionCombination)==ConstantArray[0,n] (* no repayments *)
  && fnGetActionMessage/@actionCombination!=ConstantArray["n",n] (* not all say "no" *)},
(* Rule (3a) *)
{-P, fnGetActionMessage[actionCombination[[playerKey]]]=="n" (* player said "no" *)
  &&-Plus@@(fnGetActionRepayment/@actionCombination)>=n R (* but loan is repaid *)},
(* Rule (3b) *)
{0,   Union[Delete[(fnGetActionMessage/@actionCombination),playerKey]]=={"n"} (*others say no*)
  &&Union[Delete[(fnGetActionRepayment/@actionCombination),playerKey]]=={0} (*others repay zero
      *)
  &&-fnGetActionRepayment[actionCombination[[playerKey]]]==n R (* player pays all alone *)},
(* Rule (4) *)
{-P, True}
}]];

(* REWARD FUNCTION: lender to borrower flow ---- *)
(* Rule (3b) *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
{+n R+e, Union[Delete[(fnGetActionMessage/@actionCombination),playerKey]]=={"n"} (*others say no
    *)
  &&Union[Delete[(fnGetActionRepayment/@actionCombination),playerKey]]=={0} (*others repay zero
    *)
  &&-fnGetActionRepayment[actionCombination[[playerKey]]]==n R (* player pays all alone *)},
{0, True}
}]];

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerNr_]:=0;

(* ASSEMBLING PAYOFF LAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+punishmentFunction[
    abilityCombination,actionCombination,playerNr]+rewardFunction[abilityCombination,
    actionCombination,playerNr]+sanctionFunction[abilityCombination,actionCombination,
    playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
P>2R, (* intensity punishment *)
e>0]; (* intensity reward *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

INDIVIDUAL LIABILITY

(cross-reporting as in [RS13], corrected)
High punishment intensity ($2R < P$, $0 < e$)

Punishment strategy

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	0 | 0	-P | -P	-P | -P	-P | e+2R
P1:0R-y	-P | -P	-P | -P	-P | -P	0 | 0
P1:1R	-P | -P	-P | -P	0 | 0	0 | 0
P1:2R	e+2R | -P	0 | 0	0 | 0	0 | 0

Ai: ability to repay for borrower *i*
Pi: action of player *i*
× infeasible actions

Map of equilibria

	A2:0	A2:-R	A2:-2 R
A1:0	0R-n | 0R-n 0R-y | 0R-y	0R-n | 0R-n 0R-y | 0R-y	0R-y | 2R
A1:-R	0R-n | 0R-n 0R-y | 0R-y	0R-n | 0R-n 0R-y | 0R-y 1R | 1R	0R-y | 2R 1R | 1R
A1:-2 R	2R | 0R-y	1R | 1R 2R | 0R-y	0R-y | 2R 1R | 1R 2R | 0R-y

Map of lender payoffs

	A2:0	A2:-R	A2:-2 R
A1:0	0 0	0 0	2R
A1:-R	0 0	0 0 2R	2R 2R
A1:-2 R	2R	2R 2R	2R 2R 2R

Map of deadweight loss

	A2:0	A2:-R	A2:-2 R
A1:0	0 -2P	0 -2P	0
A1:-R	0 -2P	0 -2P 0	0 0
A1:-2 R	0	0 0	0 0 0

Traceable solution

A2:0

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	×	×
P1:0R-y	[-P,-P]	[-P,-P]	×	×
P1:1R	×	×	×	×
P1:2R	×	×	×	×

A1:0

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	×	×
P1:0R-y	[-P,-P]	[-P,-P]	×	×
P1:1R	[-P-R,-P]	[-P-R,-P]	×	×
P1:2R	×	×	×	×

A1:-R

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	×	×
P1:0R-y	[-P,-P]	[-P,-P]	×	×
P1:1R	[-P-R,-P]	[-P-R,-P]	×	×
P1:2R	[e,-P]	[-2 R,0]	×	×

A1:-2 R

A2:-R

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	[-P,-P-R]	×
P1:0R-y	[-P,-P]	[-P,-P]	[-P,-P-R]	×
P1:1R	×	×	×	×
P1:2R	×	×	×	×

A1:0

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	[-P,-P-R]	×
P1:0R-y	[-P,-P]	[-P,-P]	[-P,-P-R]	×
P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	×
P1:2R	×	×	×	×

A1:-R

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	[-P,-P-R]	×
P1:0R-y	[-P,-P]	[-P,-P]	[-P,-P-R]	×
P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	×
P1:2R	[e,-P]	[-2 R,0]	[-2 R,-R]	×

A1:-2 R

A2:-2 R

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	[-P,-P-R]	[-P,e]
P1:0R-y	[-P,-P]	[-P,-P]	[-P,-P-R]	[0,-2 R]
P1:1R	×	×	×	×
P1:2R	×	×	×	×

A1:0

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	[-P,-P-R]	[-P,e]
P1:0R-y	[-P,-P]	[-P,-P]	[-P,-P-R]	[0,-2 R]
P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	[-R,-2 R]
P1:2R	×	×	×	×

A1:-R

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	[0,0]	[-P,-P]	[-P,-P-R]	[-P,e]
P1:0R-y	[-P,-P]	[-P,-P]	[-P,-P-R]	[0,-2 R]
P1:1R	[-P-R,-P]	[-P-R,-P]	[-R,-R]	[-R,-2 R]
P1:2R	[e,-P]	[-2 R,0]	[-2 R,-R]	[-2 R,-2 R]

A1:-2 R

B.11 Cross-reporting in [RS04]

```
(* GAME DEFINITION - cross-reporting as in RS04 *)

(* Auxilary functions to transform an actionCombination into an actionKeyCombination *)
fnActionToKey[playerKey_, action_]:=Sequence@@@Position[actionSet[playerKey],action];
fnActionToKeyCombination[actionCombination_]:=Flatten[MapIndexed[fnActionToKey[Sequence@@
        #2,#1]&,actionCombination,{1}]];

(* case in the paper where h>=2R and thus A=min[h,2R]=2R *)
A=2R

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R-n",0,"n"},
{"0R-y",0,"y"},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentRule[1]={
{-R, -(A+e), -A, -P},
{-P, -P,     -P, -(2R-A)},
{ 0,  -P,     0,  0},
{ 0,   0,    -P, 0}};
punishmentRule[2]=Transpose[punishmentRule[1]];

(* Punishment function chooses the adequate entries from the matrices defined above *)
punishmentFunction[abilityCombination_, actionCombination_, playerKey_]:=punishmentRule[playerKey][[
        Sequence@@fnActionToKeyCombination[actionCombination]]];

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardRule[1]={
{0, 0,  0,  0},
{0, 0,  0,  0},
{0, 0,  0,  R+e},
{A-R+e, 0, 0,  0}};
rewardRule[2]=Transpose[rewardRule[1]];

(* Reward function chooses the adequate entries from the matrices defined above *)
rewardFunction[abilityCombination_, actionCombination_, playerKey_]:=rewardRule[playerKey][[
        Sequence@@fnActionToKeyCombination[actionCombination]]];

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_, actionCombination_, playerNr_]:=0;

(* ASSEMBLING PAYOFF LAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_, actionCombination_, playerNr_]:= punishmentFunction[
        abilityCombination, actionCombination, playerNr] + rewardFunction[abilityCombination,
        actionCombination, playerNr] + sanctionFunction[abilityCombination, actionCombination,
        playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R > 0, (* sign gross interest *)
P > 2R, (* intensity punishment *)
0 < e < P-A; (* intensity reward *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

(with cross-reporting as in [RS04])
High punishment intensity
($2R < P$, $0 < e < P - 2R$)

Punishment and reward strategy

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	-R | -R	-e-2R | -P	-2R | 0	-P | e+R
P1:0R-y	-P | -e-2R	-P | -P	-P | -P	0 | 0
P1:1R	0 | -2R	-P | -P	0 | 0	e+R | -P
P1:2R	e+R | -P	0 | 0	-P | e+R	0 | 0

Symbol in [RS04]	Here
h	none
$\Psi > h$	P
$\bar{R}$	R
$A \equiv \min\{h, 2\bar{R}\}$	$2R$
$0 < \varepsilon < \Psi - h$	e
$(0, \mu)$	"0R-y"

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R-n | 0R-n	0R-n | 0R-n 0R-n | 1R	0R-y | 2R
A1:-R	0R-n | 0R-n 1R | 0R-n	0R-n | 0R-n 1R | 1R	1R | 1R
A1:-2 R	2R | 0R-y	1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0 R	2R
A1:-R	0 R	0 2R	2R
A1:-2 R	2R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2R	-2R -2R	0
A1:-R	-2R -2R	-2R 0	0
A1:-2 R	0	0	0

Traceable solution

A2:0

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[-R,-R]	[-e-2 R,-P]	×	×
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	×	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:-R	P1:0R-n	[-R,-R]	[-e-2 R,-P]	×	×
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	×	×
	P1:1R	[-R,-2 R]	[-P-R,-P]	×	×
	P1:2R	×	×	×	×
		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:-2 R	P1:0R-n	[-R,-R]	[-e-2 R,-P]	×	×
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	×	×
	P1:1R	[-R,-2 R]	[-P-R,-P]	×	×
	P1:2R	[e-R,-P]	[-2 R,0]	×	×

A2:-R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[-R,-R]	[-e-2 R,-P]	[-2 R,-R]	×
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	[-P,-P-R]	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:-R	P1:0R-n	[-R,-R]	[-e-2 R,-P]	[-2 R,-R]	×
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	[-P,-P-R]	×
	P1:1R	[-R,-2 R]	[-P-R,-P]	[-R,-R]	×
	P1:2R	×	×	×	×
		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:-2 R	P1:0R-n	[-R,-R]	[-e-2 R,-P]	[-2 R,-R]	×
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	[-P,-P-R]	×
	P1:1R	[-R,-2 R]	[-P-R,-P]	[-R,-R]	×
	P1:2R	[e-R,-P]	[-2 R,0]	[-P-2 R,e]	×

A2:-2 R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	[-R,-R]	[-e-2 R,-P]	[-2 R,-R]	[-P,e-R]
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	[-P,-P-R]	[0,-2 R]
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:-R	P1:0R-n	[-R,-R]	[-e-2 R,-P]	[-2 R,-R]	[-P,e-R]
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	[-P,-P-R]	[0,-2 R]
	P1:1R	[-R,-2 R]	[-P-R,-P]	[-R,-R]	[e,-P-2 R]
	P1:2R	×	×	×	×
		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:-2 R	P1:0R-n	[-R,-R]	[-e-2 R,-P]	[-2 R,-R]	[-P,e-R]
	P1:0R-y	[-P,-e-2 R]	[-P,-P]	[-P,-P-R]	[0,-2 R]
	P1:1R	[-R,-2 R]	[-P-R,-P]	[-R,-R]	[e,-P-2 R]
	P1:2R	[e-R,-P]	[-2 R,0]	[-P-2 R,e]	[-2 R,-2 R]

B.12 Cross-reporting in [RS04] improved

```
(* GAME DEFINITION - cross-reporting as in RS04 *)

(* Auxilary functions to transform an actionCombination into an actionKeyCombination *)
fnActionToKey[playerKey_, action_ ]:= Sequence@@@Position[actionSet[playerKey],action ];
fnActionToKeyCombination[actionCombination_]:=Flatten[MapIndexed[fnActionToKey[Sequence@@
        #2,#1]&,actionCombination,{1}]];

(* case in the paper where h>=2R and thus A=min[h,2R]=2R *)
A=2R

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R-n",0,"n"},
{"0R-y",0,"y"},
{"1R",-R,""},
{"2R",-2R,""}};
actionSet[2]= actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentRule[1]={
{-M, -(A+e), -M, -P },
{-P, -P,      -P, -(2R-A) },
{ 0,  -P,      0,  0 },
{ 0,   0,     -P,  0 }};
punishmentRule[2]=Transpose[punishmentRule[1]];

(* Punishment function chooses the adequate entries from the matrices defined above *)
punishmentFunction[ abilityCombination_, actionCombination_, playerKey_]:=punishmentRule[playerKey
        ][[ Sequence@@fnActionToKeyCombination[actionCombination]]];

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardRule[1]={
{0, 0,  0,  0},
{0, 0,  0,  0},
{0, 0,  0,  R+e},
{A-R+e, 0, 0,  0}};
rewardRule[2]=Transpose[rewardRule[1]];

(* Reward function chooses the adequate entries from the matrices defined above *)
rewardFunction[ abilityCombination_, actionCombination_, playerKey_]:=rewardRule[playerKey ][[
        Sequence@@fnActionToKeyCombination[actionCombination]]];

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[ abilityCombination_, actionCombination_, playerNr_ ]:=0;

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[ abilityCombination_, actionCombination_, playerNr_]:= punishmentFunction[
        abilityCombination, actionCombination, playerNr] + rewardFunction[ abilityCombination,
        actionCombination, playerNr]+ sanctionFunction[ abilityCombination, actionCombination,
        playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R> 0, (* sign gross interest *)
P>2R, (* intensity high punishment *)
R<M<2R, (* intensity medium punishment *)
0<e<P-A]; (* intensity reward *)
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

JOINT LIABILITY

(cross-reporting [RS04] improved)
High punishment intensity
($2R < P$, $0 < e < P - 2R$, $R < M < 2R$)

Punishment and reward strategy

	P2:0R-n	P2:0R-y	P2:1R	P2:2R
P1:0R-n	-M | -M	-e-2R | -P	-M | 0	-P | e+R
P1:0R-y	-P | -e-2R	-P | -P	-P | -P	0 | 0
P1:1R	0 | -M	-P | -P	0 | 0	e+R | -P
P1:2R	e+R | -P	0 | 0	-P | e+R	0 | 0

A*i*:	ability to repay for borrower *i*
P*i*:	action of player *i*
×	infeasible actions

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R-n | 0R-n	0R-n | 1R	0R-y | 2R
A1:-R	1R | 0R-n	1R | 1R	1R | 1R
A1:-2 R	2R | 0R-y	1R | 1R	1R | 1R

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	R	2R
A1:-R	R	2R	2R
A1:-2 R	2R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	-2M	-M	0
A1:-R	-M	0	0
A1:-2 R	0	0	0

Traceable solution

A2:0

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×	×
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×	×
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×	×
	P1:1R	×[-R,-M]	×[-P-R,-P]	×	×
	P1:2R	×	×	×	×
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×	×
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×	×
	P1:1R	×[-R,-M]	×[-P-R,-P]	×	×
	P1:2R	×[e-R,-P]	×[-2 R,0]	×	×

A2:-R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×[-M,-R]	×
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×[-P,-P-R]	×
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×[-M,-R]	×
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×[-P,-P-R]	×
	P1:1R	×[-R,-M]	×[-P-R,-P]	×[-R,-R]	×
	P1:2R	×	×	×	×
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×[-M,-R]	×
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×[-P,-P-R]	×
	P1:1R	×[-R,-M]	×[-P-R,-P]	×[-R,-R]	×
	P1:2R	×[e-R,-P]	×[-2 R,0]	×[-P-2 R,e]	×

A2:-2 R

		P2:0R-n	P2:0R-y	P2:1R	P2:2R
A1:0	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×[-M,-R]	×[-P,e-R]
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×[-P,-P-R]	×[0,-2 R]
	P1:1R	×	×	×	×
	P1:2R	×	×	×	×
A1:-R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×[-M,-R]	×[-P,e-R]
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×[-P,-P-R]	×[0,-2 R]
	P1:1R	×[-R,-M]	×[-P-R,-P]	×[-R,-R]	×[e,-P-2 R]
	P1:2R	×	×	×	×
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R	P2:2R
	P1:0R-n	×[-M,-M]	×[-e-2 R,-P]	×[-M,-R]	×[-P,e-R]
	P1:0R-y	×[-P,-e-2 R]	×[-P,-P]	×[-P,-P-R]	×[0,-2 R]
	P1:1R	×[-R,-M]	×[-P-R,-P]	×[-R,-R]	×[e,-P-2 R]
	P1:2R	×[e-R,-P]	×[-2 R,0]	×[-P-2 R,e]	×[-2 R,-2 R]

B.13 Cross-reporting with individual liability

```
(* GAME DEFINITION - cross-reporting with individual liability *)

(* NUMBER OF PLAYERS ---- *)
n=2;

(* ACTIONSET for each player ---- *)
actionSet[1]={
{"0R-n",0,"n"},
{"0R-y",0,"y"},
{"1R-n",-R,"n"},
{"1R-y",-R,"y"},
{"2R-n",-2R,"n"},
{"2R-y",-2R,"y"}};
actionSet[2]=actionSet[1];

(* PUNISHMENT FUNCTION: borrower to deadweight, controlled by lender ---- *)
punishmentFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
(* anti-coordination-failure-rule *)
{-2M, (fnGetActionMessage/@actionCombination)==ConstantArray["y",n](* if both say "yes" *)
&& -Plus@@(fnGetActionRepayment/@actionCombination)==0 (* and nobody repays *)},
(* anti-lying-rule one *)
{-M, -Plus@@Delete[(fnGetActionRepayment/@actionCombination),playerKey]>=R(* if peer repays *)
&& fnGetActionMessage[actionCombination[[playerKey]]]=="n" (* but player said "no" *)},
(* anti-lying-rule two *)
{-M, -Plus@@Delete[(fnGetActionRepayment/@actionCombination),playerKey]==0(* if peer defaults *)
&& fnGetActionMessage[actionCombination[[playerKey]]]=="y" (* but player said "yes" *)},
(* anti-strategic-default-rule *)
{-M, Delete[(fnGetActionMessage/@actionCombination),playerKey]=={"y"}(* if peer "yes" *)
&& fnGetActionRepayment[actionCombination[[playerKey]]]==0 (* but player defaults *)},
{0, True}
}];

(* REWARD FUNCTION: lender to borrower flow ---- *)
rewardFunction[abilityCombination_,actionCombination_,playerKey_]:=Refine[Piecewise[{
(* sincerity -reward-rule *)
{+e+R, Delete[(fnGetActionMessage/@actionCombination), playerKey]== ConstantArray["n",n-1] (*
    peer says "no" *)
&& -fnGetActionRepayment[actionCombination[[playerKey]]]>=R (*player repays despite peer message*)
    },
{0,True}
}];

(* SANCTIONS FUNCTION: borrower to deadweight flow ---- *)
sanctionFunction[abilityCombination_,actionCombination_,playerNr_]:=0;

(* ASSEMBLING PAYOFFLAYERS (apart from repayment) ---- *)
payoffLayers[abilityCombination_,actionCombination_,playerNr_]:=+ punishmentFunction[
    abilityCombination,actionCombination,playerNr] + rewardFunction[abilityCombination,
    actionCombination,playerNr]+ sanctionFunction[abilityCombination,actionCombination,
    playerNr];

(* ASSUMPTIONS ---- *)
$Assumptions=True; (* Hygiene: Clearing list of global assumptions first. *)
$Assumptions=And[
R>0, (* sign gross interest *)
M>R, (* intensity punishment *)
e>0(* intensity reward *)
];
(* Use == for comparisons! No contradictory assumptions! No ambiguous assumptions like <=! *)
```

INDIVIDUAL LIABILITY

(with cross-messaging)
Medium punishment intensity
($R < M, 0 < e$)

Punishment and reward strategy

	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	0 | 0	-M | -M	-M | e+R	-M | e-M+R	-M | e+R	-M | e-M+R
P1:0R-y	-M | -M	-2M | -2M	0 | 0	-M | -M	0 | 0	-M | -M
P1:1R-n	e+R | -M	0 | 0	e-M+R | e-M+R	-M | e+R	e-M+R | e-M+R	-M | e+R
P1:1R-y	e-M+R | -M	-M | -M	e+R | -M	0 | 0	e+R | -M	0 | 0
P1:2R-n	e+R | -M	0 | 0	e-M+R | e-M+R	-M | e+R	e-M+R | e-M+R	-M | e+R
P1:2R-y	e-M+R | -M	-M | -M	e+R | -M	0 | 0	e+R | -M	0 | 0

Map of equilibria

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0R-n | 0R-n	0R-y | 1R-n	0R-y | 1R-n
A1:-R	1R-n | 0R-y	1R-y | 1R-y	1R-y | 1R-y
A1:-2 R	1R-n | 0R-y	1R-y | 1R-y	1R-y | 1R-y

Map of lender payoffs

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	R	R
A1:-R	R	2R	2R
A1:-2 R	R	2R	2R

Map of deadweight loss

	A2:0	**A2:-R**	**A2:-2 R**
A1:0	0	0	0
A1:-R	0	0	0
A1:-2 R	0	0	0

Traceable solution

	A2:0					
	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	[0,0]	[-M,-M]	×	×	×	×
P1:0R-y	[-M,-M]	[-2 M,-2 M]	×	×	×	×
A1:0 P1:1R-n	×	×	×	×	×	×
P1:1R-y	×	×	×	×	×	×
P1:2R-n	×	×	×	×	×	×
P1:2R-y	×	×	×	×	×	×

	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	[0,0]	[-M,-M]	×	×	×	×
P1:0R-y	[-M,-M]	[-2 M,-2 M]	×	×	×	×
A1:-R P1:1R-n	[e,-M]	[-R,0]	×	×	×	×
P1:1R-y	[e-M,-M]	[-M-R,-M]	×	×	×	×
P1:2R-n	×	×	×	×	×	×
P1:2R-y	×	×	×	×	×	×

	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	[0,0]	[-M,-M]	×	×	×	×
P1:0R-y	[-M,-M]	[-2 M,-2 M]	×	×	×	×
A1:-2 R P1:1R-n	[e,-M]	[-R,0]	×	×	×	×
P1:1R-y	[e-M,-M]	[-M-R,-M]	×	×	×	×
P1:2R-n	[e-R,-M]	[-2 R,0]	×	×	×	×
P1:2R-y	[e-M-R,-M]	[-M-2 R,-M]	×	×	×	×

	A2:-R					
	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	[0,0]	[-M,-M]	[-M,e]	[-M,e-M]	×	×
P1:0R-y	[-M,-M]	[-2 M,-2 M]	[0,-R]	[-M,-M-R]	×	×
A1:0 P1:1R-n	×	×	×	×	×	×
P1:1R-y	×	×	×	×	×	×
P1:2R-n	×	×	×	×	×	×
P1:2R-y	×	×	×	×	×	×

	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	[0,0]	[-M,-M]	[-M,e]	[-M,e-M]	×	×
P1:0R-y	[-M,-M]	[-2 M,-2 M]	[0,-R]	[-M,-M-R]	×	×
A1:-R P1:1R-n	[e,-M]	[-R,0]	[e-M,e-M]	[-M-R,e]	×	×
P1:1R-y	[e-M,-M]	[-M-R,-M]	[e,-M-R]	[-R,-R]	×	×
P1:2R-n	×	×	×	×	×	×
P1:2R-y	×	×	×	×	×	×

	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y
P1:0R-n	[0,0]	[-M,-M]	[-M,e]	[-M,e-M]	×	×
P1:0R-y	[-M,-M]	[-2 M,-2 M]	[0,-R]	[-M,-M-R]	×	×
A1:-2 R P1:1R-n	[e,-M]	[-R,0]	[e-M,e-M]	[-M-R,e]	×	×
P1:1R-y	[e-M,-M]	[-M-R,-M]	[e,-M-R]	[-R,-R]	×	×
P1:2R-n	[e-R,-M]	[-2 R,0]	[e-M-R,e-M]	[-M-2 R,e]	×	×
P1:2R-y	[e-M-R,-M]	[-M-2 R,-M]	[e-R,-M-R]	[-2 R,-R]	×	×

	A2:-2 R							
	P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y		
A1:0	P1:0R-n	{0,0}	[-M,e]	[-M,e-M]	[-M,e-R]	[-M,e-M-R]		
	P1:0R-y	[-M,-M]	[-2 M,-2 M]	[0,-R]	[-M,-M-R]	[0,-2 R]	[-M,-M-2 R]	
	P1:1R-n	×	×	×	×	×	×	
	P1:1R-y	×	×	×	×	×	×	
	P1:2R-n	×	×	×	×	×	×	
	P1:2R-y	×	×	×	×	×	×	
A1:-R		P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y	
	P1:0R-n	{0,0}	[-M,-M]	[-M,e]	[-M,e-M]	[-M,e-R]	[-M,e-M-R]	
	P1:0R-y	[-M,-M]	[-2 M,-2 M]	[0,-R]	[-M,-M-R]	[0,-2 R]	[-M,-M-2 R]	
	P1:1R-n	[e,-M]	[-R,0]	[e-M,e-M]	[-M-R,e]	[e-M,e-M-R]	[-M-R,e-R]	
	P1:1R-y	[e-M,-M]	[-M-R,-M]	[e,-M-R]	[-R,-R]	[e,-M-2 R]	[-R,-2 R]	
	P1:2R-n	×	×	×	×	×	×	
	P1:2R-y	×	×	×	×	×	×	
A1:-2 R		P2:0R-n	P2:0R-y	P2:1R-n	P2:1R-y	P2:2R-n	P2:2R-y	
	P1:0R-n	{0,0}	[-M,-M]	[-M,e]	[-M,e-M]	[-M,e-R]	[-M,e-M-R]	
	P1:0R-y	[-M,-M]	[-2 M,-2 M]	[0,-R]	[-M,-M-R]	[0,-2 R]	[-M,-M-2 R]	
	P1:1R-n	[e,-M]	[-R,0]	[e-M,e-M]	[-M-R,e]	[e-M,e-M-R]	[-M-R,e-R]	
	P1:1R-y	[e-M,-M]	[-M-R,-M]	[e,-M-R]	[-R,-R]	[e,-M-2 R]	[-R,-2 R]	
	P1:2R-n	[e-R,-M]	[-2 R,0]	[e-M-R,e-M]	[-M-2 R,e]	[e-M-R,e-M-R]	[-M-2 R,e-R]	
	P1:2R-y	[e-M-R,-M]	[-M-2 R,-M]	[e-R,-M-R]	[-2 R,-R]	[e-R,-M-2 R]	[-2 R,-2 R]	

Bibliography

[AB13] Lutz Arnold and Benedikt Booker. Good intentions pave the way to … the local moneylender. *Economics Letters*, 118(3):466–469, 2013.

[AG00] Beatriz Armendáriz and Christian Gollier. Peer group formation in an adverse selection model. *Economic Journal*, 110(465):632–643, 2000.

[Agu09] Itai Agur. On the possibility of credit rationing in the Stiglitz-Weiss model: a comment. Working Paper 237, De Nederlandsche Bank, December 2009. Available at: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.5885&rep=rep1&type=pdf`.

[AH00] Reinnhold Adrian and Thomas Heidorn. *Der Bankbetrieb*. Gabler Verlag, fifteenth edition, August 2000.

[Ahl15] Christian Ahlin. The role of group size in group lending. *Journal of Development Economics*, 115:140–155, 2015.

[Ake70] George Arthur Akerlof. The market for ”lemons”: Quality Uncertainty and the Market Mechanism. *Quarterly Journal of Economics*, 84(3):488–500, August 1970.

[All16] Treb Allen. Optimal (partial) group liability in microfinance lending. *Journal of Development Economics*, 121:201–216, 2016.

[AM10] Beatriz Armendáriz and Jonathan Morduch. *The Economics of Microfinance*. MIT Press, second edition, 2010.

[Ame16] American Law Institute. *What Can We Learn from Credit Markets*, 2016. 93rd Annual Meeting, Washington, D.C., May 18, 2016. Available at: `https://www.ali.org/media/filer_public/41/2f/412f21ed-e4a3-49c6-aadc-8c7cca0a750f/simkovic-remarks.pdf`.

[Ani03] Kumar Aniket. Sequential group lending with moral hazard. Working Paper. Available at: `www.researchgate.net`, June 2003.

C. Prem, *The Theory of Credit Contracts*,
https://doi.org/10.1007/978-3-658-29362-8

[AR09] Lutz Arnold and John Riley. On the possibility of credit rationing in the Stiglitz-Weiss model. *American Economic Review*, pages 2012–2021, 2009.

[Arm99] Beatriz Armendáriz. On the design of a credit agreement with peer monitoring. *Journal of Development Economics*, 60(1):79–104, 1999.

[Arm11] Beatriz Armendáriz. Women and microsavings. In *The Handbook of Microfinance*, pages 503–516. World Scientific Publishing Co. Pte. Ltd., 2011.

[Arn16] Lutz Arnold. *Makroökonomik*. Mohr Siebeck Tübingen, fifth edition, 2016.

[ARS13] Lutz G. Arnold, Johannes Reeder, and Susanne Steger. On the viability of group lending when microfinance meets the market: A reconsideration of the besley—coate model. *Journal of Emerging Market Finance*, 12(1):59–106, 2013.

[ASP15] Ashta Arvind, Khan Saleh, and Otto Philipp. Does Microfinance Cause or Reduce Suicides? Policy Recommendations for Reducing Borrower Stress. *Strategic Change*, 24(2):165–190, March 2015.

[AT07] Christian Ahlin and Robert M. Townsend. Selection into and across credit contracts: Theory and field research. *Journal of Econometrics*, 136(2):665–698, February 2007.

[Bal78] Ernst Baltensperger. Credit Rationing: Issues and Questions. *Journal of Money, Credit and Banking*, 10(2):170–183, 1978.

[Bar76] Robert J. Barro. The loan market, collateral, and rates of interest. *Journal of Money, Credit and Banking*, 8(4):439–456, November 1976.

[BC95] Timothy Besley and Stephen Coate. Group lending, repayment incentives and social collateral. *Journal of Development Economics*, 46(1):1–18, 1995.

[BD05] Patrick Bolton and Mathias Dewatripont. *Contract Theory*. MIT Press, 2005.

[BDN05] John H. Boyd and Gianni De Nicoló. The Theory of Bank Risk Taking and Competition Revisited. *The Journal of Finance*, 60(3):1329–1343, June 2005.

[Bes85] Helmut Bester. Screening vs. rationing in credit markets with imperfect information. *The American Economic Review*, 75(4):850–855, 1985.

[Bes87] Helmut Bester. The role of collateral in credit markets with imperfect information. *European Economic Review*, 31(4):887–899, 1987.

[BK93] David Besanko and George Kanatas. Credit Market Equilibrium with Bank Monitoring and Moral Hazard. *The Review of Financial Studies*, 6(1):213–232, 1993.

[BK04] Philip Bond and Arvind Krishnamurthy. Regulating exclusion from financial markets. *The Review of Economic Studies*, 71(3):681–707, 2004.

[BKS14] Allen N. Berger, Thomas Kick, and Klaus Schaeck. Executive board composition and bank risk taking. *Journal of Corporate Finance*, 28:48–65, 2014.

[BO99] Tamer Basar and Geert Jan Olsder. *Dynamic Noncooperative Game Theory*. Society for Industrial and Applied Mathematics, second edition, 1999.

[BO10] Bharat Bhole and Sean Ogden. Group lending and individual lending with strategic default. *Journal of Development Economics*, 91:348–363, 2010.

[BR02] Philip Bond and Ashok Rai. Collateral substitutes in microfinance. Working Paper. Available at: `http://finance.wharton.upenn.edu/~{}pbond/research/limits-jul2-02.pdf`, July 2002.

[BS90] Patrick Bolton and David S. Scharfstein. A Theory of Predation Based on Agency Problems in Financial Contracting. *American Economic Review*, 80(1):93–106, March 1990.

[BS94] John H. Boyd and Bruce D. Smith. How Good Are Standard Debt Contracts? Stochastic Versus Nonstochastic Monitoring in a Costly State Verification Environment. *Journal of Business*, 67(4):539–561, October 1994.

[BS08] Dyuti Banerjee and Anupama Sethi. Intra-Group Transfers And Group Formation. Working paper, Monash University, Department of Economics, 2008. Available at: `https://www.monash.edu/business/economics/research/publications/2008/2408intragroupbanerjeesethi.pdf`.

[BS12] Sylvain Bourjade and Ibolya Schindele. Group lending with endogenous group size. *Economics Letters*, 117(3):556–560, 2012.

[BSW14] Jean-Marie Baland, Rohini Somanathan, and Zaki Wahhaj. Group lending and endogenous social sanctions. Discussion Paper No. 1415, University of Kent, School of Economics, 2014. Available at: `https://www.econstor.eu/bitstream/10419/129983/1/814788696.pdf`.

[BU90] Allen N. Berger and Gregory F. Udell. Collateral, loan quality and bank risk. *Journal of Monetary Economics*, 25(1):21–42, 1990.

[BZ06] Giovanni Busetta and Alberto Zazzaro. Mutual Loan-Guarantee Societies in Credit Markets with Adverse Selection: Do They Act as a Sorting Device? Working Paper. Available at: `www.researchgate.net`, 2006.

[Cho05] Prabal Roy Chowdhury. Group-lending: Sequential financing, lender monitoring and joint liability. *Journal of Development Economics*, 77(2):415–439, August 2005.

[Cho07] Prabal Roy Chowdhury. Group-lending with sequential financing, contingent renewal and social capital. *Journal of Development Economics*, 84(1):487–506, 2007.

[Chu11] Craig Churchill. Insurance for the Poor: Definitions and Innovations. In *The Handbook of Microfinance*, pages 537–562. World Scientific Publishing, 2011.

[CK85] Yuk-Shee Chan and George Kanatas. Asymmetric Valuations and the Role of Collateral in Loan Agreements. *Journal of Money, Credit and Banking*, 17(1):84–95, February 1985.

[Coc99] Giuseppe Coco. Collateral, heterogeneity in risk attitude and the credit market equilibrium. *European Economic Review*, 43(3):559–574, 1999.

[Coc00] Giuseppe Coco. On the use of collateral. *Journal of Economic Surveys*, 14(2):191–214, 2000.

[Con99] Jonathan Conning. Outreach, sustainability and leverage in monitored and peer-monitored lending. *Journal of Development Economics*, 60(1):51–77, 1999.

[Con05] Jonathan Conning. Monitoring by Delegates or by Peers? Joint Liability Loans under Moral Hazard. Working Paper. Available at:

http://econ.hunter.cuny.edu/wp-content/uploads/sites/6/RePEc/papers/HunterEconWP407.pdf, June 2005.

[CS05] Prabirendra Chatterjee and Sudipta Sarangi. Enforcement with Costly Group Formation. *Economics Bulletin*, 15(9):1–8, February 2005.

[DD05] Jean-Pierre Danthine and John Donaldson. *Intermediate Financial Theory*. Elsevier Academic Press, second edition, 2005.

[Dia84] Douglas W. Diamond. Financial Intermediation and Delegated Monitoring. *Review of Economic Studies*, 51(3):393–414, 1984.

[DMW87] David De Meza and David C. Webb. Too Much Investment: A Problem of Asymmetric Information. *Quarterly Journal of Economics*, 102(2):281–292, 1987.

[DNDLV10] Gianni De Nicolò, Giovanni Dell'Ariccia, Luc Laeven, and Fabian Valencia. Monetary Policy and Bank Risk Taking. IMF Staff Position Note SPN/10/09, International Monetary Fund, July 2010. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.813&rep=rep1&type=pdf.

[Dos16] Anastasios Dosis. On Signalling and Screening in Markets with Asymmetric Information. Working Paper 1608, ESSEC Business School, 2016. Available at: https://hal-essec.archives-ouvertes.fr/hal-01285190v2/document.

[DQFG16] Jonathan De Quidt, Thiemo Fetzer, and Maitreesh Ghatak. Group lending without joint liability. *Journal of Development Economics*, 121:217–236, 2016.

[DR94] Prajit K. Dutta and Roy Radner. Moral Hazard. In R. J. Aumann and S. Hart, editors, *Handbook of Game Theory*, volume 2, chapter 26, pages 869–903. Elsevier Science B. V., 1994.

[DS63] John M. Dutton and William H. Starbuck. On Managers and Theories. *Management International*, 3(6):25–35, 1963.

[EMS02] Paul Embrechts, Alexander McNeil, and Daniel Straumann. Correlation and Dependence in Risk Management: Properties and Pitfalls. In M. A. H. Dempster, editor, *Risk Management: Value at Risk and Beyond*, pages 176–223. Cambridge University Press, 2002.

[Fei59] Hermann Feifel. *Die Anwendbarkeit der modernen Kreditschöpfungslehre auf die besondere Art des Sparkassengeschäfts*. Duncker & Humblot, Berlin, 1959.

[FG10] Greg Fischer and Maitreesh Ghatak. Repayment Frequency in Microfinance Contracts with Present-Biased Borrowers. Working paper, London School of Economics, July 2010. Available at: `http://eprints.lse.ac.uk/58184/1/__lse.ac.uk_storage_LIBRARY_Secondary_libfile_shared_repository_Content_STICERD_EOPP_eopp21.pdf`.

[FG11] Greg Fischer and Maitreesh Ghatak. Spanning the Chasm: Uniting Theory and Empirics in Microfinance Research. In *The Handbook of Microfinance*, chapter 3, pages 59–75. World Scientific Publishing, 2011.

[FP08] Erica Field and Rohini Pande. Repayment Frequency and Default in Microfinance: Evidence from India. *Journal of the European Economic Association*, 6(2-3):501–509, May 2008.

[FR08] Xavier Freixas and Jean-Charles Rochet. *Microeconomics of Banking*. MIT Press, second edition, 2008.

[FR15] Robert Frontczak and Stefan Rostek. Modeling loss given default with stochastic collateral. *Economic Modelling*, 44:162–170, January 2015.

[GG99] Maitreesh Ghatak and Timothy W. Guinnane. The economics of lending with joint liability: theory and practice. *Journal of Development Economics*, 60:195–228, 1999.

[GGH+13] Ludwig Gramlich, Peter Gluchowski, Andreas Horsch, Klaus Schäfer, and Gerd Waschbusch. *Gabler Bank-Lexikon*. Springer Gabler, fourteenth edition, 2013.

[GH85] Douglas Gale and Martin Hellwig. Incentive-Compatible Debt Contracts: The One-Period Problem. *Review of Economic Studies*, 52(4):647–663, 1985.

[Gha99] Maitreesh Ghatak. Group lending, local information and peer selection. *Journal of Development Economics*, 60(1):27–50, 1999.

[Gha00] Maitreesh Ghatak. Screening by the company you keep: Joint liability lending and the peer selection effect. *Economic Journal*, 110(465):601–631, July 2000.

[GK14] Xavier Giné and Dean S. Karlan. Group versus individual liability: Short and long term evidence from Philippine microcredit lending groups. *Journal of Development Economics*, 107:65–83, 2014.

[GL14] Shubhashis Gangopadhyay and Robert Lensink. Asymmetric group loans, non-assortative matching and adverse selection. *Economics Letters*, 124(2):185–187, 2014.

[GMRS11] Isabelle Guérin, Solène Morvant-Roux, and Jean-Michel Servet. Understanding the Diversity and Complexity of Demand for Microfinance Services: Lessons from Informal Finance. In *The Handbook of Microfinance*, pages 101–122. World Scientific Publishing, 2011.

[Gro94] Denis Gromb. Renegotiation in debt contracts. Working Paper. Available at: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.512.4264&rep=rep1&type=pdf`, May 1994.

[Gut08] Joel M. Guttman. Assortative matching, adverse selection, and group lending. *Journal of Development Economics*, 87(1):51–56, 2008.

[GW11] Christophe J. Godlewski and Laurent Weill. Does Collateral Help Mitigate Adverse Selection? A Cross-Country Analysis. *Journal of Financial Services Research*, 40(1):49–78, October 2011.

[Han01] Frank Hanser. *Die Struktur von Kreditbeziehungen*. Gabler Verlag und Deutscher Universitäts-Verlag, first edition, 2001.

[HCG06] Valentina Hartarska, Steven B. Caudill, and Daniel M. Gropper. The Cost Structure of Microfinance Institutions in Eastern Europe and Central Asia. Working Paper Number 809, William Davidson Institute at the University of Michigan, January 2006. Available at: `https://deepblue.lib.umich.edu/bitstream/handle/2027.42/40195/wp809.pdf?sequence=3`.

[HL07] Niels Hermes and Robert Lensink. Impact of Microfinance: A Critical Survey. *Economic and Political Weekly*, 42(6):462–465, February 2007.

[Hol79] Bengt Holmström. Moral Hazard and Observability. *Bell Journal of Economics*, 10(1):74–91, 1979.

[Hol82] Bengt Holmström. Moral Hazard in Teams. *Bell Journal of Economics*, 13(2):324–340, 1982.

[HT97] Bengt Holmstrom and Jean Tirole. Financial intermediation, loanable funds, and the real sector. *Quarterly Journal of Economics*, 112(3):663–691, 1997.

[Imb05] Kathryn Imboden. Building inclusive financial sectors: The road to growth and poverty reduction. *Journal of International Affairs*, 58(2):65–86, 2005.

[Imp98] Gregorio Impavido. Credit rationing, group lending and optimal group size. *Annals of Public and Cooperative Economics*, 69(2):243–260, 1998.

[Inn90] Robert D. Innes. Limited liability and incentive contracting with ex-ante action choices. *Journal of Economic Theory*, 52(1):45–67, 1990.

[Jea03] Tirole Jean. Incomplete Contracts: Where Do We Stand? *Econometrica*, 67(4):741–781, December 2003.

[Jen09] Mark William Jenkins. *Essays on Consumer Credit Markets*. phdthesis, Department of Economics at Stanford University, December 2009.

[JN02] Robert A. Jones and David Nickerson. Mortgage Contracts, Strategic Options and Stochastic Collateral. *Journal of Real Estate Finance and Economics*, 24(1):35–58, Jan 2002.

[JPB05] Tullio Jappelli, Marco Pagano, and Magda Bianco. Courts and Banks: Effects of Judicial Enforcement on Credit Markets. *Journal of Money, Credit and Banking*, 37(2):223–244, April 2005.

[Kar07] Dean S. Karlan. Social Connections and Group Banking. *Economic Journal*, 117:F52–F84, February 2007.

[Kee79] William R. Keeton. *Equilibrium Credit Rationing*. Garland Publishing, 1979.

[Key36] John Maynard Keynes. *The General Theory of Employment, Interest and Money*. Macmillan, London, 1936.

[KL12] Tomek Katzur and Robert Lensink. Group lending with correlated project outcomes. *Economics Letters*, 117(2):445–447, 2012.

[KOU14] Kei Kawai, Ken Onishi, and Kosuke Uetake. Signaling in Online Credit Markets. Working paper. Available at: `https://vlab.decisionsciences.columbia.edu/newsletter/Mar0413/signaling%20in%20online.pdf`, August 2014.

[KP98] Fahad Khalil and Bruno M. Parigi. Loan size as a commitment device. *International Economic Review*, 39(1):135–150, February 1998.

[KS94] David M. Kreps and Joel Sobel. Signalling. In R. J. Aumann and S. Hart, editors, *Handbook of Game Theory*, volume 2, chapter 25, pages 849–867. Elsevier Science, 1994.

[KS14] Artashes Karapetyan and Bogdan Stacescu. Does information sharing reduce the role of collateral as a screening device? *Journal of Banking and Finance*, 43:48–57, 2014.

[KV00] Stefan Krasa and Anne P. Villamil. Optimal Contracts When Enforcement is a Decision Variable. *Econometrica*, 68(1):119–134, January 2000.

[KV11] Dean Karlan and Martin Valdivia. Teaching Entrepreneurship: Impact of Business Training on Microfinance Clients and Institutions. *Review of Economics and Statistics*, 93(2):510–527, May 2011.

[Laf03] Jean-Jacques Laffont. Collusion and group lending with adverse selection. *Journal of Development Economics*, 70(2):329–348, 2003.

[Led98] Joanna Ledgerwood. *Microfinance Handbook: An Institutional and Financial Perspective*. World Bank Publications, 1998.

[Man28] Thomas Mann. *The Magic Mountain*. Secker & Warburg London, third edition, 1928. Translated from the German, 1971 Reprint.

[MCWG95] Andreu Mas-Colell, Michael D. Whinston, and Jerry R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[MFE05] Alexander J. McNeil, Rüdiger Frey, and Paul Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools*. Princeton University Press, first edition, 2005.

[MNS06] Lukas Menkhoff, Doris Neuberger, and Chodechai Suwanaporn. Collateral-based lending in emerging markets: Evidence from Thailand. *Journal of Banking and Finance*, 30(1):1–21, 2006.

[MO14] Sana Mohsni and Isaac Otchere. Risk taking behavior of privatized banks. *Journal of Corporate Finance*, 29:122–142, 2014.

[MP86] Arie Melnik and Steven Plaut. Loan commitment contracts, terms of lending, and credit allocation. *Journal of Finance*, 41(2):425–435, 1986.

[MP89] Dilip Mookherjee and Ivan Png. Optimal Auditing, Insurance, and Redistribution. *Quarterly Journal of Economics*, 104(2):399–415, May 1989.

[MQ02] Michael Magill and Martine Quinzii. *Theory of Incomplete Markets*, volume 1. MIT Press, 2002.

[MR88] Hellmuth Milde and John G. Riley. Signaling in credit markets. *Quarterly Journal of Economics*, 103(1):101–129, 1988.

[MS02] Eric Maskin and Tomas Sjöström. Implementation Theory. In K. J. Arrow, A. K. Sen, and K. Suzumura, editors, *Handbook of Social Choice and Welfare*, volume 1, chapter 5, pages 237–288. Elsevier Science, 2002.

[MS13] R. Mersland and R. Øystein Strøm. Microfinance: Costs, Lending Rates, and Profitability. In Gerard Caprio, editor, *Handbook of Key Global Financial Markets, Institutions, and Infrastructure*, chapter 44, pages 489–499. Elsevier, first edition, 2013.

[MW13] Sebastian C. Moenninghoff and Axel Wieandt. The Future of Peer-to-Peer Finance. *Zeitschrift für Betriebswirtschaftliche Forschung*, 65:466–487, September 2013.

[Nel06] Roger B. Nelsen. *An Introduction to Copulas*. Springer, New York, second edition, 2006.

[Nii09] Juha-Pekka Niinimäki. Screening in the credit market when the collateral value is stochastic. Discussion Paper 19, Bank of Finland Research, 2009. Available at: `https://helda.helsinki.fi/bof/bitstream/handle/123456789/7830/163910.pdf`.

[Nii11] Juha-Pekka Niinimäki. Nominal and true cost of loan collateral. *Journal of Banking and Finance*, 35(10):2782–2790, October 2011.

[NS05] Noam Nisan and Shimon Schocken. *The Elements of Computing Systems*. The MIT Press, 2005. Paperback edition, 2008.

[Par00] Cheol Park. Monitoring and structure of debt contracts. *Journal of Finance*, 55(5):2157–2195, October 2000.

[PHS14] Luminita Postelnicu, Niels Hermes, and Ariane Szafarz. Defining Social Collateral in Microfinance Group Lending. In Roy Mersland and R. Øystein Strøm, editors, *Microfinance Institutions: Financial and Social Performance*, chapter 10, pages 187–207. Palgrave Studies in Impact Finance, first edition, 2014.

[PMF17] Achim Peters, Bruce S. McEwen, and Karl Friston. Uncertainty and stress: Why it causes diseases and how it is mastered by the brain. *Progress in Neurobiology*, 156:164–188, September 2017.

[Rah99] Aminur Rahman. Micro-credit Initiatives for Equitable and Sustainable Development: Who Pays? *World Development*, 27(1):67–82, 1999.

[Ras07] Eric Rasmusen. *Games and Information: An Introduction to Game Theory*. Blackwell Publishing, fourth edition, 2007.

[RC12] David Rowell and Luke B. Connelly. A History of the Term "Moral Hazard". *Journal of Risk and Insurance*, 79(4):1051–1075, 2012.

[Ric12] Stefan Richter. *Schadenszurechnung bei deliktischer Haftung für fehlerhafte Sekundärmarktinformation*. Mohr Siebeck, Tübingen, 2012.

[RS70] Michael Rothschild and Joseph E. Stiglitz. Increasing Risk: I. A Definition. *Journal of Economic Theory*, 2(3):225–243, September 1970.

[RS00] Rafael Repullo and Javier Suarez. Entrepreneurial moral hazard and bank monitoring: A model of the credit channel. *European Economic Review*, 44(10):1931–1950, 2000.

[RS04] Ashok S. Rai and Tomas Sjöström. Is Grameen Lending Efficient? Repayment Incentives and Insurance in Village Economies. *Review of Economic Studies*, 71(1):217–234, 2004.

[RS13] Ashok Rai and Tomas Sjöström. Redesigning Microcredit. In Nir Vulkan, Alvin E. Roth, and Zvika Neeman, editors, *The Handbook of Market Design*, chapter 9, pages 249–265. Oxford University Press, 2013.

[Rut11] Stuart Rutherford. Boosting the Poor's Capacity to Save: A Note on Instalment Plans and their Variants. In *The Handbook of Microfinance*, pages 517–535. World Scientific Publishing, 2011.

[Sch01] Mark Schreiner. Seven Aspects of Loan Size. *Journal of Microfinance*, 3(2):27–47, 2001.

[Sha03] Tridib Sharma. Optimal Contracts When Enforcement Is a Decision Variable: A Comment. *Econometrica*, 71(1):387–390, January 2003.

[Sha12] Andrey Shatelyuk. *Sanierungskredite in der Krise und in der Insolvenz von Unternehmen: Eine vergleichende Untersuchung des deutschen und russischen Rechts*. De Gruyter, 2012.

[Sim56] Herbert A. Simon. Rational Choice and the Structure of the Environment. *Psychological Review*, 63(2):129–138, 1956.

[Ste10] Susanne E. Steger. *Kreditmarktunvollkommenheiten und Mikrokredite*. PhD thesis, University of Regensburg, 2010.

[Ste11] Christian Stegbauer. *Reziprozität - Einführung in soziale Formen der Gegenseitigkeit*. Verlag für Sozialwissenschaften, second edition, 2011.

[Sti90] Joseph E. Stiglitz. Peer Monitoring and Credit Markets. *World Bank Economic Review*, 4(3):351–366, September 1990.

[SW81] Joseph E. Stiglitz and Andrew Weiss. Credit Rationing in Markets with Imperfect Information. *American Economic Review*, 71(3):393–410, 1981.

[SW06] Sam Schulhofer-Wohl. Negative assortative matching of risk-averse agents with transferable expected utility. *Economics Letters*, 92(3):383–388, 2006.

[SZ06] Franklin Simtowe and Manfred Zeller. Determinants of Moral Hazard in Microfinance: Empirical Evidence from Joint Liability Lending Programs in Malawi. Working paper, Munich Personal RePEc Archive, October 2006. Available at: `https://mpra.ub.uni-muenchen.de/461/1/MPRA_paper_461.pdf`.

[Sza97] Nick Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), September 1997. Available at: `http://firstmonday.org/ojs/index.php/fm/article/view/548`.

[Tow79] Robert M. Townsend. Optimal Contracts and Competitive Markets with Costly State Verification. *Journal of Economic Theory*, 21:265–293, 1979.

[Wei03] Karl E. Weick. Theory and Practice in the Real World. In Haridimos Tsoukas and Christian Knudsen, editors, *The Oxford Handbook of Organization Theory*, chapter 16, pages 453–475. Oxford University Press, 2003.

[Zel98] Manfred Zeller. Determinants of Repayment Performance in Credit Groups: The Role of Program Design, Intragroup Risk Pooling, and

Social Cohesion. *Economic Development and Cultural Change*, 46(3):599–620, April 1998.

[Zha08] Wei Zhang. Ex-ante Moral Hazard and Repayment Performance under Group Lending. *Journal of East Asian Studies*, 6:145–171, 2008.