Software technology for learning and teaching

Part 3: Programming tutors

Tim Olmer, Alex Gerdes, and Hieke Keuning

Bastiaan Heeren[1] and Johan Jeuring[1,2]

[1] Open Univerity of the Netherlands
[2] Utrecht University

26 January 2015, IPA course, Eindhoven

Open Universiteit
www.ou.nl

Overview

1. Introduction
2. Haskell Expression Evaluator
3. Ask-Elle
4. A tutor for imperative programming

Learning programming

- Programming is difficult
- Individual support for students in large classes is hard
- Can programming tutors help?

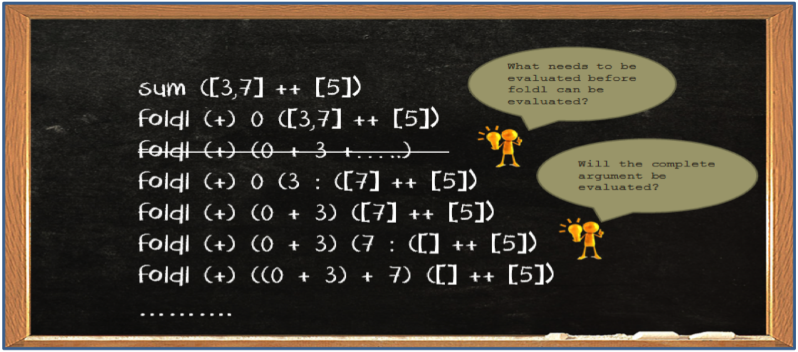
Programming tutors

We have developed three tools supporting learning programming:

- Haskell Expression Evaluator
- Ask-Elle
- Imperative Programming Tutor

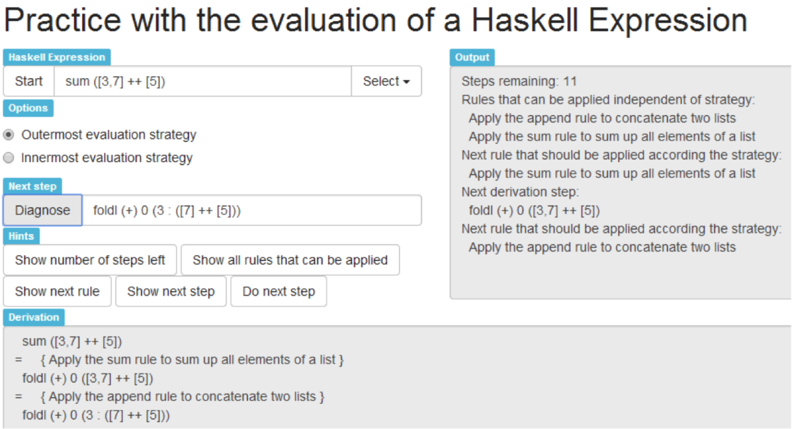
Outline of presentation

Haskell Expression Evaluator

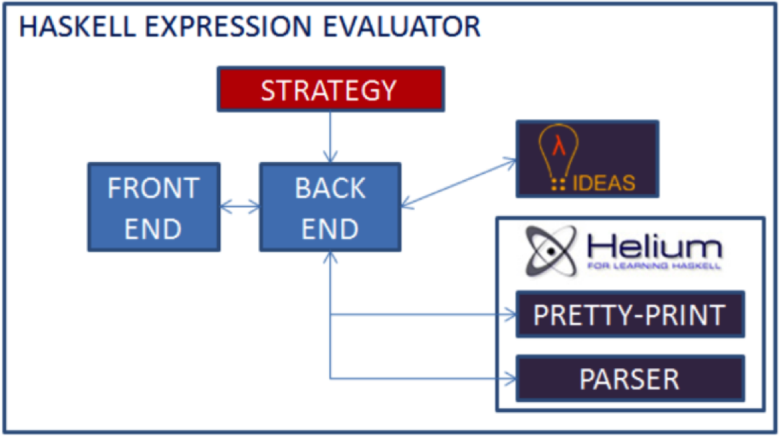


Frontend

http://ideas.cs.uu.nl/HEE/index.html



Architecture



Calculating sums

$$sum = foldl\ (+)\ 0$$

$$\begin{aligned} foldl\ \_\ v\ [\,] &= v \\ foldl\ f\ v\ (x : xs) &= foldl\ f\ (f\ v\ x)\ xs \end{aligned}$$

$$\begin{aligned} [\,] +\!\!+\ ys &= ys \\ (x : xs) +\!\!+\ ys &= x : (xs +\!\!+\ ys) \end{aligned}$$

$$[3,7] = 3 : 7 : [\,]$$

Step-wise evaluation example

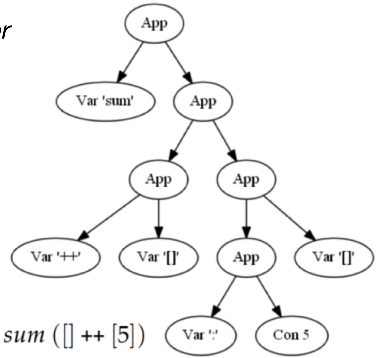
$$
\begin{aligned}
& \mathit{sum}\ ([3,7] \mathbin{+\!\!+} [5]) \\
= & \quad \{ \text{definition } \mathit{sum} \} \\
& \mathit{foldl}\ (+)\ 0\ ([3,7] \mathbin{+\!\!+} [5]) \\
= & \quad \{ \text{definition } \mathbin{+\!\!+} \} \\
& \mathit{foldl}\ (+)\ 0\ (3 : ([7] \mathbin{+\!\!+} [5])) \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& \mathit{foldl}\ (+)\ (0+3)\ ([7] \mathbin{+\!\!+} [5]) \\
= & \quad \{ \text{definition } \mathbin{+\!\!+} \} \\
& \mathit{foldl}\ (+)\ (0+3)\ (7 : ([] \mathbin{+\!\!+} [5])) \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& \mathit{foldl}\ (+)\ ((0+3)+7)\ ([] \mathbin{+\!\!+} [5]) \\
= & \quad \{ \text{definition } \mathbin{+\!\!+} \} \\
& \mathit{foldl}\ (+)\ ((0+3)+7)\ [5] \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& \mathit{foldl}\ (+)\ (((0+3)+7)+5)\ [] \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& (((0+3)+7)+5) \\
= & \quad \{ \text{applying } + \} \\
& ((3+7)+5) \\
= & \quad \{ \text{applying } + \} \\
& (10+5) \\
= & \quad \{ \text{applying } + \} \\
& 15
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{sum}\ ([3,7] \mathbin{+\!\!+} [5]) \\
= & \quad \{ \text{definition } \mathit{sum} \} \\
& \mathit{foldl}\ (+)\ 0\ ([3,7] \mathbin{+\!\!+} [5]) \\
= & \quad \{ \text{definition } \mathbin{+\!\!+} \} \\
& \mathit{foldl}\ (+)\ 0\ (3 : ([7] \mathbin{+\!\!+} [5])) \\
= & \quad \{ \text{definition } \mathbin{+\!\!+} \} \\
& \mathit{foldl}\ (+)\ 0\ (3 : 7 : ([] \mathbin{+\!\!+} [5])) \\
= & \quad \{ \text{definition } \mathbin{+\!\!+} \} \\
& \mathit{foldl}\ (+)\ 0\ [3,7,5] \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& \mathit{foldl}\ (+)\ (0+3)\ [7,5] \\
= & \quad \{ \text{applying } + \} \\
& \mathit{foldl}\ (+)\ 3\ [7,5] \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& \mathit{foldl}\ (+)\ (3+7)\ [5] \\
= & \quad \{ \text{applying } + \} \\
& \mathit{foldl}\ (+)\ 10\ [5] \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& \mathit{foldl}\ (+)\ (10+5)\ [] \\
= & \quad \{ \text{applying } + \} \\
& \mathit{foldl}\ (+)\ 15\ [] \\
= & \quad \{ \text{definition } \mathit{foldl} \} \\
& 15
\end{aligned}
$$

Datatype for expressions

$$
\begin{aligned}
\textbf{data } \mathit{Expr} = \; & \mathit{App}\ \mathit{Expr}\ \mathit{Expr} \\
\mid \; & \mathit{Abs}\ \mathit{String}\ \mathit{Expr} \\
\mid \; & \mathit{Var}\ \mathit{String} \\
\mid \; & \mathit{Con}\ \mathit{Int}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{appN} :: \mathit{Expr} \to [\mathit{Expr}] \to \mathit{Expr} \\
&\mathit{appN} = \mathit{foldl}\ \mathit{app}
\end{aligned}
$$

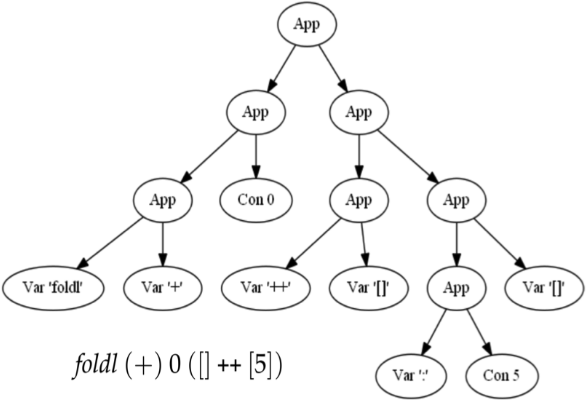


Rewrite strategy outermost

- Descend to function until node is not an *App*
- Try to apply beta reduction
  - If current node lambda abstraction *App* (*Abs* *x* *e*) *a*, substitute variable *x* by *a* in expression *e*
- Or try to apply one of the evaluation strategies for definitions
  - Check function name and number of arguments
  - If needed bring argument(s) in WHNF (apply outermost strategy recursively)
  - Apply rewrite rule

Example outermost rewriting

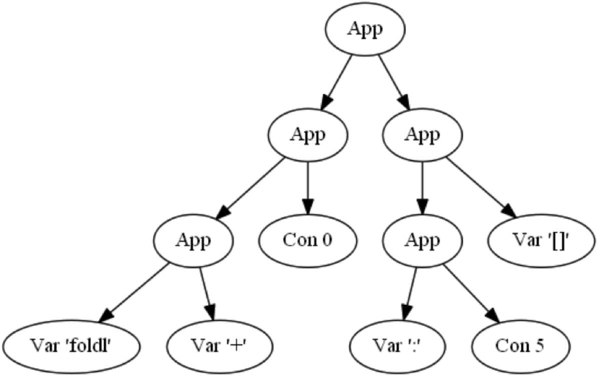
$$
\begin{aligned}
&foldl\ \_\ v\ [] &&= v \\
&foldl\ f\ v\ (x : xs) &&= foldl\ f\ (f\ v\ x)\ xs \\
&[]\ {+}{+}\ ys &&= ys \\
&(x : xs)\ {+}{+}\ ys &&= x : (xs\ {+}{+}\ ys)
\end{aligned}
$$



$foldl\ (+)\ 0\ ([]\ {+}{+}\ [5])$

Example outermost rewriting

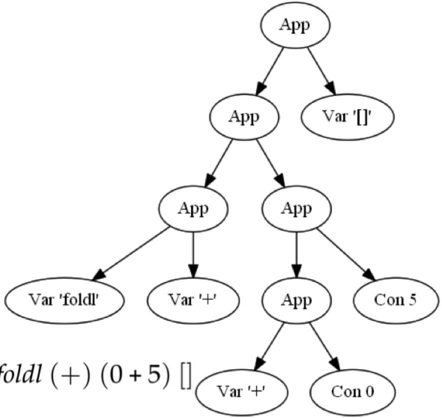
$$
\begin{aligned}
foldl\ \_\ v\ [\,] &= v \\
foldl\ f\ v\ (x : xs) &= foldl\ f\ (f\ v\ x)\ xs \\
[\,] \mathbin{+\!\!+} ys &= ys \\
(x : xs) \mathbin{+\!\!+} ys &= x : (xs \mathbin{+\!\!+} ys)
\end{aligned}
$$



$foldl\ (+)\ 0\ [5]$

Example outermost rewriting

$$
\begin{aligned}
&\mathit{foldl}\ \_\ v\ [\,] &&= v\\
&\mathit{foldl}\ f\ v\ (x : xs) &&= \mathit{foldl}\ f\ (f\ v\ x)\ xs\\
&[\,] \mathbin{++} ys &&= ys\\
&(x : xs) \mathbin{++} ys &&= x : (xs \mathbin{++} ys)
\end{aligned}
$$



$\mathit{foldl}\ (+)\ (0 + 5)\ [\,]$

User-defined function definitions

$$sum = foldl\ (+)\ 0$$

$$Var\ \texttt{"sum"} \mapsto appN\ (Var\ \texttt{"foldl"})\ [Var\ \texttt{"(+)"}, Con\ 0]$$

- Wish:
  - Easily add support for new functions
  - Rewrite rules and evaluation strategies are very similar
- Possible solution:
  - One configuration file on the server
  - Use annotations to add a description
  - Let the evaluator generate rewrite rules and strategies
- Future: determine from function definition
  - Number of arguments
  - Which argument(s) must be in WHNF

Future work

- Support user-defined function definitions
- Configure the step size of a function
- Lazy evaluation
  - Can be supported by introducing let expressions to label arguments
  - Place arguments in a heap and make the heap visible

Conclusions

- Prototype to support students in better understanding
  - How Haskell expressions evaluate
  - Programming concepts (recursion, higher-order functions, pattern-matching)
  - Evaluation strategies (innermost and outermost evaluation)
- Prototype uses rewrite rules and rewrite strategies
- Evaluation process is driven by
  - Rewrite rules
  - Evaluation strategy (multiple variants)
- Feedback uses IDEAS services
- User defined function definitions can be supported by
  - Parsing function definitions
  - Generate rewrite rules/evaluation strategy

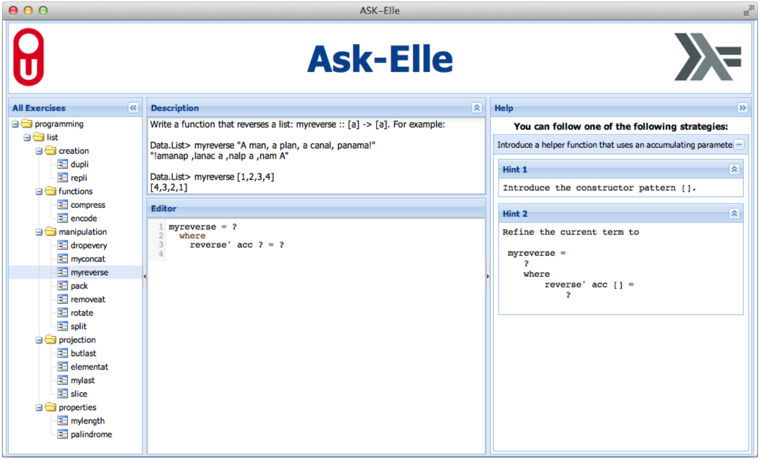
Outline of presentation

1. Introduction
2. Haskell Expression Evaluator
3. Ask-Elle
4. A tutor for imperative programming

Ask-Elle: basic ideas

- Incrementally construct a program
- Get feedback on each intermediate step:
  - syntax, dependencies, types (Helium)
  - equal to, or transformable to, part of a model solution (IDEAS/Ask-Elle)
  - property testing (QuickCheck)
- Ask for a hint

Ask-Elle: demo



Ask-Elle for assessment

- We used Ask-Elle to assess a lab assignment in 2009
- 94 submissions
  - 72 correct (sometimes with superfluous input checks)
  - 64 recognised (89%) from 4 model solutions
  - improved on hand-grading

Ask-Elle for tutoring

- We used Ask-Elle for tutoring in 2013
- 83% of the 3.500 submitted programs were correctly diagnosed as right or wrong
- 56% of the ‘correct’ programs are recognised as parts of model solutions
- With better program transformations: 81%

Underlying technologies

- Model solutions
- Program annotations
- Program refinements
- Programming strategies
- Program transformations
- Deep search

Model solutions

For each task, Ask-Elle uses one or more model solutions:

$$
\begin{aligned}
&\mathit{myreverse} = \mathit{reverse}\ [\,]\\
&\quad \textbf{where}\\
&\qquad \mathit{reverse}\ \mathit{acc}\ [\,] \qquad\quad = \mathit{acc}\\
&\qquad \mathit{reverse}\ \mathit{acc}\ (x : xs) = \mathit{reverse}\ (x : \mathit{acc})\ xs
\end{aligned}
$$

$$\mathit{myreverse} = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]$$

Program annotations


```
{-# DESC Use the prelude function foldl #-}
myreverse =
  {-# FEEDBACK foldl takes an operator and a ... #-}
    (foldl {-# FEEDBACK Use flip and (:) #-}
          (flip (:))
          []
    )
```

Program refinements

In Ask-Elle, a student refines a program:

$$\begin{array}{l} myreverse = reverse\ [\,] \\ \quad \textbf{where}\ reverse\ \ ?\ ? = ? \end{array}$$

can be refined to

$$\begin{array}{l} myreverse = reverse\ [\,] \\ \quad \textbf{where}\ reverse\ \ acc\ [\,] = ? \end{array}$$

Refinement rules

Each (combination of) abstract syntax construct(s) leading to a visible change of a program gives rise to a refinement rule

? $\mapsto$ **if** ? **then** ? **else** ?

Programming strategies

A programming strategy specifies how a program

$$myreverse = foldl\ (flip\ (:))\ [\,]$$

is constructed using refinement rules:

Introduce a pattern binding
<∗> *Introduce the pattern var* `"myreverse"`
<∗> *Introduce an application*
<∗> *Introduce the var* `"foldl"`
<∗> ([...*Introduce the first argument of foldl*...]
<%>
Introduce con [ ])

Constructing programming strategies

- Turn library functions into strategies
  - choice between name and definition
- Turn model solutions into strategies
  - top-down using `<*>`, arguments and list of declarations using `<%>`, annotations are included as labels
- Take the `<|>` of the model strategies

Analysing student programs

- Parse a student program
- Normalise it
- Use the programming strategy to construct a tree of 'all' intermediate programs
- Check that the student program occurs somewhere in this tree
- 'Parallel' Tomita-like parsing

Program transformations

- Desugaring
- Inlining
- Constant arguments
- Alpha, beta, eta

Example transformations

$$encode :: Eq\ a \Rightarrow [a] \rightarrow [(Int, a)]$$

$$> encode\ [1, 2, 2, 3, 2, 4]$$
$$[(1, 1), (2, 2), (1, 3), (1, 2), (1, 4)]$$

Example transformations model solution

$$
\begin{aligned}
&\mathit{encode}\ [\,] &&= [\,]\\
&\mathit{encode}\ (x : \mathit{xs}) &&= (n+1, x) : \mathit{encode}\ (\mathit{drop}\ n\ \mathit{xs})\\
&\quad \textbf{where}\ n = \mathit{length}\ (\mathit{takeWhile}\ ({==}\ x)\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encode}\ [\,] &&= [\,]\\
&\mathit{encode}\ (x : \mathit{xs}) &&= ((\mathit{length}\ (\mathit{takeWhile}\ ({==}\ x)\ \mathit{xs}) + 1, x)\\
&\quad : \mathit{encode}\ (\mathit{drop}\ (\mathit{length}\ (\mathit{takeWhile}\ ({==}\ x)\ \mathit{xs}))\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encode}\ [\,] &&= [\,]\\
&\mathit{encode}\ (x : \mathit{xs}) &&= (1 + (\mathit{length}\ (\mathit{takeWhile}\ ({==}\ x)\ \mathit{xs}), x)\\
&\quad : \mathit{encode}\ (\mathit{drop}\ (\mathit{length}\ (\mathit{takeWhile}\ ({==}\ x)\ \mathit{xs}))\ \mathit{xs})
\end{aligned}
$$

Example transformations student solution

$$
\begin{aligned}
&\mathit{encode}\ [\,] &&= [\,] \\
&\mathit{encode}\ (x : xs) &&= (\mathit{length}\ \$\ x : \mathit{takeWhile}\ (== x)\ xs, x) \\
& &&\quad : \mathit{encode}\ (\mathit{dropWhile}\ (== x)\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encode}\ [\,] &&= [\,] \\
&\mathit{encode}\ (x : xs) &&= (1 + \mathit{length}\ (\mathit{takeWhile}\ (== x)\ xs), x) \\
& &&\quad : \mathit{encode}\ (\mathit{dropWhile}\ (== x)\ xs)
\end{aligned}
$$

{-# ALT *dropWhile p xs = drop (length (takeWhile p xs)) xs* #-}

Deep search

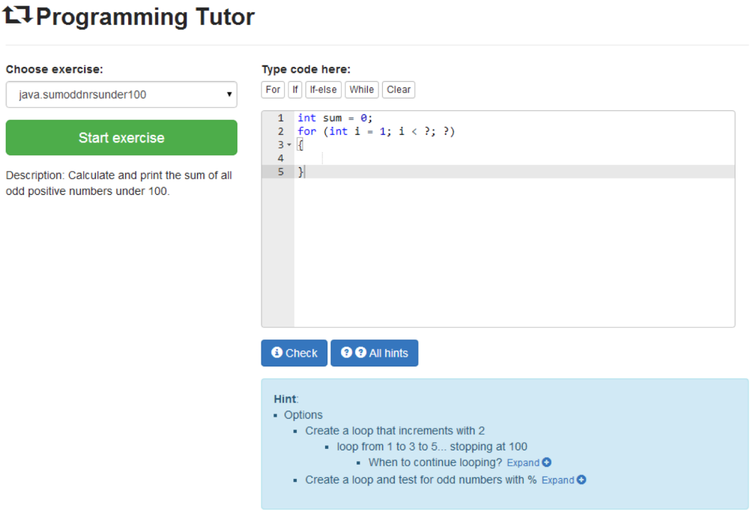
- Diagnose a single step, multiple steps, or a complete program
- Huge search space!
- Using that the order of refinements does not matter makes the problem tractable

Future work

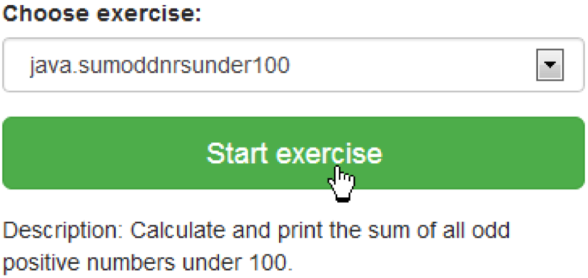
- More transformations
- Contracts
- Refactoring

Outline of presentation

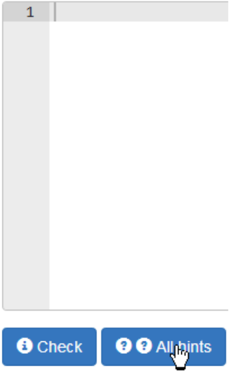
A tutor for imperative programming



A tutoring session I



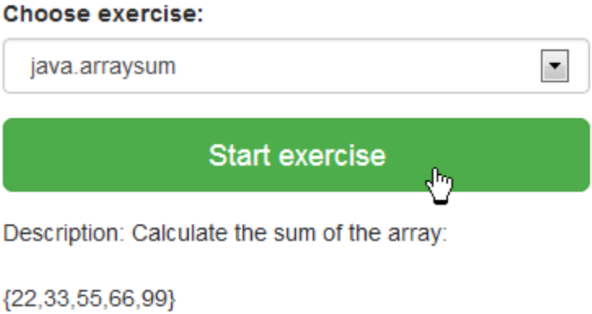
A tutoring session I



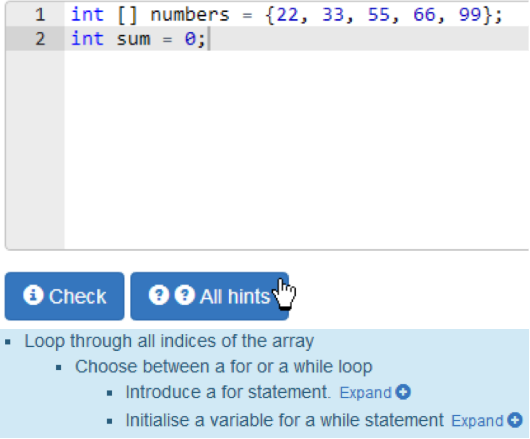
Options:

- Create a loop that increments with 2
    - Introduce a variable declaration.
        - Type code int ?;
- Create a loop and test for odd numbers with % Expand
- Perform a smart calculation Expand

A tutoring session II



A tutoring session II



A tutoring session II


```java
int [] numbers = {22, 33, 55, 66, 99};
int sum = 0;
for (int i = 0; i < numbers.length; i++)
{
    sum = sum + numbers[i];
}
System.out.println("sum");
```

Check   All hints

Error: The output is incorrect

A tutoring session II


```java
int [] numbers = {22, 33, 55, 66, 99};
int sum = 0;
for (int i = 0; i < numbers.length; i++)
{
    sum = sum + numbers[i];
}
System.out.println(sum);
```

Check | All hints

You are done!

Feedback: Correct.

Components

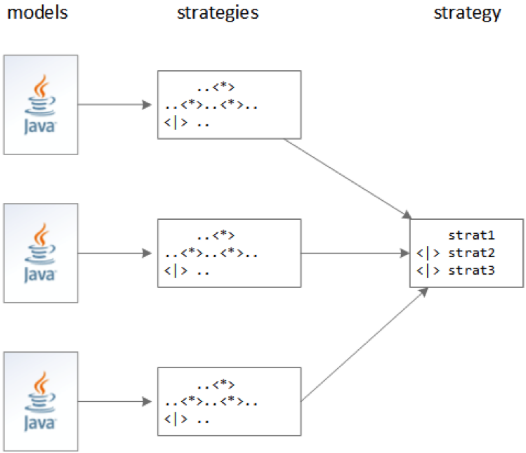
- Abstract syntax, parser and pretty-printer
- A strategy generator
- Feedback services
- Annotations

Abstract syntax

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Stat} = & \mathit{Block} & [\mathit{Stat}] \\
\mid & \mathit{If} & \mathit{Expr}\ \mathit{Stat} \\
\mid & \mathit{IfElse} & \mathit{Expr}\ \mathit{Stat}\ \mathit{Stat} \\
\mid & \mathit{While} & \mathit{Expr}\ \mathit{Stat} \\
\mid & \mathit{For} & [\mathit{Expr}]\ [\mathit{Expr}]\ [\mathit{Expr}]\ \mathit{Stat} \\
\mid & \mathit{Print} & \mathit{Expr} \\
\mid & \mathit{VarDecls} & \mathit{DataType}\ [\mathit{Expr}] \\
\mid & \mathit{ExprStat} & \mathit{Expr} \\
\mid & \mathit{Empty} & \\
\mid & \mathit{Break} & \\
\mid & \mathit{Continue} &
\end{array}
$$

Strategies for imperative programming

Rules (steps) and a strategy that combines rules.



Append rule


```
x = 5;
```

⇨

```
x = 5
y = 7;
```

⇨

```
x = 5;
y = 7;
avg =(x+y)/2;
```

⇨

```
x = 5;
y = 7;
avg =(x+y)/2;
print(avg);
```

Refinement rule

```
avg = ?;
```

⇨

```
avg = ? / ?;
```

⇨

```
avg = (? + ?) / ?;
```

⇨

```
avg = (x + y) / ?;
```

⇨

```
avg = (x + y) / 2;
```

Strategy for if

$$
\begin{aligned}
&\textit{genStrat loc pref } (\textit{If cond body}) = \mathbf{do}\\
&\quad (\textit{hole}, \textit{cond}') \quad\ \leftarrow \textit{genStratWithLoc pref cond}\\
&\quad (\textit{block}, \textit{body}') \leftarrow \textit{genStratWithLoc pref body}\\
&\quad \textit{app} \qquad\qquad\quad \leftarrow \textit{appRule } (\textit{If hole block})\\
&\quad \textit{return } \$ \textit{ app} \mathbin{\langle\star\rangle} \textit{cond}' \mathbin{\langle\star\rangle} \textit{body}'
\end{aligned}
$$

```
if (?) {
}
```

⇨

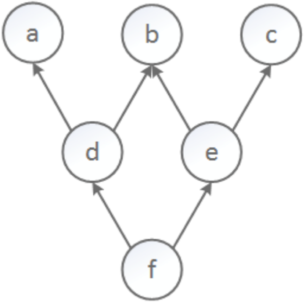
```
if (isOk) {
}
```

⇨

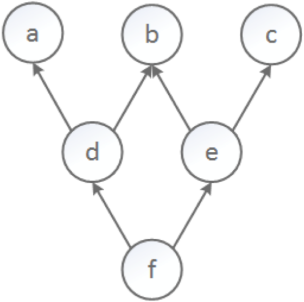
```
if (isOk) {
  call();
}
```

Strategy for *Block*

$$a = 1;$$
$$b = 2;$$
$$c = 3;$$
$$d = a + b;$$
$$e = b + c;$$
$$f = d + e;$$



Topological sort



$$
\begin{array}{l}
(a \mathbin{<\!\star\!>} ((b \mathbin{<\!\star\!>} \quad (\quad (c \mathbin{<\!\star\!>} \quad (\\
\qquad\qquad\qquad\qquad\qquad\qquad (d \mathbin{<\!\star\!>} e \mathbin{<\!\star\!>} f) \\
\qquad\qquad\qquad\qquad\qquad \mathbin{<\!|\!>} (e \mathbin{<\!\star\!>} d \mathbin{<\!\star\!>} f))) \\
\qquad\qquad\qquad \mathbin{<\!|\!>} (d \mathbin{<\!\star\!>} c \mathbin{<\!\star\!>} e \mathbin{<\!\star\!>} f))) \\
\qquad \mathbin{<\!|\!>} \quad (c \mathbin{<\!\star\!>} \,..))) \\
\mathbin{<\!|\!>} (b \mathbin{<\!\star\!>} \,..) \\
\mathbin{<\!|\!>} (c \mathbin{<\!\star\!>} \,..)
\end{array}
$$

Semantics-preserving Variations

Xu & Chee 2003:

	Description	AST	Strategy
SPV1	Different algorithms		✓
SPV2	Different source code formats	✓	
SPV3	Different syntax forms	✓	✓
SPV4	Different variable declarations		✓
SPV5	Different algebraic expression forms		
SPV6	Different control structures		✓
SPV7	Different Boolean expression forms		
SPV8	Different temporary variables		
SPV9	Different redundant statements		
SPV10	Different statement orders		✓
SPV11	Different variable names		
SPV12	Different program logical structures		
SPV13	Different statements		✓

Semantics-preserving Variations

Xu & Chee 2003:

	Description	AST	Strategy	Normalisation
SPV1	Different algorithms		✓	
SPV2	Different source code formats	✓		
SPV3	Different syntax forms	✓	✓	✓
SPV4	Different variable declarations		✓	✓
SPV5	Different algebraic expression forms			✓
SPV6	Different control structures		✓	
SPV7	Different Boolean expression forms			✓
SPV8	Different temporary variables			
SPV9	Different redundant statements			
SPV10	Different statement orders		✓	
SPV11	Different variable names			✓
SPV12	Different program logical structures			
SPV13	Different statements		✓	

Normalisation

Transforming a program into a canonical form:

- Syntax desugaring
- Renaming variables
- Rewriting expressions
- ...

Feedback services

DeepDiagnose from Ask-Elle:

$$
\begin{aligned}
\textbf{data}\ \mathit{Diagnosis}\ a = {} & \mathit{Buggy} && \ldots \\
\mid {} & \mathit{NotEquivalent} && \ldots \\
\mid {} & \mathit{Similar} && \ldots \\
\mid {} & \mathit{WrongRule} && \ldots \\
\mid {} & \mathit{Expected} && \ldots \\
\mid {} & \mathit{Detour} && \ldots \\
\mid {} & \mathit{Correct} && \ldots \\
\mid {} & \mathit{Unknown} && \ldots
\end{aligned}
$$

Feedback services

AllHints from Ask-Elle:

Introduce a loop statement:

- Introduce a for statement
  - Type code for (?; ?; ?)
- Initialise a variable for a while statement
  - Expand ? to a variable assignment
    - Type code i = ?;

Labels

$$\mathit{return}\ \$\ \mathit{app}$$
$$\quad \langle\star\rangle\ \mathit{label}\ \texttt{"if-condition"}\ \mathit{cond}'$$
$$\quad \langle\star\rangle\ \mathit{label}\ \texttt{"if-true"}\ \mathit{body}'$$

```
feedback if-condition =
  What do you want to check?
feedback if-true      =
  What do you want to do if the condition is true?
```

Adapting feedback


```
/* DESC Implement the Quicksort algorithm */

/* PREF 2 DIFF Hard */


/* FEEDBACK Calculate the average of the two results */
double avg = (x + y) / 2;


/* ALT x = Math.max(a,b); */
if (a > b) x = a;
else x = b;
/* MUSTUSE */for (int i = 1; i <= 10; i ++);
```

Conclusions

- Rewriting strategies, feedback services, and domain reasoners can be used to develop various programming tutors
- The development of programming tutors is still quite a lot of work
- Lots of opportunities to use software technology to improve the tutors