Program Transformations

Refinements from (formal) specification to actual programs

- Also known as “Transformational Programming”
- Formal mathematical approach
- Problem description in mathematical notation,
  e.g.: \( \text{queens}(N) \) where
  \( \text{funct} \ \text{queens} = (\text{nat} \ n) \ \text{bool}: \exists \ \text{set of} \ \{(1..n),(1..n)\} \ m: |m| = n \land \text{nonconflict}(m) \)

Program Transformations

- Refinements from (formal) specification to actual programs
- Refactorings (object-oriented programs)
- Transformation of legacy code
  - Syntactical transformations
  - Semantical transformations

Program Transformations

Transformation rules, such as fold and unfold:

\[
E' \ \text{where} \ m \ x = E \quad \text{DEF} \quad \{f(E)\} \\
\]

\( f(E) \)

Syntactic constraints:
- \( \text{NOTOCCURS}[x \ \text{in} \ E] \)
- \( \text{DECL}[f] = \text{funct}(m \ x) \ n: E' \)

Fold

\[
E' \ \text{where} \ m \ x = E \quad \text{DEF} \quad \{f(E)\} \\
\]

\( f(E) \)

Syntactic constraints:
- \( \text{NOTOCCURS}[x \ \text{in} \ E] \)
- \( \text{DECL}[f] = \text{funct}(m \ x) \ n: E' \)

Unfold
Program Transformations
Refinements from (formal) specification to actual programs

**Step 1:** unfold \textit{credits}:

```plaintext
funct \textit{newbal} = (int \textit{oldbal}, \textit{tfile} t) int:
oldbal + \textit{credits}(t) - \textit{debits}(t) \textit{-num}(t) * fee where

funct \textit{credits} = (\textit{tfile} t) nat:
  if \textit{t} =<> then 0
  elsif \textit{kind}(\textit{first} \textit{t}) = \textit{C} then \textit{amount}(\textit{first} \textit{t}) + \textit{credits}(\textit{rest} \textit{t})
    else \textit{credits}(\textit{rest} \textit{t}) fi

funct \textit{debits} = (\textit{tfile} t) nat:
  if \textit{t} =<> then 0
  elsif \textit{kind}(\textit{first} \textit{t}) = \textit{C} then \textit{debits}(\textit{rest} \textit{t})
    else \textit{amount}(\textit{first} \textit{t}) + \textit{debits}(\textit{rest} \textit{t}) fi

funct \textit{num} = (\textit{tfile} t) nat:
  if \textit{t} =<> then 0 else 1 + \textit{num}(\textit{rest} \textit{t}) fi
```

**Step 2:** distributivity of operations over conditional

```plaintext
if \textit{t} =<> then
  oldbal + \textit{amount}(\textit{first} \textit{t}) + \textit{credits}(\textit{rest} \textit{t})
    - \textit{debits}(\textit{t}) - \textit{num}(\textit{t}) * fee
  \textit{kind}(\textit{first} \textit{t}) = \textit{C} then
    \textit{amount}(\textit{first} \textit{t}) + \textit{debits}(\textit{rest} \textit{t})
      - \textit{debits}(\textit{t}) - \textit{num}(\textit{t}) * fee fi
else
  \textit{oldbal} + \textit{credits}(\textit{rest} \textit{t}) - \textit{debits}(\textit{t}) - \textit{num}(\textit{t}) * fee fi
```

**Step 3:** instantiate:

- \textit{debits}(\textit{t}) = 0 [if \textit{t} =<> holds]
- \textit{num}(\textit{t}) = 0 [if \textit{t} =<> holds]
- \textit{debits}(\textit{t}) = \textit{debits}(\textit{rest} \textit{t}) [if \textit{t} =<> and \textit{kind}(\textit{first} \textit{t}) = \textit{C} hold]
- \textit{debits}(\textit{t}) = \textit{amount}(\textit{first} \textit{t}) + \textit{debits}(\textit{rest} \textit{t}) [if \textit{t} =<> and \textit{kind}(\textit{first} \textit{t}) = \textit{C} hold]
- \textit{num}(\textit{t}) = 1 + \textit{num}(\textit{rest} \textit{t}) [if \textit{t} =<> holds]

if \textit{t} =<> then \textit{oldbal} + 0 - 0 * fee
  elsif \textit{kind}(\textit{first} \textit{t}) = \textit{C}
    then \textit{oldbal} + \textit{amount}(\textit{first} \textit{t}) + \textit{credits}(\textit{rest} \textit{t})
      - (1 + \textit{num}(\textit{rest} \textit{t})) * fee
  else \textit{oldbal} + \textit{credits}(\textit{rest} \textit{t}) - (\textit{amount}(\textit{first} \textit{t}) + \textit{debits}(\textit{rest} \textit{t}))
    - (1 + \textit{num}(\textit{rest} \textit{t})) * fee fi
```

**Step 4:** apply arithmetic simplification rules:

if \textit{t} =<> then \textit{oldbal}
  elsif \textit{kind}(\textit{first} \textit{t}) = \textit{C}
    then \textit{oldbal} + \textit{amount}(\textit{first} \textit{t}) - \textit{fee} + \textit{credits}(\textit{rest} \textit{t})
      - \textit{debits}(\textit{rest} \textit{t}) - \textit{num}(\textit{rest} \textit{t}) * fee
  else \textit{oldbal} - \textit{amount}(\textit{first} \textit{t}) - \textit{fee} + \textit{credits}(\textit{rest} \textit{t})
    - \textit{debits}(\textit{rest} \textit{t}) - \textit{num}(\textit{rest} \textit{t}) * fee fi
Program Transformations
Refinements from (formal) specification to actual programs

Step 5: fold newbal:

\[
\text{funct } \text{newbal} = (\text{int } \text{oldbal}, \text{tfile } t) \text{ int}:
\]

\[
\begin{align*}
\text{if } t &=<> \text{ then } \text{oldbal} \\
\text{else } \text{kind}(\text{first } t) &= C \\
\text{then } \text{newbal}(\text{oldbal} + \text{amount}(\text{first } t) – \text{fee}, \text{rest } t) \\
\text{else } \text{newbal}(\text{oldbal} - \text{amount}(\text{first } t) – \text{fee}, \text{rest } t) \\
\end{align*}
\]

fi

Further reading: Specification and transformation of programs by Helmut A. Partsch

Program Transformations
Refactorings (object-oriented programs)

- Refactoring: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
- Refactoring:
  - Improves the design of software
  - Makes software easier to understand
  - Helps you find bugs
  - Increases your productivity

Program Transformations
Refactorings (object-oriented programs)

- When should you refactor:
  - When you do something for the third time
  - When you add a function
  - When you need to fix a bug
  - When you do a code review

- Well-known approach: CIP-L/CIP-S
  - CIP = Computer-Aided Intuition-Guided Programming
  - CIP-L = specification language
  - CIP-S = interactive program transformation environment

- Tedious approach, not very popular
- “Follow-up” in the functional programming community: Bird-Meertens formalism
**Program Transformations**

Refactorings (object-oriented programs)

- Bad code smells:
  - Duplicated code
  - Long methods
  - Large classes
  - Long parameter lists
  - Message chains
  - Etc.

Some refactorings:

- Extract/inlinemethod
- Inline temporary variable
- Introduce explaining variable
- Split temporary variable, prevents reuse of variables
- Move method, field (to another class)
- Extract/inline class
- Etc.

Further reading: Refactoring by Martin Fowler

---

**Program Transformations**

Transformation of legacy code

- From program analysis to program transformation
- Semantic preserving transformations of (COBOL) source code
- Get rid of old/obsolete/deprecated language constructs
- NOT from procedural to object-oriented
- Define once, apply often (factory approach)
- Code improvement

In a number of projects ASF+SDF has been used to define program transformation rules for COBOL:

- Dialect conversion: from COBOL74 to COBOL85
- Removal of company specific COBOL extensions
- General code improvement

Simple transformations, e.g., introduction of missing keywords, removal of optional keywords
Program Transformations
Transformation of legacy code

• More complex transformations:
  – Control flow improvement, by means of GOTO elimination
• Problematic flow of control
• No real subroutines
• Each modification hinders future modifications

Program Transformations
Transformation of legacy code

1. Derivation of grammar
2. Preprocessing
3. Actual program transformation
4. Pretty printing
5. Postprocessing

Program Transformations
Transformation of legacy code

• Derivation of the grammar
  – Copying ANSI standard into SDF (done by a student)
  – Derivation of SDF definition based on source base (a collection of non-trivial COBOL programs)
  – Automatic derivation from IBM reference manual
• Problematic issues:
  – Embedded grammars: SQL, CICS, Assembler, CICS
  – Company specific alternations and extensions to the standard.

Program Transformations
Transformation of legacy code

• The IBM Cobol VS2 grammar
Program Transformations
Transformation of legacy code

- Unstructured version in LLL (a kind of EBNF)
  - Formalism of Grammar Development Kit

\begin{verbatim}
Add-stat1 : "ADD" cobword+ "TO" cobword+
on-size-error-statement-list? not-on-size-error-statement-list? "END-ADD"? ;
on-size-error-statement-list :
  "ON"? "SIZE" "ERROR" statement-list ;
How to parse?:
START SECTION.
ADD A TO B ON SIZE ERROR DISPLAY "error"
DISPLAY "next statement"
. 
\end{verbatim}

- Grammar Development Kit allows:
  - Restructuring/refactoring of grammars
  - Transformation to various formalisms:
    - SDF
    - LEX/YACC
    - Etc.
  - Developed at the Vrije Universiteit by Ralf Lämmel and Jan Kort

From GDK/LLL to SDF
- Some additional SDF-information has been added.
  - Addition of avoid/prefer attributes to productions
- GDK exports final LLL-grammar to SDF
  - Every sort, and all its productions, are assigned to a module
  - This gives an SDF grammar consisting of
    - 16 modules
    - 552 sorts
    - 906 productions
Program Transformations
Transformation of legacy code

Grammatical issues

- Disambiguation of nested statements

```
IF AA = 1 CONTINUE
ELSE
  IF BB = 2 CONTINUE
  END-IF
IF CC = 3 CONTINUE
```

SDF offers following disambiguation mechanisms:

- Lexical disambiguation, e.g., prefer longest match
- Associativity and priorities
- Avoid/prefer mechanism

Program Transformations
Transformation of legacy code

Grammatical issues

- None of these mechanisms is powerful enough.

- Disambiguation via rewrite rules:

```
[amb-statement-list2]
amb(#Statement-list,
  #If-statement #If-statement-non-closed) = #Statement-list

#Statement-list corresponds with inner-most binding of end-if

• Remove unwanted trees/parses
```

Program Transformations
Transformation of legacy code

Transformations

- Legacy Cobol code
- Improve structure and modifiability
- Code isolation and goto elimination
- Transformation algorithm for automatic application
- Applied to over 5 million LOC
- 60-80% goto-free code

Program Transformations
Transformation of legacy code

- Control-flow: three ways

```
GO TO LABEL
...  ↓  ↓  ↓
...  LABEL
...  ↓  ↓  ↓
goto statement
```

```
PERFORM LABEL
...  ↓  ↓  ↓
...  LABEL
...  ↓  ↓  ↓
perform statement
```

```
LABEL-1
...  ↓
LABEL-2
...  ↓
fall-through
```
Program Transformations
Transformation of legacy code

- Inserting new code

Program Transformations
Transformation of legacy code

- Improving modifiability

Program Transformations
Transformation of legacy code

- Collection of (simple) transformations
- Eliminates fall-through and goto statements via subroutines
- Algorithm for automatic application
- Extensions for and adaptation to a base-line system
- Large-scale application to industrial COBOL systems
**Program Transformations**
Transformation of legacy code

- Algorithm: isolate **GOTO-free** code

```
Isolate-code

Eliminate-Goto

Algorithm

PERFORM LABEL-1
PERFORM LABEL-2
PERFORM LABEL-3
...
GOBACK
LABEL-1
LABEL-2
LABEL-3
...

Resulting source code
```

- Extensions and adaptation to base-line system
  - Base-line system is used to calibrate the transformations
  - 87 IBM COBOL programs, a real-life system of a bank
  - 80,000 LOC containing 2938 GOTOs
  - Implemented about 25 GOTO elimination patterns
  - 80.6% of GOTOs eliminated!

**Program Transformations**
Transformation of legacy code

- Transformations implemented/specified in ASF+SDF
- Powerful pattern-matching based on concrete syntax
- Efficient rewriting (compiled to C code)
- Example pattern:

```
Label.
IF Condition
Statements1
GO TO Label
END-IF.

Label.
PERFORM UNTIL NOT Condition
Statements1
END-PERFORM.
```

**Program Transformations**
Transformation of legacy code

- 10 SDF modules for the COBOL grammar:
  - 780 production rules
- 30 modules for the various transformations:
  - 190 production rules (functions)
  - 230 rewrite rules
- Total:
  - 2070 LOC SDF
  - 2800 LOC ASF
Program Transformations
Transformation of legacy code

- Heavily based on ASF traversal functions
- Bottom-up, continue
  - Useful for visiting all occurrences of a production rule
- Top-down, break
  - More 'control' over traversal
- Choosing a strategy is based on:
  - Efficiency
  - Minimal number of equations

Program Transformations
Transformation of legacy code

- Partial parsetree of a code fragment

```
PAR1
  DISPLAY '1'
  DISPLAY '2'
PAR2
  DISPLAY '3'
  IF A > B THEN
    DISPLAY '4'
    DISPLAY '5'
  ELSE
    DISPLAY '6'
    DISPLAY '7'
  END-IF
```

Program Transformations
Transformation of legacy code

- Bottom-up, continue: to visit all statements

```
PAR1
  DISPLAY '1'
  DISPLAY '2'
PAR2
  DISPLAY '3'
  IF A > B THEN
    DISPLAY '4'
    DISPLAY '5'
  ELSE
    DISPLAY '6'
    DISPLAY '7'
  END-IF
```

Program Transformations
Transformation of legacy code

- Top-down, break: visit only nonnested statements

```
PAR1
  DISPLAY '1'
  DISPLAY '2'
PAR2
  DISPLAY '3'
  IF A > B THEN
    DISPLAY '4'
    DISPLAY '5'
  ELSE
    DISPLAY '6'
    DISPLAY '7'
  END-IF
```
Traversal functions allow visiting a specific type of node (sort/nterminal) and skip all intermediate nodes.

Example count all conditional in a Pico program:

```plaintext
module Count
imports
languages/pico/syntax/Pico-Syntax
basic/Integers
exports
  context-free syntax
  count(PROGRAM) -> Integer
  count((STATEMENT ";")*, Integer) -> Integer [traversal(accu, top-down, continue)]
  count(STATEMENT, Integer) -> Integer [traversal(accu, top-down, continue)]

hiddens
  variables
    "D"[0-9]* -> DECLS
    "S"[0-9]* -> (STATEMENT ";")*
    "E"[0-9]* -> EXP
    "I"[0-9]* -> Integer

equations
  [1] count(begin D S* end) = count(S*, 0)
  [2] count(if Exp then S*1 else S*2 fi, I) = I+1
```

Count all non-nested conditions in a Pico program:

```plaintext
module Count
imports
languages/pico/syntax/Pico-Syntax
basic/Integers
exports
  context-free syntax
  count(PROGRAM) -> Integer
  count((STATEMENT ";")*, Integer) -> Integer [traversal(accu, top-down, break)]
  count(STATEMENT, Integer) -> Integer [traversal(accu, top-down, break)]

hiddens
  variables
    "D"[0-9]* -> DECLS
    "S"[0-9]* -> (STATEMENT ";")*
    "E"[0-9]* -> EXP
    "I"[0-9]* -> Integer

equations
  [1] count(begin D S* end) = count(S*, 0)
  [2] count(if Exp then S*1 else S*2 fi, I) = I+1
```

Count all conditions in a Pico program using breaks:

```plaintext
module Count
imports
languages/pico/syntax/Pico-Syntax
basic/Integers
exports
  context-free syntax
  count(PROGRAM) -> Integer
  count((STATEMENT ";")*, Integer) -> Integer [traversal(accu, top-down, break)]
  count(STATEMENT, Integer) -> Integer [traversal(accu, top-down, break)]

hiddens
  variables
    "D"[0-9]* -> DECLS
    "S"[0-9]* -> (STATEMENT ";")*
    "E"[0-9]* -> EXP
    "I"[0-9]* -> Integer

equations
  [1] count(begin D S* end) = count(S*, 0)
  [2] count(S*1, I1) = I2,
  count(S*2, I2) = I3
  count(if Exp then S*1 else S*2 fi, I1 = I3+1
```