*NET.OBJECT*DAYS2000

Transformation Systems: Theory, Implementation, Survey

Ira D. Baxter, Ph.D.
Michael Mehlich, Dr.rer.nat.

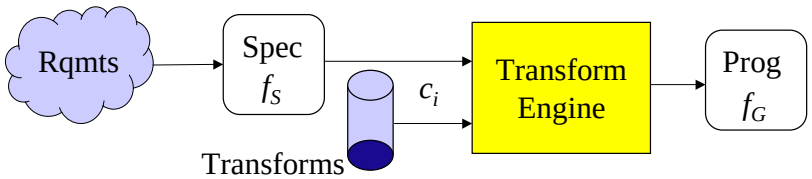
Semantic Designs, Inc.
12636 Research Blvd, Suite C214
Austin, Texas 78759
{idbaxter,mmehlich}@semdesigns.com

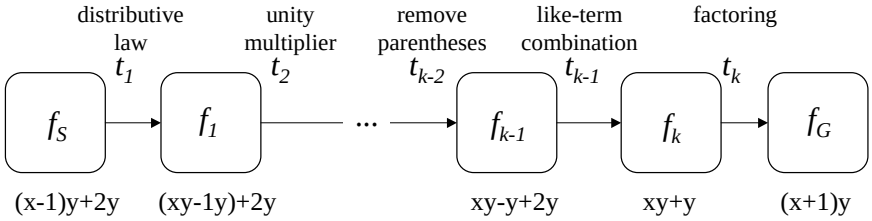
October 10, 2000

© Semantic Designs 2000

Transformation Systems

Stepwise Semiautomatic Conversion of Specs to Code





© Semantic Designs 2000

Tutorial Schedule

None!

It wouldn't be met anyway ...

Contents

Introduction
- Theory
- Implementation
- Some Transformation Systems
- Summary

Who are we?

- Dr. Ira D. Baxter
  - 30 years Software Engineering:
    Operating Systems, Compilers, Transform Systems, Reuse
  - Architect of Design Maintenance System
- Dr. Michael Mehlich
  - Background in formal methods (algebraic specifications)
  - Interested in making formal methods more practical
  - Lead Engineer on the Design Maintenance System
- Semantic Designs, Inc.
  - Formed to realize Design Maintenance System concept
  - 1996: $2M NIST ATP award to develop DMS
  - Transformational Modification/Synthesis consulting

Introduction

Why Transformation Systems?

- Conventional Software Lifecycle and its Flaws
- Ideal Solution: Fully Automatic Programming
- Concession to Reality: Semiautomatic Transformation Systems

- Applications of Transformation Systems

Conventional Software Lifecycle

- Requirements Analysis: *discovery of needs*
- (Requirements) Specification: *informal document defining product*
- Design: *determine structure of product*
- Code: *detailed software construction*
- Test/Debug: *find/fix bugs, verify operation*
- Maintain: *repeat above indefinitely*

An informal process producing complex products

Flaws of the Conventional Approach

- Misinterpreted requirements/specifications
- Difficult to validate steps
- Buggy implementations
- Implementation knowledge hidden
- Programmer quality reflected in product
- Poor traceability of requirements to code
- Design documents decay easily
- *Little hope for automation*

Automation of the Software Engineering Process

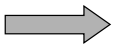
- Benefits:
  - *Faster*: Sooner to market
  - *Better*: Raising reliability, performance, quality of product
  - *Cheaper*: Lowering engineering costs
- Prerequisites:
  - Formal descriptions of desires and programs
  - Ability to write descriptions for real problems
  - Mechanizable domain and programming expertise

Refactoring as Code Quality Enhancement

"*… changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*"
Refactoring (Fowler99, Addison-Wesley)

Extract Method, p. 110

```
void printOwing(double amount) {
    printBanner();
    // print details
    System.out.println("Name:" = _name);
    System.out.println("Amount" + amount);
}
```

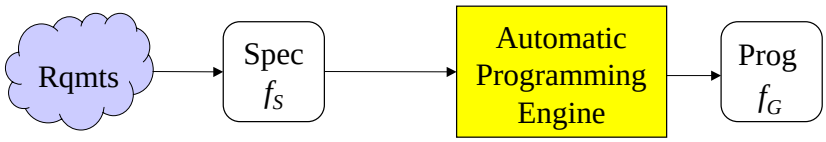


```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println("Name:" = _name);
    System.out.println("Amount" + amount);
}
```

Mechanization details follow later

Automatic Programming



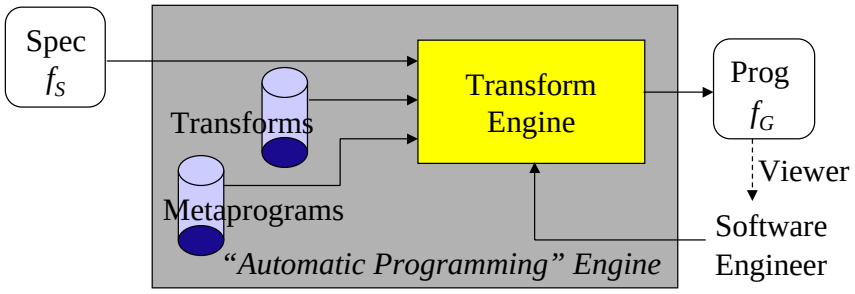
- Conventional Compilers
  - translate "simple" specification to machine code
    (usually implemented "ad hoc")
- Constructive Theorem Provers
  - extract program from proof
- Transformation Systems
  - stepwise transformation from specification to code

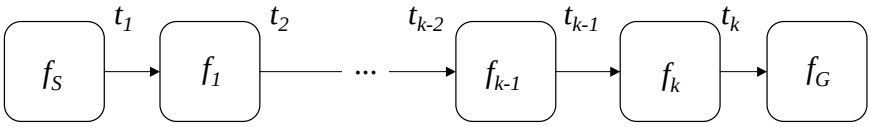
Fully Automatic Programming

- Problems:
  - Insufficent abstraction level for specifications
    that can be implemented automatically
    (Gödels incompleteness theorem)
  - Unsuitable notation to describe problem
    (who implements the AP engine for "my" problem domain?)
  - Limited control over performance of implementation
    (why does the $\in$-test on sets need linear time in the size of the set?)
    (why doesn't yacc produce JOVIAL code?)
- Solution:
  - Use highly configurable semi-automatic engine

Transformation Systems

Stepwise Semiautomatic Conversion of Specs to Progs





transformations preserve specification semantics!

A Simple Example

print **{1≤i≤50 ∧ prime(i): i}**; f_S

print block s←∅;
foreach i : i ∈ 1..50 do
if prime(i)
then s ← s ∪ {i};
return s
endblock; f_1

s← ∅;
do i = 1 to 50
if **prime(i)**
then **s ← s ∪ {i};**
foreach e ∈ s do print e; f_2

s← ∅;
do i = 1 to 50
if forall j: 2≤j<i ∧ (i mod j)≠0
then insert(s,i);
foreach e ∈ s do print e; f_3

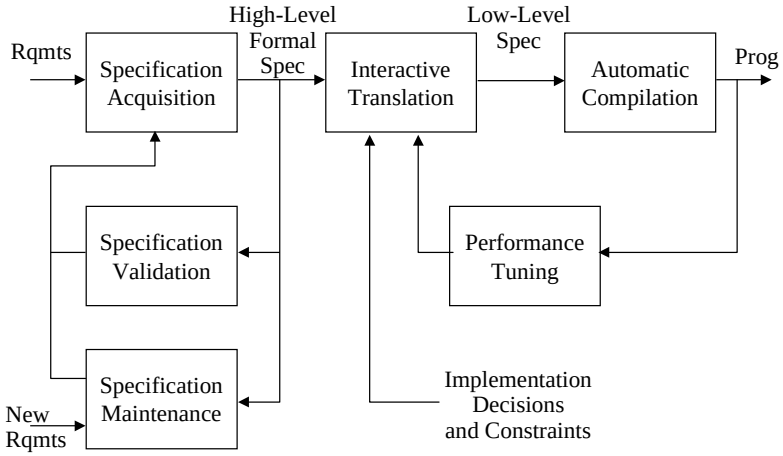
s← ∅;
do i = 1 to 50
if forall j: 2≤j<i ∧(i mod j)≠0
then **insert(s,i);**
do j=1 to size(s)
e← choose(s);s← s-e; print e f_4

s← ∅;
do i = 1 to 50
if forall j: 2≤j<i ∧ (i mod j)≠0
then **insert(s,i); e← choose(s);**
s← s-e; print e endif f_5

s← ∅;
do i = 1 to 50
if forall j: 2≤j<i ∧ (i mod j)≠0
then print i; f_6

do i = 1 to 50
if **forall j: 2≤j<i ∧ (i mod j)≠0**
then print i; f_G

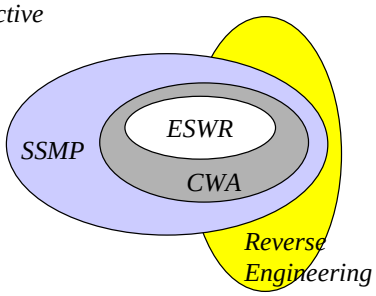
Transformational Lifecycle Model



[Balzer1983]

Transformation Systems – Incarnations

- Expert system with rewrite rules
  - *symbolic (mathematics) manipulation view*
  - Mathematica
- Compilation with advice
  - *automatic implementation perspective*
  - Predator
- Semantically sound, mechanically-supported program modification
  - *broader perspective*
  - DMS, Refine



Transformation Systems – Promises

- Agreed, formal meaning of specification
- "Provably" correct implementation steps
- Reliable application of implementation steps
- "Provably" correct products
- Accurate documentation by recording steps
- Specification traceability via retracing recorded steps
- Incremental acquisition/validation of implementation knowledge
- Capture and reuse of implementation knowledge
- At least partial automation of process
- Knowledge Base provides hints for manual steps

Introduction

- Why Transformation Systems?
  - Conventional Software Lifecycle and its Flaws
  - Ideal Solution: Fully Automatic Programming
  - Concession to Reality: Semiautomatic Transformation Systems

Applications of Transformation Systems

Applications of Transformation Systems

☺ Used to manipulate specifications

…Programs are just a special kind of specification!

Implement Specifications

Translate/Port Programs

Optimize Programs

Restructure Code

Reverse Engineering

Applications: *Forward Generation*

- Mathematical Formula Handling
  - Mathematica, Macsyma, Axiom, non-resolution theorem provers
- Translation of specifications to code
  - Compilers, YACC/LEX, SQL, Rational98: ROSE UML to C++ (ad hoc)
  - Neighbors84: Draco (domain specific synthesis), Smith98: Planware (schedulers)
  - CRPC/Rice98: ADIFOR (automatic differentiation of Fortran)
  - Simonyi98: Intentional Programming
- Optimization of existing code
  - Compilers, RDBMS query optimizers
  - Anders90: PartialEvaluation
  - Kennedy92: ParaScope (Interactive Fortran parallelization)
- Porting Software
  - Arango86: TMM (LISP dialects), EDS/Liu92: COBOL dialects

Applications: *Reverse Engineering*

- Specification Recovery
  - Sneed88: COBOLtoJSD
  - Baxter/Mehlich00: RE is Reverse Forward Engineering (theory)
  - Ward00: ASM370 to C
- Design Recovery
  - Waters90: Programmer's Apprentice (Lisp/ADA cliches to concepts)
  - Wills90: graph-parsing (slow!)
  - Quilici/Woods99: scalable constraint driven graph matching
  - Ning92: BAL (expert system: assembler to concepts)
- Robust Domain Analysis
  - Arango86:TMM (theory)
  - Baxter99: Clone detection

Applications: *Reengineering*

- Sweeping, complex editing changes
  - Feather90: EvolutionTransforms (functionality maintenance deltas)
  - Reasoning92: REFINE/C
- Reengineering
  - Newcomb94: FortranToDataflow (extracts changed-data dependent code)
  - Reasoning94: SoftwareRefinery, Reasoning99: COBOL Y2K
  - Brant98: RefactoringBrowser
  - SemanticDesigns00: DMS (Clone removal, #ifdef cleanup)
- Software Maintenance
  - Ward89: Maintainers' Assistant
  - Waters90: Programmer's Apprentice
  - Baxter92: DMS (theory)

Contents

- Introduction
  Theory
- Implementation
- Some Transformation Systems
- Summary

Theory

Concepts and Components

- Transforms and Transformations
- Specifications
- Semantics
- The meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

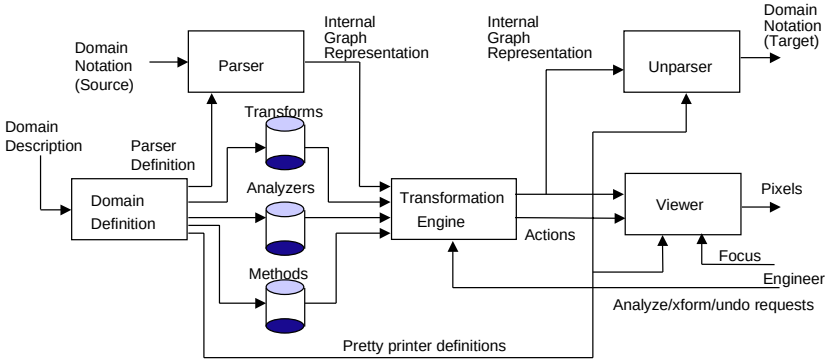
Transformation System *Concepts*

- Domain — *selects instance set of following concepts*
- Schema — *changed by transformation system*
- Semantics — *define meaning of schema*
- Performance Measure — *compute qualities of schemas*
- Performance Predicate — *compares qualities*
- Specification — *defines desired qualities (functional + performance)*
- State — *specification + cached inferences*
- Transform — *state modifiers*
- Locator/locale — *place/area in state*
- Binding — *mapping for variables in transform via locator*
- Transformation — *applied, bound transform*
- Property preservation — *notion of implementation*
- Metaprogram — *program which applies transforms*

Transformation System *Components*

- Formal *Source Language* (**with semantics**) — *specification*
- Formal *Target Language* (**with semantics**) — *program*
- (Semantic) definition of *Implementation*
- A *Representation* for source/target instances
- A set of *Transforms* applicable to source
- A mechanical *Tool* to apply transforms
- (optional) Means to *apply multiple transforms*
- (optional) Source *Parser*
- (optional) Target *PrettyPrinter*
- (optional) Means to *display* intermediate *states*
- (optional) Means to *undo* transformations
- (optional) Means for performing *analyses*
- (optional) Means for *augment*ing Transformation System databases

Transformation Engine Core



Transformation System *Components*: Example

Wang's Rules

simplify propositional calculus formulas

Given boolean formula over variables with operators:
and, or, not, implies
*determine if formula is a **tautology** or not*

true	*tautology*
or(a,not(a))	*tautology*
and(a,b)	*not tautology*
implies(not(or(a,b)),and(b,not(a)))	*status?*

(Useful as part of precondition evaluator!)

Wang's Rules in DMS

DMS rewrite rules are extracted from Spectrum-SD specifications, e.g. by interpreting equations from left to right as rewrite rules

```
WANG =
 { sort Boolean;

   true, false: Boolean;

   Boolean generated by true, false;

   not: Boolean → Boolean;
   and, or: Boolean × Boolean → Boolean associative commutative;
   implies, equivalent: Boolean × Boolean → Boolean;

   axioms
     –+3 simplification rules for not
       «not1» not(true) = false;
       «not2» not(false) = true;
       «not3» not(not(x)) = x;
     –+ definition of 'equivalent'
       «eq»   equivalent(x,y) = and(implies(x,y),implies(y,x));
     –+ definition of 'implies'
       «impl» implies(x,y) = or(not(x),y);
     –+ definition of 'and'
       «and» and(x,y) = not(or(not(x),not(y)));
     –+14 simplification rules for 'or'
       «or01» or(true,x) = true;
       «or02» or(false,x) = x;
       «or03» or(x,not(x)) = true;
       «or04» or(x,or(not(x),y)) = true;
       «or05» or(x,x) = x;
       «or06» or(x,or(x,y)) = or(x,y);
       «or07» or(not(or(x,y)),x) = or(x,not(y));
       «or08» or(not(or(x,y)),or(x,z)) = or(x,or(not(y),z));
       «or09» or(not(or(x,y)),not(x)) = not(x);
       «or10» or(not(or(x,y)),or(not(x),z)) = or(not(x),z);
       «or11» or(not(or(not(x),y)),x) = x;
       «or12» or(not(or(not(x),y)),or(x,z)) = or(x,z);
       «or13» or(not(or(not(x),y)),not(or(x,y))) = not(y);
       «or14» or(not(or(not(x),y)),or(not(or(x,y)),z)) = or(not(y),z);
   endaxioms;
 }
```


Wang's Rules … (DMS) in action

Proposition:

implies(not(or(a,b)),and(b,not(a)))

Transformation Steps:

Step implies(not(or(a,b)),**and(b,not(a))** ⇒ implies(not(or(a,b)),**not(or(not(not(a)),not(b)))**
by and(*x*,*y*) → not(or(not(*x*),not(*y*))) ***«and» rewrite***

Step implies(not(or(a,b)),not(or(**not(not(a))**,not(b)))) ⇒ implies(not(or(a,b)),not(or(**a**,not(b))))
by not(not(*x*)) ⇒ *x* ***«not3» rewrite***

Step **implies(not(or(a,b)),not(or(a,not(b)))) ⇒ or(not(not(or(a,b))),not(or(a,not(b))))**
by implies(*x*,*y*) → or(not(*x*),*y*) ***«impl» rewrite***

Step or(**not(not(or(a,b)))**,not(or(a,not(b)))) ⇒ or(**or(a,b)**,not(or(a,not(b))))
by not(not(*x*)) ⇒ *x* ***«not3» rewrite***

Step **or(or(a,b),not(or(a,not(b)))) ⇒ or(b,a)**
by or(not(or(not(*x*),*y*)),or(*x*,*z*)) ⇒ or(*x*,*z*) ***«or12» rewrite***

Normal form:

or(b,a)

Transformation System Components *for Wang's Rules in DMS*

- Formal Source Language — *e.g. Propositional terms*
- Formal Target Language — *e.g. Propositional terms*
- Definition of Implementation — *Normalization of propositions*
- Correctness of Implementation — *Rules preserve truth table*
- Representation for Language instances — *Abstract Syntax Trees*
- Tool to apply transforms — *DMS associative-commutative term rewrite engine*
- Means to apply multiple transforms — *e.g. Exhaustive bottom-up application*
- Source Language Parser — *e.g. Propositional term parser*
- Target Language Pretty Printer — *e.g. Propositional term printer*
- Means to display intermediate states — *Trace rewrite applications*
- (optional) Means to undo transformations
  Application of rewrite rule backwards at same position
  Generic undo mechanism on Abstract Syntax Trees
- Means for augmenting Transformation System databases
  Storing internal representation of Spectrum-SD specification and rewrite rules
  Using Spectrum-SD means having a parser for this formal source language and a transform to produce rewrite rules from an instance

Variations between Transformation Systems

- Kinds of specification languages(s)
  - simple, abstract, algebraic, domain-specific, graphical
  - expression, conventional, functional, wide-spectrum languages
- Types of transforms used
  - simple syntactic rewrites, complex rewrites, arbitrary procedures
- View of correctness of transforms
  - intuitively correct, provably correct, constructively correct
- Content and structure of transform library
  - small generative sets, domain-specific groups, huge catalog
- Degree of automation of transform application
  - manual, simplifying, goal-directed, mixed-initiative, planning
- Direction of application of transforms
  - forward synthesis, backward design recovery
- Method for recording/replaying history
  - none, derivation, metaprograms, design histories

Theory

- Concepts and Components
  Transforms and Transformations
- Specifications
- Semantics
- The meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

What is a *transform*?

A partial function from specs/programs to specs/programs

t: Spec → Spec

Compilers, YACC, VLSI synthesizers, program restructurers

Often represented as a *rewrite rule with pattern variables*:

eliminate-additive-identity: $x+0 \Rightarrow x$ *optimization*

t(Locator): Spec → Spec

implement-sum: *refinement*

```
sum(var,limit,vector) ⇒
        begin local s=0,var;
                do var=1 to limit;
                        s=s+vector(var); enddo
                return s
        end
```

Transforms for Mathematics

eliminate-additive-identity:	$x+0 \Rightarrow x$
cancellation-of-terms:	$x - x \Rightarrow 0$
distributive-law:	$x * (y+z) \Rightarrow x * y + x * z$
reverse-distributive-law:	$x * y + x * z \Rightarrow x * (y+z)$
integer-additive-arithmetic:	$1+1 \Rightarrow 2$ $1+2 \Rightarrow 3$ ... $n1+n2 \Rightarrow n3$ **if** $n3$=sum($n1$,$n2$)
an-equality-rule:	$x+n = x \Rightarrow$ false **if** $n \neq 0$

Transform *Applicability Conditions*

Domain of transform defined as applicability predicate
(transforms are partial functions!)

$x+0 \Rightarrow x$ **if** true

$x / x \Rightarrow 1$ **if** $x \neq 0$

bnf $\Rightarrow$ yacc(*bnf*) if isLALR(*bnf*)

var = *expr*;
call *proc*; $\Rightarrow$ call *proc*;
var = *expr*; **if** no-side-effects(*proc*) and *var* $\notin$ references(*proc*)

analyzers

Transform*ations*
Application of a Transform

Usually applied at a place (*locator*) in the program

t(Locator): Spec → Spec

apply *eliminate-additive-identity @ 2:6*

[token]

[line]

```
[1]   do j = 1 to 10                do j = 1 to 10
[2]         s=s+3+0+2       ⇒             s=s+3+2
[3]         p=p+0                         p=p+0
[4]   end                           end
```

[token 6]

... rewrite binds *x* to s+3 rewriting s+3+0 ⇨ s+3

Transforms as Procedures
the way they "really" work

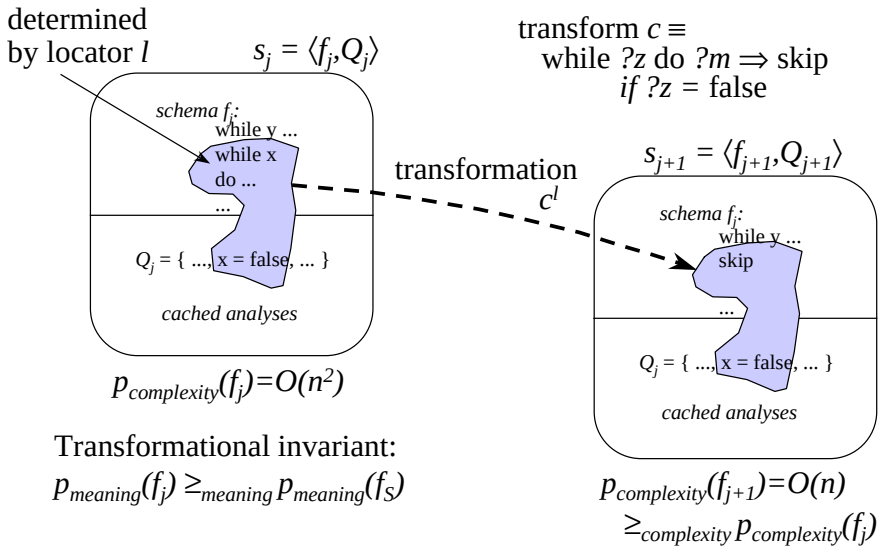
- Arbitrary procedure for each transform

  eliminate-additive-identity: $x+0 \Rightarrow x$

```
procedure eliminate-additive-identity(line,token)
  if      is-token(@(line,token),["+"])
    and  is-token(@(line,token)+1,["0"])
    and  is-last-token-of-additive-expression(@(line,token)-1)
    and  not is-token(@(line,token)+2,["*","/","%"])
  then
    delete-tokens([@(line,token)+1,@(line,token)])
  else fail
end procedure
```

- Rewrites are implemented as a *distinguished* procedure
  - implements pattern matching and replacement

Transformations Change States



Correctness-preserving Transforms

maintain properties of transformee

- Spec *s* is defined in some modelling language
  - *s* |= *p*, i.e. *s* satisfies *p*, iff all models of *s* satisfy *p*
  - Spec *s* has properties *P(s) = { p: s |= p }*
- Transform *t: S → S* is **correctness-preserving** iff
  - forall *s*, properties of *s* are preserved in *t(s), i.e. P(s) ⊆ P(t(s))*
  - *P(t(s))* is not inconsistent

Correctness-preserving is Not "Syntactic"

- Spec *s* is defined in some system of logic:
  - set of inference rules : $\vdash : p^* \rightarrow p$ *entailment calculus*
  - Spec *s* has entailable properties $E(s) = \{ p: s \vdash^* p \}$
- Definitions
  - $\vdash$ is **sound** iff $s \vdash^* p \Rightarrow s \models p$, *i.e.* $E(s) \subseteq P(s)$ $\vdash$ *is "useful"*
  - $\vdash$ is **complete** iff $s \models p \Rightarrow s \vdash^* p$, *i.e.* $P(s) \subseteq E(s)$ $\vdash$ *is "sufficient"*
- Theorem
  There is a sound and complete $\vdash$ for first order logic.
- Theorem (Gödel)
  There is no sound and complete $\vdash$ for systems of logic $\supseteq$ Peano arithmetic.

Correctness-preserving vs. Engineering Reality

Problem:

It is not cost-effective (and often really difficult) to develop a modelling language that allows to describe all and only the properties of interest.

Solution:

Distinguish properties into

- Properties that must be preserved
  e.g. fib(n) computes n-th Fibonacci number
- Properties that may improve but most not deteriorate
  e.g. fib(n) computes its result in O(T(n)) steps
  e.g. $\in O(n^2)$ *and* $\notin O(n)$ *to* $\in O(n)$
- Properties that are irrelevant
  e.g. fib(n-1) is not computed during computation of fib(n)

Classification of Transforms: *Effect*
Semantics ("Correctness") preserving

- equivalences: *preserve functionality without change or enhancement*
- optimizations: *preserve functionality, enhance some performance aspect*
- simplification: *equivalence, decreasing size of representation*
- partial evaluation: *simplification of procedure with fixed argument*
- unfold: *replace symbol with definition, essentially inlining* (*inverse:* fold)
- refinements: *preserve function, add detail, often change domain*
- theory morphisms: *algebraic refinements*
- data refinements: *refinements of abstract data types*
- finite differencing: *replacement of batch by incremental computation*
- jittering: *preparation for other correctness-preserving transforms*

Classification of Transforms: *Effect*
Noncorrectness-preserving

- component: *one of several required to be correctness-preserving*
- erroneous: *they happen, must be fixed*
- *almost*-correctness preserving: *common engineering*
- evolution transforms: *standard types of functionality deltas*
- functionality deltas: *intentional change of functionality*
- performance deltas: *intentional change of performance*
- support deltas: *intentional change to transformation system databases*

Classification of Transforms: *Other*

- Syntactic *written in source domain terms*
  - alternative: write against internal representation
- Procedural *coded in arbitrary language*
  - alternative: rewrite rule with left- and right- patterns
  - variation: metaprogram procedure
- Conditional *not always applicable*
  - alternative: unconditional
- Correctness (status) *reliability of transform*
  - axiomatic, manually or automatically verified, constructively correct
- Synthesized *dynamically constructed*
  - alternative: statically predefined

Theory

- Concepts and Components
- Transforms and Transformations
  Specifications
- Semantics
- The meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

Specifications: Stating Desires

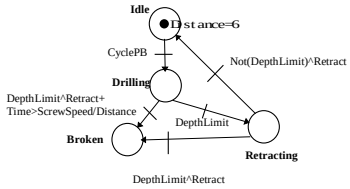
Specification: an acceptance predicate *P* over a desired artifact *s* is an acceptable program iff *P(s)* is true

Issues

- What needs to be expressed?
  - *Inputs/Outputs, Function, runtime costs, runtime environment, construction costs, etc..*
- How ambitious are our measurements?
  - *sorted, O(n^2), "low coupling", "usable", "cheap", "profitable", "beautiful"*
- What do we know how to express?
  - *Function, Time, Cost, Metrics*
- How to express?
  - *How formal? Which formalism? One formalism? A Combination?*

Practical Specifications

***Somehow** state what programs are acceptable*



- **Functional (f)** specifications:
  what a program computes

abstract computations	sum(i,10,quantity(i)*price(i))
pre-/post-conditions	post: exists i: prime(vector(i))
domain specific notations	*Colored Petri net, above right*

- **Performance (P)** specifications:
  how well it computes it

target language	targetlanguage=Pascal
size, complexity, structuredness,...	complexity=O(n ln n)

Both required: $P = f_0 \wedge P_{rest}$

Specifications
state more than just what is computed

A full specification (S):

M(S) = f: length(x) = n ⇒
$\forall\ i \in 1..n\text{-}1$: $f(x)[i] \le f(x)[i+1]$
} f_S *functionality*

and complexity(S) = O(n ln n)
and target-language(S) = Pascal
and souce-lines-of-code(S) ≤ 50
and no-gotos(S)
} P_{rest} *performance*

Types of Specifications

- Natural Language — *you can read it*
  structured(?) English
- Operational — *can execute it*
  procedural, functional
- Nonoperational — *can reason about it*
  predicate calculus, pre-/post- conditions
- Algebraic — *can structure it*
  abstract data types and their properties
- Performance — *say more than function*
  any of the above
- Unusual properties
  divergence, nontermination
- Domain Specific — *problem experts' notation*
- Real-time — *significant time constraints*
- Mixtures — *address larger problems*

Natural Language Specification

- The zero-th Fibonacci number is 1.
- The first Fibonacci number is 1.
- Each other Fibonacci number is the sum of the two largest smaller Fibonacci numbers.

Operational Specification

```
function Fibonacci(x)
  local result
  if x < 2 then
    result = 1
  else begin
    result = Fibonacci(x-1)
    result = result
             + Fibonacci(x-2)
  end
  return result
end
```

Procedural

```
function Fibonacci(x) =
  if x < 2 then
    1
  else
      Fibonacci(x-1)
    + Fibonacci(x-2)
```

Functional

essentially programming in an abstract language

Nonoperational Specifications

No hint on how to execute!

- I/O Pairs *too easy to forget a case, impossible to write all cases*

  f(2)=2, f(3)=3, f(4)=5, f(5)=7, ... *what's f(6)?*

- Relational I/O specification

  $R \subseteq I \times O$ *complex functions hard to specify*

  $R = \{\langle n,p\rangle \mid \forall\langle m,q\rangle \in R: n \bmod q \neq 0 \;\&$
  $\text{size}(\{q \mid \langle m,q\rangle \in R \;\&\; q< p\})=n-1\}$

- Hoare-style specifications *difficult because of quantifiers*

  pre: *condition* *abstraction* post: *condition*

  pre: *what was true before implementation executed*

  post: *what is true after implementation runs*

  pre: $i = n \wedge \forall j \in 1..n-1: \text{prime}(f(j)) \wedge \forall j \in 2..n-1: f(j-1)<f(j)$
  $\wedge \forall k. \neg\text{prime}(k) \vee k>f(n-1) \vee \exists j \in 1..n-1. k=f(j)$

  computenextprime

  post: $\forall j \in 1..n: \text{prime}(f(j)) \wedge \forall j \in 2..n: f(j-1)<f(j)$
  $\wedge \forall k: \neg\text{prime}(k) \vee k>f(n) \vee \exists j \in 1..n-1. k=f(j)$

Algebraic Specifications

- Specify primitive datatypes by **functional** *signatures* and *axioms*
- Specify more complex types by combining algebraic specifications *(an algebra over algebras!)*

Spectrum

```
SPECNAME =
 {  sort type1;
    ...
    sort typeN;

    function1: argtype1 → resulttype1;
    ...
    functionM: argtypeM → resulttypeM;

    axioms
       formula1 = formula2;
       formula3 ⇒ formula4 = formula5;
       ...
    endaxioms;
 }
```

signature

axioms

The Quintessential Stack

"Stackness" defined algebraically

```
STACK =
 {  sort Stack α;

    empty: Stack α;
    push: Stack α × α → Stack α;

    Stack α generated by empty, push;

    pop: Stack α → Stack α;
    top: Stack α → α;

    axioms
       pop(push(s, e)) = s;
       top(push(s, e)) = e;
        ¬ DEF(pop(empty));
       ¬ DEF(top(empty);
    endaxioms;
 }
```


"Incompleteness" is your friend ...

- Completely specified systems are often over-specified:
  "*Fibonacci*(-3) = 12"
- Over-specified systems harder to implement
  - *must work to implement useless detail*
  - *customer often doesn't really care*

⇒ Specify only what you really desire

… so is Nondeterminism

- Nondeterministic choices found in real life:
  - roll-dice() ∈ 1..6
  - dealcard(deck) ∈ deck
- Nondeterministic choices found in implementations:
  - choose(set) with set being implemented as an unordered list (deterministic choose more difficult to implement on unordered lists)

“Incompleteness” ≠ Nondeterminism

Specification:

- f returns 1 or 2 or 3

Interpretations:

- incomplete (but deterministic):
  - **either** f always returns 1 **or** f always returns 2 **or** f always returns 3
- nondeterministic (but complete):
  - f sometimes returns 1 **and** f sometimes returns 2 **and** f sometimes returns 3 and f always returns either 1 or 2 or 3

Some Formal Specification Languages

- LOTOS
- RAISE/RSL
- CASL
- OBJ
- Z
- VDM

Theory

- Concepts and Components
- Transforms and Transformations
- Specifications
  Semantics
- The Meaning of "Implementation"
- Control: Navigating the Implementation Space
- Design Capture

What is Semantics?

Communication is done using *Syntax*

Understanding each other comes from a common "agreement" about the *Interpretation* of the *Syntax*

Semantics is the
"agreement" about the *Interpretation* of *Syntax*

Semantics: *Syntax* → Set(*Interpretation*)

also called "meaning function"

Note the set of interpretations in the meaning function!

For its necessity consider e.g. the English word **band** and its interpretations:

- a thin strip of flexible material
- a group of players who perform as an ensemble

Types of Semantics

- Informal Semantics
  - *Everybody is pretty sure they understand*
- Operational Semantics
  - *Executable by well-defined interpreter*
- Transformational Semantics
  - *Map to notation with known semantics*
- Denotational Semantics
  - *Map to lambda calculus*
- Models and Algebraic Semantics
  - *Domain with known semantics ensures theorem truth*

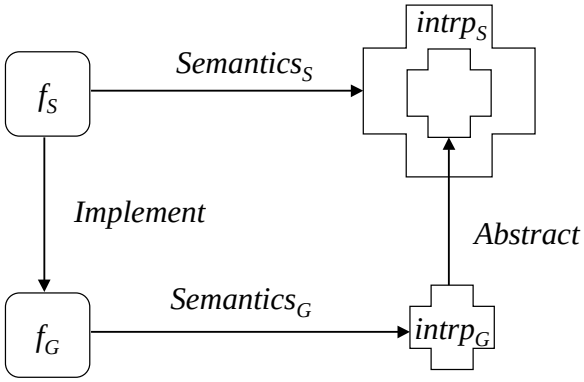
Theory

- Concepts and Components
- Transforms and Transformations
- Specifications
- Semantics

  The Meaning of “Implementation”
- Control: Navigating the Implementation Space
- Design Capture

Implementation - Semantics View

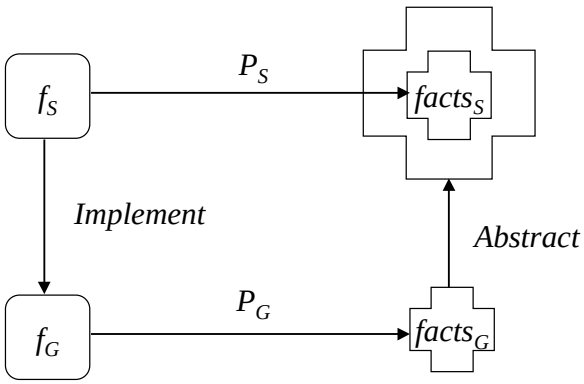
How do we know a program implements a specification?



$Semantics_S(f_S) \supseteq Abstract(Semantics_G(Implement(f_S)))$

Implementation - Properties View

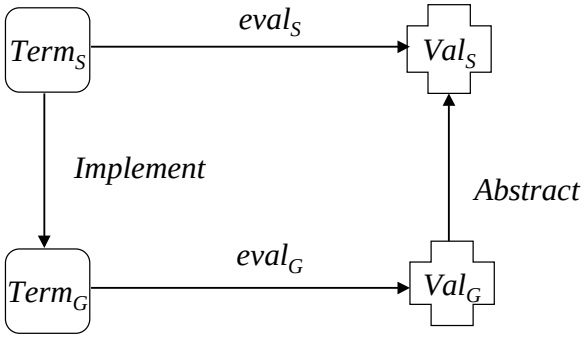
How do we know a program implements a specification?



$P_S(f_S) \subseteq Abstract(P_G(Implement(f_S)))$

Data Refinements

Refinements of functional expressions (Goldberg83)



$eval_S(Term_S) = Abstract(eval_G(Implement(Term_S)))$

Implement may be set-valued

Abstract may be a partial function

Refinements

- Refinements are transforms that augment the meaning of a specification
- They often incidentally introduce properties into the specification

- Classical Examples of Refinements:
  - Implement set by list, ordered list, hash table, ...
  - Unfold: replace a concept by an implementation
  - Finite differencing

Morphisms

- Theory Morphisms (Goguen, Maibaum, Blaine, Nord)
  maps within a formalism
- Institution Morphisms (Goguen, Sanella)
  maps across formalisms preserving $\models$
- Logics Morphisms (Meseguer)
  maps across formalisms preserving $\models$ *and* $\vdash^*$
- Logical System Morphisms (Meseguer)
  maps across formalisms preserving $\models$, $\vdash^*$ *and proof calculi*

Note: There are several proof calculi for e.g. first-order logic: Hilbert-style, natural deduction, sequent, ...

Theory Morphisms - Example I
Tiny Lisp Algebra

```
TINY_LISP =
 {  sort Atom; sort Seq;

    listify: Atom → Seq; atomize: Seq → Atom;
    nil: Seq; cons: Seq × Seq → Seq; list: Seq → Seq;
    car: Seq → Seq; cdr: Seq → Seq; cadr: Seq → Seq;

    Seq generated by nil, listify, cons;

    axioms
        atomize(listify(a)) = a;
        s = listify(a) ⇒ listify(atomize(s)) = s;
        car(cons(s1, s2)) = s1; cdr(cons(s1, s2)) = s2;
        cadr(s) = car(cdr(s));
        cons(s,nil) = list(s);
    endaxioms;
 }
```

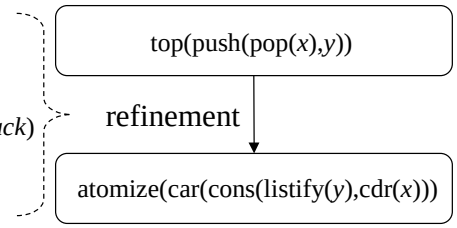
Theory Morphisms - Example II
A Morphism

- Definition:
  - The source sorts and operations are translated to target sorts and operations in a compatible way by a map *TM*
  - the source axioms are translated by *TM* to target theorems, i.e. target-axioms |= *TM*(source-axioms)
- Theory morphism for stacks to tiny lisp:

stack	⇒ seq
empty	⇒ nil
push(*stack*,*element*)	⇒ cons(listify(*element*),*stack*)
pop(*stack*)	⇒ cdr(*stack*)
top(*stack*)	⇒ atomize(car(*stack*))

Theory Morphisms - Example III
The Morphism as a Program Refinement

stack ⇒ seq
empty ⇒ nil
push(*stack*,*element*)
⇒ cons(listify(*element*),*stack*)
pop(*stack*) ⇒ cdr(*stack*)
top(*stack*) ⇒ atomize(car(*stack*))



The morphism also translates axioms:

top(push(*stack*,*element*)) = *element*

⇒

atomize(car(cons(listify(*element*),*stack*))) = *element*

Theory Morphisms - Example IV
Multiple Morphisms between Theories

- Another theory morphism for stacks to tiny lisp:

stack	⇒ seq
empty	⇒ cons(nil,nil)
push(*stack*,*element*)	⇒ cons(car(*stack*),cons(listify(*element*),cdr(*stack*)))
pop(*stack*)	⇒ cons(car(*stack*),cdr(cdr(*stack*)))
top(*stack*)	⇒ atomize(car(cdr(*stack*)))

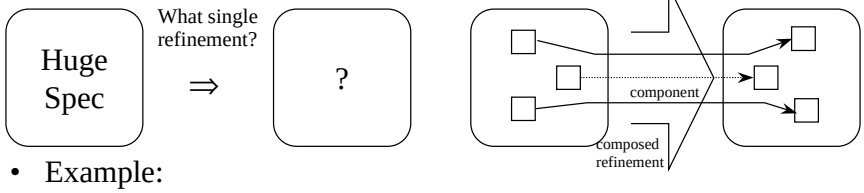
- Program Refinement for this Morphism

top(push(pop(*x*),*y*))
⇒
atomize(car(cdr(cons(car(cons(car(*x*),cdr(cdr(*x*)))),
cons(listify(*y*),cdr(cons(car(*x*),cdr(cdr(*x*)))))))))

Different morphisms result in different performance

The Consistent Refinement Problem

- Not always practical to *refine* specification as monolith
  - so must "refine" parts of spec "independently"
  - must have separate "refinements" for parts (*component transforms*)
  - *what guarantees that set of component transforms forms a refinement?*

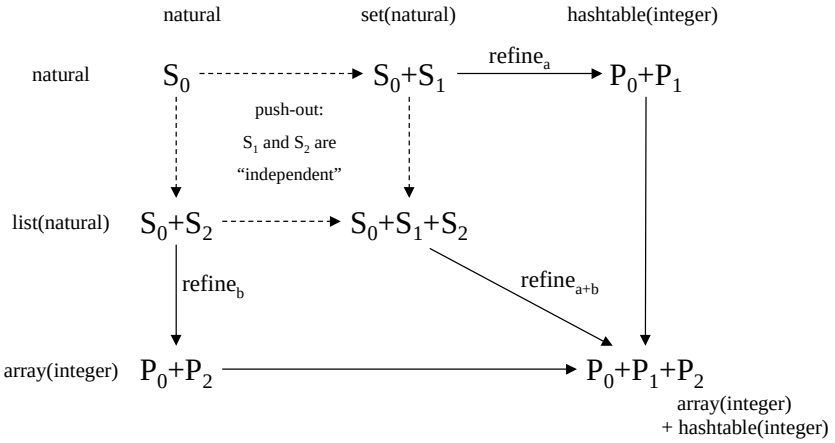


- Example:
  - Want to refine stack spec having **push** and **pop** actions
  - "Refine" **push** by adding new cell to linked list
  - "Refine" **pop** by decrementing pointer to array
  - Resulting program obviously doesn't work!
  - The pair **push** ⇒ linked list & **pop** ⇒ array is not a refinement

Solutions to Consistent Refinement Problem

- Only Monolithic Refinements *not usually practical*
- Predefined, consistent component transform packages
  many packages must be predefined
  - Package as set of transforms to be applied atomically
  - Apply set implicitly via metaprogram
- Synthesis time, dynamic consistent package assembly
  - synthesis time proof of consistency *need theorem prover*
  - consistency by construction *need decomposition scheme*
  - design rule checks to enforce consistency *weak check*
- Smart engineer

Consistent Refinements and Modules



Assumption: $P_0 = \text{refine}_a(S_0) = \text{refine}_b(S_0) = P_0$

Theory

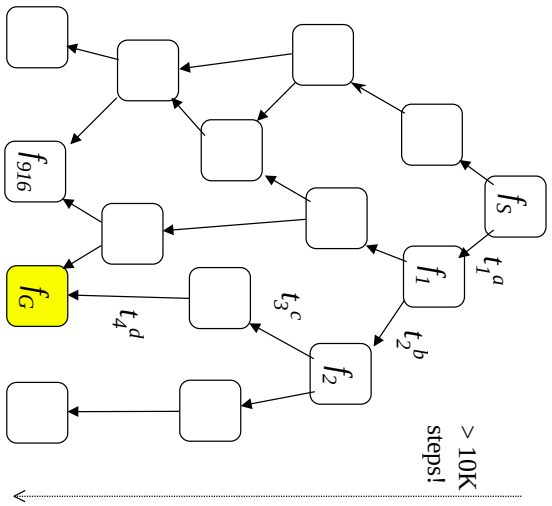
- Concepts and Components
- Transforms and Transformations
- Specifications
- Semantics
- The Meaning of "Implementation"
  Control: Navigating the Implementation Space
- Design Capture

The Control Problem
How to make implementation decisions?

- Transformational implementation of a specification goes through a huge number of intermediate states
- At each intermediate state many transforms are applicable at many locales leading to different possible implementations
- *How do we choose the "right" transformations?*

Control

Navigating the implementation space
What and Why?



Transformations on specification define
Huge Implementation Space

Navigation Issues

- **Efficacy** (Effectiveness) of transformation choice:
  - How to choose transformation approaching
    - Executability?
    - Desired Performance?

    How do we measure "approaching" effectiveness?
  - How to choose consistent transformations?

    (cf. consistent refinement problem)
- **Efficiency** of transformation choice:
  - How easy for software engineer to choose transformation?
  - How fast for tool to execute transformation?

Navigation Techniques

- Efficacy:
  - Manual selection of transformations
  - Heuristics
  - Repeated domain refinement/optimization
  - Performance-directed search
  - Metaprograms
  - Replay of previous transformation sequences
- Efficiency
  - Top down or bottom up exhaustive application of set of transforms
  - Heuristics
  - Searches
  - Confluent and terminating (Church-Rosser) system of transforms

    application order irrelevant to achieve goal

    e.g derived from equational theory by Knuth-Bendix completion algorithm
  - Metaprograms

Metaprograms

Control which transforms are applied to functionality to achieve desired level of performance

procedural:

name lower-abstraction-level
achieves abstractionlevel(*spec*)=low
performs **foreach** *locale* **in** *spec* **with** *spec@locale* = sum **do**
apply implement-sum@*locale*

declarative:

name implement-in-Pascal
achieves targetlanguage(*spec*)=Pascal
performs **achieve** abstractionlevel(*spec*)=low
then
call implement-using-Pascal-primitives**(***spec***)**

Metaprogram (Planner) Constructs

Methods: Procedures with well-defined performance postconditions

let *locale = locale-expr*	*computes and names locales*
apply *transform@locale*	*execute a transformation*
call methodname(*args*)	*invoke a method by name*
achieve *performance*	*nonprocedural invocation*
achieve *performance* **by** *action*	*preferred action for effect*
require *performance*	*verify effect of heuristic*
action1 **then** *action2*	*sequence actions*
action1 **and** *action2*	*perform both actions*
action1 **or** *action2*	*perform either action*
action1 **else** *action2*	*preference for actions*
plan { *label***:** *action* }, *partial-order*	*generalization of above*

Ensuring a Desired Level of Performance

O(n*ln(n)) for median computation

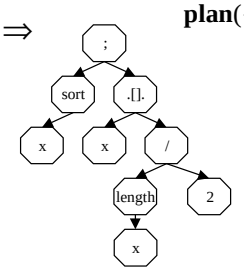
Refinements for Median:

$C_{median\text{-}u}$: median(*x*) ⇒

```
let sc = smallest(x); nc = #(x,sc) in
    while (nc < (length(x)/2))
        sc = next-larger(x,sc)
        nc += #(x,sc) endwhile;
    sc endlet
```

$C_{median\text{-}t}$: median(*x*) ⇒

```
sort(x);
x[(length(x)/2)]
```



A Planner Program for Median:

```
name      implement-median
achieves  complexity(spec) = O(n*ln(n))
performs
    imp = apply C_median-t@root(spec)
  then
    plan({1: achieve
                complexity(root(imp):1)
                  = O(n*ln(n)),
              2: achieve
                complexity(root(imp):2:2:1)
                  = O(n)}, ∅)
```

Metaprograms
are just transforms!

- Metaprograms satisfy criteria for transforms, i.e. they map schema to schema
- Metaprograms are complex transforms built from simple ones
- Metaprograms should be part of the transform library

Theory

- Concepts and Components
- Transforms and Transformations
- Specifications
- Semantics
- The Meaning of “Implementation”
- Control: Navigating the Implementation Space

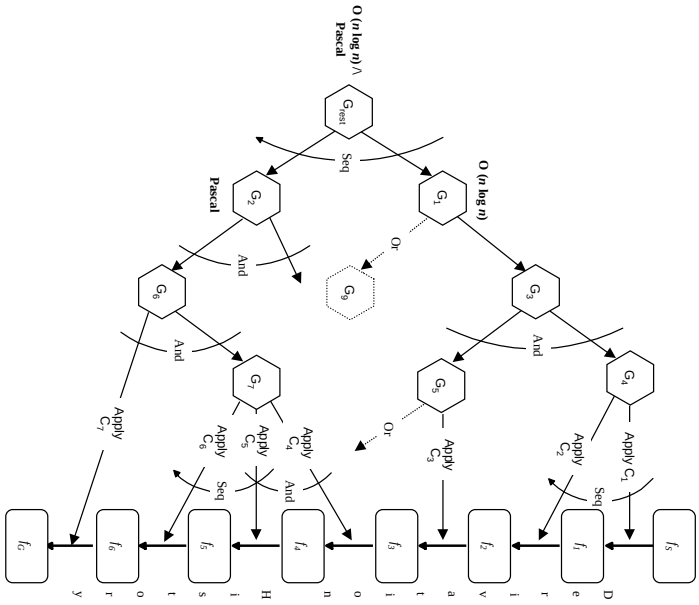
Design Capture

Capturing Designs

- Designs can be captured by recording
  - sequence of applied transforms
  - methods selecting the transforms and locales
- Recorded design
  - is formal explanation for implementation
  - can be (partially) replayed to implement changed specification by repeating and reorganizing steps

  ⇒enables design maintenance

Transformational Design



Contents

- Introduction
- Theory

  Implementation
- Some Transformation Systems
- Summary

Implementation of Transformation Systems

Constructing a Transformation System is Hard

- Representations of Specifications and Programs
- Implementation of Transforms
- Implementation of Planners
- ...

Constructing a Transformation System is Hard

Transformation System Components

- Formal *Source Language*
- Formal *Target Language*
- (Semantic) definition of *Implementation*
- A *Representation* for source/target instances
- A set of *Transforms* applicable to source
- A mechanical *Tool* to apply transforms
- Means to *apply multiple transforms*
- Source *Parser*
- Target *PrettyPrinter*
- Means to *display* intermediate *states*
- Means to *undo* transformations
- Means for performing *analyses*
- Means for *augment*ing Transformation System databases

- Each component
  - must be defined
  - must be implemented
- Transformation systems should also provide
  - large number of languages
  - large transform database
  - large metaprogram supply

*Let's look at **some(!)** details*

Implementation of Transformation Systems

- Constructing a Transformation System is Hard
  Representations of Specifications and Programs
- Implementation of Transforms
- Implementation of Planners
- ...

Specification and Program Representation

- Primitive Text (aka character string)
  standard representation for many languages
  - only useful for very simple transformations (sed, awk, …)
- Tokenized Text
  - pedagogical but still not very practical
- Abstract Syntax Trees (AST)
  - technology: parsing, term rewriting
- Resolved Abstract Syntax Trees (R-AST)
  - abstract syntax trees and symbol table forming a directed acyclic graph
  - technology: term graph rewriting
- Data- and Control-flow Representations
  - simplify analysis of procedural code for interesting relations
  - technology: graph rewriting
- Arbitrary Graph
  - can capture arbitrary relations

Tokenized Text

The Program as Text:

```
s = 0.0 ;
for ( i = 1 ; i < 5 ; i ++ )
  s =   i % 2 == 1
      ? 2 * s + 0
      : s + i * i ;
```

The Machine View:

```
s = 0.0 ; for ( i = 1
; i < 5 ; i ++ ) s =
i % 2 == 1 ? 2 * s +
0 : s + i * i ;
```

Rewrite Rule: x `+ 0` $\Rightarrow$ x

- Rule supplier (engineer) has to
  - describe restrictions for x
    "multiplicative expression"
  - describe restrictions for context
    "no factorial sign after 0"

  Hard to express over tokens!
- Machine has to
  - find locale to apply rewrite rule
  - find binding for x
  - check whether binding satisfies restrictions for x
  - check whether locale satisfies context restrictions

Primitive Text to Tokenized Text

- Technology
  - Lexical Specification (with "Regular Expressions") + Lexer Generator

    single "global" lexical spec often insufficent ⇒ need multiple lexical specs
- Example Lexical Specification (Fragment):

```
#skip "[\ \t\r\n]+"
#token 'for' "for"
#token identifier [STRING]  "[a-zA-Z][a-zA-Z0-9_]*"
  << (ConvertTokenStringToString (. ?:LiteralString) ? 0 0) >>
#token integer     [INTEGER] "[0-9]+"
  << ?:LiteralInteger = (ConvertDecTokenStringToNatural ? 0 0) >>
#token ':'   ":"          #token ';'    "\;"
#token '+'   "\+"         #token '*'    "\*"
#token '('   "\("         #token ')'    "\)"
```

ignorable characters

type of token value

regular expression

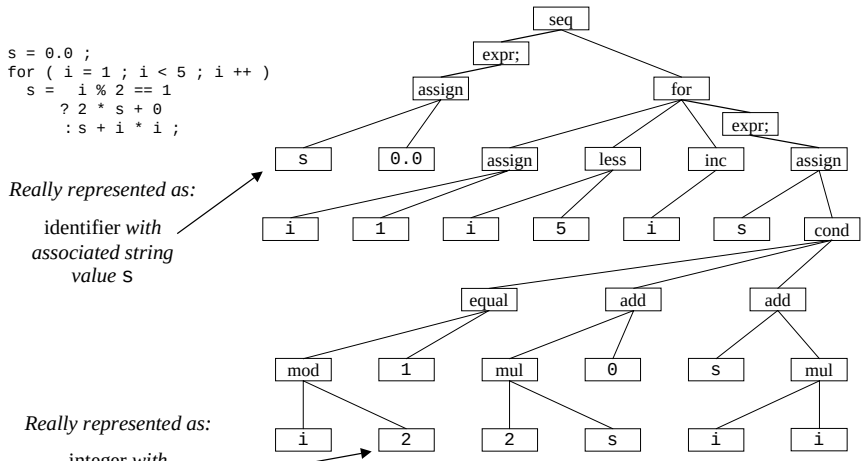
action to be performed when token is recognized

token name

- Note:
  - *Lexical Specification* is *Source Language* Instance
  - *Lexer Generator* is *Transform* producing *Target Language* Code

Abstract Syntax Trees

```
s = 0.0 ;
for ( i = 1 ; i < 5 ; i ++ )
  s =  i % 2 == 1
     ? 2 * s + 0
     : s + i * i ;
```



Really represented as:

identifier *with associated string value* `s`

Really represented as:

integer *with associated integer value* `2`

Tokenized Text to Abstract Syntax Trees

- Technology:
  - Context Free Grammar + Parser Generator
    LL(1), LR(1) often not sufficent ⇒ need more general parser
- Example Grammar (Fragment):

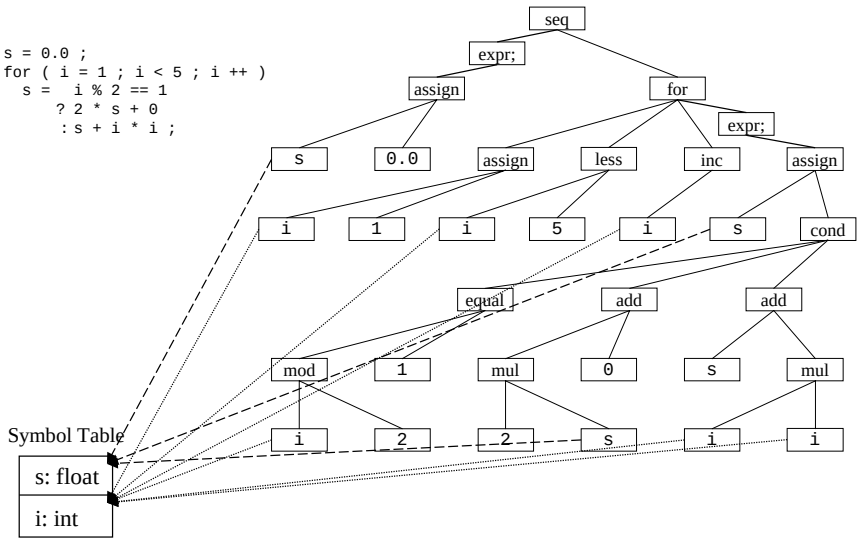
```
iteration
    = 'for' '(' expression ';' expression ';' expression ')'
      statement;
statement = expression ';';
expression = identifier '=' conditional;
conditional = expression '?' additive ':' additive;
additive = additive '+' multiplicative;
additive = multiplicative;
primary = identifier;
primary = '(' expression ')';
```

nonterminal
token
grammar production
terminal token produced by lexer

- Note:
  - *Grammar* is *Source Language* Instance
  - *Parser Generator* is *Transform* producing *Target Language* Code

(Resolved) Abstract Syntax Trees

```
s = 0.0 ;
for ( i = 1 ; i < 5 ; i ++ )
  s =  i % 2 == 1
     ? 2 * s + 0
     : s + i * i ;
```



Resolving Identifiers in Abstract Syntax Trees

- Technology:
  - Code Symbol Table — *core shared by different languages*
  - Tree Walk
    - Attribute Grammar (modifying symbol table) + Attribute Evaluator Generator
- Example Attribute Grammar (Fragment):

```
statement = expression ';';                    attribute (inherited)
  <<Resolver>>: { expression.symbol-table = statement.symbol-table; }
iteration = 'for' '(' expression ';' expression ';' expression ')'
                statement;
  <<Resolver>>: { expression[1].symbol-table = iteration.symbol-table;
                  expression[2].symbol-table = iteration.symbol-table; ... }
expression = identifier '=' conditional;
  <<Resolver>>: { Add(expression.symbol-table,identifier.name);
                  conditional.symbol-table = expression.symbol-table; }
primary = identifier;
   <<Resolver>>: { Map(identifier.,
                       Lookup(primary.symbol-table,identifier.name)); }
identifier;
   <<Resolver>>: { identifier.name = GetString(identifier.); }
                                               attribute (derived)
```

attribute evaluation pass

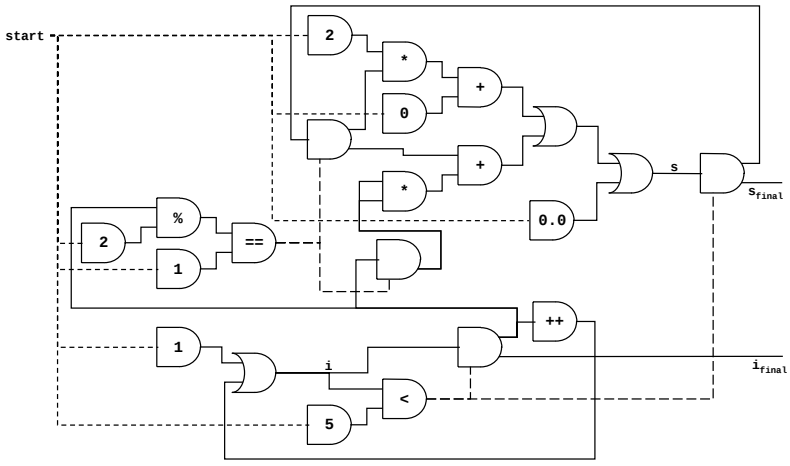
Advantages of Attribute Grammars

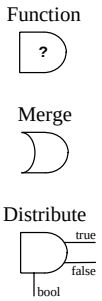
- they hide the concrete representation of abstract syntax trees
  sequences represented by lists or by spines?
- they are "declarative" and, thus, easier to write
  attribute evaluator generator ensures data dependencies being respected
- they can be used for other kinds of analyses

- Note:
  - *Attribute Grammar* is *Source Language* Instance
  - *Attribute Evaluator Generator* is *Transform* producing *Target Language* Code

Control and Data-flow Graphs

```
s = 0.0 ;
for ( i = 1 ; i < 5 ; i ++ )
  s =  i % 2 == 1
     ? 2 * s + 0
     : s + i * i ;
```





Resolved Abstract Syntax Trees to Control and Data-flow Graphs

- Technology:
  - Derive Analyzer from semantic description of language

    needs semantic description
  - Use only partial information about language semantics?

… and Back to Primitive Text

- Reason:
  - Show engineer transformed specification
  - Provide transformed specification to other tools
- Technology:
  - Hand coded
  - Pretty Printer Specification + Pretty Printer Generator *e.g. box based*
- Example:

```
iteration-statement
    = 'for' '(' expression ';' expression ';' expression ')' statement;
   <<PrettyPrinter>>: { V(H('for','(',expression,';',expression,';',
                           expression,')',I(statement)); }
additive = additive + multiplicative;
   <<PrettyPrinter>>: { V(I(expression[1]),H('+',factor)); }
```

vertical *indent* *horizontal*

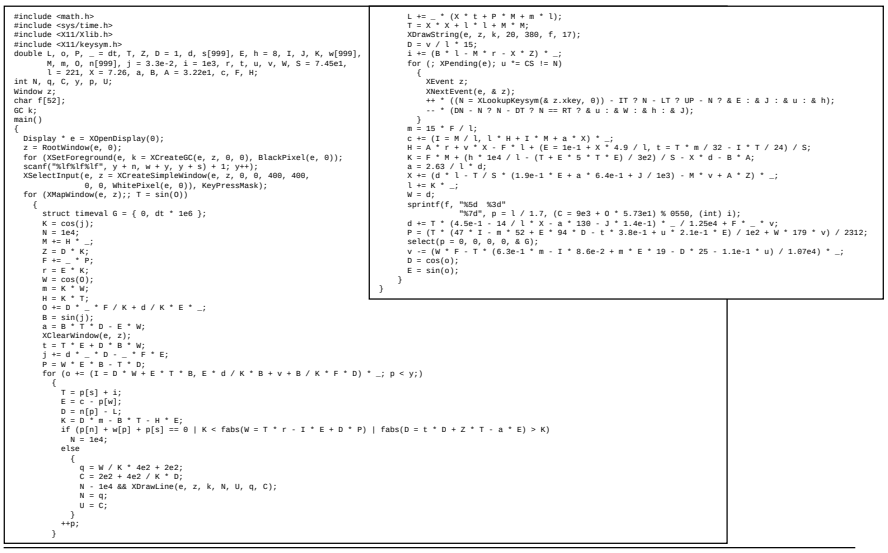
- Note:
  - *Pretty Printer Specification* is *Source Language* Instance
  - *Pretty Printer Generator* is *Transform* producing *Target Language* Code

Winner: Obfuscated "C" Contest

The Maintenance Programmer's Nightmare

```
#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
double L,o,P
,-=dt,T,Z,D=1,d,
s[999],E,h=.8,I,
J,K,w[999],M,m,O
,n[999],j=.53e-3,z=
7.5,i=.12,c,v,p,U;
a,B,A=32.2,c,F,H;
int N,q,C,y,p,u;
Window Z; char f[32]
;
GC k; main(){ Display*e=
XOpenDisplay(0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC(e,z,0,0),BlackPixel(e,0)
; scant("%lf%lf%lf",y+n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0, WhitePixel(e,0)),KeyPressMask); for(XMapWindow(e,z); ;T=sin(O){ struct timeval G={0,dt*1e6}
; XClearWindow(e,z),XDrawLine(e,z,k,N,u,Q,C); N=q; u=C; } +p; } L+=.* (X*t,+P*M+m*L); T=X*X+ l*l+M*M;
+T=B,F=d*.8+v+B/K*r*O)-p*y/K; T=p[s+1]; E=c-p[M]; D=n[p]-L; k=O*m+dT*.1*E; if(p [n]*w[ p]*p/s
s)+(.03*j-T*E+W*.2*K*.2*E+W*W*.3e-3*E+P*W*.3e-3*E*E+P*P*.3e-3*E*E*E+P*P*P*.3e-3*E*E*E*E)*dT
*+B,F=d*.8+v+B/K*r*O)-p*y/K; T=p[s+1]; E=c-p[M]; D=n[p]-L; k=O*m+dT*.1*E; if(p [n]*w[ p]*p/s
]-= 0|K <fabs(M*T-.1*E *O*P) |fabs(O=t *D+Z *T-a *E)>K)N=104; else{ q=W/K *4E2+202; C= 2E2+40/K
*D; N-.1E4&& XDrawLine(e,z,k,N ,U,Q,C); N=q; u=C; } +p; } L+=.* (X*t,+P*M+m*L); T=X*X+ l*l+M*M;
XDrawString(e,z,k ,20, 380,f,17); D=v/l*15; l+=(B *l-W*r -X*Z)*.; for(; XPending(e); u -=CS!=N){
XEvent z; XNextEvent(e ,&z);
++r(!N=XLookupKeysym
(&z.xKey,0)-IT?
K:IT ):(O=(W*B *B -1)*.
DM -W? N:DT ?M=
RT?0.: & W:0h!&J
); } m=15*F/l;
c+=(I=M/ l,l*H
+I*M+a*X)*.; H
=A*r+v*X-F*l+(
E=-.1*X*4.9/l,I
=(M/.2Z-I*W/l,
/S-X*d-BA;
a=2.63 /!d;
X+=( d*l-T/S
*(.19*E +a
*.64+J/183
)*.W* V +A*
Z/ ; l +=
K *.; W+=p;
sprintf(f,
"%5d %3d"
"%7d",p =l
/l.7; (G=9E3+
0*57.3%9559,(int)i); d+=I*(.45-14/l*
X+a*138.J*.14)*-/25e2+F*-*v; P=(T*(47
-1.=M* .5Z+E*94 *D-t*.38+u*.27*E) /1e2+W*
1/+W*.7*(.23*E+.69*a*1.69e+W*E*.19-D*25; .1t-(
W*F+.7*(.69*a*1.69e+W*E*.19-D*25; .1t-u
/l0?e7)-_
D=cos(O); E=sin(O); } }
```

Pretty Printing to *Uno*bfuscate



```
#include <math.h>
#include <sys/time.h>
#include <X11/Xlib.h>
#include <X11/keysym.h>
double L, o, P, _, dt, T, Z, D = 1, d, s[999], E, h = 8, I, J, K, w[999],
       M, m, O, n[999], j = 3.3e-2, i = 1e3, r, t, u, v, W, S = 7.45e1,
       l = 221, X = 7.26, a, B, A = 3.22e1, c, F, H;
int N, q, C, y, p, U;
Window z;
char f[52];
GC k;
main()
{
  Display *e = XOpenDisplay(0);
  z = RootWindow(e, 0);
  for (XSetForeground(e, k = XCreateGC(e, z, 0, 0), BlackPixel(e, 0));
  scanf("%lf%lf%lf", y + n, w + y, y + s) + 1; y++);
  XSelectInput(e, z = XCreateSimpleWindow(e, z, 0, 0, 400, 400,
                  0, 0, WhitePixel(e, 0)), KeyPressMask);
  for (XMapWindow(e, z);; T = sin(O))
    {
      struct timeval G = { 0, dt * 1e6 };
      K = cos(j);
      N = 1e4;
      M += H * _;
      Z = D * K;
      F += _ * P;
      r = E * K;
      W = cos(O);
      m = K * W;
      H = K * T;
      O += D * _ * F / K + d / K * E * _;
      B = sin(j);
      a = B * T * D - E * W;
      XClearWindow(e, z);
      t = T * E + D * B * W;
      j += d * _ * D - _ * F * E;
      P = W * E * B - T * D;
      for (o += (I = D * W + E * T * B, E * d / K * B + v + B / K * F * D) * _; p < y;)
        {
          T = p[s] + i;
          E = C - p[w];
          D = n[p] - L;
          K = D * m - B * T - H * E;
          if (p[n] + w[p] + p[s] == 0 | K < fabs(W = T * r - I * E + D * P) | fabs(D = t * D + Z * T - a * E) > K)
            N = 1e4;
          else
            {
              q = W / K * 4e2 + 2e2;
              C = 2e2 + 4e2 / K * D;
              N - 1e4 && XDrawLine(e, z, k, N, U, q, C);
              N = q;
              U = C;
            }
          ++p;
        }
      L += _ * (X * t + P * M + m * l);
      T = X * X + l * l + M * M;
      XDrawString(e, z, k, 20, 380, f, 17);
      D = v / l * 15;
      i += (B * l - M * r - X * Z) * _;
      for (; XPending(e); u != CS != N)
        {
          XEvent z;
          XNextEvent(e, & z);
          ++ * ((N = XLookupKeysym(& z.xkey, 0)) - IT ? N - LT ? UP - N ? & E : & J : & u : & h);
          -- * (DN - N ? N - DT ? N == RT ? & u : & W : & h : & J);
        }
      m = 15 * F / l;
      c += (I = M / l, l * H + I * M + a * X) * _;
      H = A * r + v * X - F * l + (E = 1e-1 + X * 4.9 / l, t = T * m / 32 - I * T / 24) / S;
      K = F * M + (h * 1e4 / l - (T + E * 5 * T * E) / 3e2) / S - X * d - B * A;
      a = 2.63 / l * d;
      X += (d * l - T / S * (1.9e-1 * E + a * 6.4e-1 + J / 1e3) - M * v + A * Z) * _;
      l += K * _;
      W = d;
      sprintf(f, "%5d  %3d"
                 "%7d", p = l / 1.7, (C = 9e3 + O * 5.73e1) % 0550, (int) i);
      d += T * (4.5e-1 - 14 / l * X - a * 130 - J * 1.4e-1) * _ / 1.25e4 + F * _ * v;
      P = (T * (47 * I - m * 82 + E * 94 * D - t * 3.8e-1 + u * 2.1e-1 * E) / 1e2 + W * 179 * v) / 2312;
      select(p = 0, 0, 0, 0, & G);
      v -= (W * F - T * (6.3e-1 * m - I * 8.6e-2 + m * E * 19 - D * 25 - 1.1e-1 * u) / 1.07e4) * _;
      D = cos(O);
      E = sin(O);
    }
}
```


Implementation of Transformation Systems

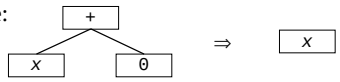
- Constructing a Transformation System is Hard
- Representations of Specifications and Programs
  Implementation of Transforms
- Implementation of Planners
- …

How to Implement Transforms

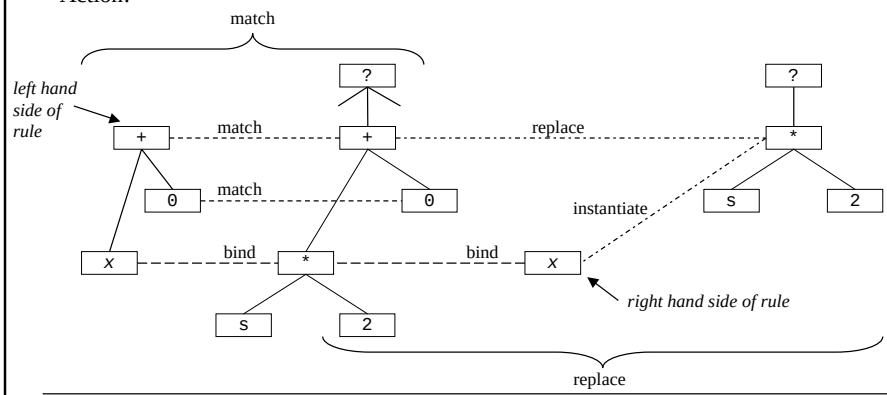
- Arbitrary Procedure
  - Full access to schema representation
  - Arbitrary navigation in and modification of schema representation
  - Fast execution
- Rewrite Rules
  - For chosen schema representation prebuild matching/replacing engine
  - Usually require precondition evaluator
  - Desired: Compilation of set of rewrite rules to efficient procedure

Rewrites on Abstract Syntax Trees

Rewrite Rule:

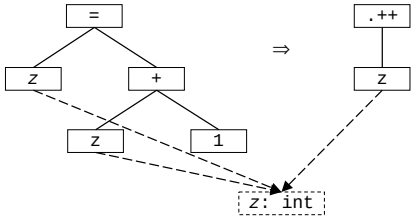


Action!



Rewrites on Resolved Abstract Syntax Trees

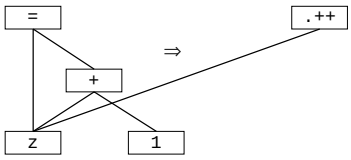
Rewrite Rule:



Technology :

- Use conditional to check whether `z` is declared as an integer
- Use graph nodes to represent symbol table entries and rewriting on directed acyclic graphs, i.e. term graph rewriting *what if* `z` *is not a declared symbol but updatable, e.g.* `*a[3]`*?*

Unresolved abstract syntax trees and rewrite rules can also be represented as directed acyclic graphs:



Matching Modulo Equality Theories

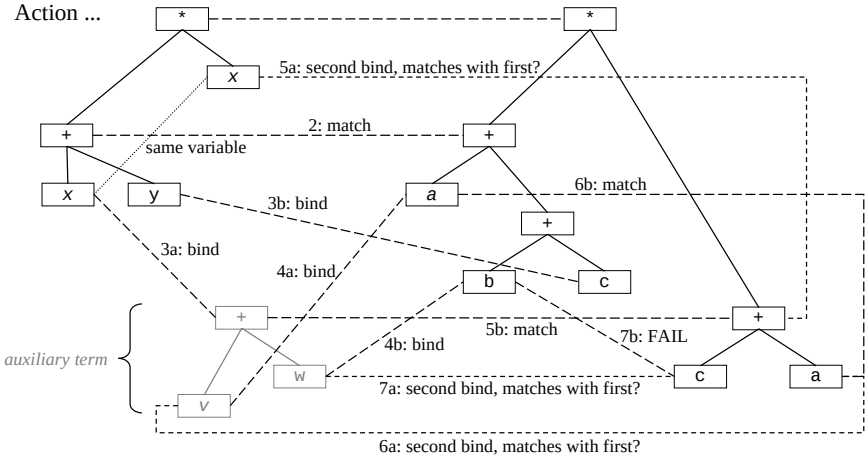
- + is commutative (but not associative):
  Add rule `0 + x ⇒ x`
- + is associative (but not commutative):
  Just add some rules?
  - Normalize "parentheses", e.g. `(x + y) + z ⇒ x + (y + z)`
  - Add **associative extended** rule `x + (0 + y) ⇒ x + y`

  Done?
  - Rule: `(x + y) * x ⇒ x * x + x * y`
  - Abstract Syntax Tree: `(a + (b + c)) * (a + b)`

  Oops, still cannot match, thus cannot rewrite!

  – Possible solutions:
  - Add further rule `(x + (y + z)) * (x + y) ⇒ …`
    what about `(a + (b + (c + d))) * (a + b + c)`*?*
  - Transform rule to set of rules implementing desired effect
    exhaustive rewriting is Turing-capable!
  - Build knowledge about associativity into matcher
- + is associative and commutative *same idea, different details*

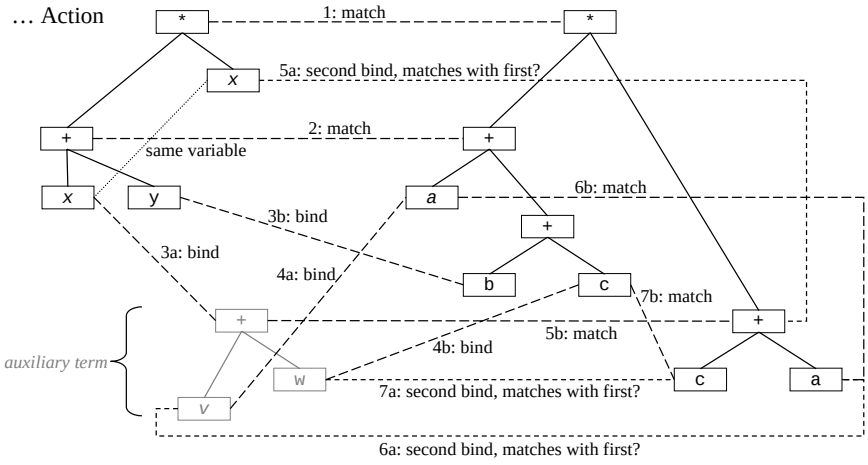
Associative-Commutative Matching I



A failed match! Remedies:

- Compute all alternatives in parallel
- Backtrack to last matching state allowing alternative if match for chosen alternative fails

Associative-Commutative Matching II



A successful match: $x \rightarrow$ `a + c`, $y \rightarrow$ `b`

Higher Order Program Transforms

- Consider:
  - Program Transform (Huet, Lang 1978):

```
f(x) = if c(x) then b else h(f(p(p(x))),f(p(x)))
     where  ∀x. ¬c(x) ∧ c(p(x)) ⇒ c(p(p(x)))
       and  ∀x. h(x,⊥) = ⊥                    fibonacci scheme
⇒
f(x) = g(x,b,b)
g(u,v,w) = if c(u) then v else g(p(u),h(w,v),v)
```

    Second order pattern variables in scheme: *c*, *h*, *p*

- Add Machinery for Higher-Order Matching and Rewriting
  - Have to fight for termination!

    only 2nd order, (relaxed) higher-order patterns, resource bounds, ...

Fibonacci Scheme Applied

Fibonacci specification:

```
f(n) = if n<=1 then 1 else f(n-1)+f(n-1-1)
```

Match results in binding: *assuming + being associative-commutative*

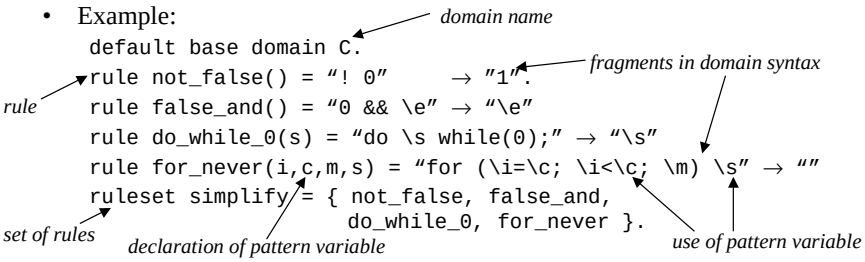
```
c ⇒ λn. n<=1
b ⇒ 1
h ⇒ λn,m.n+m
p ⇒ λn.n-1
```

Fibonacci implementation:

```
f(n) = g(n,1,1)
g(n,v,w) = if n<=1 then v else g(n-1,v+w,v)
```

Specifying Rewrite Rules

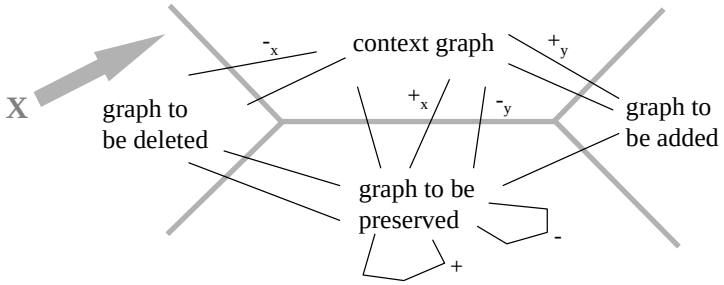
- Technology:
  - Specification in source language with pattern variables + Pattern Parser
- Example:



- Note:
  - *Rule Specification Language* Instance
    ⇒ Another language to define and implement.

Graph Rewrites

X notation defines graph rewrite with 4 parts and cross-arcs

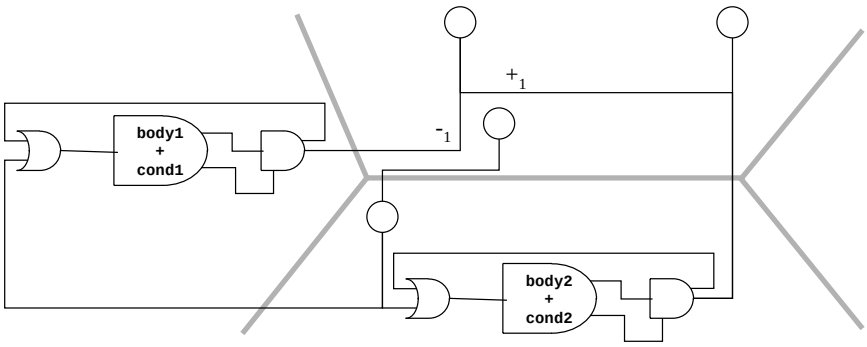


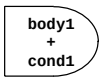
Given a graph transform as an **X**-rewrite, use by:

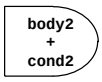
- Match to graph consisting of graph to be deleted and graph to be preserved
- Remove match to deleted graph, arcs tagged "-", "$-_z$"
- Insert added graph, arcs tagged "+", and corresponding arcs "$+_z$"

Dataflow Graph Rewrite

Eliminate Duplicate Loop



if 

=


Difficult to code on abstract syntax trees!

Precondition Evaluator Technology

- Determine if application condition for transform is true
- Technology:
  - Symbolic simplifier trying to prove truth by reducing a Boolean expression to true using rewrite rules

    only for simple conditions, but uses rewrite infrastructure
    completion may help finding proper set of rewrite rules
  - Proof system for mathematical logic

    hard to build good (semi-)automatic proof systems!
  - Arbitrary procedures

```
        variable = expression; call procedure;
   ⇒    call procedure; variable = expression;
        if  variable ∉ references(procedure)
        ∧   side-effect-free(procedure)
```

Precondition Evaluator
Symbolic Simplifier Example

A *Confluent* and *terminating* rewrite system for Boolean algebra where `and` and `xor` are associative and commutative (Hsiang 1982):

```
or(x,y)          ⇒ not(and(not(x),not(y)))
not(x)           ⇒ xor(x,true)
and(x,true)      ⇒ x
and(x,false)     ⇒ false
and(x,x)         ⇒ x
xor(x,false)     ⇒ x
xor(x,x)         ⇒ false
and(xor(x,y),z)  ⇒ xor(and(x,z),and(y,z))
```

A rewrite engine supporting associative-commutative theories can use this set of rewrite rules to *decide* equivalences of Boolean expressions.

Precondition Evaluator
Unproven Preconditions – What Now?

- Precondition evaluators are usually incomplete

⇒ Not all preconditions can be proven (or disproven)

- Remedies for unproven preconditions:
  - Get stronger precondition evaluator
  - Weaken precondition *only useful if precondition was too strong*
  - Try to prove converse *demonstrates only that transform does not apply*
  - Form conditional on precondition in code
    assumes precondition codable in target language
    - Code conditional
    - Apply transform to branch for satisfied precondition
  - Claim (by engineer) condition being true
  - Revise specification with stronger precondition to enable proof
    if specification is more generic than necessary

Implementation of Transformation Systems

- Constructing a Transformation System is Hard
- Representations of Specifications and Programs
- Implementation of Transforms
  Implementation of Planners
- ...

A Planner for Difference Equations

A generic planner to solver difference equations:

```
name implement-difference-equation
performs
             apply simplify-recursion
        then      apply eliminate-recursion
             else apply localize-function
   else           apply solve-difference-equation
        then      apply implement-using-fpu
             else apply reduce-floating-point-computations
```

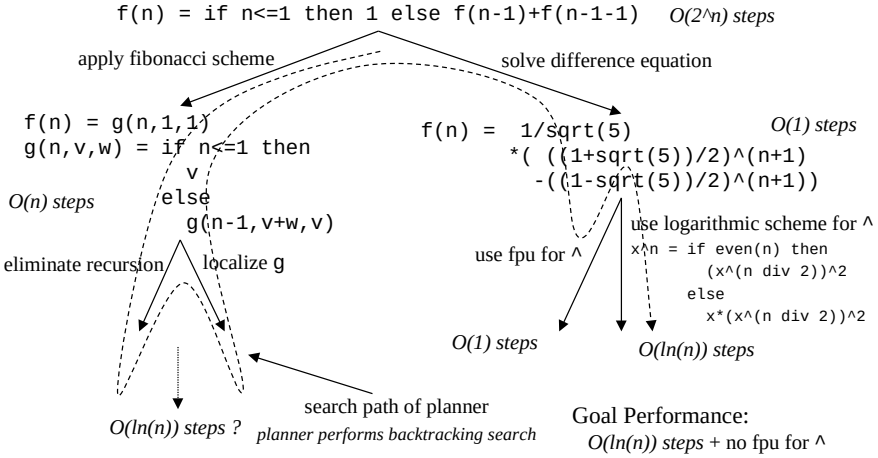
A planner for Fibonacci:

```
name implement-fibonacci
performs
   achieve O(log(n)) steps and no fpu for ^
   by implement-difference-equation
```

Note:

Transform Control Language Instance
⇒ Yet another language to define and implement.

An Implementation Space for Fibonacci



Planner

- Search for implementation by *execution mechanism*
  - Backtracking *need to store states at points with alternatives e.g. by capturing the design history!*
  - Parallel exploitation of alternatives *split in design history!*
- Planner programs are *execution guidance*
  - tactics / heuristics to explore space for implementation with desired properties

Implementation of Transformation Systems

- Constructing a Transformation System is Hard
- Representations of Specifications and Programs
- Implementation of Transforms
- Implementation of Planners

  ...

...

Transformation systems further need:

- Data Bases
  - for transforms, metaprograms
  - for design history
- Development Environment for semi-automatic application of transforms
  - View specifications and transforms
  - Apply transforms
  - Undo transforms
- …
- *Lots of Languages, Transforms, Metaprograms!*

Contents

- Introduction
- Theory
- Implementation

  Some Transformation Systems
- Summary

Some Transformation Systems

Compilers

- Intentional Programming (IP)
- Design Maintenance System (DMS)

Compilers as Transformation Systems

- Procedural or functional specification with language semantics
- Parses to intermediate representation
- Machine code target with architecture semantics
- “Prettyprints” machine code as “object file”
- Informal definition of implementation as “simulates”
- Syntax/semantic analysis of input specification
- Optimization and Code Generation as 2 sets of transforms
- Fixed, automatic sequence of transformation steps

Transformation System Concepts for Compilers

• Domain	*programming language*
• Schema	*often abstract syntax tree*
• Semantics	*informal in reference manual*
• Performance Measure	*object code size, speed*
• Performance Predicate	*nonexistent*
• Specification	*program code + command line switches*
• State	*schema + symbol table + data/control flow*
• Transform	*optimization rules, code generator*
• Locator/locale	*pointer to tree node*
• Binding	*implicit*
• Transformation	*implicit*
• Property preservation	*realization of language in machine code*
• Metaprogram	*sequence of compiler steps*

Transformation System Components for Compilers

- Formal Source Language — *programming language*
- Formal Target Language — *target machine code*
- Definition of Implementation — *resulting code simulations language computations*
- A Representation for source/target instances — *abstract syntax tree/binary*
- A set of Transforms — *code motion, inlining, ...*
- Tool to apply transforms — *optimizer + code generator*
- Means to apply multiple transforms — *main compiler*
- Specification Parser — *e.g. generated by yacc*
- Target PrettyPrinter — *relocatable binary output*
- Means to display intermediate states — *none*
- Means to undo transformations — *none*
- Means for performing analyses — *embedded procedures*
- Means for augmenting Transformation System databases — *none*

Some Transformation Systems

- Compilers
  Intentional Programming (IP)
- Design Maintenance System (DMS)

Intentional Programming (IP)

Programming with Intentions *Simonyi95*

- *Concept:* allow evolution of programming languages
  - Allows domain-specific language extensions
  - Gives programmer opportunity to fix bad code generation
  - A conventional programming language base ("C")
  - Programmer can extend language by adding arbitrary intentions
  - Intentions semantically defined/implemented by transformations
- Intentional programming environment
  - Programs stored as Abstract Syntax Trees
  - Intentions are programmer defined abstract syntax tree nodes
  - Abstract syntax tree direct manipulation editor
  - Transformation engine, with programmer defined:
    - Editing transforms: modify program permanently
    - Compiling transforms: transform intentions to conventional C constructs
  - Integrated development infrastructure (compiler, debuggers, …)

IP Direct Manipulation Editing

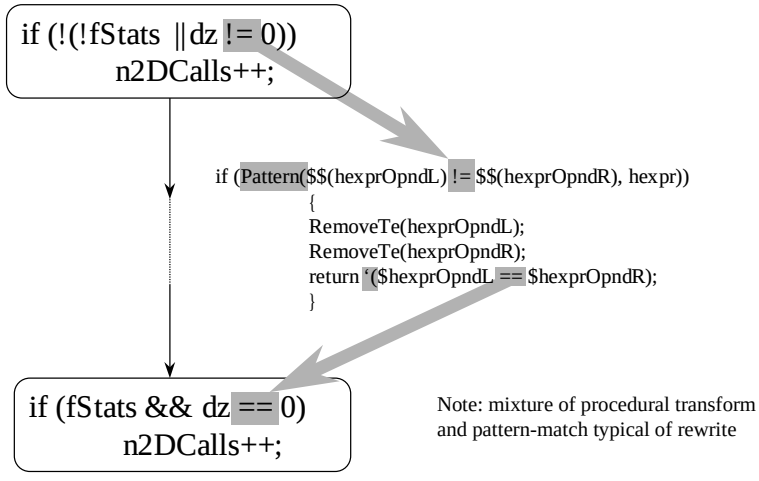
- Each AST node has screen display method
  - needed to allow arbitrary intentions
  - displays subtree in any desired format: notation, fonts, colors, ...
  - establishes map of screen positions to abstract syntax tree nodes
- WYSIWYG editing of internal IP abstract syntax tree
  - user moves cursor to place of interest and selects locale
    tree, left subtree, right subtree, content, root of tree, ...
  - effect on abstract syntax tree determined by selection and user typing
  - user can invoke editing transforms on locale
  - very convenient variable renaming due to resolved abstract syntax tree

IP "Enzymes" (Transforms)

are IP procedures that operate on abstract syntax trees

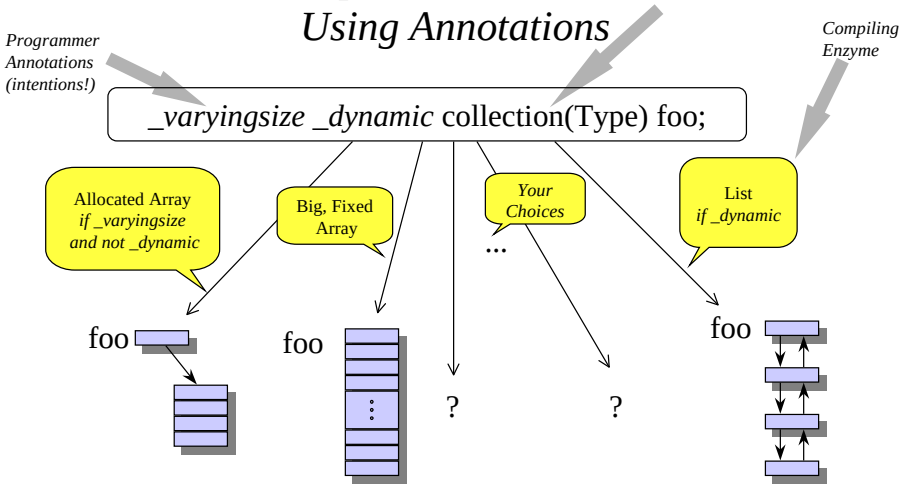
- Enzymes perform arbitrary changes to tree
- Many enzymes are built into base IP system
- Enzymes are IP programs, they can be entered via IP editor
  - easy addition of new compiling enzymes
  - easy addition of new types of intentions
- Metaprograms are IP transforms that call others
- Representation is resolved abstract syntax tree with annotations
  - identifier "use" nodes point to identifier "declaration" nodes
  - annotations for marking abstract syntax tree with hints: *_dynamic*
  - annotations for caching analysis results

An IP Editing Enzyme



Note: mixture of procedural transform and pattern-match typical of rewrite

Efficient Implementation of **Intention**
Using Annotations



Data structure implementations

Transformation System Concepts in IP

• Domain	*C augmented by extensions, other*
• Schema	*resolved abstract syntax tree*
• Semantics	*informal*
• Performance Measure	*none*
• Performance Predicate	*informal*
• Specification	*C code decorated with intentions, other*
• State	*resolved abstract syntax tree*
• Transform	*IP procedures with tree matching/substitution*
• Locator/locale	*pointer to tree node interpreted as node or subtree*
• Binding	*explicit substitution from matching*
• Transformation	*explicit data structure holding <transform,locale>*
• Property preservation	*informal*
• Metaprogram	*programmer annotations, intention reduction order*

Transformation System Components in IP

- Formal Source Language — *C + intentions, other*
- Formal Target Language — *C + intentions, object code, other*
- Definition of Implementation — *preservation of functionality*
- A Representation for source/target instances — *resolved abstract syntax tree*
- A set of Transforms — *coded as procedures inside IP*
- Tool to apply transforms — *procedure call, IP reduction engine*
- Means to apply multiple transforms — *IP defined tree walk*
- Specification Parser — *direct manipulation editor*
- Target PrettyPrinter — *pretty printer per abstract syntax tree node*
- Means to display intermediate states — *direct manipulation editor*
- Means to undo transformations — *fine-grain version control*
- Means for performing analyses — *hand-coded procedures*
- Means for augmenting Transformation System databases — *IP libraries*

Some Transformation Systems

- Compilers
- Intentional Programming (IP)
  Design Maintenance System (DMS)

Design Maintenance System I

- Goal: *Software Maintenance* and Synthesis
  - Software Construction important
  - Software Evolution *more* important
- Domain Oriented
  - Different notations and semantics
  - Different levels of abstraction
- Large Scale
  - Ambition: 10 million source lines
  - Runs on commercial parallel workstations
- Design Capture and Revision
  - Required to make changes on Scale

NIST ATP 1995

Design Maintenance System II

- Software maintenance costs dominate
- Key problem: Absence of system design
- Make *design* **the** *product of development*
- Capture complete design
- *Can use tools to aid design revision*

⇒Faster, Better, Cheaper

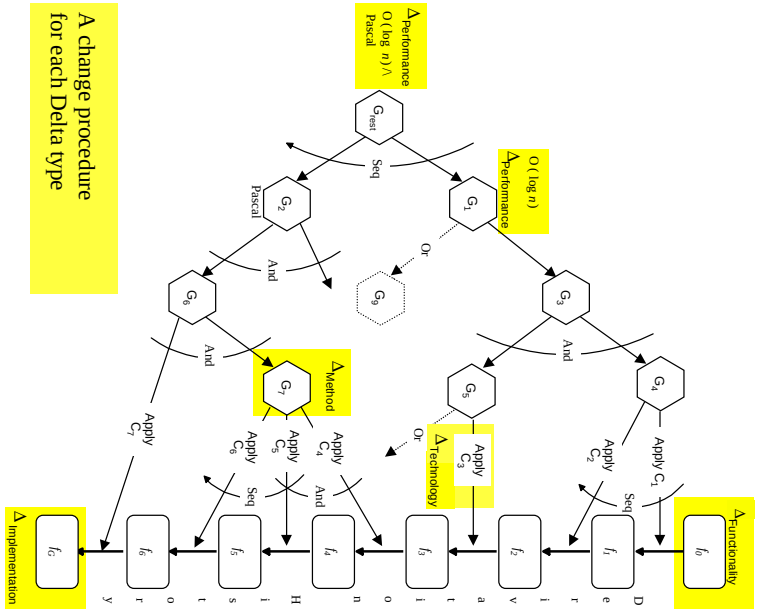
Baxter1992

Maintenance Deltas

Maintenance Deltas state formally how aspects of the software development are changed:

- Specification Deltas *changes to problem definition*
  - $\Delta_{functionality}$: `Lucas-numbers ↘ Fibonacci-numbers`
  - $\Delta_{performance}$: `O(n) steps ↘ O(ln(n)) steps`
- Support Deltas *changes to transform system infrastructure*
  - $\Delta_{technology}$: $x * 0 \Rightarrow x \searrow x + 0 \Rightarrow x$
  - $\Delta_{definition}$: `sloc = #CR-LF ↘ sloc = #semicolon`

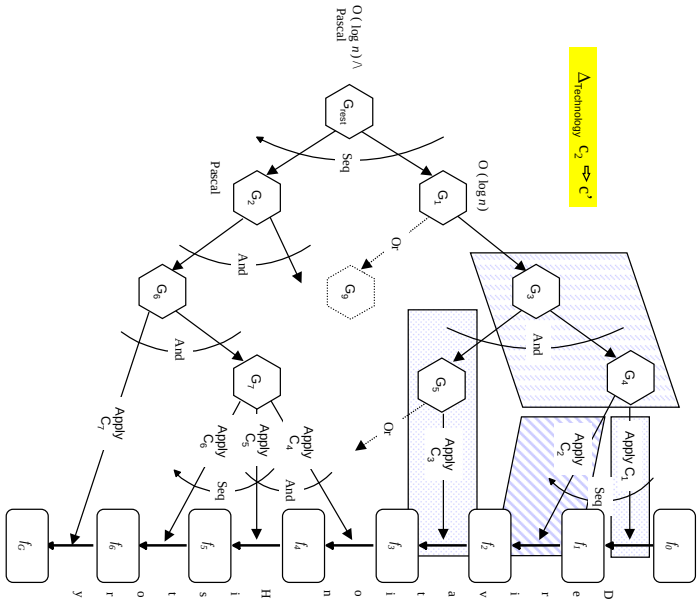
Initial Delta Impact



Design Modification I

`capture` explicit Δ
`modify` design aspect directly affected **`by`** Δ
`repeat`
  `mark` actions subsidiary to action(s) affected
  `delete` marked actions
  `remove` marked transformations
    (possibly affecting more actions)
`end`
`run` transform engine to **repair** partial history

Mark/Delete Affected Actions

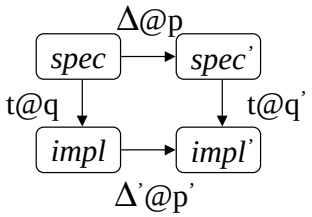


Design Modification II

- Efficient Design Modification ...
  - Preserve transformations not affected by change
- … by Rearranging the Derivation History
  - preserve transforms whenever possible
  - delay transforms when preserving is not possible

Preserving a Transformation
by pushing down the change

Push down change Δ@p
by finding Δ'@p' and q':

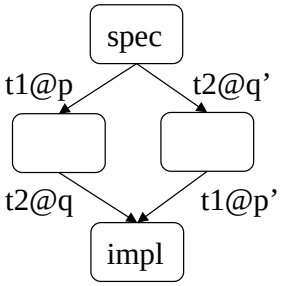


Δ does **not** affect t@q if the diagram commutes, i.e. if
Δ'@p'(t@q(*spec*))=t@q'(Δ@p(*spec*))

Often only the locale changes

Delaying a Transformation

Transformations often commute:

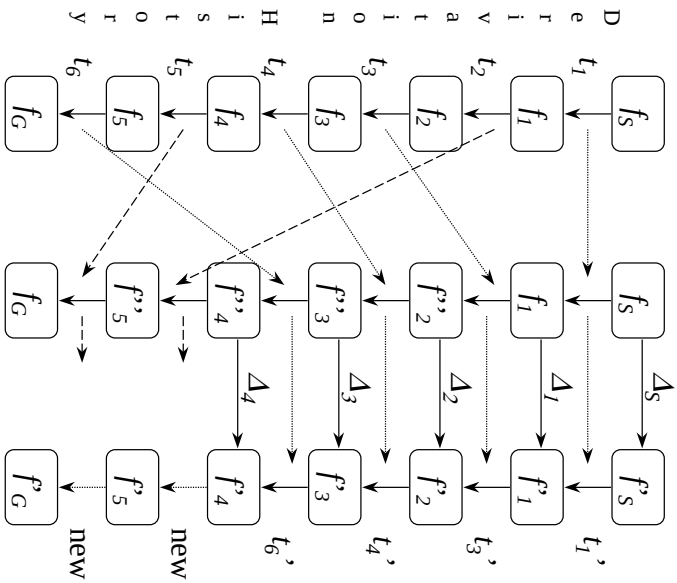


If Δ affects t1@p, delay t1@p
by commuting it with following transformations

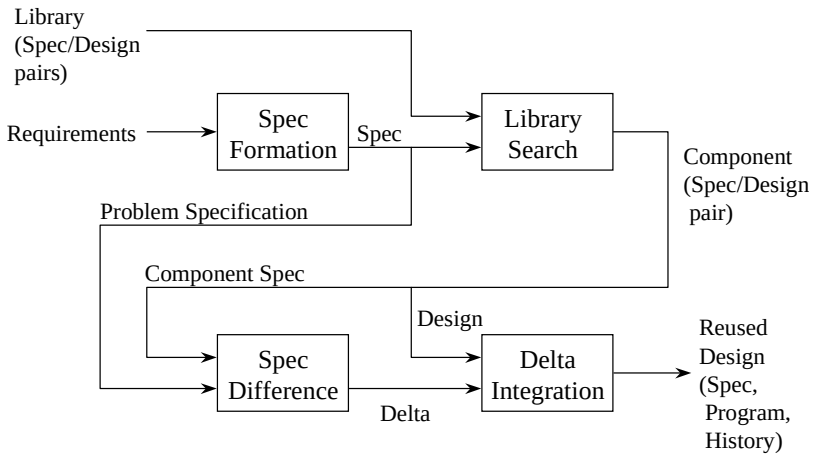
We win if Δ does not affect e.g. t2@q'

Derivation Revision

Abstract view

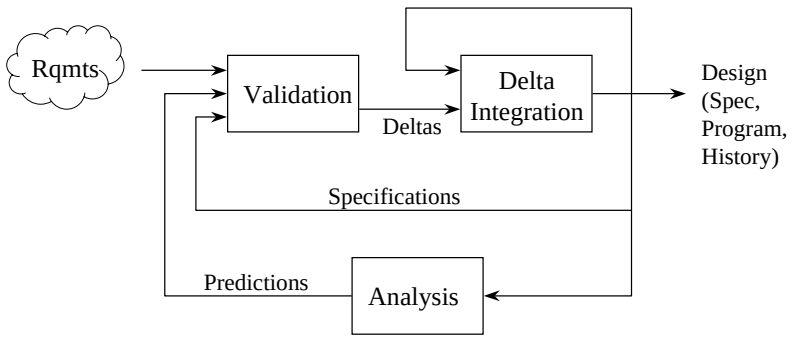


Component Reuse by Design Modification



Design Maintenance explains how component modification happens

Design Phase: Early Continuous Evolution



Design and Maintenance Phases are not different!

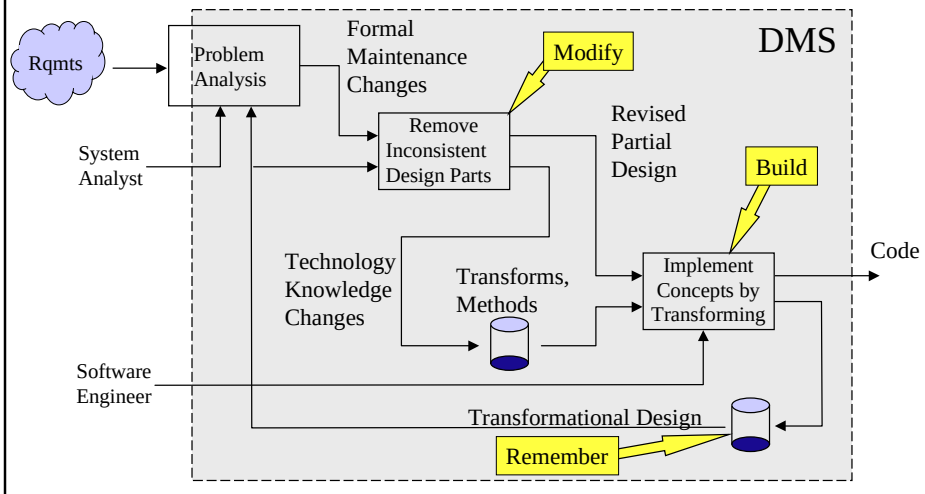
Transformational Maintenance – Summary

- Code Maintenance is the major software cost
- Can capture formal design of software *transformationally*
- Can modify formal design to get new software
- Rationalize maintenance, lower cost, raise quality
- Reuse enabled by design modification of component
- Design phase: just early continous software evolution

☞ *Design* should be **the** *product of software process*

... recover design of legacy software to apply?

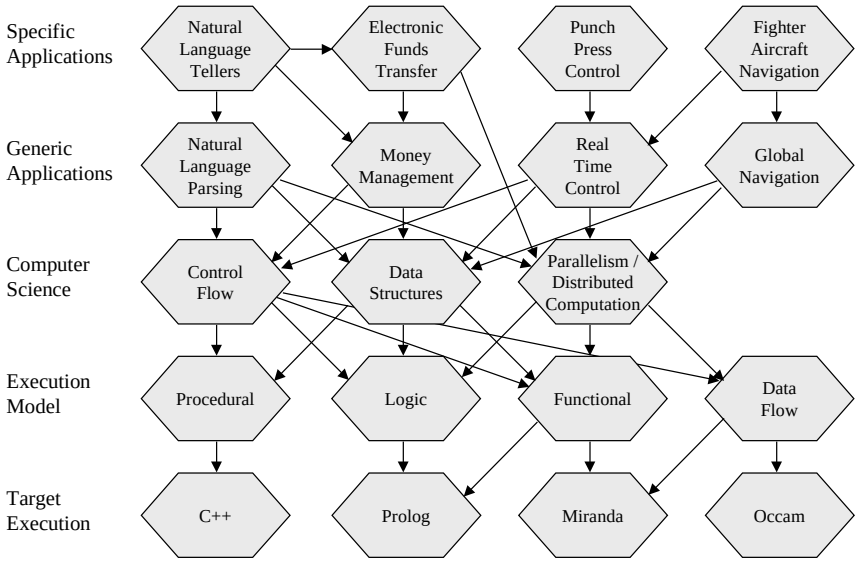
Design Maintenance System – Concept



DMS Domain Parts

- External Form — *what you can say: string or graphical*
- Internal Form — *how DMS stores it*
- Semantics — *what the Internal Form means*
- Parser — *how to convert external form to internal form*
- Unparser — *how to display the Internal Form*
- Transformations
  - Optimizations — *how to optimize in the domain*
  - Refinements — *how to transform Internal Form to another Internal Form*
  - Jittering transformations — *how to make other transformations applicable*
- Analyzers — *how to analyze in the domain*

A Domain Interconnection Network



Transformation System Concepts in DMS

- Domain — *arbitrary string or graph language*
- Schema — *hypergraph with domain-imposed topology*
- Semantics — *informal to algebraic/denotational*
- Performance Measure — *algebraically defined computation*
- Performance Predicate — *first order logic + order relations on measures*
- Specification — *domain instance + performance predicate*
- State — *schema + cached analyses*
- Transform — *domain-specific procedures + tree/graph rewrites*
- Locator/locale — *domain schema/topology-specific*
- Binding — *substitution for rewrites*
- Transformation — *explicit data structure holding <transform,locale,bindings>*
- Property preservation — *requirement for functional preservation*
- Metaprogram — *domain specific procedures, transform control language*

Transformation System Components in DMS

- Formal Source Language — *domain syntax*
- Formal Target Language — *domain syntax*
- Definition of Implementation — *preservation of functionality*
- A Representation for source/target instances — *hypergraph*
- A set of Transforms — *explicitly defined per domain*
- Tool to apply transforms — *procedures, term/graph rewrite engines*
- Means to apply multiple transforms — *rewrite sets, transform control language*
- Specification Parser — *generalized LR parser*
- Target PrettyPrinter — *box specification + box layout algorithm*
- Means to display intermediate states — *structure editor*
- Means to undo transformations — *design history*
- Means for performing analyses — *procedures, attribute grammars, rewrites*
- Means for augmenting Transformation System databases — *domain definitions*

Design Recovery for Legacy Code I

Constructing a Plausible Transformational Design

- Problem: only the code exists
- Solution: run transformation system "backwards"
- Purpose:
  - document design
  - prerequisite for
    - re-engineering
    - porting
    - maintenance
  - domain analysis/domain engineering
    - by discovery of transformational support needed
      - domain concepts/idioms
      - idiom optimizations and refinements,
      - transformational methods

Design Recovery Problems

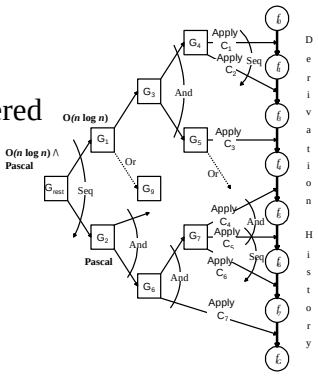
- Conditions: how to check preconditions?
- Control: Which transformation next?
  - implication that specification is roughly known!
- Does transform exist in the transform library?
  - if not, must capture it first
  - same for metaprograms, performance measures
- Code is often not correct!

⇒Design Recovery also requires continuous:

- Domain analysis and engineering
- Commitment of abstraction error

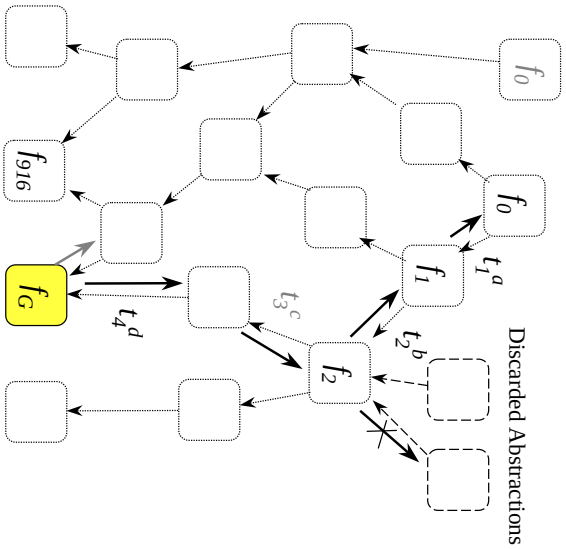
Design Recovery for Legacy Code II
Constructing a Plausible Design

- Method:
  - Use existing or proposed support
  - apply transform system in reverse
  - Fix code/support errors as encountered
- Record steps and justifications
- Extract desired aspect
  - *Specification*
  - *Derivation/Justification*
  - *Transforms/Metaprograms used*



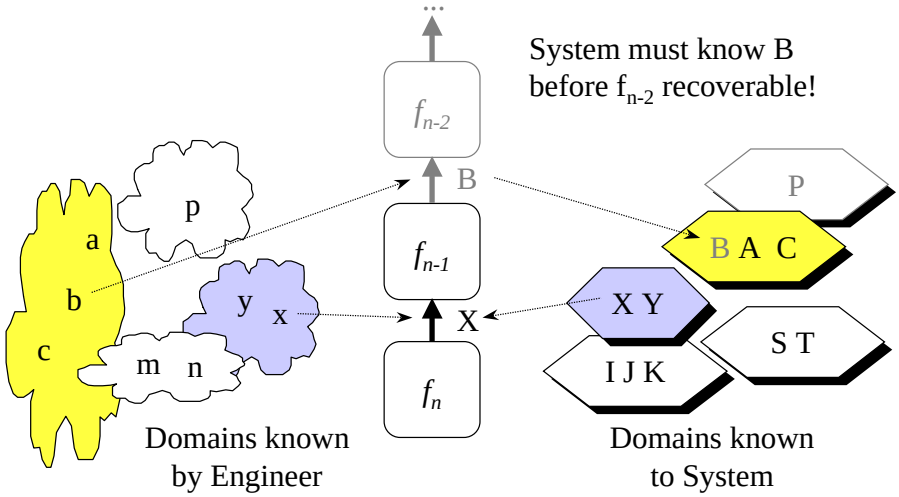
Observation: can't reverse engineer until you can forward engineer

Design Recovery – Control



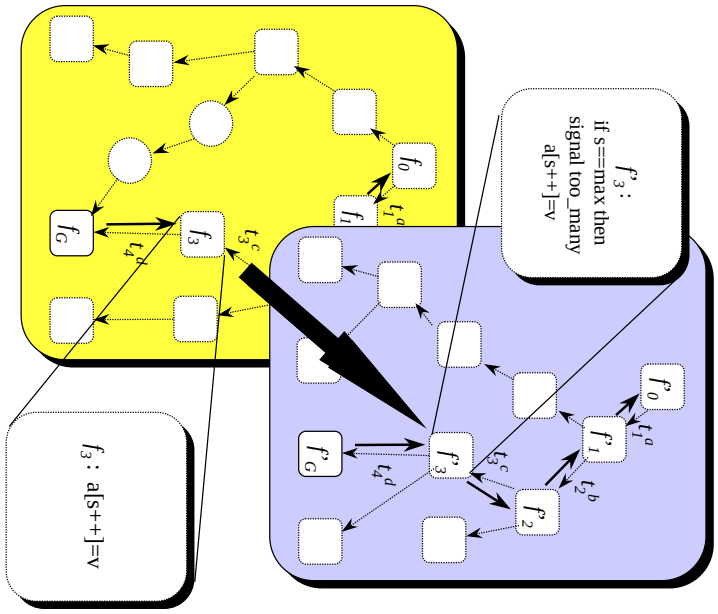
Bottom-up rediscovery of an implementation space

Design Recovery
Intertwined with Domain Engineering



Design Recovery
Commitment of Abstraction Error

Jumping between implementation spaces
to fix buggy code



Design Maintenance System – Status I

- DMS Support Domains
  - DMSLexical – Lexical Specifications
    with UNICODE, comment handling support, literal conversion support
  - DMSStringGrammar – (Context Free) Grammars
    with comment handling support, position information in abstract syntax tree
  - DMSAttributeGrammar – Attribute Grammars
  - DMSBoxPrettyPrinter – Pretty Printer Specifications
    pretty printing or fidelity printing
  - DMSRuleSpecificationLanguage – Definition of rules and rule sets
    with rewrite conditionals, procedural conditionals with procedural right hand sides of rules
- Other Domains (lexical specification and context free grammar)
  - COBOL85 (with name resolution and offset computation), C with preprocessor directives (with name resolution if preprocessor directives are expanded), C++ with preprocessor directives, ECMAScript, FORTRAN, Java, Pascal, PL/1, Spectrum, Verilog, VHDL, VisualBasic, XML, Zed, …

Design Maintenance System – Status II

- Rewrite Engine
  - conditional rewrite rules
    with procedural conditionals, procedural right hand sides of rules
  - modulo associative, commutative, associative-commutative 1st order theories
  - 2nd order, relaxed higher-order patterns

Clone DR
A Transformation System based on DMS

- Clone Detector *generic, detects exact copies and near misses*
- Clone Remover *domain specific transform to remove clones after detection*

Clones

- Statistically 10% of a large application system is *cloned* code

  your mileage may vary

  – Exact copies: caused by inlined code and lazy programmers
  – Near misses: "copy-n-hack" mode (good and bad reuse)
- Problems with Clones
  – A fixed bug in one code block often missed in its clone
    - higher cost to fix bugs
    - different fixes for same bug ⇒ code blocks appear less clone-like
  – Clone is often hint at a missed domain abstraction
    - capturing as procedure can clarify code and enhance design

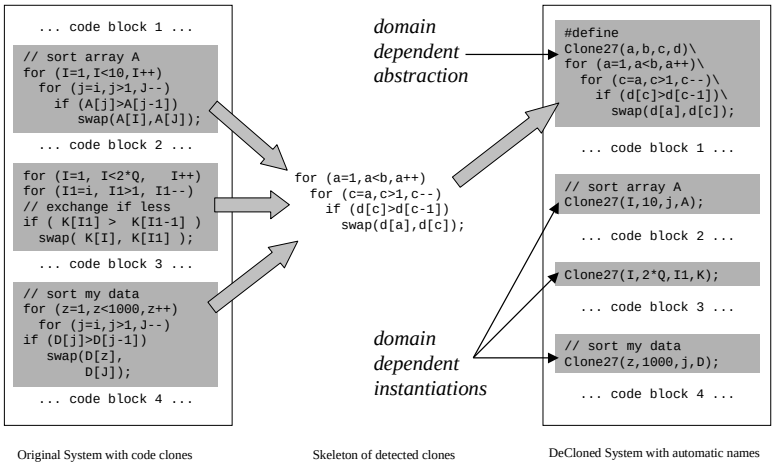
⇒ Manage Clones

  – Locate clones to fix bugs in all copies
  – Eliminate clones *$1/line/year to maintain code*

Clone Detection

- Clone detection is performed on abstract syntax trees
  – independent of whitespace, comments, representations of literals
  – independent of external form ⇒ can perform on any (string based) domain
  – clones and near misses based on language structure
- Clone detection is expensive
  – need to compare each subtree with each other subtree
    n nodes ⇒ n^2 tree comparisons *cost can be controlled by hashing*
    cost can be reduced linearly by parallel execution

Clone Removal



Example Clone Detection and Removal

California Community Colleges (CCC) Application

Course Inventory Construction for each campus

- Statistics:
  - 77,631 SLOC ANSI COBOL 85 → 774,645 AST nodes
    - 128 second parse time, 40 minutes for clone detection (early algorithm)
    - 500 Mb RAM, 6 CPUs
  - Number of exact clone pairs = 78, near miss pairs = 95
  - Largest clone: 5 copies, 1017 lines each, 1 parameter!
  - Number of cloned lines = 30727 → can remove 15363
- Actual SLOC reduction by removing clones = 19.8%

Sample COBOL Clones

```
Similarity = .99178082191781
from 35179 to 35204 file = example.cbl

  9700-OUTPUT-REPORT-TOTALS.
      MOVE CURRENT-COLLEGE-ID
          TO REPORT-CODE1  REPORT-CODE2  REPORT-CODE3.
      SET EDIT-ERROR-LITERAL-INDEX TO 1.
      SET DISTRICT-COUNT-ROW-INDEX TO 1.
      PERFORM 9710-OUTPUT-TOTALS1
          UNTIL EDIT-ERROR-LITERAL-INDEX > 30.
      PERFORM 9720-OUTPUT-TOTALS2.
      PERFORM 9730-OUTPUT-TOTALS3.


  9710-OUTPUT-TOTALS1.
      MOVE REPORT-SUM1 TO TOTALS-ID1.
      MOVE ELEMENT-NUMBER (EDIT-ERROR-LITERAL-INDEX)
              TO DED-NUMBER.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  1)
              TO EXCEPT-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  2)
              TO UNKNOWN-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  3)
              TO REASON-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  4)
              TO GRP3-COUNT.
      WRITE REPORT-TOTALS-RECORD1.
      SET EDIT-ERROR-LITERAL-INDEX UP BY 1.
      SET DISTRICT-COUNT-ROW-INDEX UP BY 1.
```

#1

#2

```
-----------------------------------
from 15368 to 15393 file = example.cbl

  9700-OUTPUT-REPORT-TOTALS.
      MOVE HOLD-DISTRICT-ID
          TO REPORT-CODE1  REPORT-CODE2  REPORT-CODE3.
      SET EDIT-ERROR-LITERAL-INDEX TO 1.
      SET DISTRICT-COUNT-ROW-INDEX TO 1.
      PERFORM 9710-OUTPUT-TOTALS1
          UNTIL EDIT-ERROR-LITERAL-INDEX > 17.
      PERFORM 9720-OUTPUT-TOTALS2.
      PERFORM 9730-OUTPUT-TOTALS3.


  9710-OUTPUT-TOTALS1.
      MOVE REPORT-SUM1 TO TOTALS-ID1.
      MOVE ELEMENT-NUMBER (EDIT-ERROR-LITERAL-INDEX)
              TO DED-NUMBER.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  1)
              TO EXCEPT-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  2)
              TO UNKNOWN-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  3)
              TO REASON-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  4)
              TO GRP3-COUNT.
      WRITE REPORT-TOTALS-RECORD1.
      SET EDIT-ERROR-LITERAL-INDEX UP BY 1.
      SET DISTRICT-COUNT-ROW-INDEX UP BY 1.
```

Report by College-ID

```
#2  ==>  30
    ==>  17
---------
#1
==>     CURRENT-COLLEGE-ID
==>     HOLD-DISTRICT-ID
```

Report by District-ID

The Generated COPYLIB

```
9700-OUTPUT-REPORT-TOTALS .
    MOVE PARAMETER-1
        TO REPORT-CODE1  REPORT-CODE2  REPORT-CODE3 .
    SET EDIT-ERROR-LITERAL-INDEX TO 1 .
    SET DISTRICT-COUNT-ROW-INDEX TO 1 .
    PERFORM 9710-OUTPUT-TOTALS1
        UNTIL EDIT-ERROR-LITERAL-INDEX > PARAMETER-2 .
    PERFORM 9720-OUTPUT-TOTALS2 .
    PERFORM 9730-OUTPUT-TOTALS3 .
9710-OUTPUT-TOTALS1 .
    MOVE REPORT-SUM1 TO TOTALS-ID1 .
    MOVE ELEMENT-NUMBER ( EDIT-ERROR-LITERAL-INDEX )
            TO DED-NUMBER .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX 1 )
            TO EXCEPT-COUNT .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX  2 )
            TO UNKNOWN-COUNT .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX 3 )
            TO REASON-COUNT .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX 4 )
            TO GRP3-COUNT .
    WRITE REPORT-TOTALS-RECORD1 .
    SET EDIT-ERROR-LITERAL-INDEX UP BY 1 .
    SET DISTRICT-COUNT-ROW-INDEX UP BY 1 .
```

Source File Change

```
MOVE DISTRICT-INT-CNT (11)
     TO INT-CNT-OUT-B.
WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-B
     AFTER ADVANCING 2 LINES.
MOVE DISTRICT-INT-CNT (12)
     TO INT-CNT-OUT-C.
WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-C
     AFTER ADVANCING 2 LINES.
```

```
MOVE DISTRICT-INT-CNT (11)
     TO INT-CNT-OUT-B.
WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-B
     AFTER ADVANCING 2 LINES.
MOVE DISTRICT-INT-CNT (12)
     TO INT-CNT-OUT-C.
WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-C
     AFTER ADVANCING 2 LINES.
```

```
9700-OUTPUT-REPORT-TOTALS.
    MOVE HOLD-DISTRICT-ID
        TO REPORT-CODE1, REPORT-CODE2, REPORT-CODE3.
    SET EDIT-ERROR-LITERAL-INDEX TO 1.
    SET DISTRICT-COUNT-ROW-INDEX TO 1.
    PERFORM 9710-OUTPUT-TOTALS1
        UNTIL EDIT-ERROR-LITERAL-INDEX > 17.
    PERFORM 9720-OUTPUT-TOTALS2.
    PERFORM 9730-OUTPUT-TOTALS3.


9710-OUTPUT-TOTALS1.
    MOVE REPORT-SUM1 TO TOTALS-ID1.
    MOVE ELEMENT-NUMBER (EDIT-ERROR-LITERAL-INDEX)
         TO DED-NUMBER.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 1)
         TO EXCEPT-COUNT.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 2)
         TO UNKNOWN-COUNT.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 3)
         TO REASON-COUNT.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 4)
         TO GRP3-COUNT.
    WRITE REPORT-TOTALS-RECORD1.
    SET EDIT-ERROR-LITERAL-INDEX UP BY 1.
    SET DISTRICT-COUNT-ROW-INDEX UP BY 1.
```

```
COPY CDR_clone10
     REPLACING PARAMETER-1
                    BY == HOLD-DISTRICT-ID ==
               PARAMETER-2
                    BY == 17 ==
```

```
9720-OUTPUT-TOTALS2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (1)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '01' TO INTEGRITY-ERROR-CODE.
    WRITE REPORT-TOTALS-RECORD2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (2)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '02' TO INTEGRITY-ERROR-CODE.
```

```
9720-OUTPUT-TOTALS2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (1)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '01' TO INTEGRITY-ERROR-CODE.
    WRITE REPORT-TOTALS-RECORD2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (2)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '02' TO INTEGRITY-ERROR-CODE.
```


Example Clone Detection and Removal

Java Development Kit sources 117_v3

- Statistics:
  - 681 Files; 151,255 SLOC → ~750K AST nodes
    - 170 second parse time, 22 minutes for clone detection
    - 500 Mb RAM, 8 CPUs (200Mhz Pentium Pro)
  - Number of exact clone pairs = 115, near miss pairs = 843
  - Largest clone: 2 copies, 253 lines each, 6 parameters
  - Most clones: 56 copies, 47 lines, 4 parameters
  - Number of cloned lines = 31741 → can remove 18259
- Estimated SLOC reduction by removing clones = 12%

Clone Detection/Removal Statistics

Application	*Language*	*SLOC*	*Removable SLOC (est.)*	*Removed SLOC*
Process Control	C	427921	54346 12.7%	N.A. (pre removal capability for C 6/98)
PARLANSE Compiler	C	42413	4133 9.7%	2568 6.0%
CCC Registration	COBOL85	77631	15363(39967) 19.8 (51.5)%	15034 19.3%
Roccade Accounting	IBM COBOL	567798	130494 23%	N.A. (not yet performed)
SLC Insurance	PROGRESS	356413	58783 16.5%	N.A. (not yet performed)
SWING Toolkit	Java	241460	26355 10.9%	N.A. (not yet performed)

Refactoring

"*… changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*"
Refactoring (Fowler99, Addison-Wesley)

Extract Method, p. 110

```
void printOwing(double amount) {
    printBanner();
    // print details
    System.out.println("Name:" = _name);
    System.out.println("Amount" + amount);
}
```



```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println("Name:" = _name);
    System.out.println("Amount" + amount);
}
```

☞ Formal, behavior preserving transforms ⇒ *Capture in Tool*!
- "Refactoring Object-oriented Frameworks" (Opdyke92)
- C++ refactoring (Tokuda98/99)
- http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser/Brant98 (Rewrite Rule Editor)

Extract Method as a *Mechanical* Transform

Mix of syntax matching and procedure

```
default base domain Java.

rule ExtractMethod(...):class->class
  = "\parent_class_modifiers \class_header
     { \some_declarations
       \method_modifiers \type \method_name \parameters \brackets \throw_list
          { \leader_statements
            \extractable_statements
            \follower_statements
          }
       \more_declarations }"
  rewrites to
    "\parent_class_modifiers \class_header
     { \some_declarations
       \extracted_method(\extractable_statements)
       \method_modifiers \type \method_name \parameters \brackets \throw_list
          { \leader_statements
            \new_methodcall(\extractable_statements)
            \follower_statements
          }
       \more_declarations
    }"
    if extractable(\leader_statements,\extractable_statements) -- no bad side effects

-- Properties of computed new method:
--   method name not member parameter_names(\class_header) or declared(\leader_statements)
--   method throw list = exceptions_escaping(\extractable_statements)
--   declarations = declared(\leader_statements) & escaping-referred(\executable_statements)
--   argumentnames = (parameter_names(\class_header) ∪
                      declared(\leader_statements)) ∩ reads(\executable_statements)
--   argumentbindings =
```

Java pattern

computes new method

computes call to method

OO Promise inhibited by Scale

- Typical 21st Century Application:
  IBM's "San Francisco" (ERP)
  - 2 million lines Java
  - 9000 Class definitions for ERP framework
- Scale refactoring by manual methods Very Difficult
  - Impractical to carry out thoroughly and reliably
  - Failure to refactor ⇒ Framework evolution frozen ⇒ Future ?
- Scalable re-engineering tools will become *necessary*
- *Transformation systems can provide that foundation*
  - Can support library of 100s of Refactoring operations (Fowler99)
  - Can support other useful transforms (optimizations, etc.)

Contents

- Introduction
- Theory
- Implementation
- Some Transformation Systems

  Summary

When to Use A Transformation System

- W*ell-defined* problem domain
  - with high payoff for automation
    - used often
    - high demands for reliability
  - with rapid evolution of problem domain or implementation knowledge
- Special optimizations/modifications for large bodies of code
  - many optimizations hard for software engineers to implement reliably on scale
- Re-engineering/porting

Knowledge Acquisition – The Big Cost

- Domain Analysis
  - Requires deep understanding of issues in domain
  - Must define *implementable* specification language
  - Domain *will* evolve as tool is used
  - Domain analysis methods not well understood yet
- Domain Engineering
  - Domain Analysis must be complete
  - Discovery/definition of transforms/refinements/methods to map to existing "lower level" domains
  - May require construction of intermediate domains
  - Domain engineering methods also not well understood
  - Need to capture lower-level domains
- May require man-years for complex domains

How Do We Validate a Specification?

Impossible to formally validate against informal requirements!

- Build formal specification to natural language paraphraser
  readable by non-experts
- Build consistency checker
  tests whether specification has an implementation
- Build completeness checker
  tests whether all functional properties are determined
- Build redundancy into specification
  more difficult to ensure property preservation
- Build multiple specifications with different perspectives
  how to ensure consistency between specifications?
- Symbolically evaluate specification
  test functional behavior
- Rapid Prototyping: build functional implementation fast, disregard properties
  test functional behavior

Open Issues

- How to specify performance metrics
- How to acquire domain knowledge
- How to construct specifications
- How to debug the design history
- How to control navigation of implementation space
- How to scale to develop large software systems
- How to interface with software engineer / with conventional software engineering activities
- How to (re)train large base of software engineers
- ...

The Future

- Demand for automation will push technology
- Progress requires compounded investment
- Domain-specific synthesis systems will appear
- Several commercial technology vendors
- Design capture/traceability between specifications, code, and domain/implementation knowledge
- Safety-critical systems built this way
- Software engineer's role moves towards specification development and specification debugging

Summary

- Transformation systems convert specifications to implementations
- No single, ideal specification language useful for all problem domains
  no "one size fits all"
- Transforms
  - preserve functionality
  - optimize performance
- Metaprograms provide automation by searching the implementation space and selecting transformation sequences
- Sophisticated engines required for support
- Acquiring knowledge to drive system is hard
- Payoff is reliable, semiautomatic implementation!

Where to Learn More

- [Partsch90] *Specification and Transformation of Programs*
  - Excellent overview of transforms and wide-spectrum languages
- [Feather86] *A Survey of Transformation Techniques*
  - Survey of Transformation Systems and Methods
- [Turski86] *The Specification of Computer Programs*
  - Theory of specification and implementation
- [Waters92] *Programmer's Apprentice*
  - Transformational Design Recovery and Modification
- [Smith90] *KIDS: A Semi-Automatic Development System*
  - Advanced transformation system
- [Neighbors84] *Draco Approach to Constructing Software*
  - Domain-specific transformations