Building a Generic Feedback System for Rule-Based Problems

Nico Naus[1(✉)] and Johan Jeuring[1,2]

[1] Utrecht University, Utrecht, The Netherlands
n.naus@uu.nl
[2] Faculty of Management, Science and Technology,
Open University of the Netherlands, Heerlen, The Netherlands

Abstract. We present a generic framework that provides hints on how to achieve a goal to users of software supporting rule-based problem solving from different domains. Our approach consists of two parts. First, we present a DSL that relates and unifies different rule-based problems. Second, we use generic search algorithms to solve various kinds of problems. This solution can then be used to calculate a hint for the user. We present three rule-based problem frameworks to illustrate our approach: the Ideas framework, PuzzleScript and iTasks. By taking real world examples from these three example frameworks and instantiating feedback systems for them, we validate our approach.

1 Introduction

Many software frameworks and systems support, model, or automate the process of human problem solving. With a problem we mean anything like a game or a puzzle, solving an exercise in physics, or search and rescue people in need. Typical examples of systems supporting problem solving are workflow management systems, intelligent tutoring systems, and expert systems.

A user of a system supporting problem solving sometimes needs help in making a decision or taking a step towards a particular goal. In the case of a game or a puzzle, a user might get stuck, and need a step in the right direction. For supporting a student solving an exercise in an intelligent tutoring system, hints are essential [24]. In search and rescue systems, hints can quickly give insight in the current situation, and can help a user in understanding why a next step has to be taken. A user has to take a decision under pressure of time and potential danger. Automatically suggesting and explaining the best option to perform may reduce the chance of human error, while still allowing intervention.

In all of the above examples a user follows a potentially flexible process, and needs information about where she is in the process, where she should go next, and why she should go there [5]. In this paper, we attempt to answer the question: how can we construct a generic framework that provides users of rule-based problem solving systems with feedback? To answer the question, we look at research performed in the intelligent tutoring community. In this community,

D. Van Horn and J. Hughes (Eds.): TFP 2016, LNCS 10447, pp. 172–191, 2019.
https://doi.org/10.1007/978-3-030-14805-8_10

a lot of research has been performed on how to build frameworks that provide the user with feedback on how to solve exercises [25], allows teachers to describe their exercises [17], and deals with different problem-domains [6]. The results are not directly applicable to the rule-based problems described above, but the central ideas inspire our approach.

There are many forms of hints and feedback possible. In this paper, we focus on next-step hints. This kind of hint indicates which of the steps that can currently be taken, is the best choice. For example, in an intelligent tutoring system, the next step that a student should apply is returned, for example "Eliminate constants" or "Remove double negation". In the case of a puzzle, we want to inform a player on what to do next, for example "Move left" or "Apply action x". In the case of a search and rescue system we want to report what immediate action needs to be taken, for example "Inform unit x" or "Escalate incident to level 2". The best sentence to use when presenting such a next-step hint is probably best determined by a teacher, but *which* next step to take is something we want to and can calculate automatically.

This paper proposes a unified framework to describe processes for problem solving. For this purpose, we use a domain specific language (DSL). Giving a hint in an intelligent tutoring system for solving equations often amounts to returning the next steps prescribed by the solving strategy, where providing a hint for a more complicated problem such as the traveling salesman problem requires more involved problem solving techniques. We obtain these different instances of problem solving processes by interpreting our DSL in different ways. Thus we have a unified framework for describing problem solving processes, which can be instantiated for different purposes by selecting different interpretations. The novelty of our framework is the way in which it relates rule-based problems, to make them tractable to standard, generic solving algorithms.

This paper is organized as follows. Section 2 discusses some examples for which problem-solving assistance is desirable. Section 3 introduces a DSL for describing the rule-based problem solving processes. Section 4 presents several methods for solving the various problems. In Sect. 5, we validate our approach and in Sect. 6 we compare our approach to previous work. Section 7 concludes.

2 Examples

This section illustrates and motivates our goal of providing help to people using a rule-based problem solving system by giving three examples: the Ideas framework [7], PuzzleScript [13], and the iTasks system [18]. Each of these frameworks can describe a variety of problems. We briefly introduce each framework, show an example problem described in the framework, and explain what kind of problem solving assistance is desired.

2.1 Ideas

The Ideas framework is used to develop services to support users when stepwise solving exercises in an intelligent tutoring system for a domain like mathematics

```
dnfStrategy = label "Constants"     (repeat (topDown constants))
            <*> label "Definitions" (repeat (bottomUp definitions))
            <*> label "Negations"   (repeat (topDown negations))
            <*> label "Distribution" (repeat (somewhere distribution))
```

Fig. 1. A problem solving strategy in Ideas

or logic. It is a general framework used to construct the expert knowledge of an intelligent tutoring system (ITS). The framework has been applied in the domains of mathematics [7], programming [4], and communication skills [9].

The central component of the expert knowledge for an ITS is expressed as a so-called *strategy* in Ideas. For example, Fig. 1 gives part of a strategy for the problem of rewriting a logic expression to disjunctive normal form (for the complete strategy see Heeren et al. [7]). The framework offers various services based on this strategy, among which a service that diagnoses a step from a student, and a service that gives a next step to solve a problem. The student receives a logic expression, and stepwise rewrites this expression to disjunctive normal form using services based on the above strategy. At each step, the student can request a hint, like "Eliminate constants" or "Eliminate implications", or ask for feedback on her current expression. If no rules can be applied any more, the expression is in normal form.

The *dnfStrategy* Ideas strategy describes a rule-based process that solves the problem of converting an expression to disjunctive normal form. It is expressed in terms of combinators like <*> (sequence), *repeat* and *somewhere*, and further sub-strategies. Additionally, a *label* combinator is available, to label sub-strategies with a name.

2.2 PuzzleScript

PuzzleScript is an open source HTML5 Puzzle Game Engine [13]. It is a simple scripting language for specifying puzzle games. Its central component is a DSL for describing a game. PuzzleScript compiles a puzzle described in this DSL into an HTML5 puzzle game. Using the DSL, the game programmer describe a puzzle as a list of objects, rules that define the behavior of the game, a win condition, collision information, and one or more levels.

The hello-world example for PuzzleScript is given in Fig. 2. It describes a simple crate-pusher game, also called Sokoban. Objects are: background, walls, crates, the player and the targets for the crates. There is a single rule that states if a player moves into a crate, the crate moves with the player. Objects appearing on the same line in the collision layers are not allowed to pass trough each other. The winning condition is reached when all targets have a crate on them. Finally, a start-level is specified under *LEVELS*.

In a difficult game, we want to offer next-step hints to the player on how to proceed. Based on the state of the game, the *RULES*, *COLLISIONLAYERS* and *WINCONDITIONS*, an algorithm can calculate a hint for a user [14]. This

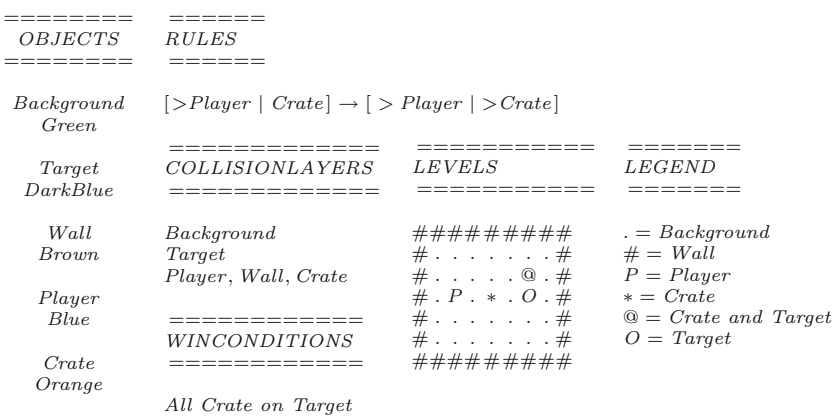

Fig. 2. Partial definition of the hello-world example of PuzzleScript

same information can also be used to check if a game can still be solved in the current state. For example, the game cannot be solved any more if a crate gets stuck in a corner.

2.3 iTasks

iTasks [18] supports task-oriented programming in the pure functional programming language Clean [19]. It allows for rapid workflow program development, by using the concept of task as an abstraction. Clean is very similar to Haskell, with a few exceptions. A data declaration starts with ::, types of function arguments are not separated by a function arrow (→) but by a space, and class contexts are written at the end of a type, starting with a |.

An iTasks program is composed out of base tasks, task combinators, and standard Clean functions. A task is a monadic structure. Its evaluation is driven by events and handling an event potentially changes a shared state. Tasks can be combined using combinators. The most common combinators are ≫= (sequence), ≫⋆ (step), −||− (parallel) and −&&− (choice). The step combinator can be seen as a combination of sequence and choice. It takes a task and attaches a list of actions to it, from which the user can choose. The chosen action receives a result value from the first task. The action, which is of type *TaskStep*, is a regular task combined with an action to trigger it, and a condition that must hold for the action to be available.

Figure 3 shows the partial source code of an iTasks program for a Command and Control (C2) system, as illustrated in Fig. 4. The illustration represents a ship with rooms and doors between them. Alice is a worker on the ship. This system is a simplified version of the C2 system built by the iTasks team in cooperation with the Royal Netherlands Navy [23].

The record type *ShipState* holds the state of the ship and the state of the worker. *shipTask* implements the C2 system. First, it uses the standard task for

```
shipStore :: Shared ShipState

shipTask :: Task ShipState
shipTask = viewSharedInformation "Ship" [] shipStore
         >>*  [OnAction (Action "Move"     []) (always moveTask)
               , OnAction (Action "Pick up" []) (ifValue hasInventory
                                                 (λst → set (applyPickup st) shipStore >>| shipTask))
               , OnAction (Action "Extinguish" []) (ifValue canExtinguish
                                                    (λst → set (applyExtinguish st) shipStore >>| shipTask))]

moveTask :: ShipState → Task ShipState

hasInventory    :: ShipState → Bool
applyPickup     :: ShipState → ShipState
canExtinguish   :: ShipState → Bool
applyExtinguish :: ShipState → ShipState
```

Fig. 3. Example iTasks program, formulated by composing tasks

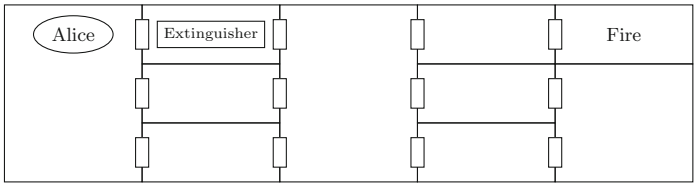

Fig. 4. Rendering of an example initial state for the simplified C2 system

viewing information, stored in the Shared Data Sources [18] (SDS), to display the current state. Then, it uses the step combinator $\gg\star$ to combine the viewing task with the tasks offering the possible options.

When running *shipTask*, the iTasks system renders the *shipStore* together with three buttons that allow the worker to move, pick up an item, or extinguish a fire. If an action is not applicable (for example, an extinguish action when the worker does not hold an extinguisher), the button is disabled. When a user clicks an enabled button, the *shipStore* is updated accordingly.

The goal of this system is to extinguish a fire on the ship. We want to extend the functionality of our program to give a user a next-step hint if she does not know how to proceed. We could implement this in an ad-hoc fashion by developing hint functionality for each iTask program for which we want to give hints to the end user. Alternatively, we will use the same framework as we want to use in the previous examples. By using this framework, we do not have to reimplement the hint functionality for every iTask program from scratch.

```
:: Rule a        =  Rule Name (Effect a)
:: Name          :== String
:: Effect a      :== a → a
:: Goal a        :== Predicate a
:: Predicate a   :== a → Bool
```

```
:: RuleTree a
   = Seq [RuleTree a]
   | Choice [RuleTree a]
   | Parallel [RuleTree a]
   | Condition (Predicate a) (RuleTree a)
   | Leaf (Rule a)
   | Empty
```

Fig. 5. Types of the components of our DSL

3 Problem Formalization

Russell and Norvig [21] define a well-defined artificial intelligence (AI) problem as consisting of the following components:

Initial state. The state of the problem that you want to solve.
Operator set. The set of steps that can be taken, together with their effects.
Goal test. A predicate that is True if the problem is solved.
Path Cost function. A function that describes the cost of each operation.

We use a slightly simplified definition of an AI problem. If there is a cost associated with a certain operation, we encode this as an effect on the state. Therefore, we do not need a path cost function. We have an initial state, represented by a value of type a, and an operator set, represented by a *RuleTree*, and a goal test, represented by the predicate *Goal a*.

Figure 5 gives the types of the components of our DSL. A *Rule* consists of a *Name* and an *Effect*. The *Name* is used to identify the rule to the end user in a given hint. Therefore, these names should be informative and unambiguous. The *Goal* can be reached by performing one or more rules after each other to arrive at a state where the *Goal* condition is met. The solution depth is the number of rules that have to be applied to reach the *Goal*.

The components in the DSL follow naturally from the Russell and Norvig definition, except for the *RuleTree*. The design of the *RuleTree* is loosely based on strategy languages from the Ideas framework [7], iTask combinators [18], and the strategy language presented by Visser and others [26].

We implement the DSL as an embedded DSL in Clean. This allows us to use standard Clean functions to construct for example a rule tree. We chose not to implement recursion in our DSL, but instead make use of recursion in the host language. The advantage of this is that we can keep our DSL simple and small. Implementing recursion in the DSL requires adding abstraction and application, making the DSL significantly more complex. Another notable omission is support for multiple users, for example by means of an assign combinator. This restricts the set of rule-based problems that can be encoded to single-user problems. Apart from multi-user problems, any rule-based problem can be encoded in this DSL, and as long as there is an appropriate solving algorithm available, our framework can generate hints for it.

```
shipSimulation :: RuleTree ShipState
shipSimulation = Seq
  [Choice [Choice (map (λx → Condition (isValidMove x)
                                       (Leaf (Rule (toName x)
                                                   (applyMove x))))
                       [1 .. maxRoomID])
          , Condition hasInventory (Leaf (Rule "Pickup" applyPickup))
          , Condition canExtinguish (Leaf (Rule "Extinguish" applyExtinguish))]
  , shipSimulation]


shipNotOnFire :: ShipState → Bool
shipNotOnFire  {ship}    = (foldr (∧) True (map (notOnFire) (flatten ship)))
```

Fig. 6. C2 in our DSL

3.1 C2 in Our DSL

To build intuition on how to translate a rule-based problem into our DSL, we have taken the iTasks example program from Sect. 2, listed in Fig. 3 and transformed it. First we build the *RuleTree* for our problem. We do this by using the constructors as combinators. At the top-level of the tree structure, we have two subtrees in sequence. The first subtree allows a choice between three options, namely the three actions the user can take: Move, Pick up, and Extinguish. In the case of Move, we construct the list of possible rooms that Alice can move to, by mapping a function that constructs a Condition over the list of rooms. This condition validates the move, before allowing this rule to be offered and applied. The Condition constructor is also used in the case of Pick up and Extinguish. The condition makes sure that there is something to pick up and something to extinguish, respectively. The second subtree is a recursive call to the whole tree. As mentioned earlier, we use the recursion from Clean here to construct recursive *RuleTree*s, recursion is not part of our DSL.

Now that we have our *RuleTree*, we need to construct the goal condition. This is simply a predicate over the state, which indicates if we have achieved our goal. In this case, we don't want the ship to be on fire, so we do a *foldr* over the rooms of the ship to check that none of the rooms are on fire.

4 Solving Implementations

The DSL introduced in the previous section offers a uniform approach to describe rule-based problems. Different classes of problems require different approaches to solving such problems. This section describes how we can view the DSL as an interface for which we can provide different interpretations to obtain different ways to approach a problem, and to obtain various services, in particular for providing hints.

```
allFirsts :: (RuleTree a) a → [Name]
allFirsts    t              s  =  map toName (topRules s t)

topRules :: a (RuleTree a    ) → [Rule a]
topRules    _ Empty            =  []
topRules    _ (Leaf r        ) =  [r]
topRules    _ (Seq []        ) =  []
topRules    s (Seq [rt : rts]) =  case rt of
                                     Condition c t | c s     = topRules s (Seq [t : rt])
                                                   | True    = []
                                     _             = case topRules s rt of
                                                       []    = topRules s (Seq rts)
                                                       x     = x
topRules    s (Choice rts    ) =  flatten (map (topRules s) rts)
topRules    s (Parallel rts  ) =  flatten (map (topRules s) rts)
topRules    s (Condition c rt) |  c s     = topRules d rt
                               |  True    = []
```

Fig. 7. Definition of the *allFirsts* hint service

This section shows the implementations of four services for giving hints for different classes of problems. All implementations take the strategy *RuleTree*, the current state in the form of a value of type *a* and the goal test *Goal*, and return the names of zero or more steps that can be taken at this point in solving the problem. As will become clear in the coming sections, not all implementations require a goal test. Some implementations require an additional scoring function like fitness or a heuristic. We also state what guarantees can be given for the implementation.

The implementations we give in this section all use Clean syntax.

4.1 All Firsts

allFirsts, listed in Fig. 7, returns the first steps that can be taken given a state and a *RuleTree* value. Since it does not take the goal into account, these steps are only relevant for problem domains where it is possible to precisely describe the next step to be taken in a solution towards a goal using a *RuleTree* value. Examples of such domains are mathematical and logic exercises. The *allFirsts* service is used if the tree describes only the steps that are always on a path to the goal. This is the case for example in the Ideas framework.

For the *allFirsts* algorithm, we cannot give any guarantee about the given hint. Since it simply returns all steps a user can take, it is completely up to the programmer to guarantee that only steps towards the goal can be taken.

```
fitnessHint :: (a → Int) (RuleTree a) a → [Name]
fitnessHint   f          rt           s =
  map toName (findBest (map (λ(Rule n e) → ((Rule n e), f (e s))) (topRules s rt)))

findBest :: [(Rule a, Int)] → [Rule a]
findBest   []               = []
findBest   [(r, n) : xs]    = fst (helper ([r], n) xs)

helper :: ([Rule a], Int) [(Rule a, Int)] → ([Rule a], Int)
helper   a                []              = a
helper   (rs, n)          [(r, n2) : xs]  | n ≡ n2 = helper (rs ++ [r], n) xs
                                          | n < n2  = helper (rs, n) xs
                                          | n > n2  = helper ([r], n2) xs
```

Fig. 8. Definition of the fitness-based hint service

4.2 Fitness Scoring

Just as *allFirsts*, *fitnessHint*, listed in Fig. 8, only looks at the immediate next steps specified by the *RuleTree*. From these next steps, it selects the best by calculating the fitness of each step, and taking the step with maximum fitness. A fitness function is defined as a function from a to *Int*, where a lower score is better. The algorithm assumes that the fitness function ensures that a user gets closer to the goal if she follows the hints, but of course this depends on the relation between the fitness function and the goal. If a programmer passes a good fitness function as argument, that is, a step selected by the fitness function brings a user closer to the goal and not to a local optimum, then the step returned by *fitnessHint* is part of a sequence of steps that leads to the goal.

4.3 Brute Force

Figure 9 gives a brute force algorithm *bfHint*. It takes a *Goal*, a state a and a *RuleTree*, and returns a list of first steps that can be taken. It uses the function *bfStep*, which returns all paths that reach the goal. Since we only need the first step to produce a hint, we map the *traceToName* function over it. *bfStep* works recursively. It first takes the previous expansions, and checks if there are any states that fulfill the goal condition. If so, we return the traces of these expansions as a solution. If no expansions fulfill the goal, all states are expanded and *bfStep* is called recursively. This means that we search breadth-first. If multiple solutions are found at the same depth, all of these solutions are returned. Therefore, the return type of *bfHint* is a list of *Name*. We only give the type of *expand*.

If *bfHint* returns a hint, then by definition this is a step in a sequence of steps that achieves the goal. Not only that, but since it searches breadth-first for a solution, it is guaranteed to be the first step on the shortest path to the goal.

We have also implemented a special version of *bfHint* that allows the programmer to pass an additional pruning function called a *TraceFilter*. This special

```
bfHint :: (Goal a) (RuleTree a) a → [Name]
bfHint   g          rt              s = map traceToName (bfStep g [([], rt, s)])

bfStep :: (Goal a) [([Name], RuleTree a, a)] → [[Name]]
bfStep   g          items = case [h \\ (h, _, d) ← items | g d] of
                                [] → bfStep g (flatten (map expand items))
                                x → x

expand :: ([Name], RuleTree a, a) → [([Name], RuleTree a, a)]

traceToName :: [Name] → Name
traceToName   []      = ""
traceToName   [x, _]  = x

bfHintFilter :: (TraceFilter a) (Goal a) (RTree a) a → [Name]
```

Fig. 9. Brute force algorithm

version, *bfHintFilter*, applies the filter to the list of expansions after every expansion to prune the search space. Some uses for this are limiting the search depth, pruning duplicate expansions or removing expansions that will never lead to a solution.

4.4 Heuristic Search

A potential problem with the brute force algorithm is that it expands every state until a state fulfills the goal predicate. This might be computationally very expensive. We can try to reach the goal using fewer resources by using a heuristic. A heuristic is a function $hr :: a \to Int$, with $\forall s : hr\ s \geqslant 0$, and $hr\ s \equiv 0$ if and only if s fulfills the goal condition. This heuristic function is used in the search algorithm to search for a solution in a more informed way. A heuristic function differs from a fitness function in the sense that it is no longer required that the function does not lead to a local optimum. The implementation takes this into account by keeping track of states that have already been observed. These states are considered to be closed, and will not be expanded twice.

Figure 10 gives our *heuristicHint* algorithm, which implements a best-first-search algorithm using a heuristic function.

heuristicHint initializes the arguments for *hDecide*.

hDecide looks at the highest scoring expansion. If it fulfills the goal condition, *hDecide* returns the trace of that expansion. If not, it checks if the *RuleTree* in the expansion is empty. If so, the expansion cannot be expanded further and is discarded. *hDecide* is called on the remaining list. If the expansion does not fulfill the goal and has a non-empty *RuleTree*, *hStep* is called to perform the next expansion.

hStep then performs an expansion step. It expands the states that have the lowest score. It then checks if any of the new states have already been observed before. If so, then they are discarded as the expansion is redundant. It adds the new states to the list of observed states, and scores them. Now, the whole list of scored states is sorted, and *hDecide* is called.

```
heuristicHint :: (a → Int) (Goal a) (RuleTree a) a → Name
heuristicHint  f           g        rt           s  =
  traceToName (hDecide f g [(0, ([], rt, s))])

hStep :: (a → Int)   (Goal a) ([a], [(Int, ([Name], RuleTree a, a))]) → [Name]
hStep  _             _        (_  , []                              ) = []
hStep  h             g        (obs, [(n, t) : xs]                   ) =
  hDecide h g (obs ++ newObs, (sortScore ((map (λ(his, t, d) → (h d, (his, t, d)))
                                               (filteredCnds)) ++ tail)))
    where
      candidates  = [t : (map snd (takeWhile (λ(i, _) → i ≡ n) xs))]
      tail        = filter (λ(i, _) → i ≢ n) xs
      filteredCnds = filter (λ(_, _, d) → ¬ ∘ (isMember d obs)) (flatten (map expand candidates))
      newObs      = map (λ(_, _, d) → d) closedExpansion

hDecide :: (a → Int) (Goal a) ([a], [(Int, ([Name], RuleTree a, a))]) → [Name]
hDecide  h           g        (_,   [(_, (his, _       , d)) : xns]  ) | g d = his
hDecide  h           g        (obs, [(_, (_  , Empty, _)) : xns]  ) = hDecide h g (obs, xns)
hDecide  h           g        x                                     = hStep h g x
```

Fig. 10. The *heuristicHint* algorithm

When *hDecide* encounters an expansion that fulfills the goal condition, the trace of this expansion is returned, and *heuristic* takes the first step in the trace and returns it as a hint.

heuristicHint differs from *bfHint* in two ways. First, *heuristicHint* performs only one expansion at a time and does not do this in a breadth-first manner, but best-first. A consequence of using best-first search is that the result trace we get, is not guaranteed to be the shortest path to the goal.

Second, in order to be able to escape local optima, *heuristicHint* prunes away expansions that lead to states already observed. This prevents cyclic expansions, something *bfHint* does not take into account. However, this has no effect on the result of the algorithm since two states will have the same solution.

4.5 Other Algorithms

The algorithms described in this section have been implemented in our framework. There are many other algorithms that support solving problems of the kind described in Sect. 3. A programmer can implement these once, and then solve multiple problems using the same implementation in our framework. Some common algorithms not listed above are A*, Hill climbing, and probabilistic annealing.

```
dnfRT :: RuleTree Expr
dnfRT = Seq [rptRule constantsR, rptRule definitionsR
           , rptRule negationsR, rptRule distributionR]

rptRule rule = Choice [Condition (canApply rule) (Seq [Leaf rule, rptRule rule])
                      , Condition (¬ ∘ (canApply rule)) Empty]

canApply (Rule n e) s = (e s) ≢ s
```

Fig. 11. DNF exercise in our DSL

5 Validation

Sections 3 and 4 fully describe our method for generating feedback systems. To summarize, we first describe the rule-based problem in our DSL. Once the problem is uniformly described, we get a hint function for free by means of the generic solving algorithms. In this section, we validate our approach.

We take real-world examples from each of the three rule-based problem frameworks introduced in Sect. 2, including the examples presented in that section and two new problems, and instantiate feedback systems for them using our proposed approach of first describing the problem using our DSL, and then applying a generic solving algorithm. By doing this, we validate that our approach indeed allows easy instantiation of a feedback system for different rule-based problems.

5.1 Ideas

The Ideas framework introduced in Sect. 2.1 is used to build intelligent tutoring systems. We have taken two examples, with different domains and problems, of actual systems implemented in Ideas: calculating the disjunctive normal form of a logic expression (see Fig. 11), and reducing a matrix to echelon form (see Fig. 12).

Disjunctive Normal Form. Figure 11 lists the description of the disjunctive normal form exercise in our DSL. This is almost a direct translation from the Ideas strategy listed in Sect. 2.1, Fig. 1.

Figure 11 lists the *RuleTree* for the DNF strategy. In order to encode it compactly, an additional combinator function is used called the *rptRule*. This function checks if the rule applies. If so, the rule can be applied, after which *rptRule rule* is called again. If not, the *Empty RuleTree* is returned and this will end the recursion. This means that the *rule* that *rptRule* is applied to, should have an effect on the condition, otherwise this recursion will never terminate.

The *RuleTree* is all that is required to build the hint-function. Since all steps offered by the *RuleTree* are on a path to the goal, we can just return them by using the *allFirsts* algorithm.

```
toReducedEchelon = label "Gaussian elimination" (forwardPass <*> backwardPass)

forwardPass = label "Forward pass" (
  repeat   (  label "Find j-th column"      ruleFindColumnJ
         <*> label "Exchange rows"          (try ruleExchangeNonZero)
         <*> label "Scale row"              (try ruleScaleToOne)
         <*> label "Zeros in j-th column"   (repeat ruleZerosFP)
         <*> label "Cover up top row"       ruleCoverRow))

backwardPass = label "Backward pass" (
  repeat (label "Uncover row" ruleUncoverRow <*> label "Sweep" (repeat ruleZerosBP)))
```

Fig. 12. Gaussian elimination strategy in Ideas

The *hint* function below takes an expression in the domain, and returns steps that can be taken at this point. If no steps are returned, the exercise is solved.

```
hint :: Expr → [Name]
hint = allFirsts dnfRT
```

Gaussian Elimination. Our second example is in the domain of linear algebra. The exercise at hand is to reduce a matrix to echelon form, using Gaussian elimination.

Figure 12 lists the strategy of Gaussian elimination that is used in the Ideas framework. It describes what steps must be applied to a matrix in order to transform it to the reduced echelon from, by means of Gaussian elimination.

The forward pass is applied to the matrix as often as possible. When this procedure no longer applies, the backwards pass is applied exhaustively. If no rules from either two phase apply, the matrix has been reduced. We leave out the exact details of what each rule in the passes does, they are available elsewhere [7].

In order to transform this description into our DSL, we need to introduce two new combinator-functions. Namely to deal with the *repeat* and the *try*. Figure 13 lists the complete description of Gaussian elimination in our DSL, together with these combinator-functions.

Since we are again dealing with a *RuleTree* where all the offered steps are on a path to the goal, no goaltest function is needed to build the hint-function.

```
hint :: Expr → [Name]
hint = allFirsts toReducedEchelonRT
```

As with the DNF example, applying the *RuleTree* to the allFirsts algorithm instantiates the hint-function.

```
toReducedEchelonRT :: RuleTree Expr
toReducedEchelonRT = Seq [forwardPassRT, backwardsPassRT]

forwardPassRT = rptRT Seq [Leaf ruleFindColumnJ,  tryRule ruleExchangeNonZero
                          , tryRule ruleScaleToOne, rptRule ruleZerosFP
                          , Leaf ruleCoverRow]

backwardPassRT = rptRT Seq [Leaf ruleUncoverRow, rptRule ruleZerosBP]

rptRT rt = Choice [Condition (done rt) Empty
                  , Condition (¬ ∘ (done rt)) (Seq [rt, rptRT rt])]

tryRule rule = Choice   [Condition (canApply rule) (Leaf rule)
                        , Condition (¬ ∘ (canApply rule)) Empty]
```

Fig. 13. Gaussian elimination exercise in our DSL

5.2 PuzzleScript

PuzzleScript, as introduced in Sect. 2.2, is a puzzle game framework. We have taken the most well known puzzle game implemented in PuzzleScript, Sokoban, and built a hint system for it.

Sokoban. Figure 2 in Sect. 2.2 lists the source code for Sokoban, written in PuzzleScript. Since PuzzleScript is written in JavaScript and not in Clean, we cannot directly reuse auxiliary functions, like we did when transforming the C2 program.

Figure 14 lists the *RuleTree* for Sokoban. We first define *GameState* which models our state. It contains the *LevelState*, as well as the position of the player pX, pY. *sokobanRT* defines the *RuleTree*. In sequence, it offers choice from one of the four moves, and then recurses. All moves are conditional, they can only be chosen if they can actually be applied. We only supply the types of the functions *validMove* and *applyMove*.

On first attempt, we take the brute force algorithm, and use it to construct our hint-function. For trivial levels, this suffices, but once we have a solution depth of 15, we have to explore $3^{15} \approx 1.4 \times 10^7$ states, assuming that there are on average three valid moves per state.

Brute force clearly will not work. We have to come up with something a bit more clever. Literature on Sokoban [10] points to heuristics and search space pruning to help us order and restrict the search space, and construct a hint-function. Lim and Harrell have generalized these Sokoban heuristics to apply to most PuzzleScript games [14].

```
:: GameState = {lvl :: LevelState, pX :: Int, pY :: Int}
:: LevelState :== [[[GameObject]]]

sokobanRT :: RuleTree GameState
sokobanRT =
  Seq [Choice [Condition (validMove LeftMove)
                         (Leaf (Rule "Move Left"  (applyMove LeftMove  )))
              ,Condition (validMove RightMove)
                         (Leaf (Rule "Move Right" (applyMove RightMove)))
              ,Condition (validMove UpMove)
                         (Leaf (Rule "Move Up"    (applyMove UpMove   )))
              ,Condition (validMove DownMove)
                         (Leaf (Rule "Move Down"  (applyMove DownMove)))]
      ,sokobanRT]

sokobanGoal :: GameState → Bool

validMove :: GameMove GameState → Bool
applyMove :: GameMove GameState → GameState
```

Fig. 14. Sokoban in our DSL

We implement a simple deadlock pruning filter to improve performance. A simple deadlock occurs when a crate is in an unsafe position, from where it will never reach a target. Removing these states reduces the search space.

Implementing heuristics for sokoban, like the mentioned work suggests, can be quite involved. This is beyond the scope of this paper, but could be implemented using the *heuristicHint* algorithm.

To perform simple deadlock detection, we first build a list of unsafe positions. To do this, we find all the corners in the game. After locating the corners, we generate a list of all horizontal and vertical paths from corner to corner. Paths that are not along a wall, or that have walls or targets on them, are removed. The cells on the remaining paths, together with the corners, form the list of unsafe positions. To determine if a state has a deadlock, we simply inspect all unsafe positions. If a state has a crate on an unsafe position, it is removed and thus not further expanded.

Below, the hint function is implemented. For the simple deadlock pruning function, we only provide the type.

```
noDeadlock :: GameState → Bool

hint :: GameState → Name
hint = bfHintFilter (λ(_, _, b) → noDeadlock b) sokobanGoal sokobanRT
```

5.3 iTasks

The iTasks framework introduced in Sect. 2.3 is used to build workflow systems using a notion of task as an abstraction. We have taken two examples, with different domains, of actual systems implemented in iTasks: a C2 workflow system (see Fig. 3), and a sliding puzzle game (see Fig. 16).

```
:: GameState = { board :: [Int], dim :: Int, hole :: Int }
:: Dir = North | East | South | West
boardStore :: Shared GameState

slidePuzzle :: Task GameState
slidePuzzle =
  viewSharedInformation "Sliding Puzzle" [ViewWith viewBoard] boardStore
  >>* map (λdir → OnAction (Action (toName dir) []) (ifValue (checkStep dir)
                                  (λst → set (applyStep dir st) boardStore
                                                              >>| slidePuzzle)))
          [North, East, South, West]

viewBoard :: GameState      → HtmlTag
checkStep :: Dir GameState → Bool
applyStep :: Dir GameState → GameState
```

Fig. 15. Sliding puzzle program written in iTasks

ShipAdventure. Figure 3 in Sect. 2.3 lists the partial source code of a C2 system written in iTasks. We already explored what this problem would look like in our DSL in Sect. 3.1. The *RuleTree* and *shipNotOnFire*-goal-test are listed in Fig. 6.

If we now want to build the hint-function that takes the state of the system and returns a hint for the user how to keep the ship from burning down, we can take the brute force algorithm from Fig. 9 and give it the *RuleTree* and goal-test as shown below.

$$hint :: (SimulationState \rightarrow [Name])$$
$$hint = bfHint\ shipNotOnFire\ shipSimulation$$

Sliding Puzzle. To demonstrate and experiment with iTasks, we implemented a simple sliding puzzle (also called n-puzzle). Figure 15 gives the (partial) source code of the iTasks program that we constructed. In this puzzle, the player arranges all tiles in order, by using the hole to slide the tiles over the board, as shown in Fig. 16.

The record type *GameState* holds the board configuration, the dimension of the puzzle, and the position of the hole. *Dir* defines the kind of moves a player can perform and *slidePuzzle* implements the puzzle.

As with the C2 system, *slidePuzzle* uses the standard task for viewing information to display the current state. Then, it uses the step combinator $\gg\star$ to combine the viewing task with the tasks offering the possible options. We use a *map* to generate the four options a player can choose from.

The goal of the puzzle is to move all tiles in positions so that they appear in order, as shown in Fig. 16b. We now want to add hints to the iTasks program. If the player gets stuck, we want to help out by providing a hint step.

Figure 17 lists the *RuleTree* and *goalTest* for the sliding puzzle. The functions *checkStep* and *applyStep* are the same Clean functions used by the iTasks implementation. The only additional function needed is the *goalTest*, that compares the current board to the solution-state.

(a) Initial state (b) Goal state

Fig. 16. Instance of a block sliding puzzle, of dimension 3×3

```
slidePuzzle :: RuleTree GameState
slidePuzzle = Seq
  [Choice [Condition (checkStep North) (Rule "Move up"    (applyStep North))
           ,Condition (checkStep South) (Rule "Move down"  (applyStep South))
           ,Condition (checkStep West)  (Rule "Move left"  (applyStep West))
           ,Condition (checkStep East)  (Rule "Move right" (applyStep East))]
  ,slidePuzzle]

goalTest :: GameState → Bool
goalTest {board, dim} = [0..((dim * dim) − 1)] ≡ board
```

Fig. 17. Sliding puzzle in our DSL

The n-puzzle problem is too complex to apply a brute force algorithm. An 8-puzzle for example has an average branching factor of 2.67 [15], and an average solution length of 21.97 [20]. We can calculate that we have to visit $2.67^{21.97} \approx 3.39 \times 10^8$ states on average before a solution is found using brute force search.

This calls for a more informed algorithm. Russell and Norvig propose two heuristics for the n-puzzle problem [21]. The first, h_1, is the number of tiles out of place. h_2 is the sum of the (Manhattan) distances of the tiles from their goal positions. With help of h_1, we can construct the following hint function.

```
hint :: GameState → [Name]
hint = heuristicHint h1 goalTest slidePuzzle

h1 :: GameState    → Int
h1 {board, dim} = bDiff board [0..((dim * dim) − 1)]
                  where
                    bDiff []     []     = 0
                    bDiff [x:xs] [y:ys] | x ≢ y = 1 + bDiff xs ys
                                        | True  = bDiff xs ys
```

We use the heuristic search function instead of brute force. This expands the state space in an ordered way. When we now run the original program, in parallel with the hint function, we indeed get a hint for each possible state of the game.

6 Related Work

We follow in a long tradition of creating (domain specific) languages that allow uniform description of rule-based problems, such as planning problems. Some of the early languages written for this purpose are STRIPS [2], PLANNER [8] and SITPLAN [3]. Most of these are based on the same principles as our approach, namely to describe state, operator set and goal test. For example, a STRIPS problem is defined as $\langle P, O, I, G \rangle$, where P is the set of states the problem can be in, O the set of operators, I is the initial state, and G the goal state [1].

A more recent language is PDDL [16]. Version one of the language, from 1998, consists of a domain description, action set, goal description and effects. Again, these ideas coincide with our notion of a problem formalization. The PDDL standard has been updated several times [11], and there are many variants currently in use. These variants include MA-PDDL [12], which can deal with multiple agents, and PPDDL [27], which supports probabilistic effects.

The language we present is different from all of the aforementioned languages in several ways. Our language is a DSL, embedded in Clean. This means that the programmer can use the full power of Clean when constructing the problem description in our DSL. The languages mentioned above are not embedded in any language and therefore the programmer is limited to the syntax of the DSL in constructing the problem description. Another big difference is the fact that in all of the other languages mentioned, except PDDL, the state-space is finite. For example, in SITPLAN, part of the problem description is a finite set of possible situations, and in STRIPS, the set of states is defined as finite a set of conditions that can be either true or false. In our DSL, we do not limit the set of possible states. This allows us to describe many more problems in our DSL, but at the same time makes solving them harder.

The second part of our approach is to solve the problem described in our DSL. When comparing to other approaches, both SITPLAN and PDDL rely on general solvers, just like our approach. In fact, PDDL was initially designed as a uniform language to compare different planning algorithms in the AIPS-98 competition [16]. STRIPS and PLANNER however, do include a specific solving algorithm.

For each of the frameworks that we discussed in this work, there has been some research on generically solving problems. Lim and Harrell [14] present a generic algorithm for evaluating PuzzleScript that discovers solutions for different PuzzleScript games. From these solutions, one could take the first step in the sequence as a hint. The Ideas framework includes a set of feedback services to generate hints for the user. For example, the basic.allfirsts service generates all steps that can be taken at a certain point in the exercise [6]. For the iTasks framework, a system was developed to inspect current executions by using dynamic blueprints of tasks [22]. It can give additional insight in the current and future states, but does not act as a hint-system and does not take a goal into account.

7 Conclusions

With this paper, we set out to answer the question of how to construct a generic feedback framework for rule-based problems. Ideas from the intelligent tutoring community inspired our approach. We first construct a DSL that provides a uniform way to describe many different rule-based problems. Then we can use a generic algorithm to generate feedback, in the form of hints. In order to validate our approach, we have demonstrated that it is indeed possible to encode and instantiate a feedback system for many different problems.

7.1 Future Work

In the future we would like to extend our approach in several ways. First, we would like to extend our DSL to support multiple users. When dealing with for example workflow systems, it is almost always the case that there is more than one user. Secondly, we would like to extend the kind of feedback that we can give to the user. At this point, our system only returns one or more steps that serve as a hint. Other kinds of feedback could for example include a sequence of steps towards the goal or contextual information about where the user is in the greater system. When our DSL would support multiple users, we also need to offer other kinds of feedback. Imagine the situation where a user has to wait on a different user in order to reach her goal, or that a certain step has priority because other users are waiting on that step to be performed.

We are also very interested to see what questions and challenges would come up when our system would be integrated into one of the aforementioned rule-based frameworks. We think that the iTasks framework would be the most interesting candidate, since it allows for a vast amount of completely different problems to be encoded in it.

Acknowledgments. This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organization for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

References

1. Bylander, T.: The computational complexity of propositional STRIPS planning. Artif. Intell. **69**(1–2), 165–204 (1994)
2. Fikes, R., Nilsson, N.J.: STRIPS: a new approach to the application of theorem proving to problem solving. Artif. Intell. **2**(3–4), 189–208 (1971)
3. Galagan, N.I.: Problem description language SITPLAN. Cybern. Syst. Anal. **15**(2), 255–266 (1979)
4. Gerdes, A., Jeuring, J., Heeren, B.: An interactive functional programming tutor. In: Lapidot, T., Gal-Ezer, J., Caspersen, M.E., Hazzan, O. (eds) Proceedings of ITICSE 2012: The 17th Annual Conference on Innovation and Technology in Computer Science Education, pp. 250–255. ACM (2012)
5. Hattie, J., Timperley, H.: The power of feedback. Rev. Educ. Res. **77**(1), 81–112 (2007)
6. Heeren, B., Jeuring, J.: Feedback services for stepwise exercises. Sci. Comput. Program. **88**, 110–129 (2014)
7. Heeren, B., Jeuring, J., Gerdes, A.: Specifying rewrite strategies for interactive exercises. Math. Comput. Sci. **3**(3), 349–370 (2010)
8. Hewitt, C.: PLANNER: a language for proving theorems in robots. In: Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, May 1969, pp. 295–302 (1969)
9. Jeuring, J., et al.: Communicate!—A serious game for communication skills. In: Conole, G., Klobučar, T., Rensing, C., Konert, J., Lavoué, É. (eds.) EC-TEL 2015. LNCS, vol. 9307, pp. 513–517. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24258-3_49

10. Junghanns, A., Schaeffer, J., Sokoban: a challenging single-agent search problem. In: IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch (1997)
11. Kovacs, D.L.: BNF definition of PDDL 3.1 (2011). http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/OtherContributions/kovacs-pddl-3.1-2011.pdf
12. Kovacs, D.L.: A multi-agent extension of PDDL3. In: WS-IPC 2012, p. 19 (2012)
13. Lavelle, S.: PuzzleScript (2016). https://github.com/increpare/PuzzleScript
14. Lim, C.-U., Fox Harrell, D.: An approach to general videogame evaluation and automatic generation using a description language. In: Proceedings of IEEE CIG 2014: Conference on Computational Intelligence and Games, pp. 1–8 (2014)
15. Luger, G.F.: Artificial Intelligence: Structures and Strategies for Complex Problem Solving. Pearson Education, London (2005)
16. McDermott, D., et al.: PDDL-The Planning Domain Definition Language (1998)
17. Murray, T.: An overview of intelligent tutoring system authoring tools: updated analysis of the state of the art. In: Murray, T., Blessing, S.B., Ainsworth, S. (eds.) Authoring Tools for Advanced Technology Learning Environments, pp. 491–544. Springer, Dordrecht (2003). https://doi.org/10.1007/978-94-017-0819-7_17
18. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.W.M.: Task-oriented programming in a pure functional language. In: Proceedings of PPDP 2012: Principles and Practice of Declarative Programming, pp. 195–206. ACM (2012)
19. Plasmeijer, R., van Eekelen, M.: Clean language report version 2.1 (2002)
20. Reinefeld, A.: Complete solution of the eight-puzzle and the benefit of node ordering in IDA. In: Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambéry, France, 28 August–3 September 1993, pp. 248–253 (1993)
21. Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3 International Edition). Pearson Education, London (2010)
22. Stutterheim, J., Achten, P., Plasmeijer, R.: Static and dynamic visualisations of monadic programs. In: Implementation and Application of Functional Languages, Koblenz, Germany, pp. 1–13, December 2015
23. Stutterheim, J., Achten, P., Plasmeijer, R.: C2 demo (2016). https://gitlab.science.ru.nl/clean-and-itasks/iTasks-SDK/tree/master/Examples/Applications/c2-demo
24. VanLehn, K.: The behavior of tutoring systems. Int. J. Artif. Intell. Educ. **16**(3), 227–265 (2006)
25. VanLehn, K., et al.: The Andes physics tutoring system: lessons learned. Int. J. Artif. Intell. Educ. **15**(3), 147–204 (2005)
26. Visser, E., Benaissa, Z.-E.-A., Tolmach, A.P.: Building program optimizers with rewriting strategies. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP 1998), Baltimore, Maryland, USA, 27–29 September 1998, pp. 13–26 (1998)
27. Younes, H.L.S., Littman, M.L.: PPDDL1. 0: the language for the probabilistic part of IPC-4. In: Proceedings of the International Planning Competition (2004)

Author Index