REFLECTIONS ON OPERATOR SUITES FOR REFACTORING AND EVOLUTION

Ralf Lämmel, VU & CWI, Amsterdam

joint work with Jan Heering, CWI

QUESTIONS

- What's the distance between refactoring and evolution?
- What are the ultimate properties of operator suites?
- How much language parametricity can be expected?
- What's a low-level API for evolution?
- How do we derive more complex operators?
- What are the limits of a suite-based approach?
- Is there a higher form of XP-like smells, say odours?
- What's there besides refactoring and evolution?

What's the distance between refactoring and evolution?

- Evolution also includes enhancement steps.
- ... and clean-up steps.
- Consequently, different properties are involved.

Also, our interest is in change tracking.

DEMO — THE RULE EVOLUTION KIT

- REK supports evolution of rule-based programs.
- REK is Prolog-based at object and meta-level.
- REK provides a suite of evolution operators.
- Examples in the interpreter domain.

`http://www.cs.vu.nl/rek/`

What are the ultimate properties of operator suites?

- Orthogonality
- Completeness
- Efficiency
- Self-checking
- Composability
- Self-disabling/-reinsuring
- Preservation properties
- Metrics-driven
- Type-driven
- Analysis-driven

PRESERVATION PROPERTIES

- Semantics-preserving or “$<$” and “$>$”
- Type-preserving/-increasing/-decreasing
- Productivity-... and Reachability-...
- Definedness-... and Usedness-...
- ...

Modulo properties:

- modulo renaming
- modulo projection
- modulo restriction
- ...

PROPERTIES OF OPERATORS FOR GRAMMAR EVOLUTION

Operator	Preservation	Inverse
Refactoring		
preserve	strict	**preserve**
fold	introducing	**unfold**
unfold	strict	**fold**
introduce	introducing	**eliminate**
eliminate	eliminating	**introduce**
rename	modulo renaming	**rename**
Construction		
generalise	increasing	**restrict**
include	increasing	**exclude**
resolve	resolving	**reject**
unify	essentially resolving	**separate**
Destruction		
restrict	decreasing	**generalise**
exclude	decreasing	**include**
reject	rejecting	**resolve**
separate	essentially rejecting	**unify**
delete	"zig-zag"	

How much language parametricity can be expected?

- Operators?
- Analyses?
- Composition principles?
- Properties?
- Suites?

VARIATIONS ON GENERIC EXTRACTION

Paradigm	*Focus*	*Abstraction*
OO programming	statements	method
OO programming	features	class
Functional programming	expression	function
Functional programming	type expression	datatype
Functional programming	functions	type class
Logic programming	literal	predicate
Syntax definition	EBNF phrase	nonterminal
Preprocessing	code fragment	macro
Document processing	content particle	element type
Cobol programming	sentences	paragraph
Cobol programming	sentences	subprogram
Cobol programming	data description entries	copy book
Cobol programming	data description entries	group field

PARAMETRICITY CHALLENGES

- Syntax abstraction
- Type system abstraction
- Data-flow abstraction
- Control-flow abstraction
- Hot spots for customisation
- Reusable APIs at various levels
- Hierarchical conceptualisation
- Generic testing
- Approved instantiation

WHAT'S A LOW-LEVEL API FOR EVOLUTION?

- Abstraction: named forms of abstraction including their application.
- Reference: entities that serve as references in scopes.
- Parameter: entities that can be specialised and generalised.
- Aggregate: entities that can aggregate parts or alternatives.
- Composite: aggregates that allow for grouping.

(Some bias to rule-based systems.)

ABSTRACTION

- *eliminate*: Safe removal of an unused abstraction.
- *introduce*: Insertion of a new abstraction.
- *duplicate*: Duplicate an abstraction.
- *unfold*: Replace an application of an abstraction by its body.
- *fold*: Reverse the effect of unfolding.
- *move*: Move an abstraction from one scope to the other.

REFERENCE

- *rename*: Rename the reference in the sense of consistent replacement.
- *redirect*: Replace a reference such that it refers elsewhere.

PARAMETER

- *specialise*: Instantiate constraints.
- *generalise*: Generalise constraints.

AGGREGATE

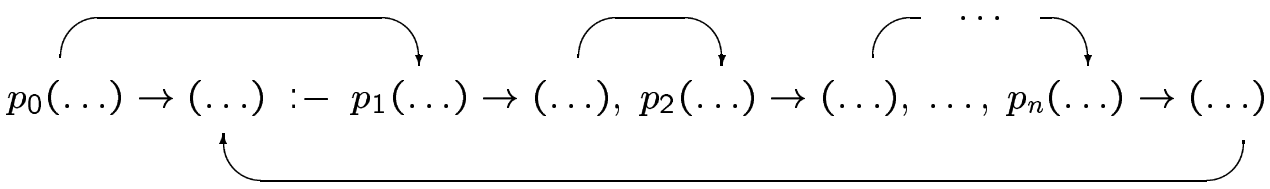
- *add*: Add a component to the aggregate.
- *rm*: Remove a component from the aggregate.
- *permute*: Permute the various components in the aggregate.

COMPOSITE

- *wrap*: Wrap a number of components as a single component.
- *unwrap*: Unwrap a component so that subcomponents are inlined.

How do we derive more complex operators?

An example: state passing



Steps:

- Add input positions
- Add output positions
- Unify positions
- Initialise thread

ISSUES REGARDING COMPOSITION

- Quantification or iteration
- Correctness of intermediate results
- Static feasibility
- Joint preconditions
- MPC-like side conditions

What are the limits of a suite-based approach?

There tend to be schematic transformations
that are not amenable to derivation by composition.

- Big-step to small-step style and vice versa
- Elimination of left recursion
- CPS conversion
- Monad introduction
- ...

Is there a higher form of XP-like smells, say odours?

- Distance
- Coupling
- Size
- Style
- Conciseness
- Extensibility
- Generality
- Readability
- . . .

What's there besides refactoring and evolution?

- Transformational program development
- Program derivation
- Stepwise enhancement
- Stepwise refinement
- Slicing in program comprehension
- Aspect weaving and mining