Specifying Strategies for Exercises

Bastiaan Heeren [1] Johan Jeuring [1,2]
Arthur van Leeuwen [2] Alex gerdes [1]

[1] Open Universiteit Nederland
[2] Universiteit Utrecht, The Netherlands

July 30, 2008 (MKM'08)
Birmingham

Overview

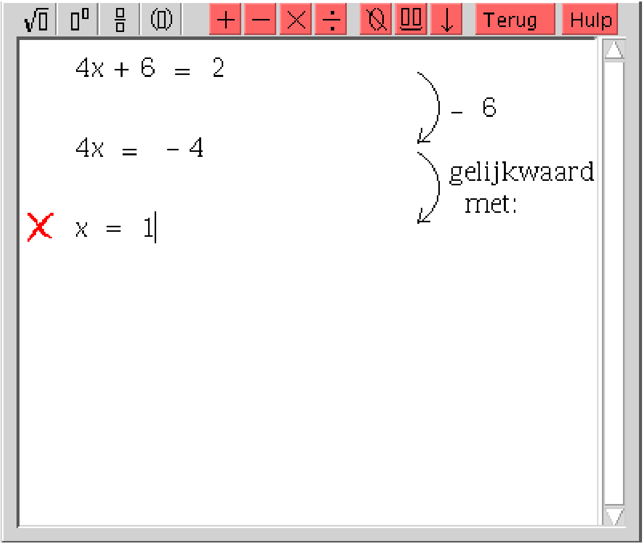
Procedural skills

In many subjects students have to acquire procedural skills:

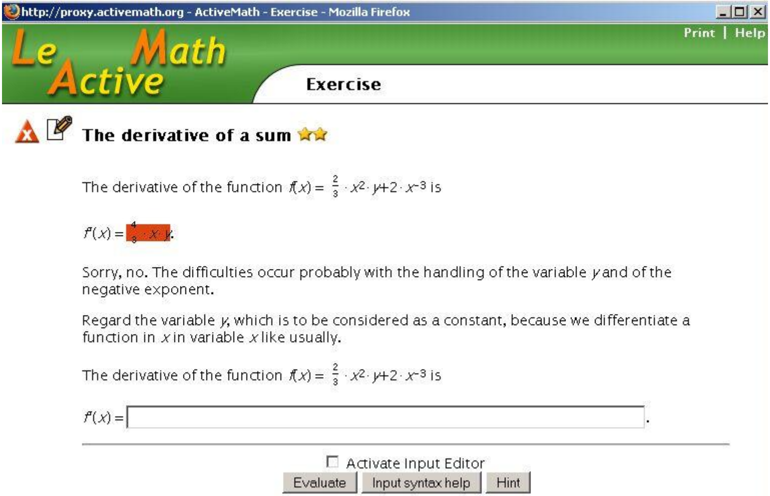
- Mathematics:
  - calculate the value of an expression
  - solve a system of linear equations
  - differentiate a function
  - invert a matrix
- Logic: rewrite a logical expression to disjunctive normal form
- Computer Science: construct a program from a specification using Dijkstra's calculus
- Physics: calculate the resistance of a circuit
- Biology: calculate inheritance values using Mendel's laws

Tutoring tools for procedural skills

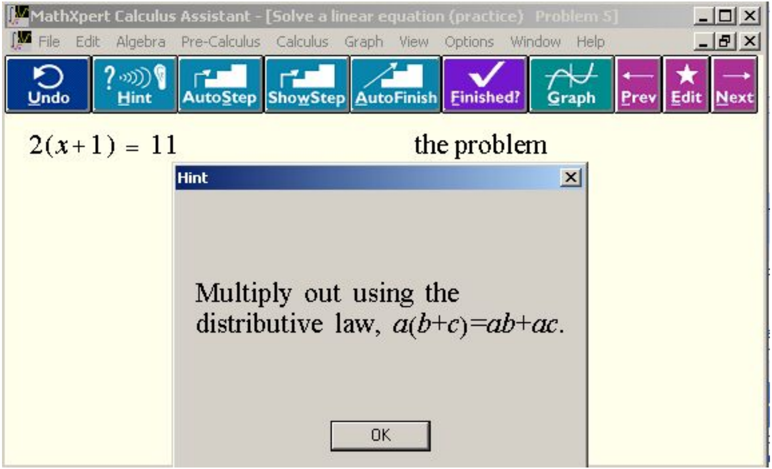
Wisweb (Freudenthal Instituut)



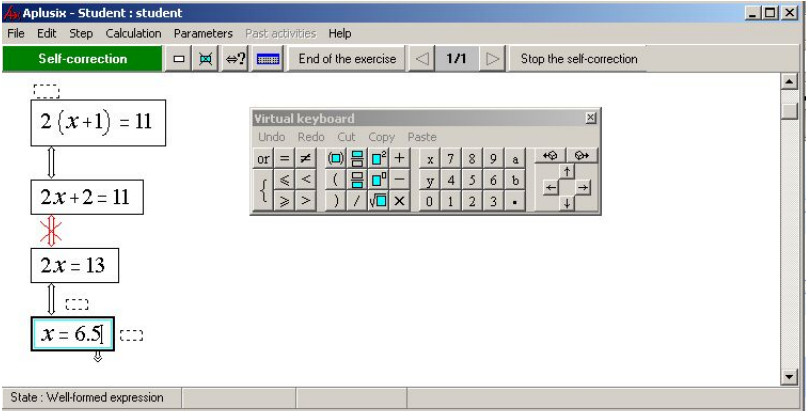
LeActiveMath



MathXPert


MathXpert Calculus Assistant - [Solve a linear equation (practice) Problem 5]
File Edit Algebra Pre-Calculus Calculus Graph View Options Window Help
Undo
Hint
AutoStep
ShowStep
AutoFinish
Finished?
Graph
Prev
Edit
Next
2(x+1) = 11
the problem
Hint
Multiply out using the distributive law, a(b+c)=ab+ac.
OK

Aplusix


Aplusix - Student : student
File Edit Step Calculation Parameters Past activities Help
Self-correction
End of the exercise
1/1
Stop the self-correction
$2(x+1) = 11$
$2x+2 = 11$
$2x = 13$
$x = 6.5$
Virtual keyboard
Undo Redo Cut Copy Paste
State : Well-formed expression

Tutoring tools for procedural skills

- Tutoring tools for practicing procedural skills:
  - generate exercises
  - support stepwise construction of a solution
  - select a rewriting rule, or apply a transformation
  - determine whether a solution is correct/incorrect
- Such tools offer many advantages to users:
  - work at any time
  - select material and exercises
  - a tool can select exercises based on a user-profile
  - a tool can log user errors, and can report common errors back to teachers
  - a tool can give immediate feedback
- There exist many tools for practicing procedural skills
- How are procedures represented?

Representing strategies

- Strategies (procedures) are almost always specified informally
- If a tool can deal with a strategy, it is often hard-wired
- Different teachers sometimes use different strategies for solving problems
- Strategies need to be adaptable and programmable
- If we want diagnose user errors, and give automatic feedback based on a strategy for an exercise, we need an explicit description of the strategy

Rewriting to disjunctive normal form (1)

Rewrite rules for logical propositions:

$$\neg\neg p \Rightarrow p \qquad p \wedge (q \vee r) \Rightarrow (p \wedge q) \vee (p \wedge r)$$

$$\neg(p \wedge q) \Rightarrow \neg p \vee \neg q \qquad (p \vee q) \wedge r \Rightarrow (p \wedge r) \vee (q \wedge r)$$

$$\neg(p \vee q) \Rightarrow \neg p \wedge \neg q$$

- Exercise: bring proposition to disjunctive normal form

$$\neg(\neg(p \vee q) \wedge r)$$

Rewriting to disjunctive normal form (1)

Rewrite rules for logical propositions:

$$\neg\neg p \Rightarrow p \qquad p \wedge (q \vee r) \Rightarrow (p \wedge q) \vee (p \wedge r)$$

$$\neg(p \wedge q) \Rightarrow \neg p \vee \neg q \qquad (p \vee q) \wedge r \Rightarrow (p \wedge r) \vee (q \wedge r)$$

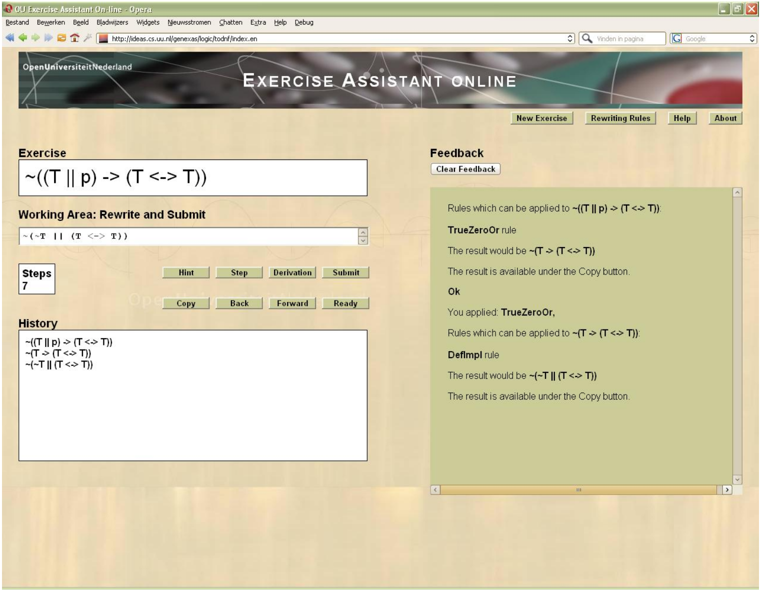
$$\neg(p \vee q) \Rightarrow \neg p \wedge \neg q$$

- Exercise: bring proposition to disjunctive normal form

$$\begin{array}{ll} & \neg(\neg(p \vee q) \wedge r) \\ \Rightarrow & \neg\neg(p \vee q) \vee \neg r \\ \Rightarrow & p \vee q \vee \neg r \end{array}$$

- Exercise is solved in just two steps

Rewriting to disjunctive normal form (2)



Rewriting to disjunctive normal form (3)

- A different derivation (same proposition):

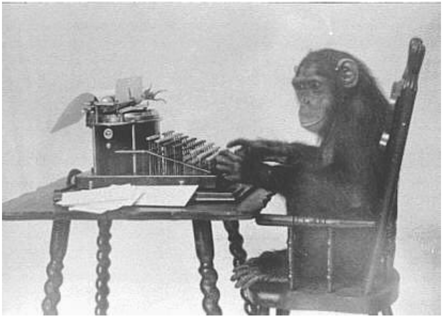
$$
\begin{array}{ll}
 & \neg(\neg(p \vee q) \wedge r) \\
\Rightarrow & \neg((\neg p \wedge \neg q) \wedge r) \\
\Rightarrow & \neg(\neg p \wedge \neg q) \vee \neg r \\
\Rightarrow & \neg\neg p \vee \neg\neg q \vee \neg r \\
\Rightarrow & p \vee \neg\neg q \vee \neg r \\
\Rightarrow & p \vee q \vee \neg r
\end{array}
$$

- Same answer, more steps

Three strategies for disjunctive normal form (1)

Monkey strategy

Apply rules for propositions exhaustively



Three strategies for disjunctive normal form (1)

Monkey strategy

Apply rules for propositions exhaustively

Not very attractive, since it allows

$$\begin{array}{ll} & \neg\neg(p \vee q) \\ \Rightarrow & \neg(\neg p \wedge \neg q) \\ \Rightarrow & \neg\neg p \vee \neg\neg q \\ \Rightarrow & p \vee \neg\neg q \\ \Rightarrow & p \vee q \end{array}$$

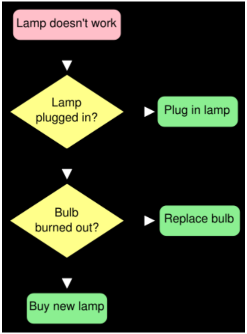
instead of

$$\begin{array}{ll} & \neg\neg(p \vee q) \\ \Rightarrow & p \vee q \end{array}$$

Three strategies for disjunctive normal form (2)

Algorithmic strategy

- Remove constants
- Unfold definitions of implication and equivalence
- Push negations inside (top-down)
- Then use the distribution rule



Three strategies for disjunctive normal form (2)

Algorithmic strategy

- Remove constants
- Unfold definitions of implication and equivalence
- Push negations inside (top-down)
- Then use the distribution rule

Better, but it doesn't take tautologies into account

$$
\begin{array}{ll}
 & (p \vee q) \leftrightarrow (p \vee q) \\
\Rightarrow & ((p \vee q) \wedge (p \vee q)) \vee (\neg(p \vee q) \wedge \neg(p \vee q)) \\
\Rightarrow & \ldots
\end{array}
$$

Three strategies for disjunctive normal form (3)

Expert strategy

- Apply the algorithmic strategy
- Whenever possible, use rules for tautologies and contradictions

Modelling intelligence

To model intelligence in a computer program, Bundy (*The Computer Modelling of Mathematical Reasoning*, 1983) identifies three important, basic needs:

1. The need to have knowledge about the domain
2. The need to reason with that knowledge
3. The need for knowledge about how to direct or guide that reasoning

In our running example,

1. the domain consists of logical expressions
2. reasoning uses rewrite rules for logical expressions
3. strategies guide that reasoning

Specifying a strategy

From the informal specifications of the strategies for DNF we infer that we need the following concepts for specifying a strategy:

- apply a basic rewrite rule *("$\wedge$ distributes over $\vee$")*
- sequence *("first ... then ...")*
- choice *("use one of the rules for $\neg$")*
- apply exhaustively *("repeat ... as long as possible")*
- traversals *("apply ... top down")*

These concepts all appear in (program) transformation languages such as Stratego: a similar language for specifying strategies seems feasible

A strategy language

- The rewrite rules of the domain are the basic ingredients of our strategies.
- A rule can be applied to a term. The application may succeed or fail.
- On top of the basic rules we have the following basic combinators:

Strategy combinators

1. Sequence $s <\star> t$
2. Choice $s <|> t$
3. Unit elements *succeed*, *fail*
4. Labels *label* ℓ s
5. Recursion *fix* f

Concepts

- Just as a rule, a strategy can be applied to a term
- Labels are used to mark positions in a strategy
- Combinators are inspired by context-free grammars
- In fact, this is an embedded domain specific language (in Haskell) and more combinators can be added:

$$\mathit{many}\ s = \mathit{fix}\ (\lambda x \rightarrow \mathit{succeed} <\!|\!> (s <\!\star\!> x))$$

$$\mathit{repeat}\ s = \mathit{many}\ s <\!\star\!> \mathit{not}\ s$$

Traversals

- *once s* applies strategy *s* once to one of the immediate children of the argument term (specific for the domain)

$$
\begin{aligned}
\mathit{once}\ s\ (p \wedge q) &= \{p' \wedge q \mid p' \leftarrow s\ p\} \cup \{p \wedge q' \mid q' \leftarrow s\ q\}\\
\mathit{once}\ s\ (\neg p) &= \{\neg p' \mid p' \leftarrow s\ p\}\\
\mathit{once}\ s\ \mathit{True} &= \emptyset\\
\ldots
\end{aligned}
$$

Traversals

- *once s* applies strategy *s* once to one of the immediate children of the argument term (specific for the domain)

$$
\begin{aligned}
&\mathit{once}\ s\ (p \land q) &&= \{p' \land q \mid p' \leftarrow s\ p\} \cup \{p \land q' \mid q' \leftarrow s\ q\} \\
&\mathit{once}\ s\ (\neg p) &&= \{\neg p' \mid p' \leftarrow s\ p\} \\
&\mathit{once}\ s\ \mathit{True} &&= \emptyset \\
&\ldots
\end{aligned}
$$

With *once* we can now define:

- *somewhere s*: apply *s* once to a subterm
- *bottomUp s*: apply *s* bottom up
- *topDown s*: apply *s* top down

DNF strategies revisited (1)

Monkey strategy:

```
dnfStrategy1 = repeat (somewhere basicRules)

basicRules  =  label "Basic rules"
               (   constants <|> definitions
               <|> negations <|> distribution)

constants   =  label "Constant rules"
               (   andTrue <|> andFalse <|> orTrue
               <|> orFalse <|> notTrue  <|> notFalse)
```

DNF strategies revisited (2)

Algorithmic strategy:

$$
\begin{aligned}
\mathit{dnfStrategy2} = \; & \\
& \mathit{label}\ \texttt{"step 1"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants})) \\
\lt\!\star\!\gt\; & \mathit{label}\ \texttt{"step 2"}\ (\mathit{repeat}\ (\mathit{bottomUp}\ \mathit{definitions})) \\
\lt\!\star\!\gt\; & \mathit{label}\ \texttt{"step 3"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{negations})) \\
\lt\!\star\!\gt\; & \mathit{label}\ \texttt{"step 4"}\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{distribution}))
\end{aligned}
$$

Using strategies for error diagnosis and feedback

Given a term, a strategy, and a step made by a student, we can give several kinds of feedback:

- Feedback after a step
- Progress
- Strategy unfolding
- Hint
- Completion problems
- Buggy strategies

How

Conceptually:

- View the strategy specification as a grammar
- Solving an exercise is now constructing a sentence using the grammar
- Prefix parsers can be used to diagnose errors and give feedback

The details of how this is done is worth another presentation

Related work

- Anderson: Rules of the mind
- VanLehn: Mind bugs – the origins of procedural misconceptions
- Collections of condition-action rules
- Leading argument: the procedural language should be psychologically plausible
- Our point: the diagnosis and feedback should be psychologically plausible
- Our language satisfies VanLehn's requirements
- In many other approaches, rules and strategies are hard-wired into the tool

Current status and future work

Current status:

- A library with which we can define strategy recognizers
  See `ideas.cs.uu.nl/trac/`
- Tested on logic expressions, linear algebra, arithmetic expressions, and relational algebra
- Used to a limited extent in some courses
- Most of the forms of error diagnosis and feedback
- Available as web services

Future work:

- Investigate strategies for constructing programs
- Give a formal account of strategies
- Flexible strategies
- Simplify adding a new domain

Conclusions

- We have introduced a strategy language with which we can specify strategies in many domains
- A strategy is specified as a context-free grammar, extended with some non-context-free constructs
- The formulation of a strategy as a context-free grammar allows us to automatically calculate several kinds of feedback and error diagnosis
- Separating strategy specification from error diagnosis and feedback calculation makes it possible to calculate different kinds of feedback
- Ours is not the first strategies for exercises language, but it is the first that allows automatic calculation of different kinds of error diagnosis and feedback