Tutoring by single-stepping interpretation of a functional program

R&D questions raised in the ISAC-project

Walther Neuper

Institute for Softwaretechnology
Graz University of Technology

TU Wien, Institut für Computersprachen
Jun.2, 2010
Outline

1. **ISAC**
   - Project and product
   - Architecture
   - Demo

2. **Programming language**
   - Syntax
   - Example programs
   - Program interpreter

3. **Open R&D tasks**
   - Re-engineer the language
   - Re-engineer the interpreter

4. **Summary**
1. ISAC
   Project and product
   Architecture
   Demo

2. Programming language
   Syntax
   Example programs
   Program interpreter

3. Open R&D tasks
   Re-engineer the language
   Re-engineer the interpreter

4. Summary
Tutoring by Interpretation
Walther Neuper

ISAC project and product

- History of the ISAC project
  - 1992/93: UNU/IIST project by D.Bjørner, B.Buchberger, P.Lucas
  - 1994 - 99: design by B.Buchberger, P.Lucas, W.Neuper
  - 1999/2001: math-engine implemented by W.Neuper
  - 2002 – : front-end implemented by 20 student projects
  - 2008 – : field tests at schools
  - 2010 – : (re-)start academic R&D

- Experimental ISAC prototype
  - 20.000 LOCs math-engine (SML, exploiting Isabelle); programming language is heart of math-engine
  - 17.000 LOCs Isabelle theories by several projects
  - 40.000 LOCs front-end (Java)
  - 100.000 LOCs knowledge by several projects; generated from ISAC’s Isabelle theories + html-content
**Tutoring by Interpretation**

**Walther Neuper**

**ISAC**

Project and product

Architecture

Demo

Programming language

Syntax

Example programs

Program interpreter

**Open R&D tasks**

Re-engineer the language

Re-engineer the interpreter

**Summary**

---

**ISAC project and product**

- **History of the ISAC project**
  - 1992/93: UNU/IIST project by D.Bjørner, B.Buchberger, P.Lucas
  - 1994 - 99: design by B.Buchberger, P.Lucas, W.Neuper
  - 1999/2001: math-engine implemented by W.Neuper
  - 2002 – : front-end implemented by 20 student projects
  - 2008 – : field tests at schools
  - 2010 – : (re-)start academic R&D

- **Experimental ISAC prototype**
  - 20.000 LOCs math-engine (SML, exploiting Isabelle); programming language is heart of math-engine
  - 17.000 LOCs Isabelle theories by several projects
  - 40.000 LOCs front-end (Java)
  - 100.000 LOCs knowledge by several projects; generated from ISAC’s Isabelle theories + html-content
Outline

1. **ISAC**
   - Project and product
   - Architecture
   - Demo

2. Programming language
   - Syntax
   - Example programs
   - Program interpreter

3. Open R&D tasks
   - Re-engineer the language
   - Re-engineer the interpreter

4. Summary
ISAC’s present experimental version
Version under construction (Isars contexts)
Next version: generalized front-end

Editor (jEdit, Lyx)
Parser

Isabelle
- Provers
- Theories
- rewriting
- matching
- pretty printing

Contexts

ML toplevel

Script Diff (f::real, ...)
Repeat Try ...
Rewrite ...

Scala Isar toplevel

Re-engineer the language
Re-engineer the interpreter

Open R&D tasks

Programming language
- Syntax
- Example programs
- Program interpreter

ISAC
- Project and product
- Architecture
- Demo

Tutoring by Interpretation
Walther Neuper
Could there be Standard ML instead the Isabelle terms?
Outline

1. **ISAC**
   - Project and product
   - Architecture
   - Demo

2. Programming language
   - Syntax
   - Example programs
   - Program interpreter

3. Open R&D tasks
   - Re-engineer the language
   - Re-engineer the interpreter

4. Summary
The demonstration briefly showed the

1. example “bending lines”
   - (interactive) calculation and specification combined
   - knowledge *transparent* to the user
   - 3 dimensions of (human readable) knowledge

2. example “differentiation”
   - system “knows” (program calculates) the next step
   - checks an input rule (“tactic”) . . .
   - checks an input formula (using a *prover*) . . .
   . . . and system knows next step again
   (program resumes in the *continuation* of last step.)

Tutoring in stepwise calculation is a side-effect of a program interpreted in single-stepping mode!
The demonstration briefly showed the

1. example “bending lines”
   - (interactive) calculation and specification combined
   - knowledge *transparent* to the user
   - 3 dimensions of (human readable) knowledge

2. example “differentiation”
   - system “knows” (program calculates) the next step
   - checks an input rule ("tactic") . . .
   - checks an input formula (using a prover) . . .
   . . . and system knows next step again
   (program resumes in the *continuation* of last step.)

Tutoring in stepwise calculation is a side-effect of a program interpreted in single-stepping mode!
The demonstration briefly showed the

1. example “bending lines”
   - (interactive) calculation and specification combined
   - knowledge *transparent* to the user
   - 3 dimensions of (human readable) knowledge

2. example “differentiation”
   - system “knows” (program calculates) the next step
   - checks an input rule (“tactic”) . . .
   - checks an input formula (using a *prover*) . . .
   . . . and system knows next step again
   (program resumes in the *continuation* of last step.)

Tutoring in stepwise calculation is a side-effect of a program interpreted in single-stepping mode!
Tutoring in stepwise calculation is a side-effect of a program interpreted in single-stepping mode!
ISAC demonstration

The demonstration briefly showed the

1 example “bending lines”
   • (interactive) calculation and specification combined
   • knowledge transparent to the user
   • 3 dimensions of (human readable) knowledge

2 example “differentiation”
   • system “knows” (program calculates) the next step
   • checks an input rule ("tactic") . . .
   • checks an input formula (using a prover) . . .
   . . . and system knows next step again
   (program resumes in the continuation of last step.)

Tutoring in stepwise calculation is a side-effect of
a program interpreted in single-stepping mode!
The demonstration briefly showed the

1 example “bending lines”
   - (interactive) calculation and specification combined
   - knowledge *transparent* to the user
   - 3 dimensions of (human readable) knowledge

2 example “differentiation”
   - system “knows” *(program calculates)* the next step
   - checks an input rule (“tactic”) . . .
   - checks an input formula (using a *prover*) . . .
   . . . and system knows next step again
   *(program resumes in the *continuation* of last step.)*

*Tutoring in stepwise calculation is a *side-effect of a program* interpreted in single-stepping mode!*
Outline

1 ISAC
   Project and product
   Architecture
   Demo

2 Programming language
   Syntax
   Example programs
   Program interpreter

3 Open R&D tasks
   Re-engineer the language
   Re-engineer the interpreter

4 Summary
A functional language, BNF

\[ \text{script ::= Script id arg}^* = \text{body} \]
\[ \text{arg ::= id | ( (id :: type))} \]
\[ \text{body ::= expr} \]
\[ \text{expr ::= } \% \text{id . expr} \quad (*\text{Isabelle/HOL}*) \]
\[ \quad | \text{LET id = expr ( ; id = expr)}^* \text{ IN expr} \]
\[ \quad | \text{IF prop THEN expr ELSE expr} \]
\[ \quad | \text{listexpr} \quad (*\text{lists in Isabelle/HOL}*) \]
\[ \quad | \text{id | ( (id :: type))} \]
\[ \quad | \text{Repeat expr} \quad (*\text{tacticals}*) \]
\[ \quad | \text{Try expr} \]
\[ \quad | \text{expr Or expr} \]
\[ \quad | \text{Take ( id | listexpr )} \quad (*\text{tactics}*) \]
\[ \quad | \text{Rewrite id ( id | listexpr )} \]
\[ \quad | \text{Rewrite_Set id ( id | listexpr )} \]
\[ \quad | \text{Rewrite_Set_Inst subst id ( id | listexpr )} \]
\[ \quad | \text{Substitute subst ( id | listexpr )} \]
\[ \quad | \text{SubProblem (id, idlist, idlist) arglist} \]
\[ \quad | ... \]
Outline

1. ISAC
   - Project and product
   - Architecture
   - Demo

2. Programming language
   - Syntax
   - Example programs
   - Program interpreter

3. Open R&D tasks
   - Re-engineer the language
   - Re-engineer the interpreter

4. Summary
Isabelle/HOL code for example
“bending line”

01  Script bendingLine
02  (l_::real) (q_::real) (v_::real) (b_::real=>real) (rb_::bool list) =
03    (LET
04      (funs_:: bool list) =
05        (SubProblem (Bendingline,[bendingline,integrate],
06            [bendingline,integrate])
07            [real_ q_, real_real_ b_, real_ v_]);
08      (equs_::bool list) =
09        (SubProblem (Bendingline,[bendingline,setConstraints],
10            [bendingline,setConstraints])
11            [bools_ funs_, bools_ rb_, real_ l_]);
12      (sols_::bool list) =
13        (SubProblem (Real,[equation,system,linear],[])
14            [bools_ equs_, reals_ [c,c_2,c_3,c_4]]);
15    B_ = Take (LAST funs_);
16    B_ = ((Substitute sols_) @@
17          (Rewrite_Set_Inst [(bdv, v_)] make_ratpoly_in)) B_
18    IN B_)
Isabelle/HOL code for example “bending line”

01 Script bendingLine
02 (l_::real) (q_::real) (v_::real) (b_::real=>real) (rb_::bool list) =
03 (LET
04  (funs_:: bool list) =
05       (SubProblem (Bendingline,[bendingline,integrate],
06                      [bendingline,integrate])
07                      [real_ q_, real_real_ b_, real_ v_]);
08  (equs_::bool list) =
09       (SubProblem (Bendingline,[bendingline,setConstraints],
10                      [bendingline,setConstraints])
11                      [bools_ funs_, bools_ rb_, real_ l_]);
12  (sols_::bool list) =
13       (SubProblem (Real,[equation,system,linear],[])
14                      [bools_ equs_, reals_ [c,c_2,c_3,c_4]]);
15  B_ = Take (LAST funs_);
16  B_ = ((Substitute sols_) @@
17          (Rewrite_Set_Inst [(bdv, v_)] make_ratpoly_in)) B_
18  IN B_)
Isabelle/HOL code for example
“differentiation”

01  Script differentiate (f_::real=>real) (v_::real) =
02      LET
03          f’_ = Take (d_d v_ f_) =
04      IN Repeat
05          (Repeat (Rewrite_Inst [(bdv, v_-)] diff_sum) Or
06            Repeat (Rewrite_Inst [(bdv, v_-)] diff_prod_const) Or
07            Repeat (Rewrite_Inst [(bdv, v_-)] diff_prod) Or
08            Repeat (Rewrite_Inst [(bdv, v_-)] diff_quot) Or
09            Repeat (Rewrite_Inst [(bdv, v_-)] diff_sin) Or
10           Repeat (Rewrite_Inst [(bdv, v_-)] diff_sin_chain) Or
11            Repeat (Rewrite_Inst [(bdv, v_-)] diff_cos) Or
12           Repeat (Rewrite_Inst [(bdv, v_-)] diff_cos_chain) Or
13            Repeat (Rewrite_Inst [(bdv, v_-)] diff_pow) Or
14           Repeat (Rewrite_Inst [(bdv, v_-)] diff_pow_chain) Or
15            Repeat (Rewrite_Inst [(bdv, v_-)] diff_const) Or
16           Repeat (Rewrite_Inst [(bdv, v_-)] diff_var) Or
17            Repeat (Rewrite_Set make_polynomial)
18       ) f’_
Isabelle/HOL code for example “differentiation”

01 Script differentiate \((f_\to real \Rightarrow real) (v_\to real) =\)
02 \hspace{1em} LET
03 \hspace{2em} f' _ = Take \((d_d v_ f_ ) =\)
04 \hspace{3em} IN Repeat
05 \hspace{4em} (Repeat (Rewrite_Inst [(bdv, v_)] diff_sum) Or
06 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_prod_const) Or
07 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_prod) Or
08 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_quot) Or
09 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_sin) Or
10 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_sin_chain) Or
11 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_cos) Or
12 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_cos_chain) Or
13 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_pow) Or
14 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_pow_chain) Or
15 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_const) Or
16 \hspace{5em} Repeat (Rewrite_Inst [(bdv, v_)] diff_var) Or
17 \hspace{5em} ) f' _
Outline

1. ISAC
   - Project and product
   - Architecture
   - Demo

2. Programming language
   - Syntax
   - Example programs
   - Program interpreter

3. Open R&D tasks
   - Re-engineer the language
   - Re-engineer the interpreter

4. Summary
Interpreter for the language (SML)

signature INTERPRETER =
sig
  type initstate
  var next : script → calcstate → next
  var locate_tactic : script → calcstate → tactic → located
  var locate_formula : script → calcstate → formula → located
end

type script = term (*term from Isabelle*)
type intstate = env * location (*location in script*)
type calcstate = calctree * position (*intstate in calctree at position*)
datatype next = EndProgram | Step of calcstate
datatype located = Helpless | Steps of calcstate
Interpreter for the language (SML)

signature INTERPRETER =
  sig
    type initstate
    var next : script → calcstate → next
    var locate_tactic : script → calcstate → tactic → located
    var locate_formula : script → calcstate → formula → located
  end

type script = term (*term from Isabelle*)
type intstate = env * location (*location in script*)
type calcstate = calctree * position (*intstate in calctree at position*)
datatype next = EndProgram | Step of calcstate
datatype located = Helpless | Steps of calcstate
1. **ISAC**
   - Project and product
   - Architecture
   - Demo

2. **Programming language**
   - Syntax
   - Example programs
   - Program interpreter

3. **Open R&D tasks**
   - Re-engineer the language
   - Re-engineer the interpreter

4. **Summary**
Open tasks in language design

- reconsider the statements (different provenance from Isabelle | Isabelle/HOL | ISAC)
- extend tacticals
  
  \[
  \begin{align*}
  \text{Repeat} & \quad \text{expr} \\
  \text{Try} & \quad \text{expr} \\
  \text{expr} & \quad \text{Or} \quad \text{expr} \\
  \text{expr} & \quad \text{Parallel} \quad \text{expr}
  \end{align*}
  \]
- extend tactics
  
  \[
  \begin{align*}
  \text{Take} & \quad ( id \mid \text{listexpr} ) \\
  \text{Rewrite} & \quad id \quad ( id \mid \text{listexpr} ) \\
  & \quad \ldots \\
  \text{Cases} & \quad \text{expr} \; ( ; \; \text{expr} )^\ast \quad \text{EndCases}
  \end{align*}
  \]
- extend tactics to geometric constructions
- ? push language “down” from Isabelle/HOL to SML?
Tutoring by Interpretation

Walther Neuper

ISAC
Project and product
Architecture
Demo

Programming language
Syntax
Example programs
Program interpreter

Open R&D tasks
Re-engineer the language
Re-engineer the interpreter

Summary

Outline

1 ISAC
Project and product
Architecture
Demo

2 Programming language
Syntax
Example programs
Program interpreter

3 Open R&D tasks
Re-engineer the language
Re-engineer the interpreter

4 Summary
Open tasks in interpretation

• Re-engineer the interpreter as a debugger
  (debugger recognizes tactics as “breakpoints”)
• Extend interpreter according to language extensions
  (set of locations for parallel execution etc)
• Clarify employment of provers in continuations
  (provers check formulas input by user)
• Investigate verification of calculations
  (calculations are created by the interpreter)
• . . .
An idea from P.Lucas still seems to raise R&D questions:

- **Idea:** tutoring is a side-effect of programming  
  (programming math in a widely-used language ?!)
- **Open questions** in language design  
  (language integrating computation and deduction ?!)
- **Challenges** in interpreter construction  
  (tutoring derived from an (adapted) debugger ?!)

A window of innovation seems open presently!