CS 394P
Automatic Programming
Class Notes

Gordon S. Novak Jr.

Department of Computer Sciences
University of Texas at Austin

`http://www.cs.utexas.edu/users/novak`

Automatic Programming

Automatic Programming is the generation of programs by computer based on a specification.

Historically, assemblers and compilers were called automatic programming systems; “automatic” has meant “more automatic than current practice”.

The goal is to make the specification better than programming languages:

- Smaller
- Easier to write
- Easier to understand (closer to application concepts)
- Less error-prone

Need for Automatic Programming

Programming hasn't changed in 30 years.
– J. C. Browne

A programmer costs \$1/minute, produces 8-16 lines of code per day: \$30 - \$60 per line of code.

Computer-driven programming: Express the program in terms of what the computer can do directly.
```
x[i] = y[i] * b
```

Problem-driven programming: Express what we want:
$Sort(s) = \forall s \exists r : Permutation(s, r) \wedge Sorted(r)$

Domain-driven programming: Express the problem in terms of domain concepts, rather than computer or CS concepts.

Strategies for AI

Power-based Strategy: Use the speed and power of the computer to derive answers to problems by search, starting from a small number of first principles.

Knowledge-based Strategy: Expertise in solving problems is derived from large amounts of knowledge. The computer must similarly embody knowledge to be a good problem solver.

The knowledge-based strategy raises several questions:

- What is the knowledge that underlies expertise?
- How can the knowledge be expressed?
- How can the knowledge be used to solve a new problem?
- How much must the user *know* and *say* in order to make use of the computer's knowledge?

Problems with Programming Languages

In traditional programming languages, the *syntax* of program code reflects the *implementation* of data. This causes several problems:

1. Design decisions about data become reflected in the code. A single design decision is reiterated in many places.
2. Code becomes ossified and hard to change.
3. Code becomes hard to reuse.
4. Reuse requires conformity to design decisions made in writing the original software.

Goals of the Course

- Understanding of fundamental concepts:
  - Program transformation
  - Program generation
  - Program analysis and optimization
  - Partial evaluation
- Knowledge of useful techniques
- Experience in using several kinds of systems:
  - Pattern matching and program transformation
  - Partial evaluation
  - Program analysis
  - Object-oriented programming
  - Glisp, GEV, VIP, APS, GPS
- Familiarity with current research literature
- Preparation for research in this area

Companies

There are now a number of companies based on automated program generation:

- **Kestrel Institute / Reasoning Systems:** Cordell Green, Douglas Smith. Generation of programs from formal specifications (predicate calculus, category theory). Success with combinatoric search problems.
- **SciComp (Austin):** Elaine Kant. Generation of code to simulate systems of differential equations. Originally seismic analysis, now evaluation of derivative securities.
- **Semantic Designs (Austin):** Ira Baxter. Use of advanced program manipulation to improve commercial software.
- **Intentional Software Corp.:** Charles Simonyi. Goal is to produce advanced software tools for programmers to use.
- **NASA Ames: Automated Software Engineering:** Mike Lowry. Use of predicate calculus representation of scientific subroutines to find ways to use the subroutines to accomplish a goal in spacecraft navigation and control.

People in Austin

Many of the world's best researchers in automatic programming are in Austin. They will be invited to present guest lectures.

- UT: Gordon Novak, Don Batory, James C. Browne
- SciComp: Elaine Kant
- Semantic Designs: Ira Baxter
- SoftwareGenerators: Ted Biggerstaff

Overview

- Introduction
- Substitution and Pattern Matching
- Program Analysis and Optimization
- Partial Evaluation
- OOP and Aspect-oriented Programming
- Glisp, Views, and Graphical Programming
- Program Generation from Logic Specifications
- Active Researcher Lectures
- Research Literature

Abstract Syntax Tree

We will assume that the fundamental form of a program is the *abstract syntax tree* (AST) – not source code.

Lisp code is already in AST form, and Lisp is ideal for implementing program generation and transformation. It is easy to generate code in ordinary programming languages from Lisp.

Substitution and Pattern Matching

Both humans and automatic programming systems construct new programs by reuse of knowledge, much of it in the form of patterns:

- code patterns
- data structure patterns
- design patterns

Thus, the ability to make instances of a pattern is crucial.

Another important capability is *transformation* of programs, e.g. for optimization. A transformation can often be modeled as a pair of patterns: an input pattern, containing variables, that is matched against the given code, and an output pattern that represents the form of the code after transformation:

```
(- (- ?x ?y))  →  (- ?y ?x)
```

We will represent variables as symbols that begin with `?`.

Transformation patterns are also called *rewrite rules*.

Binding Lists

A *binding* is a correspondence of a name and a value. In Lisp, it is conventional to represent a binding as a `cons`:
`(cons` *name value* `)`

The binding will appear in *dotted pair* notation, i.e., it is printed with a dot between name and value:
`(?X . 3)`
If the value is a list, the dot and parentheses will cancel:
`(?X . (SIN THETA))` will print as `(?X SIN THETA)` .

A set of bindings is represented as a list, called an *association list*, or *alist* for short. A new binding can be added by:
`(push (cons` *name value* `)` *binding-list* `)`

A name can be looked up using `assoc`:
`(assoc` *name binding-list* `)`

```
(assoc '?y '((?x . 3) (?y . 4) (?z . 5)))
   =  (?Y . 4)
```

The value of the binding can be gotten using `cdr`:

```
(cdr (assoc '?y '((?x . 3) (?y . 4) (?z . 5))))
   =  4
```

Substitution

Substitution makes a copy of a pattern, substituting new values for variables in the pattern.

The function `subst` performs a single substitution:
`(subst` *new var pattern* `)`

```
(subst 3 '?r '(* pi (expt ?r 2)))
   =  (* PI (EXPT 3 2))
```

The function `sublis` performs multiple substitutions:
`(sublis` *binding-list pattern* `)`

```
(sublis '((?h . 5) (?w . 4)) '(* ?w ?h))
   =  (* 4 5)
```

If all variables have been substituted by constants, the result can be *evaluated*:

```
(eval (subst 3 '?r '(* pi (expt ?r 2))))
   =  28.274333882308138
```

Copying and Substitution Functions

```
(defun copy-tree (z)
  (if (consp z)
      (cons (copy-tree (first z))
            (copy-tree (rest z)))
      z) )

; substitute x for y in z
(defun subst (x y z)
  (if (consp z)
      (cons (subst x y (first z))
            (subst x y (rest z)))
      (if (eql z y) x z)) )

; substitute in z with bindings in alist
(defun sublis (alist z)
  (let (pair)
    (if (consp z)
        (cons (sublis alist (first z))
              (sublis alist (rest z)))
        (if (setq pair (assoc z alist))
            (cdr pair)
            z)) ))
```

These are system functions in Common Lisp.[1]

[1] The system functions `subst` and `sublis` copy only as much structure as necessary.

Program Transformation

Many kinds of transformations of a program are possible:

- Optimization of various kinds
- Translation to a different language
- Changing recursion to iteration
- Making code more readable
- Making code look different
- Making code *less* readable (*code obfuscation*)

Transformation could be accomplished by a program or by a set of rewrite rules.

Rewriting: Program vs. Pattern Matching

Rewriting of symbolic expressions can be accomplished by an *ad hoc* program, or by using pattern matching and rewrite rules. Each has advantages and disadvantages:

Program:

1. **Adv.:** Often is more efficient. Avoids restrictions that may be present with patterns.
2. **Dis.:** The rewriting program becomes large. Hard to find errors, omissions.

Pattern Matching:

1. **Adv.:** Declarative representation. Easier to understand. Mistakes less likely.
2. **Dis.:** May be slower.

A good compromise may be to use a pattern matcher in combination with programs for those cases that are simple or are not easily handled by patterns.

Transformation Process

Transformation of expressions can be implemented in several steps:

1. Test whether the input matches a pattern; keeping bindings between pattern variables and parts of the input.
2. Construct the output: create a version of the output pattern with substitutions from the bindings.
3. Keep trying: one transformation may enable another, so continue until the result does not change (reaches a *fixed point*).

Pattern Matching

Given a pattern and an input, matching must:

1. Test whether the input matches the pattern.
2. Determine variable bindings: the input code that corresponds to each pattern variable.

```
Pattern:      (- ?x (- ?y))

Input:        (- (sin theta) (- w))
```

The input matches the pattern; the bindings should be:

```
 ( (?x . (sin theta))
   (?y . w) )
```

Rules for Pattern Matching

Pattern matching can be thought of as a special kind of equality test:

1. Function names, constants and structure in the pattern must be matched exactly by the input.
   - `(- ?x ?y)` does not match `(+ a b)` .
   - The pattern `(+ (- ?x) ?y)` does not match the input `(+ a b)` .
2. A variable in the pattern will match anything in the input, but it must do so consistently.
   - The pattern `(- ?x ?x)`
     matches `(- (sin theta) (sin theta))`
     but not `(- (sin theta) (cos theta))` .

   We implement this by letting a variable match anything *the first time*, but it can only match *the same thing* thereafter.

Equality vs. Matching

```
; a version of 'equal' analogous to 'match'
(defun equal (pat inp)
  (if (atom pat)
      (eql pat inp)
      (and (consp inp)
           (equal (first pat) (first inp))
           (equal (rest pat)  (rest inp)))))
```

Equality vs. Matching

`match` can be considered to be a special kind of `equal`:

```
(defun match (pat inp) (matchb pat inp '((t . t))))

; returns bindings; nil --> match fails
(defun matchb (pat inp bindings)
  (and bindings
    (if (consp pat)
        (and (consp inp)
             (matchb (cdr pat)
                     (cdr inp)
                     (matchb (car pat)
                             (car inp) bindings)))
        (if (varp pat)
            (let ((binding (assoc pat bindings)))
              (if binding
                  (and (equal inp (cdr binding))
                       bindings)
                  (cons (cons pat inp) bindings)))
            (and (eql pat inp) bindings)) ) ) )


(defun varp (v)     ; Test for vars, e.g. ?x
  (and (symbolp v)
       (char= #\? (char (symbol-name v) 0))) )
```

Matching and Substitution

Matching and substitution can be combined to transform an input based on a *rewrite rule*: a list of an input pattern and an output pattern:

```
( (- (+ ?x ?y) (+ ?z ?y))   (- ?x ?z) )

(defun transf (rule input)   ; simple version
  (let (bindings)
    (if (setq bindings
              (match (first rule) input))
        (sublis bindings (second rule))) ))

(match '(- (+ ?x ?y) (+ ?z ?y))
       '(- (+ (age tom) (age mary))
           (+ (age bill) (age mary))))

((?Z AGE BILL) (?Y AGE MARY) (?X AGE TOM) (T . T))

(transf '((- (+ ?x ?y) (+ ?z ?y))
           (- ?x ?z))
         '(- (+ (age tom) (age mary))
             (+ (age bill) (age mary))))

(- (AGE TOM) (AGE BILL))
```

Dot Matching

It is possible to use “dot notation” to match a variable to the rest of a list:

```
( (progn nil . ?s)   (progn . ?s) )
```

The variable **`?s`** will match whatever is at the end of the list: 0 or more statements.

```
(transf '( (progn nil . ?s)   (progn . ?s) )
        '(progn nil (setq x 3) (setq y 7)) )

(PROGN (SETQ X 3) (SETQ Y 7))
```

More Complex Rules

It is desirable to augment rewrite rules in two ways:

1. Add a predicate to perform tests on the input; only perform the transformation if the test succeeds:
   `(and (numberp ?n) (> ?n 0))`
2. Create new variables by running a program on existing variables:

```
(transf '((intersection
             (subset (function (lambda (?x) ?p))
                     ?s)
             (subset (function (lambda (?y) ?q))
                     ?s))
            (subset (function (lambda (?x)
                      (and ?p ?qq)))
                    ?s)
            t
            ((?qq (subst ?x ?y ?q))) )
  '(intersection
     (subset #'(lambda (w) (rich w)) people)
     (subset #'(lambda (z) (famous z)) people)))

(SUBSET #'(LAMBDA (W) (AND (RICH W) (FAMOUS W)))
        PEOPLE))
```

Transformation

```
(defun transf (rule input)
  (let ((bindings (match (first rule) input))
        (test (third rule)))
    (if (and bindings
             (or (null test)
                 (eval (sublisq bindings test))))
        (progn
          (dolist (var (fourth rule))
            (push (cons (car var)
                        (eval (sublisq bindings
                                       (cadr var))))
                  bindings) )
          (sublis bindings (second rule)))
        'match-failure) ))

; sublis, quoting value of bindings
(defun sublisq (bindings form)
  (sublis
    (mapcar #'(lambda (x)
                (cons (car x) (kwote (cdr x))))
            bindings)
    form) )

(defun kwote (x) (if (constantp x) x (list 'quote x
```

Building on Transformation

The pattern matcher in the file `patm.lsp` has additional features that are needed for transformations:

- Constant folding and partial evaluation. When certain functions have constant arguments, the functions are evaluated as part of the transformation process. Note: such evaluation must not cause an error; functions that have side-effects (e.g. printing) should not be evaluated.
- Recursion and repetition. Parts of a complex expression must be recursively transformed. Sometimes one transformation will expose an opportunity for constant folding or other transformations.

  `( (+ ?n (+ ?m ?x)) (+ (+ ?n ?m) ?x) )`
  where `?m` and `?n` are numbers, will cause the input `(+ 3 (+ 7 foo))` to be transformed to `(+ (+ 3 7) foo)` and then, by constant folding, to `(+ 10 foo)` .

Pattern Matching

```
; Try to match input against optimization patterns.
(defun ptmatch (inp patwd)
 (let (patterns pattern tmp (res 'match-failure))
  (if (setq tmp (assoc inp *compile-time-constants*
                       :test #'equal))
      (cadr tmp)
      (if (and (consp inp)
               (symbolp (car inp))
               (setq patterns
                     (get (car inp) patwd)))
          (progn
            (while (and patterns
                        (eq res 'match-failure))
              (setq res
                    (transf (pop patterns) inp)) )
            (if (eq res 'match-failure) inp res))
          inp) ) ))
```

TRANS

```
(defun trans (x patwd)
 (let (xp tail tmp)
  (if (consp x)
      (if (and (not (member (first x)
                            '(print princ terpri format random)))
               (rest x)
               (symbolp (first x))
               (fboundp (first x))
               (every #'constantp (rest x)))
          (kwote (eval x))
          (progn        ; translate args first
           (setq tail (pttransl (rest x) patwd))
           (if (symbolp (first x))
               (setq xp (ptmatch
                          (if (eq tail (rest x))
                              x
                              (cons (first x) tail))
                          patwd))
               (progn (setq tmp (trans (first x) patwd))
                      (setq xp (if (and (eq tail (rest x))
                                        (eql tmp (first x)))
                                   x
                                   (cons tmp tail)))))
           (if (eq x xp) x (trans xp patwd)) ))
      (if (and (symbolp x)
               (setq tmp (assoc x *compile-time-constants*)))
          (cadr tmp)
          x)) ))
```

CPR

```
; Print stuff out in C form
(defun cpr (item &optional tabs)
 (let (newtabs)
  (if (stringp item)
      (princ item)
      (if (characterp item)
          (if (char= item #\Return)
              (progn (terpri)
                     (if tabs (spaces tabs)))
              (princ item))
          (if (symbolp item)
              (princ (string-downcase (symbol-name item)))
              (if (consp item)
                  (if (symbolp (first item))
                      (progn (cpr (first item))
                             (princ "(")
                             (mapl
                              #'(lambda (z)
                                  (if (not (eq z (rest item)))
                                      (princ ", "))
                                  (cpr (car z)))
                              (rest item))
                             (princ ")"))
                      (dolist (z item)
                        (if (and (characterp z) (char= z #\Ta
                            (setq newtabs (+ (or tabs 0) 2))
                            (cpr z (or newtabs tabs)) )))
                  (princ item))))) ))
```

Correctness of Transformations

It is not always easy to be certain that transformed code will give exactly the same results.

```
( (> (* ?n ?x)
     (* ?n ?y))          (> ?x ?y) )

( (not (not ?x))        ?x )

( (eq (if ?p ?qu ?qv)
      ?qu)              ?p )

( (rplaca ?x ?y)        (setf (car ?x) ?y) )
```

These transformations are “usually” correct, but it is possible to construct an example for each in which the transformation changes the result. We must be careful to use only correct transforms.

Top-Down Context

Sometimes context information can be sent top-down to allow transformations to be made safely.

- Logical use of result
- Result is busy outside

```
( (if (not (not ?p)) ?x)  (if ?p ?x) )

( (progn (rplaca ?x ?y)   (progn
         ?u . ?z)           (setf (car ?x) ?y)
                            ?u . ?z) )
```

Side Effects

A *side effect* is an action of a program other than returning a value:

- Changing a global variable: `(incf *count*)`
- Printing or other I/O
- Modifying structure: `(setf (car arg) 'foo)`

An otherwise correct transformation may not be correct if the code to be transformed has side-effects.

```
( (- (+ ?x ?y) (+ ?x ?z))    (- ?y ?z) )
```

does not produce the same result if applied to the code:

```
(- (+ (progn (print 'foo) 3) y)
   (+ (progn (print 'foo) 3) z))
```

Uses of Transformations

- Optimization. Low-level inefficiencies created by a program generation system can be removed.
- Specialization. Generic operations or program patterns can be specialized to the form needed for a specific implementation of an abstract data structure.
- Language translation. Transformations can change code into the syntax of the target language.
- Code expansion. Small amounts of input code can be transformed into large amounts of output code. The expansion can depend on specifications that are much smaller than the final code.
- Partial evaluation. Things that are constant at compile time can be evaluated and eliminated from code.

Optimization by Transformation

```
; Test whether a pipe floats in water
; (gldefun t3 ((p pipe)) (floats p))
(defun t3 (P)
  (> (* PI (EXPT (/ (CADDR (PROG1 P)) 2)
                 2))
     (* (- (* PI (EXPT (/ (CADDR (PROG1 P)) 2)
                       2))
           (* PI (EXPT (/ (CADR (PROG1 P)) 2)
                       2)))
        (GET (FIFTH (PROG1 P)) 'DENSITY))))
```

Patterns:

```
( (prog1 ?v)                  ?v )

( (- (* ?n ?x) (* ?n ?y))  (* ?n (- ?x ?y)) )
```

Specialization by Transformation

A generic operation can be specialized for a particular abstract data structure. Example: *sum* of a *set*.

```
(redefpatterns 'loop
  '( ((sum ?set)
      (make-loop ?set ?item (?total)
         (setq ?total 0)
         (incf ?total ?item) ?total)
      t
      ((?item (gentemp "ITEM"))
       (?total (gentemp "TOTAL"))) ) ) )

(redefpatterns 'list
  '( ((make-loop ?lst ?item ?vars
                 ?init ?action ?result)
      (let (?ptr ?item . ?vars)
        ?init
        (setq ?ptr ?lst)
        (while ?ptr
          (setq ?item (first ?ptr))
          (setq ?ptr (rest ?ptr))
          ?action)
        ?result)
      t
      ((?ptr (gentemp "PTR"))) ) ) )
```

Loop Expansion

```
>(trans '(defun za (l) (sum l)) 'loop)

(DEFUN ZA (L)
  (MAKE-LOOP L ITEM61 (TOTAL62)
    (SETQ TOTAL62 0)
    (INCF TOTAL62 ITEM61)
    TOTAL62))

>(trans * 'list)

(DEFUN ZA (L)
  (LET (PTR63 ITEM61 TOTAL62)
    (SETQ TOTAL62 0)
    (SETQ PTR63 L)
    (WHILE PTR63
      (SETQ ITEM61 (FIRST PTR63))
      (SETQ PTR63 (REST PTR63))
      (INCF TOTAL62 ITEM61))
    TOTAL62))

>(eval *)

>(za '(1 2 3 4 5))
15
```

Advantages of Expansion

- A small amount of source code expands to a large amount of output code.
- A linear $(n + m)$ set of source modules allows a combinatorial $(n * m)$ set of possible implementations.
  $Set = \{list, array, tree\}$
  $Operation = \{sum, average, max, min\}$
- A program can be specified by a small number of choices rather than by specifying all the consequences of those choices.
- A program can be specified in terms of domain concepts rather than programming concepts.

Language Translation by Transformation

Two sets of transformations, followed by a relatively simple printing program, can be used to perform source-to-source transformation from one language to another.

Restructuring:

```
( (setq ?x (+ ?x ?y))   (incf ?x ?y) )
```

Output Syntax:

```
((aref ?x ?y)         ("" ?x "[" ?y "]"))
((go ?x)              ("goto " ?x))
((setq ?x ?y)         ("" ?x " = " ?y))
((incf ?x)            ("++" ?x))
((incf ?x ?y)         ("" ?x " += " ?y))
((return ?x)          ("return " ?x))
((while ?p . ?s)      ("while ( " ?p " ) "
                         #\Tab #\Return
                         (progn . ?s)))
```

Example of Language Translation

The first stage is transformation through patterns to produce a Lisp representation of the output syntax.

```
(trans '(defun foo (x y)
             (incf y)
             (setq x (+ x y)))
       'lisptoc)

("" FOO "(" ("" X ", " ("" Y)) ")"
  #\Tab #\Return
 ("{" #\Tab #\Return
  ("" ("++" Y) ";" #\Return (""
  ("" X " += " Y )) ";")
  #\Return "}"))
```

A simple program, **cpr**, can print the result.

```
foo(x, y)
  {
    ++y;
    x += y;
    }
```

Knuth-Bendix Algorithm

The Knuth-Bendix algorithm[2] describes how to derive a complete set of rewrite rules R from an equational theory E, such that:

> If E implies that two terms s and t are equal, then the reductions in R will rewrite both s and t to the same irreducible form in a finite number of steps.

Two properties are needed:

- *Confluence*: no matter what sequence of transforms is chosen, the final result is the same.
- *Termination*: the process of applying transforms will terminate.

The Knuth-Bendix algorithm is based on a well-founded ordering of terms so that each rewriting step makes the result "smaller".

Unfortunately, rather simple systems do not have a Knuth-Bendix solution.

[2]Knuth, D. E and Bendix, P. E., "Simple word problems in universal algebras", in J. Leech (ed.), *Computational Problems in Abstract Algebra*, Pergammon Press, 1970, pp. 263-297.

Pattern Optimization Examples

```
(defun t1 (C D)
  (COND ((> (* PI (EXPT (CADDR (PROG1 C)) 2))
            (* PI (EXPT (CADDR (PROG1 D)) 2)))
          (PRINT 'BIGGER))))

(LAMBDA-BLOCK T1 (C D)
  (IF (> (ABS (CADDR C)) (ABS (CADDR D)))
         (PRINT 'BIGGER)))
```

Examples ...

```
(defun t2 (P Q)
  (LET ((DX (- (- (+ (CADDR (CURRENTDATE)) 1900)
                  (+ (CADDR (GET (PROG1 P)
                                 'BIRTHDATE))
                     1900))
               (- (+ (CADDR (CURRENTDATE)) 1900)
                  (+ (CADDR (GET (PROG1 Q)
                                 'BIRTHDATE))
                     1900))))
        (DY (- (/ (GET (PROG1 P) 'SALARY) 1000.0)
               (/ (GET (PROG1 Q) 'SALARY)
                  1000.0))))
       (SQRT (+ (* DX DX) (* DY DY)))))

(LAMBDA-BLOCK T2 (P Q)
  (LET ((DX (- (CADDR (GET Q 'BIRTHDATE))
               (CADDR (GET P 'BIRTHDATE))))
        (DY (/ (- (GET P 'SALARY)
                  (GET Q 'SALARY))
               1000.0)))
    (SQRT (+ (* DX DX) (* DY DY)))))
```

Examples ...

```
(defun t3 (P)
  (> (* PI (EXPT (/ (CADDR (PROG1 P)) 2) 2))
     (* (- (* PI (EXPT (/ (CADDR (PROG1 P)) 2) 2))
           (* PI (EXPT (/ (CADR (PROG1 P)) 2) 2)))
        (GET (FIFTH (PROG1 P)) 'DENSITY))))

(LAMBDA-BLOCK T3 (P)
  (> (EXPT (CADDR P) 2)
     (* (- (EXPT (CADDR P) 2) (EXPT (CADR P) 2))
        (GET (FIFTH P) 'DENSITY))))


(defun t4 ()
  (cond ((> 1 3) 'amazing)
        ((< (sqrt 7.2) 2) 'incredible)
        ((= (+ 2 2) 4) 'okay)
        (t 'jeez)))

(LAMBDA-BLOCK T4 () 'OKAY)
```

Examples ...

```
(defun t5 (C)
  (DOLIST
    (S (INTERSECTION
         (SUBSET #'(LAMBDA (GLVAR7289)
                     (EQ (GET (PROG1 GLVAR7289)
                              'SEX)
                         'FEMALE))
                 (GET (PROG1 C) 'STUDENTS))
         (SUBSET #'(LAMBDA (GLVAR7290)
                     (>= (STUDENT-AVERAGE
                          (PROG1 GLVAR7290))
                         95))
                 (GET (PROG1 C) 'STUDENTS))))
    (FORMAT T "~A ~A~%" (GET S 'NAME)
            (STUDENT-AVERAGE S))))

(LAMBDA-BLOCK T5 (C)
  (DOLIST (S (GET C 'STUDENTS))
    (IF (AND (EQ (GET S 'SEX) 'FEMALE)
             (>= (STUDENT-AVERAGE S) 95))
        (FORMAT T "~A ~A~%" (GET S 'NAME)
                (STUDENT-AVERAGE S)))))
```

```
; Test whether a line on the screen is being selected by the
; (gldefun draw-line-selectedp (d\:draw-line pt\:vector off\:
;   (let ((ptp (pt - off)))
;     (and (contains? (vregion d) ptp)          ; is point in
;          ((distance (line d) ptp) < 5) ) ))  ; and within d
(defun t6 (D PT OFF)
  (LET ((PTP (LIST (- (CAR (PROG1 PT)) (CAR (PROG1 OFF)))
                   (- (CADR (PROG1 PT)) (CADR (PROG1 OFF))))))
       (AND (AND (>= (CAR (PROG1 PTP))
                     (- (MIN (CAR (CADR (PROG1 D)))
                             (+ (CAR (CADR (PROG1 D)))
                                (CAR (CADDR (PROG1 D)))))
                        2))
                 (<= (CAR (PROG1 PTP))
                     (+ (- (MIN (CAR (CADR (PROG1 D)))
                                (+ (CAR (CADR (PROG1 D)))
                                   (CAR (CADDR (PROG1 D)))))
                           2)
                        (+ (ABS (CAR (CADDR (PROG1 D)))) 4)))
                 (>= (CADR (PROG1 PTP))
                     (- (MIN (CADR (CADR (PROG1 D)))
                             (+ (CADR (CADR (PROG1 D)))
                                (CADR (CADDR (PROG1 D)))))
                        2))
                 (<= (CADR (PROG1 PTP))
                     (+ (- (MIN (CADR (CADR (PROG1 D)))
                                (+ (CADR (CADR (PROG1 D)))
                                   (CADR (CADDR (PROG1 D)))))
                           2)
                        (+ (ABS (CADR (CADDR (PROG1 D)))) 4))
```

```
(< (ABS (LET ((DX (-
                    (CAR
                     (LET
                      ((GLVAR7282 (CADR (PROG1 D))
                       (GLVAR7283 (CADDR (PROG1 D)
                      (LIST
                       (+ (CAR GLVAR7282)
                          (CAR GLVAR7283))
                       (+ (CADR GLVAR7282)
                          (CADR GLVAR7283)))))
                    (CAR (CADR (PROG1 D)))))
               (DY (-
                    (CADR
                     (LET
                      ((GLVAR7284 (CADR (PROG1 D))
                       (GLVAR7285 (CADDR (PROG1 D)
                      (LIST
                       (+ (CAR GLVAR7284)
                          (CAR GLVAR7285))
                       (+ (CADR GLVAR7284)
                          (CADR GLVAR7285)))))
                    (CADR (CADR (PROG1 D))))))
              (/ (- (* DX
                       (- (CADR (PROG1 PTP))
                          (CADR (CADR (PROG1 D))))
                    (* DY
                       (- (CAR (PROG1 PTP))
                          (CAR (CADR (PROG1 D)))))
                 (SQRT (+ (EXPT DX 2) (EXPT DY 2))
   5))))
```

```
(LAMBDA-BLOCK T6 (D PT OFF)
  (LET ((PTP (LIST (- (CAR PT) (CAR OFF)) (- (CADR PT) (CADR
    (AND (AND (>= (CAR PTP)
                  (+ -2 (+ (CAR (CADR D))
                           (MIN 0 (CAR (CADDR D))))))
              (<= (CAR PTP)
                  (+ 2
                     (+ (+ (CAR (CADR D))
                           (MIN 0 (CAR (CADDR D))))
                        (ABS (CAR (CADDR D))))))
              (>= (CADR PTP)
                  (+ -2 (+ (CADR (CADR D))
                           (MIN 0 (CADR (CADDR D))))))
              (<= (CADR PTP)
                  (+ 2
                     (+ (+ (CADR (CADR D))
                           (MIN 0 (CADR (CADDR D))))
                        (ABS (CADR (CADDR D)))))))
         (< (ABS (LET ((DX (- (LET ((GLVAR7282 (CADR D))
                                    (GLVAR7283 (CADDR D)))
                                (+ (CAR GLVAR7282) (CAR GLVAR
                              (CAR (CADR D))))
                       (DY (- (LET ((GLVAR7284 (CADR D))
                                    (GLVAR7285 (CADDR D)))
                                (+ (CADR GLVAR7284) (CADR GLV
                              (CADR (CADR D)))))
                   (/ (- (* DX (- (CADR PTP) (CADR (CADR D)))
                         (* DY (- (CAR PTP) (CAR (CADR D)))))
                      (SQRT (+ (EXPT DX 2) (EXPT DY 2))))))
            5))))
```

```
>(cprfn 'rotate-x)

rotate-x(b, theta, x)
  {
    x[0][0] = b[0][0];
    x[0][1] = b[0][1];
    x[0][2] = b[0][2];
    x[0][3] = b[0][3];
    x[1][0] = ((cos(theta) * b[1][0])
               - (sin(theta) * b[2][0]));
    x[1][1] = ((cos(theta) * b[1][1])
               - (sin(theta) * b[2][1]));
    x[1][2] = ((cos(theta) * b[1][2])
               - (sin(theta) * b[2][2]));
    x[1][3] = ((cos(theta) * b[1][3])
               - (sin(theta) * b[2][3]));
    x[2][0] = ((sin(theta) * b[1][0])
               + (cos(theta) * b[2][0]));
    x[2][1] = ((sin(theta) * b[1][1])
               + (cos(theta) * b[2][1]));
    x[2][2] = ((sin(theta) * b[1][2])
               + (cos(theta) * b[2][2]));
    x[2][3] = ((sin(theta) * b[1][3])
               + (cos(theta) * b[2][3]));
    x[3][0] = b[3][0];
    x[3][1] = b[3][1];
    x[3][2] = b[3][2];
    x[3][3] = b[3][3];
    };
```

Optimization

Program optimization [3] can be defined as follows:

> Given a program P, produce a program P' that produces the same output values as P for a given input, but has a lower cost.

Typical costs are execution time and program space. Time is usually more important; fortunately, the two usually go together.

Optimization is an *economic* activity:

- **Cost:** a larger and sometimes slower compiler.
- **Benefit:**

```
Amount saved by the code improvement
  * number of occurrences in code
  * number of repetitions in execution
  * number of uses of the compiled code
```

It is not possible to optimize everything. The goal is to find *leverage*: cases where there is a large expected payoff for a small cost.

[3] Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1988, Ch. 10; Marvin Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, 1973.

Correctness of Optimization

Optimization must not introduce compiler-generated errors! A program that runs faster but produces incorrect results is not an improvement.

There are often cases where an optimization will *nearly always* be correct.

```
   if ( x * n == y * n ) ...
```

might be optimized to:

```
   if ( x == y ) ...
```

Is this correct?

In general, one must be able to *prove* that an optimized program will *always* produce the same result.

Local and Global Optimization

Local optimization is that which can be done correctly based on analysis of a small part of the program.
Examples:

- Constant folding: $2 * 3.14 \rightarrow 6.28$
- Reduction in strength: $x^2 \rightarrow x * x$
- Removing branches to branches:

```
    L1:   Goto  L2
```

Global optimization requires information about the whole program to be done correctly.
Example:

```
I * 8              ==>       R1 = I * 8

...                          ...

I * 8              ==>       R1
```

This is correct only if `I` is not redefined between the two points. Doing optimization correctly requires program analysis: a special-purpose proof that program P' produces the same output values as P.

Optimization and Automatic Programming

Optimization is relevant to Automatic Programming in several respects:

1. Program analysis is often needed for automatic programming.
2. Program transformation is a commonly used technique in automatic programming.
3. Automatic programming may generate code with poor performance; sometimes optimization can fix it.

Some Optimization Techniques

Some good optimization techniques include:

1. Generation of good code for common special cases, such as `I := 0`. These occur frequently enough to provide a good savings, and testing for them is fairly easy.
2. Generation of good code for subscript expressions. These often occur in loops.
3. Assigning variables to registers.
   - Much of code is loads and stores.
   - Easy case: assign a loop index variable to a register inside the loop.
   - General case: graph coloring.
4. Moving invariant code out of loops:

   ```
   for i := 1 to 1000 do
     x[i] := y[i] * sqrt(a);
   ```

   The code `sqrt(a)` does not change within the loop, so it could be moved above the loop and its value reused.
5. Reduction in strength: `x * 8` $\rightarrow$ `x << 3`

Program Analysis

In order for a compiler to perform certain optimizations, such as moving invariant code out of loops or reusing common subexpressions, it is necessary to have *global* information about the program. [4]

Control flow analysis provides information about the potential control flow:

- Can control pass from one point in the program to another?
- From where can control pass to a given point?
- Where are there loops in the program?

Data flow analysis provides information about the definition and use of variables and expressions. It can also provide valuable detection of certain types of programmer errors.

- Where is the value of a variable assigned?
- Where is a given assignment used?
- Does an expression have the same value at a later point that it had at an earlier point?

[4] This treatment follows Marvin Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, 1973.

Basic Block

A *basic block* (or *block* for short) is a sequence of instructions such that if any of them is executed, all of them are. That is, there are no branches in except at the beginning and no branches out except at the end.

```
begin
   i := j;
   if i > k
     then begin k := k + 1; i := i - 1 end
     else i := i + 1;
   writeln(i)
end
```

Finding Basic Blocks

Basic blocks are easily found by a compiler while processing a program.

A *leader* is the first statement of a basic block:

1. the first statement of a program
2. any statement that has a label or is the target of a branch
3. any statement following a branch

A basic block is a leader and successive statements up to the next leader.

Note that branch statements themselves do not appear in basic blocks, although the computation of the condition part of a conditional branch will be included.

In a graph representation of a program, basic blocks are the nodes of the graph, and branches are the arcs between nodes.

Relations and Graphs

The *cartesian product* of two sets A and B , denoted $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.

A *relation* between two sets is a subset of their cartesian product.

A *graph* is a pair (S, Γ) where S is a set of nodes and $\Gamma \subseteq S \times S$.

Properties of relations:

Property:	Definition:
Reflexive	$\forall a \quad (a, a) \in R$
Symmetric	$\forall a, b \quad (a, b) \in R \rightarrow (b, a) \in R$
Transitive	$\forall a, b, c \ (a, b) \in R \wedge (b, c) \in R$ $\rightarrow (a, c) \in R$
Antisymmetric	$\forall a, b \quad (a, b) \in R \wedge (b, a) \in R \rightarrow a = b$

A relation that is reflexive, symmetric, and transitive is an *equivalence relation*, which corresponds to a *partition* of the set (a set of disjoint subsets whose union is the set).

A relation that is reflexive, antisymmetric, and transitive is a *partial order*. Example: $\leq$.

Graph Notations

Let (S, Γ) be a graph and $b \in S$ be a node. [5]

$\Gamma b \; = \; \{x \in S \;\mid\; (b, x) \in \Gamma\}$
are the nodes that are *immediate successors* of b .

$\Gamma^{+} b \; = \; \{x \in S \;\mid\; (b, x) \in \Gamma^{+}\}$
are the nodes that are *successors* of b .

$\Gamma^{-1} b \; = \; \{x \in S \;\mid\; (x, b) \in \Gamma\}$
are the nodes that are *immediate predecessors* of b .

Let $A \subset S$ be a subset of the set of nodes S.

$\Gamma A \; = \; \{y \in S \;\mid\; (x, y) \in \Gamma \wedge x \in A\}$
is the set of nodes that are *immediate successors* of nodes in A .

$\Gamma^{-1} A \; = \; \{x \in S \;\mid\; (x, y) \in \Gamma \wedge y \in A\}$
is the set of nodes that are *immediate predecessors* of nodes in A .

We say (A, Γ_A) is a *subgraph* of (S, Γ) , where
$\Gamma_A x \; = \; \Gamma x \cap A$
is the set of transitions within the subgraph.

[5] Our notation generally follows that used in Marvin Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, 1973.

Bit Vector Representations

Subsets of a finite set can be efficiently represented as bit vectors, in which a given bit position is a `1` if the corresponding item is an element of the subset. Representing a 128-element set takes only 4 32-bit words of memory.

Operations on sets can be done by boolean instructions operating on whole words.

Set operation:	Bit vector operation:
$\in$	$\wedge$ with vector for element
	or test bit
$\cap$	$\wedge$
$\cup$	$\vee$
set complement of A	$\neg A$
set difference, $A - B$	$A \wedge \neg B$

Operations on the bit vector representation are $O(n/32)$, compared to $O(n \cdot m)$ with other methods.

Example use: assign a bit in a bit vector for each program variable or subexpression.

Boolean Matrix Representation of Graph

A relation R or graph on a finite set can be expressed as a boolean matrix M where:

$$M[i,j] = 1 \quad \text{iff} \quad (i,j) \in R \ .$$

Multiplication of boolean matrices is done in the same way as ordinary matrix multiplication, but using $\wedge$ for $\cdot$ and $\vee$ for $+$.

Property:	Matrix:
Identity, R^0	I_n (identity matrix)
Inverse, R^{-1} or Γ^{-1}	M^T
Reflexive	$I \subseteq M$
Symmetric	$M = M^T$
Transitive	$M^2 \subseteq M$
Antisymmetric	$M \cap M^T \subseteq I_n$
Paths of length n	M^n
Transitive closure Γ^+	$\cup_{i=1}^n M^i$
Reflexive transitive closure Γ^*	$\cup_{i=0}^n M^i$

Example: Let the set S be basic blocks of a program and Γ be transfers of control between blocks.

Dominators

Let e denote the first block of a program. A node d *dominates* a node n iff every simple path from e to n passes through d .

For a given node n, its *immediate dominator* is the dominator closest to it. A tree structure is formed by immediate dominators, with e being the root of the tree.

A loop header h dominates all the nodes in the loop. A *back edge* is an edge $n \rightarrow h$ where h dominates n.

Intervals

An *interval* is a subgraph that basically corresponds to a program loop.

An interval I with initial node h is the maximal subgraph (I, Γ_I) of (S, Γ) such that:

1. $h \in I$
2. $x \in I \rightarrow x \in \Gamma^* h$
3. $I - \{h\}$ is cycle-free
4. if $x \in I - \{h\}$, then $\Gamma^{-1} x \subset I$.

To construct an interval starting with node h:

1. initially, set $I := \{h\}$
2. repeat $I := I \cup \{x \in \Gamma I \mid \Gamma^{-1} x \subseteq I\}$
   until there are no more additions.

Members of $\Gamma I - I$ must be the heads of other intervals.

Intervals and Derived Graphs

Theorem: There is a unique partition of a program flow graph into intervals.

After a program graph has been partitioned into intervals, a *derived graph* $(\mathcal{I}, \Gamma_{\mathcal{I}})$ can be formed as follows:

- Nodes of the derived graph are the intervals of the original graph.
- Transitions of the derived graph are transitions between nodes of intervals of the original graph.
  $\Gamma_{\mathcal{I}} \ = \ \{(I_i, I_j) \ \mid \ \exists x \in I_i \ \exists y \in I_j \ \ (x, y) \in \Gamma\}$
  where $i \neq j$. Note that a transition can only be to the head node of an interval.

The process of making derived graphs is continued until a single node is reached. Some graphs are *irreducible*; these are rare in practice and can be handled by making an artificial duplicate of a node.

Definition and Reference of Variables

We assume that each variable is assigned a unique *bit number* so that it can be used in bit vectors. Likewise, each compiler variable or subexpression $\alpha \leftarrow a \circ b$ is assigned a bit number.

A variable is *defined* each time it is assigned a value. A variable is *referenced* (*used*) whenever its value is read.

The statement `x := a * b` first references `a` and `b` and then defines `x`.

The statement `x := x + 1` references `x` and then defines `x`.

A computation $a \circ b$ is *redundant* if its value is available in some variable α.

A subexpression is *computed* whenever it appears in an expression. A subexpression is *killed* if any of its components is defined or killed.

The statement `x[i*3] := a * b` computes `a * b` and `i * 3` and kills `x[`*anything*`]` .

Data Flow Analysis for a Block

Computed and *killed* vectors for a basic block can be found as follows:

- initially, $comp := \emptyset$ and $kill := \emptyset$.
- for each statement $v := a \circ b$ where $\alpha \leftarrow a \circ b$
  1. $comp := comp \cup \{\alpha\}$
  2. $kill := kill \cup kill_v$
  3. $comp := (comp - kill_v) \cup \{v\}$

  where $kill_v$ is the set of all expressions involving v directly or indirectly and $(comp - kill_v)$ is set difference.

Example: `I := I + 1`
This statement first computes the expression `I + 1`, but then it kills it because it redefines `I`.

Availability of Expressions

The expression $\alpha \leftarrow a \circ b$ is *available at a point* p if the value of the variable α is the same as the value of $a \circ b$ computed at the point p.

The expression α is *available on entry* to block b iff α is available on exit from all immediate predecessors of b.

$$avail_{entry}(b) \;=\; \cap_{x \in \Gamma^{-1}b}\, avail_{exit}(x)$$

The expression α is *available on exit* from block b iff α is available at the last point of b.

$$avail_{exit}(b) \;=\; (avail_{entry}(b) - kill(b)) \cup comp(b)$$

In general, a system of simultaneous boolean equations may have multiple consistent solutions. It is necessary to compute the *maximal solution* of the set of boolean equations for intervals at all levels of the derived graph.

Data Flow Analysis for an Interval

If the expressions that are available on entry to the head of the interval are known, the values for all blocks in the interval can be computed.

For each block b whose predecessors have had their values computed,

$$avail_{entry}(b) \;=\; \prod_{x \in \Gamma^{-1} b} avail_{exit}(x)$$

$$avail_{exit}(b) \;=\; avail_{entry}(b) \cdot \overline{kill(b)} + comp(b)$$

No expressions are available on entry to the first block of a program.

Dual Assumptions

The redundancy equations can be solved based on *dual assumptions*:

- Maximal assumption (superscript M): all subexpressions are available on entry to the head of an interval.
- Mimimal assumption (superscript m): no subexpressions are available on entry to the head of an interval.

Let variables x denote available on entry and variables y denote available on exit.

For an interval head h, set $x_h^M = 1$ and $x_h^m = 0$ (except $x_e^M = 0$ for the program entry block).

For other blocks in the interval, compute:

$$\begin{aligned} x_b^M &= \prod_{i \in \Gamma^{-1} b} y_i^M \\ x_b^m &= \prod_{i \in \Gamma^{-1} b} y_i^m \\ y_b^M &= x_b^M \cdot \overline{k_b} + c_b \\ y_b^m &= x_b^m \cdot \overline{k_b} + c_b \end{aligned}$$

Solving Equations for Derived Graph

For the head node of an interval of the derived graph, compute:

$$X_b^M \;=\; \prod_{i \in \Gamma^{-1} b} Y_b^M$$

Then, for each node c in the interval, update its output vector:

$$Y_c^M \;=\; X_b^M \cdot Y_c^M + \overline{X_b^M} \cdot Y_c^m$$

As this process is continued through levels of the derived graph, eventually it terminates because the top level is a single node whose inputs $X_e^M = 0$ are known because it is the entry block.

The algorithm can be viewed as updating the minimal and maximal available vectors until they become the same. By using the derived graphs, the updating is done efficiently.

Busy Variables

A dual notion to *available* is *busy*.

A variable is *busy* or *live* if it will be used before being defined again; otherwise, it is *dead*.

A variable is *busy on entrance* to a block b if it is used in block b before being defined, or if it is not defined or killed in block b and is busy on exit from b .

A variable is *busy on exit* from a block b if it is busy on entry to any successor of b .

We can define a bit vector *referenced*, meaning that an expression is referenced in a block before being computed or killed, and solve equations for *busy on entrance* and *busy on exit* in a manner analogous to that for the *available* equations.

Variable Uses and Register Assignment

A *def-use chain* is the connection between a definition of a variable and the subsequent use of that variable. When an expression is computed and is *busy*, the compiler can save its value. When an expression is needed and is *available*, the compiler can substitute the compiler variable representing its previously computed value.

Register allocation can be performed by graph coloring. A graph is formed in which nodes are def-use chains and (undirected) links are placed between nodes that share parts of the program flow graph.

A graph is *colored* by assigning "colors" to nodes such that no two nodes that are linked have the same color. Colors correspond to registers.

Register Allocation by Graph Coloring

An undirected graph is *colored* by assigning a “color” to each node, such that no two nodes that are connected have the same color.

Graph coloring is applied to register assignment in the following way:

- Nodes of this graph correspond to variables or subexpressions.
- Nodes are connected by arcs if the variables are busy at the same time.
- Colors correspond to registers.

A heuristic algorithm is applied to find approximately the minimum number of colors needed to color the graph. If this is more than the number of available registers, *spill code* is added to reduce the number of colors needed.

By keeping as many variables as possible in registers, the code can be significantly improved.

Overview of Global Optimization

A globally optimizing compiler will perform the following operations:

1. Perform interval analysis and compute the derived graphs.
2. Order nodes using an ordering algorithm to find dominators.
3. Find basic available and busy information for blocks.
4. Solve boolean equations to get available and busy information for each block.
5. Replace common subexpressions by corresponding compiler variables.
6. Assign registers using graph coloring.

The information provided by data flow analysis provides a special-purpose proof that the optimized program is correct (produces the same answers).

Global Optimization and Automatic Programming

The first phase of global optimization is to gather information about the program. This information can be useful for an automatic programming system.

1. Control flow: what transitions from one part of the program to another are possible?
2. Data flow: Where was the data used by a given part of the program defined? Will a given piece of data be used again after a specified point?

An automatic programming system can be careless about certain kinds of efficiency issues if it is known that optimizing compilers can fix the problems later.

Loop Transformations

Sometimes loops can be transformed to different forms that are faster.

```
for i := 1 to 1000 do
  for j := 1 to 1000 do
    x[i,j] := y[i,j];
```

This might be transformed to a single, linear loop:

```
for i := 1 to 1000000 do  x[i] := y[i];
```

Then it might be generated as a block-move instruction.

Code motion is moving code to a more favorable location, *e.g.*, moving invariant code out of loops:

```
for i := 1 to 1000 do
  x[i] := y[i] * sqrt(a);
```

The code `sqrt(a)` does not change within the loop, so it could be moved above the loop and its value reused.

Strip Mining

Getting effective performance from a multi-processor machine (i.e., getting speedup close to n from n processors) is a difficult problem.

For some matrix computations, analysis of loops and array indexes may allow “strips” of the array to be sent to different processors, so that each processor can work on its strip in parallel.

This technique is effective for a significant minority (perhaps 25%) of important matrix computations.

Induction Variable Transformation

Some compilers transform the *induction variable* to allow simplified subscripting expressions:

```
(:= I 1)
(LABEL 1)
(IF (<= I 1000)
    (PROGN ... (AREF X (+ -8 (* 8 I)))
           (:= I (+ I 1))
           (GOTO L1)))
```

might be transformed to:

```
(:= I' 0)
(LABEL 1)
(IF (<= I' 7992)
    (PROGN ... (AREF X I')
           (:= I' (+ I' 8))
           (GOTO L1)))
```

Note that the loop index has no meaning outside the loop and may not have storage assigned to it. Some machines can automatically increment an index register after it is used (called *postincrement*).

Finite Differencing

Finite differencing[6] is a general technique for optimizing expensive computations $f(i)$ that occur in a loop:

- Maintain local variables that hold previous values of the expensive computation $f(i)$ and perhaps some auxiliary values.
- Incrementally compute a new value $f(i + \delta)$ using:
  - the previous value $f(i)$
  - a *difference* from $f(i)$.

Example: $f(i) = i^2$

i	0		1		2		3		4		5
i^2	0		1		4		9		16		25
first difference:		1		3		5		7		9	
second difference:			2		2		2		2		

[6]Paige, R. and Koenig, S., "Finite Differencing of Computable Expressions", *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3 (July 1982), pp. 402-454

Example: Computing Squares

Assume that multiplication is an expensive operation. Consider the problem of computing squares of succesive integers.

```
for i := 0 to 99 do
  x[i] := i*i;
```

versus

```
next := 0;
delta := 1;
for i := 0 to 99 do
  begin
     x[i]  := next;
     next  := next + delta;
     delta := delta + 2
  end;
```

The second version has more code, but does no multiplication.

This form of computation has a long history; it was the basis of Babbage's Difference Engine.

General Case

Given an expression $f(x_1, ..., x_n)$, create a variable E to hold its value.

- **Initialize:** Create code

$$E = f(x_{1_0}, ..., x_{n_0})$$

  to establish E for the initial values of its arguments.

- **Derivative:** Replace each statement dx_i that modifies some variable x_i of E by the statements:

$$\partial^- E\langle dx_i \rangle$$

$$dx_i$$

$$\partial^+ E\langle dx_i \rangle$$

- **Redundant Code Elimination:** replace each occurrence of $f(x_1, ..., x_n)$ by E.
- **Dead Code Elimination:** remove any code that is now unused.

Finite Differencing for Set Operations

Finite differencing can be especially useful in optimizing set operations. Consider the following expression that is used in solving the "k queens" problem:

$$i \notin range(part\_sol) \wedge i \in \{1..k\}$$

This can be transformed to:

$$i \in setdiff(\{1..k\}, part\_sol)$$

A variable $unoccupied\_rows$ can be introduced for this expression. Its initial value is the set $\{1..k\}$.

An update to $part\_sol$ (by recursive call),

$$part\_sol = append(part\_sol, i)$$

leads to a corresponding change to $unoccupied\_rows$

$$unoccupied\_rows = unoccupied\_rows - \{i\}$$

This incremental update may be much cheaper than doing the original range computation or set difference every time.

Memoization

Memoization (or *memorization*) is the technique of saving previously calculated values of an expensive function $f(x)$. If a new request to compute $f(x)$ uses a value x that was used previously, the value of $f(x)$ can be retrieved from a table faster than it could be recomputed.

Compare:

- caching
- common subexpression elimination

Advanced CPU's may implement some memoization in hardware: if the CPU can determine that a computation has already been done and exists in a register, it can reuse the result.

Code Expansion

An essential topic for automatic programming is *code expansion*: converting a compact description of a computation into an expanded description of the details.

This has advantages and disadvantages:

- Expansion increases code size. In the worst case, code size could become infinite.
- Expansion often makes some things constant; optimization can then improve speed and reduce code size.

There are several common kinds of code expansion:

- Macro expansion and pattern-based expansion.
- Loop unrolling
- In-line function expansion

Macros

A *macro* is a function from code to code, usually turning a short piece of code into a longer code sequence.

Lisp macros produce Lisp code as output; this code is executed or compiled.

```
(defun neq (x y) (not (eq x y)))

(defmacro neq (x y) (list 'not (list 'eq x y)))

(defmacro neq (x y) `(not (eq ,x ,y)))
```

In C, `#define` *name pattern* specifies a textual substitution. If *pattern* contains an operation, it should be parenthesized:

```
#define sum (x + y)
```

Loop Unrolling

Loop unrolling is the compile-time expansion of a loop into repetitions of the code, with the loop index replaced by its value in each instance.

```
for i := 1 to 3 do x[i] := y[i];
```

is expanded into:

```
x[1] := y[1];
x[2] := y[2];
x[3] := y[3];
```

The second form may generate less code, and it runs faster. This is a useful optimization when the size of the loop is known to be a small constant at compile time.

Modulo unrolling unrolls a loop modulo some chosen constant. This can significantly reduce the loop overhead without expanding code size too much.

Loop Unrolling in Lisp

```
; unroll loop code: (dotimes (var n) code)
(defun unroll (docode)
  (let ((var (car (cadr docode)))
        (n   (cadr (cadr docode)))
        (code (caddr docode))
        result)
    (if (and (integerp n)
             (< n 20))
        (cons 'progn
              (dotimes (i n (nreverse result))
                (push (subst i var code)
                      result) ) )
        docode) ))
```

```
>(unroll '(dotimes (i 3)
            (setf (aref x i) (aref y i))))
(PROGN
  (SETF (AREF X 0) (AREF Y 0))
  (SETF (AREF X 1) (AREF Y 1))
  (SETF (AREF X 2) (AREF Y 2)))
```

Function Inlining

Inlining is the expansion of the code of a function at the point of call. If the code says `sqrt(x)`, `sqrt` can be invoked as a *closed* function in the usual way, or it can be expanded as an *open* or *inline* function by expanding the definition of `sqrt` at each point of call.

Inline expansion saves the overhead of subroutine call and parameter transmission; it may allow additional optimization because the compiler can now see that certain things are constant.

If code is in the form of abstract syntax trees, inlining is easy:

- Make sure the variables of the function are distinct from those of the caller.
- Generate assignment statements for the arguments.
- Copy the code of the function.

Partial Evaluation

Partial evaluation is the technique of evaluating as much of a program as possible at compile time, leaving only data-dependent computations to runtime.

- Constant folding: $\pi \cdot 2 \rightarrow 6.28$
- Operations involving special constants: $x \cdot 1 \rightarrow x$
- Simplification of `if` statements when the test is constant allows removal of interpretation.
- Method dispatch in OOP when type is known.
  `(draw x)` $\rightarrow$ `(draw-circle x)`
- Strong typing: eliminates type testing at runtime if types are known.
- Inline expansion: eliminates function call overhead, allows further optimization.

Interpretation

An *interpreter* is a program that reads instructions one at a time, decodes what they mean, then executes them.

Many parts of programs are actually interpreters:

```
pow(x, 3)
```

```
printf("i = %5d\n", i)
```

Rule of thumb: Each level of interpretation of a program costs an order of magnitude in performance.

Eliminating interpretation (increasing the degree of binding between a program and the particular application) is an important optimization.

Goal: Combine elegant, high-level expression of a program with efficient execution.

Partial Evaluation[7]

Partial evaluation specializes a function with respect to arguments that have known values. Given a program $P(x, y)$ where the values of variables x are constant, a specializing function `mix` transforms $P(x, y) \rightarrow P_x(y)$ such that $P(x, y) \;=\; P_x(y)$ for all inputs y. $P_x(y)$ may be shorter and faster than $P(x, y)$. We call x *static data* and y *dynamic data*. .

Partial evaluation involves:

- *precomputing* constant expressions involving x,
- *propagating* constant values,
- *unfolding* or *specializing* recursive calls,
- *reducing* symbolic expressions such as $x * 1$, $x * 0$, $x + 0$, $(\mathit{if}\ \mathit{true}\ S_1\ S_2)$.

Partial evaluation improves execution time by a (possibly large) constant factor, by increasing the *binding* between a program and its execution environment.

[7]Neil D. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.

Example

Suppose we have the following definition of a function `powerb(x,n)` that computes x^n :

```
(defun powerb (x n)
  (if (= n 0)
      1
      (if (evenp n)
          (square (powerb x (/ n 2)))
          (* x (powerb x (- n 1))))))
```

If this is used with a constant argument `n`, as is often the case, the function can be partially evaluated into more efficient code:

```
(gldefun t3 ((x real)) (powerb x 5))

(glcp 't3)
result type: REAL
(LAMBDA (X) (* X (SQUARE (SQUARE X))))
```

The recursive function calls and interpretation (`if` statements) have been completely removed; only computation remains. Note that the constant argument `5` is gone and has been converted into control.

Simple Partial Evaluator

```
(defun mix (code env)
 (let (args test fn)
   (if (constantp code)           ; a constant
     code                         ;   evaluates to itself
     (if (symbolp code)           ; a variable
       (if (assoc code env)       ;   bound to a constant
           (cdr (assoc code env)) ; evals to that constant
           code)                  ; else to itself
       (if (consp code)
           (progn
            (setq fn (car code))
            (if (eq fn 'if)         ; if is handled
                (progn              ;     specially
                  (setq test (mix (cadr code) env))
                  (if (eq test t)            ; if true
                      (mix (caddr code) env) ; then part
                      (if (eq test nil)      ; if false
                          (mix (cadddr code) env) ; else
                          (cons 'if
                            (cons test
                              (mapcar #'(lambda (x)
                                          (mix x env))
                                      (cddr code))))))))
```

Simple Partial Evaluator...

```
      (progn                                    ; (fn args)
        (setq args (mapcar #'(lambda (x)
                               (mix x env)) ; mix the args
                           (cdr code)))
        (if (and (every #'constantp args)   ; if all constant
                 (not (member fn '(print    ; and no
                        prin1 princ error   ; compile-time
                        format))))          ; side-effects
            (kwote (eval (cons fn args)))   ; eval it now
            (if (and (some #'constantp args); if some constan
                     (fndef fn))            ;  & symbolic fn
                (fnmix fn args)             ;  unfold the fn
                (fnopt (cons fn args)))))))  ; optimize result

(cons 'bad-code code)) ) ) ))
```

Examples

```
>(load "/projects/cs394p/mix.lsp")
>(mix 'x '((x . 4)))
4
>(mix '(if (> x 2) 'more 'less) '((x . 4)))
'MORE

(defun power (x n)
  (if (= n 0)
      1
      (if (evenp n)
          (square (power x (/ n 2)))
          (* x (power x (- n 1)))) ) )

>(fnmix 'power '(x 3))
(* X (SQUARE X))

>(specialize 'power '(x 3) 'cube)
>(fndef 'cube)
(LAMBDA (X) (* X (SQUARE X)))
> (cube 4)
64

>(fnmix 'power '(x 22))
(SQUARE (* X (SQUARE (* X (SQUARE (SQUARE X))))))
```

Examples

```
; append two lists
(defun append1 (l m)
  (if (null l)
      m
      (cons (car l) (append1 (cdr l) m))))


>(fnmix 'append1 '('(1 2 3) m))
(CONS 1 (CONS 2 (CONS 3 M)))
```

Binding-Time Analysis

Binding-time analysis determines whether each variable is static (S) or dynamic (D).

- Static inputs are S and dynamic inputs are D.
- Local variables are initialized to S.
- Dynamic is contagious: if there is a statement
  $v \ = \ f(...D...)$
  then v becomes D.
- Repeat until no more changes occur.

Binding-time analysis can be *online* (done while specialization proceeds) or *offline* (done as a separate preprocessing phase). Offline processing can *annotate* the code by changing function names to reflect whether they are static or dynamic, e.g. `if` becomes `ifs` or `ifd`.

Futamura Projections[8]

Partial evaluation is a powerful unifying technique that describes many operations in computer science.

The notation $[\![\mathtt{P}]\!]_{\mathtt{L}}$ denotes running a program `P` in language `L`. Suppose that **`int`** is an interpreter for a language `S` and **`source`** is a program written in `S`. Then:

- $$\begin{aligned} \mathtt{output} &= [\![\mathtt{source}]\!]_{\mathtt{s}}[\mathtt{input}] \\ &= [\![\mathtt{int}]\!][\mathtt{source}, \mathtt{input}] \\ &= [\![[\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}]]\!][\mathtt{input}] \\ &= [\![\mathtt{target}]\!][\mathtt{input}] \end{aligned}$$

  Therefore, $\mathtt{target} = [\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}]$.

- $$\begin{aligned} \mathtt{target} &= [\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}] \\ &= [\![[\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{int}]]\!][\mathtt{source}] \\ &= [\![\mathtt{compiler}]\!][\mathtt{source}] \end{aligned}$$

  Thus $\mathtt{compiler} = [\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{int}] = [\![\mathtt{cogen}]\!][\mathtt{int}]$

- Finally, $\mathtt{cogen} = [\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{mix}] = [\![\mathtt{cogen}]\!][\mathtt{mix}]$ is a *compiler generator*, i.e., a program that transforms interpreters into compilers.

[8] Y. Futamura, "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler", *Systems, Computers, Controls*, 2(5):45-50, 1971. `http://www.futamura.info.waseda.ac.jp/~futamura/`

Generating Extension

A *generating extension* of a program `P` is a program that generates specializations of `P` for different values of its static input.

For example, the following is a generating extension of the `power` program:

```
(defun f-gen (n) (list 'defun 'h '(x) (g n)))

(defun g (n)
  (if (= n 0)
      1
      (if (= n 1)
          'x
          (if (evenp n)
              (list 'square (g (/ n 2)))
              (list '* 'x (g (- n 1)))))))

>(f-gen 3)
(DEFUN H (X) (* X (SQUARE X)))

>(f-gen 5)
(DEFUN H (X) (* X (SQUARE (SQUARE X))))
```

Parameterized Programs

A highly parameterized program is easier to write and maintain than many specialized versions for different applications, but may be inefficient.

Example: Draw a line: (x_1, y_1) to (x_2, y_2).
Options include:

- Width of line (usually 1)
- Color
- Style (solid, dashed, etc.)
- Ends (square, beveled)

If all of these options are expressed as parameters, it makes code longer, makes calling sequences longer, and requires interpretation at runtime. Partial evaluation can produce efficient specialized versions automatically.

```
(draw-line x y (+ x 100) y 1 'black 'solid)
```

Representation Independence

In order to use a library program, it is usually necessary to understand the parameters it requires and to write code to compute these from the application data.

Example: Draw an object. It is desirable to be able to say simply `(draw` *item place* `)` .

```
(stsz (list (start vector) (size vector)))
(mkv 'line-segment 'stsz)

(glinstfnpattern 'draw '((s stsz) (myw window)))

(gldefun t21 ((s stsz) (myw window))
  (DRAW-LINE-XY MYW (P1X (LINE-SEGMENT S))
                    (P1Y (LINE-SEGMENT S))
                    (P2X (LINE-SEGMENT S))
                    (P2Y (LINE-SEGMENT S))))
(LAMBDA (S MYW)
  (LET ((QQWHEIGHT (CADDDR MYW)))
    (XDRAWLINE *WINDOW-DISPLAY* (CADR MYW)
        (CADDR MYW) (CAAR S)
        (- QQWHEIGHT (CADAR S))
        (+ (CAADR S) (CAAR S))
        (- QQWHEIGHT (+ (CADADR S) (CADAR S))))
    NIL))
```

Improvements of Partial Evaluation

There are several areas where partial evaluation technology could be improved:

- To be successfully partially evaluated, a program must be written in the right way. There should be good *binding time separation*: avoid mixing static and dynamic data (which makes the result dynamic).

  ```
  (lambda (x y z)              (lambda (x y z)
    (+ (+ x y) z))                 (+ x (+ y z)))
  ```

- The user may have to give advice on when to unfold recursive calls. Otherwise, it is possible to generate large or infinite programs.

  One way to avoid this is to require that recursively unfolding a function call must make a constant argument smaller according to a well-founded ordering. Branches of dynamic `if` statements should not be unfolded.

Improvements ...

- Repeating arguments can cause exponential computation duplication: [9]

  ```
  (defun f (n)
    (if (= n 0)
        1
        (g (f (- n 1)) ) ) )

  (defun g (m) (+ m m))
  ```

- The user should not have to understand the logic of the output program, nor understand how the partial evaluator works.
- Speedup of partial evaluation should be predictable.
- Partial evaluation should deal with typed languages and with symbolic facts, not just constants. For example, following $z = abs(x)$ we know $z \geq 0$.

[9] Jones *et al.*, p. 119.

Partial Evaluation: Matrix

In computer vision, the rotation of a point by θ around the x axis is accomplished by multiplying by the matrix:[10]

$$\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{array}$$

Many of the coefficients in this matrix are the special values `0`, `1`, or `-1`. In order to perform this operation with a standard matrix multiply subroutine, it would be necessary to:

- Create a matrix containing the 16 values.
- Call the subroutine to multiply by this matrix.

Many of the cycles consumed in the matrix multiply would be wasted because they would be trivial computations (*e.g.*, multiplying by `1` or adding `0`).

[10] D. H. Ballard and C. M. Brown, *Computer Vision*, Prentice-Hall, 1982, p. 477.

Code Expansion with Partial Evaluation

By unrolling the loops of matrix multiply, substituting the values from the coefficient matrix, and performing partial evaluation on the result, a specialized version of the matrix multiply is obtained:

```
>(fnopt 'mxmult 'rotate-x '(b theta x))
(LAMBDA-BLOCK ROTATE-X (B THETA X)
 (PROGN (SETF (AREF X 0 0) (AREF B 0 0))
        (SETF (AREF X 0 1) (AREF B 0 1))
        (SETF (AREF X 0 2) (AREF B 0 2))
        (SETF (AREF X 0 3) (AREF B 0 3))
        (SETF (AREF X 1 0)
              (- (* (COS THETA) (AREF B 1 0))
                 (* (SIN THETA) (AREF B 2 0))))
        (SETF (AREF X 1 1)
              (- (* (COS THETA) (AREF B 1 1))
                 (* (SIN THETA) (AREF B 2 1))))
     . . . ))
```

This version saves many operations:

Version:	Load	Store	Add/Sub	Mul	Total
General	128	16	48	64	256
Specialized	24	16	8	16	64

Partial Evaluation: Interpreter

This program is an interpreter for arithmetic expressions using a simulated stack machine.

```
(defun topinterp (exp)   ; interpret, pop result
  (progn (interp exp)
         (pop *stack*)))

(defun interp (exp)
  (if (consp exp)                      ; if op
      (if (eq (car exp) '+)
          (progn (interp (cadr exp))   ; lhs
                 (interp (caddr exp)) ; rhs
                 (plus))               ; add
          (if ...))      ; other ops
      (pushopnd exp)))   ; operand

(defun pushopnd (arg) (push arg *stack*))

(defun plus ()
  (let ((rhs (pop *stack*)))
    (pushopnd (+ (pop *stack*) rhs))))

>(topinterp '(+ (* 3 4) 5))
17
```

Specialization

The interpreter can be specialized for a given input expression, which has the effect of compiling that expression.

```
>(topinterp '(+ (* 3 4) 5))
17

>(specialize 'topinterp
             '('(+ (* a b) c))
             'expr1 '(a b c))

>(pp expr1)

(LAMBDA-BLOCK EXPR1 (A B C)
  (PROGN
    (PUSH A *STACK*)
    (PUSH B *STACK*)
    (TIMES)
    (PUSH C *STACK*)
    (PLUS)
    (POP *STACK*)))

>(expr1 3 4 5)
17
```

Partial Evaluation in OOP

Partial evaluation can achieve significant savings in object-oriented programming:

- When the type of an object is known at compile time, look up the function that implements a message. Replace the message send by a function call.
- Expand small message functions in-line:
  - Avoid function call overhead.
  - Allow optimization across messages.

Partial Evaluation in Automatic Programming

Partial evaluation can be especially valuable for automatic programming:

- Programming tools produce declarative program specifications
- Generic programs can be written as interpreters
- Partial evaluation produces a specialized program by performing the interpretation at compile time.

Object-oriented Programming

Object-oriented programming (OOP) has several topics of interest for Automatic Programming:

- Association of procedures with types.
- Organization of types in a hierarchy.
- Inheritance of procedures from higher types in the hierarchy.
- Generic procedures that can work for different types of data.

We will cover:

- Principles and terminology of OOP
- A simple CLOS-like OOP system implemented in Lisp
- Implementation of generic procedures
- Advantages and disadvantages of OOP

Type Checking

The *type* of data presented to a function must correspond to the operations to be performed on the data. This can be accomplished in several ways:

Programmer Type Checking: It is the programmer's responsibility to insure that a call to a function is made with the proper data types. (e.g., Fortran.)

Static Type Checking: The compiler must be able to know *at compile time* what the type of each data item is. (e.g., Pascal, Ada.)

Dynamic Type Checking: The type of each piece of runtime data can be determined. The application program tests the type of the data before performing operations on it. (e.g., Lisp.)

Static Type Checking

The compiler must be able to know *at compile time* what the type of each data item is.

```
procedure myqinsert
          (element:myelement queue:myqueue)
  ...
```

Advantages:

- Type errors detected at compile time.
- No runtime type checking required.

Disadvantage:

- Rigidity: If there are queues of different kinds of elements, the queue handling routines will have to be duplicated for each element type.

Dynamic Type Checking

The type of each piece of runtime data can be determined. The application program tests the type of the data before performing operations on it.

```
(defun copy-tree (x)
  (if (atom x) x
      (cons (copy-tree (car x))
            (copy-tree (cdr x))) ))
```

Advantages:

- Flexibility: The same code can work with many kinds of data.

Disadvantages:

- Speed: Need to test types repeatedly at runtime. Hardware support can help this problem.
- Verifiability: Harder to verify that a program will work for all combinations of data types.

Generic Functions

A *generic* or *polymorphic* function is one that performs a given operation for a variety of argument data types.

Example: `+` in Lisp:

```
(defun + (x y)
  (if (and (integerp x) (integerp y))
      (iplus x y)
      (if (and (floatp x) (floatp y))
          (fplus x y)
          ...) ) )
```

The `+` function in Lisp will also perform type *coercion* as needed. The CLOS implementation of object-oriented programming in Lisp combines the ideas of messages and generic functions. In effect, the programmer is allowed to *overload* functions – even system functions such as `+` – to specialize them for user data types.

Object-oriented Programming

Object-oriented programming (OOP) originated in the SIMULA language for discrete event simulation. The desire was to simulate large numbers of similar objects in an efficient manner. A class/instance representation achieves this goal.

- **Class:** represents the *behaviors* that are shared by all of its instances.
- **Instance:** represents the data for a particular individual.

Classes are arranged in a hierarchy, with inheritance of behaviors from higher classes.

Access to Objects

All access to objects is accomplished by sending *messages* to them.

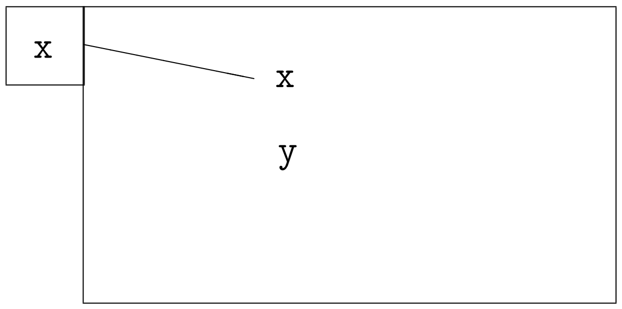
- Retrieving data values: `(send obj x)`
- Setting data values: `(send obj x:  3)`
- Requesting actions: `(send obj print)`

Internal Implementation is Hidden

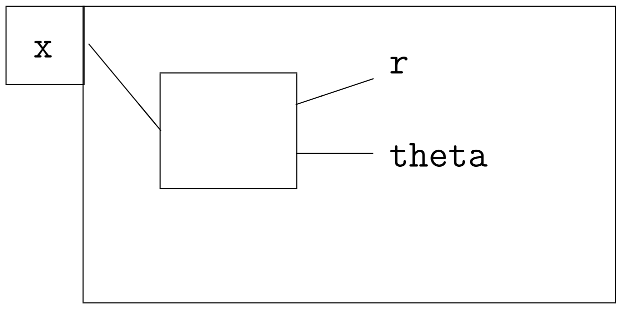
Messages define a standard interface to objects. Objects may have different internal implementations as long as the message interface is maintained.

Example: Vector `(send v x)`

- Vector type 1: `x` is stored.



- Vector type 2: `r` and `theta` are stored. `x` is computed as `r * cos(theta)`



The two kinds of vectors appear the same to the outside world.

Encapsulation with OOP

Object-oriented programming provides *encapsulation:* an external interface to an object in terms of messages is defined, but the internal implementation of the object is hidden.

Modularity: Objects are often a good way to think about the application domain.

Modifiability: The internal implementation of an object can be changed without modifying any other programs, as long as the external interface is maintained.

Expandability: New kinds of objects can be added to an existing system, as long as they present the same interface as existing objects.

Object-oriented Programming Terminology

- *Object:* typically refers to an instance (although classes may be objects too).
- *Class:* a description of a set of similar objects, the instances. This description typically includes:
  - *Instance variables:* the names of variables that are assumed to be defined for every subclass or instance of the class.
  - *Methods:* definitions of messages to which members of the class can respond.
- *Instance:* an individual member of a class. Typically an instance must be a member of exactly one class. An instance is a data structure that:
  - can be identified as being an object
  - denotes the class to which the object belongs
  - contains values of the instance variables
- *Superclass:* a class to which a given class belongs. Sometimes a class may have more than one superclass.

Terminology ...

- *Message:* an indirect procedure call. A message is *sent* to an instance object. The message contains a selector and optional arguments. The selector is looked up in the class of the object (or one of its superclasses) to find the method that implements the message. That method is called with the object to which the message was sent as its first argument.
- *Selector:* the name of a message. A generic procedure name.
- *Method:* the procedure that implements a message. Often the name of a method is the class name hyphenated with the selector, *e.g.* `square-draw`.

Message Sending

Sending a message to an object involves the following steps:

1. Find the method corresponding to the message selector.
2. Assemble the arguments:
   - Object to which the message was sent (the `self` argument)
   - Other arguments included in the message
3. Call the method function with the arguments.
4. Return the result returned by the method function.

Method Lookup

A typical method lookup procedure is:

- Look for the selector in the class of the object to which the message was sent.
- If it is not found, look in superclasses of the class.

Often, the first method found in such a depth-first search is the one used. This provides both shadowing and inheritance.

Variations include:

- Using types of all argument classes to determine the method (CLOS)
- Method Combination (Flavors)
- SendSuper (Loops)

Invoking Evaluation Explicitly

In addition to the usual ability to call a named function, Lisp makes it possible to *compute* the name of a function and explicitly cause that function to be called. There are several ways to do this. All do essentially the same thing; they differ in the form in which the arguments are presented.

`(eval` *x*`)` causes the expression *x* to be evaluated *twice*: once to find the value to be evaluated, and then to do the evaluation. That is, *x* is a computation that produces Lisp code as its output, and `eval` causes that Lisp code to be executed (evaluated).

```
(eval (subst 5 'x '(* x x)))

      =  (eval '(* 5 5))

      =  25
```

The ability to compute new pieces of program at runtime and execute them is a unique and powerful feature of Lisp; it makes Lisp an ideal substrate for implementing *embedded languages* such as Expert System tools.

APPLY and FUNCALL

`apply` and `funcall` are alternative ways to call a function whose name is computed; their arguments are supplied in different ways, so that explicit construction of Lisp code is not required, as it is for `eval`.

`(apply` *fn list-of-args*`)` applies the function *fn* to the arguments in the list *list-of-args*. Both arguments of `apply` are first evaluated.

```
(setq myfun #'+)

(apply myfun '(2 3))  =  5
```

`(funcall` *fn* arg_1 `...` arg_n `)` calls the function *fn* with the arguments arg_1 ... arg_n. All arguments of `funcall` are evaluated.

```
(funcall (get (get object 'shape)
              'drawing-program)
         (get object 'position)
         (get object 'size))
```

When to Use EVAL, APPLY, FUNCALL

The three forms `eval`, `apply`, and `funcall` allow efficient calling of functions, depending on the form of existing data.

Use `eval` when the existing data is already Lisp code:

```
(eval (subst 10.0 'r
             '(* 3.1415926 (expt r 2))))
```

Use `apply` when the existing data is already a list of arguments:

```
(setq nums '(3 7 21 14))
(apply #'+ nums)
```

Use `funcall` when the existing data is in the form of individual arguments:

```
(setq x 7)
(setq y 3)
(funcall #'+ x y)
```

FUNCALL Can Save CONSes

`funcall` can often save `cons`es compared to the other forms:

```
(funcall #'+ x y)                0 conses
(eval (list '+ x y))             3 conses
(apply #'+ (list x y))           2 conses
```

Since `cons` is a fairly expensive operation, it is worth learning how to use the appropriate form.

LAMBDA Expressions

A *lambda-expression* can be used to create an "anonymous" function (literally, one without a name); in general, a lambda-expression can be used where a function name is expected. The format of a lambda-expression is:

```
(lambda (args)  code)
```

A lambda-expression is "quoted" using the special form `function`, which can be abbreviated as `#'`.

```
(apply (function (lambda (x y) (+ x y)))
       '(2 3))

(apply #'(lambda (x y) (+ x y))
       '(2 3))
```

`function` or `#'` is analogous to `quote` or `'`, but is used instead of `quote` for lambda-expressions or function names.

Property List Representation

LISP provides for every symbol a *property list* that associates named properties with the symbol:

```
             binding
PRESIDENT --------->  BUSH
  |                     |
  | DUTIES (CINC ...)   | WIFE -----> LAURA
  |                     |               |
  | ...                 | AGE  58       |
```

This mechanism has several advantages:

1. New properties can be added at any time; there is no need to pre-declare properties.
2. Only those properties needed for each individual object need to be stored.
3. A property can have a single value or a list of values associated with it.

Property lists are a natural mechanism to use in implementing *semantic networks*.

Property Lists

Each symbol has a *property list* on which semi-permanent properties of the symbol can be stored. Each property has a *property name* or *indicator*, and a *value*. Property list values are retrieved and set by two functions:

`(get` *symbol propname*`)`
retrieves the value of the specified property for the specified symbol. If no such property exists, the value returned is `NIL`.

`(setf (get` *symbol propname*`)` *value*`)`
sets the value of the specified property for the specified symbol, displacing any previous value.

Each symbol has a single property list, which appears the same to all parts of a Lisp program and is *unaffected* by binding.

CLOS Example: ship

```
(defclass ship ()
   ((latitude    :type real)
    (longitude   :type real)
    (x-velocity  :type real)
    (y-velocity  :type real)) )

(defmethod speed ((s ship))
  (with-slots (s)
    (sqrt (+ (expt x-velocity 2)
             (expt y-velocity 2))) ))

(defmethod speedup ((s ship))
  (with-slots (s)
     (setf x-velocity (* x-velocity 2))
     (setf y-velocity (* y-velocity 2))
     s ))
```

A “virtual slot” **speed** is defined whose value is computed. The method **speedup** is defined similarly.

A Simple OOP System

A simple object-oriented programming system, called `oops`, has been constructed in Lisp; syntactically, it is similar to CLOS[11]. The `oops` system is intended to make clear what is going on "beneath the surface" in object-oriented systems and to allow experimentation with OOP.

Several things are needed for an OOP system:

- a way to define classes and store class definitions,
- a way to create new instances,
- a way to retrieve and store data in instance variables,
- a way to send messages,
- a way to define methods for a class of objects.

[11]Common Lisp Object System

Defining a Class

A class is defined using the following call:

```
(defclass class (supers)
  (slot-specs)
  class-options)
```

An example of a call to `defclass` is:

```
(defclass xyvector (vector)
  ((x :initarg :x :initform 0)
   (y :initarg :y :initform 0))
  (:documentation "A simple x-y vector"))
```

`defclass` is a macro; it calls the function `defclass-expr` with its arguments un-evaluated.

Class Implementation

A class is implemented using the property list of the symbol that is the class name. The following items are stored:

- an indicator that it is a class,
- a list of superclasses,
- a list of slot (instance variable) specifications,
- a list of options,
- an association list that maps message selectors to the corresponding function names (this list is maintained by `defmethod`).

Instance Implementation

An instance is implemented using the property list of a symbol that is generated from the name of the class. The function **`gensym`** generates new symbols; a function **`(make-atom`** *class***`)`** calls **`gensym`** to make a new symbol for an instance.

Two kinds of information need to be stored for instances:

- The class of the instance. This is stored on the property list under the property name `class`. We assume that `class` is a reserved word and cannot be the name of a user slot.
- The slot values. These are stored on the property list, using the name of the slot as the property name.

Creating an Instance

An instance object is created using the call:
`(make-instance` *class inits*`)`
where *class* is the class of the instance and *inits* is a sequence of `:initarg` names and corresponding values.

Example:

```
(make-instance 'xyvector :x 3 :y 4)
```

The macro `make-instance` calls `make-instance-expr` with the first argument evaluated but the remaining arguments un-evaluated.

Creation of an Instance

`make-instance-expr` uses `make-atom` to make a new symbol and puts in a pointer to the class.

Next, it is necessary to fill in slots and their values. There are some complications here:

1. An instance can inherit slots not only from its direct superclass, but from higher superclasses as well.
2. Initialization of the slot may be specified in the call to `make-instance` using the `:initarg` names defined for each slot (rather than the name of the slot). If an `:initarg` value is specified, it is evaluated.
3. If no `:initarg` is specified, there may be an `:initform` given in the slot specification. The `:initform` must be evaluated, e.g.:

   ```
   (creation-time
     :initform (get-universal-time))
   ```

Getting and Storing Data

Data is retrieved from an object using the call:
  `(slot-value` *object slotname*`)`
where *object* is an instance object and *slotname* is the name of the slot whose value is desired.

Data is stored into a slot using `setf`:
  `(setf (slot-value` *object slotname*`)` *value*`)`

Since an instance is implemented as a symbol, with slot values stored on its property list, `slot-value` simply becomes a call to `get`, and `set-slot-value` becomes a call to `setf` around a call to `get`. A call to `defsetf` is provided so that `setf` can be used with `slot-value`.

It is good style to use `slot-value` only within methods for the class and to define messages for the external interface:

```
(defmethod x ((v xyvector))
  (slot-value v 'x))
```

Defining Methods

Methods are defined using the call:

`(defmethod` *selector* `(`*args*`)` *code*`)`

where *selector* is the name of the message, `(`*args*`)` is a list of the arguments of the method, and *code* is the Lisp code that implements the method.

A method is actually a special form of function definition. It looks almost like a function definition, except that `(`*args*`)` has a special form. Each argument may be either a variable name, or a list `(`*variable class*`)` to declare the *class* (type) of the *variable*.

An example method definition is:

```
(defmethod area ((self circle))
  (* pi (expt (sendm self 'radius) 2)) )
```

This becomes:

```
(defun circle-area (self)
  (* pi (expt (sendm self 'radius) 2)) )
```

Implementing Method Definition

To define a method, it is necessary to do several things.

A name is made for the function that implements the method; conventionally, the class name and the selector are hyphenated. The class can be found from the type of the first argument. For example, the method `area` for class `circle` will become the function `circle-area`.

The method code is made into a function definition, then executed using `eval`.

Finally, an association between the selector and the function name is added to the class.

Associating Selector and Method

Each class has an association list of message selectors and corresponding functions. For example, a `circle` class might have the method list:

```
((radius circle-radius)
 (area   circle-area))
```

where the first item is the selector name and the second item is the corresponding function name.

`defmethod` updates this list when a new method is defined.

Sending Messages

A message is sent to an object using the call:
`(sendm` *object selector args*`)`
where *object* is the object to which the message is sent, *selector* is the name of the message (usually quoted), and *args* are the arguments of the message, if any.

For example, if `c` is a `circle` object and we want its area, we could use the call:

```
(sendm c 'area)
```

The CLOS system uses a syntax `(area c)` that is nicer but may obscure the fact that an expensive operation is occurring.

Implementing Message Sending

The first argument of **`sendm`** must be an object whose class can be determined; otherwise, it is an error.

Once the class of the object has been found, it is necessary to find the method that corresponds to the *selector* of the message.[12] This requires a search of the *class* of the object, and perhaps its superclasses, until the desired method is found.

Once the method has been found, the method function is called, using as arguments both the *object* to which the message was sent and any *args* supplied with the message.

[12]In CLOS, the method can depend on the types of all the arguments, not just the first argument.

Extending the Class System

It is nice to extend object-oriented programming to apply to ordinary Lisp objects.

An easy way to do this is by making the function `class-of` return the Lisp `type-of` data that are not instance objects.

Once basic Lisp data have classes, it is possible to overload operators and include basic data in the object system:

```
(defmethod + ((self string) ss)
  (concatenate 'string self ss))
```

oops System

```
(defmacro defclass
  (class supers slot-specs &rest class-options)
  `(defclass-expr (quote ,class) (quote ,supers)
     (quote ,slot-specs) (quote ,class-options)) )

(defun defclass-expr
  (class supers slot-specs class-options)
    (setf (get class 'classp)  t)
    (setf (get class 'supers)  supers)
    (setf (get class 'slots)   slot-specs)
    (setf (get class 'options) class-options)
    class)

(defun classp (class)
  (and (symbolp class) (get class 'classp)))

(defun class-of (object)
  (or (and (symbolp object) (get object 'class))
      (type-of object)))
```

Making Instances

```
(defun make-instance (class &rest initargs)
  (if (classp class)
    (let ((instance (make-atom class)))
      (addslots instance class initargs)
      (setf (get instance 'class) class)
      instance) ))


(defun addslots (instance class initargs)
  (let (initarg)
    (dolist (super (get class 'supers))
      (addslots instance super initargs))
    (dolist (slot (get class 'slots))
      (setf (get instance (car slot))
            (or (and (setq initarg
                           (getf (cdr slot)
                                 ':initarg))
                     (getf initargs initarg))
                (eval (getf (cdr slot)
                            ':initform)))) ) ))
```

Slot Values

```
; Get the stored value of a slot
(defun slot-value (object slot-name)
  (get object slot-name))

; Set the stored value of a slot
(defun set-slot-value (object slot-name value)
  (setf (get object slot-name) value) )

; Tell Lisp how to store a slot value,
; e.g. (setf (slot-value obj slot) val)

(defsetf slot-value set-slot-value)

; get the names of all slots defined for a class
(defun slot-names (class)
  (let (names)
    (setq names (mapcar #'car
                        (get class 'slots)))
    (dolist (super (get class 'supers))
      (setq names (union names
                         (slot-names super))) )
    names))
```

Defining Methods

```
(defmacro defmethod (selector args &rest rest)
  `(defmethod-expr (quote ,selector)
       (quote ,args) (quote ,rest)))

(defun defmethod-expr (selector args rest)
  (let (class fnname)
    (unless (and (consp (car args))
                 (setq class (cadar args))
                 (symbolp class))
      (error "Bad form - no class name ~S" args))
    (setq fnname (intern
      (concatenate 'string (symbol-name class)
                   "-" (symbol-name selector))))
    (pushnew (list selector fnname)
             (get class 'methods))
    (eval (cons 'defun (cons fnname
      (cons (mapcar #'(lambda (x)
                        (if (consp x) (car x) x))
                    args)
            rest)))) ))
```

Sending Messages

```
; Define a temporary cons cell for use by sendm
(defvar *sendmcons* (cons nil nil))

; Send a message to an object
(defun sendm (object selector &rest args)
  (let (method class)
    (unless (setq class (class-of object))
      (error "Object ~S has no Class." object))
    (if (setq method (findmethod selector class))
        (progn (rplaca *sendmcons* object)
               (rplacd *sendmcons* args)
               (apply method *sendmcons*))
        (error "No method ~A for object ~S"
               selector object)) ))

; Find a method for a given selector and class
(defun findmethod (selector class)
  (if (classp class)
      (or (cadr (assoc selector
                       (get class 'methods)))
          (some #'(lambda (super)
                    (findmethod selector super))
                (get class 'supers))) ) ) )
```

Smalltalk

ThingLab

ThingLab[13] is an interactive graphical simulation system based on Smalltalk.

- Objects have ports that can be connected graphically.
- If an object is modified interactively, it propagates changes to its ports, and thus to objects to which it is connected.
- Features of objects can be "anchored" to prevent them from changing.
- An object may do search by attempting to change different port values to make itself consistent.
- Problems are solved by value propagation or by relaxation.

[13]Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", *ACM Trans. on Programming Languages and Systems*, vol. 3, no. 4 (Oct. 1981), pp. 353-387.

ThingLab Examples

OOP and Automatic Programming

It is claimed that object-oriented programming satisfies a major goal of Automatic Programming: reuse of generic procedures. If a procedure can be reused, it is “free”.

The question to address is the effectiveness *in practice* of reuse by OOP.

Good Features of OOP

- Encapsulation: implementations of objects can be modified without modifying clients. Data types and related procedures are grouped.
- Polymorphism: some generic methods can be reused effectively.
- Inheritance: methods can be inherited within a hierarchy.

Unfortunate Features of OOP

- OOP tends to require that everything be objects. Hard to use with existing systems.
- Reusing generic procedures is not easy:
  - Programmer must know method name and arguments.
  - Arguments must respond appropriately to all messages that may be sent to them by the method or its submethods.
  - Methods may produce unexpected and unwanted side-effects.
- Slow in execution.
  - Method lookup overhead
  - Opacity of objects prevents optimization across messages.
  - A layered object structure compounds the problem.

  Some OOP systems "succeed" but have to be discarded and rewritten for efficiency.
- System structure becomes baroque and hard to change. (Just the opposite of what is advertised.)

Improving OOP Efficiency

- Use table lookup (array: class, method) to make method lookup fast (C++).
- Cache selector and method at lowest class level to avoid search (Self).
- Compile specialized methods "just in time" when first used:
  - `send`s can be converted to function calls since the class is known
  - small methods can be compiled in-line
  - partial evaluation may improve efficiency

Why OOP Is Not Enough

- OOP requires the programmer to know too many details about object and method names and implementations.
- OOP requires application data to conform to conventions of the generic procedures:
  - OOP requires an object to *be* a member of a class rather than *be viewable as* a member.
  - Some OOP systems automatically include slots of supers in all their descendants. This inhibits use of different representations.
  - An object cannot be a member of a class in more than one way.
  - No single definition of an object is likely to encompass all possible uses (or if it did, it would be too big for typical uses).
  - Uses of an object with different superclasses are likely to conflict.
- OOP is often slow in execution.

Top Ten Lies About OOP [14]

10. Objects are good for everything.

9. Object-oriented software is simpler.
(No: everything is a server, and servers are harder to write.)

8. Subclassing is a good way to extend software or libraries.

7. Object-oriented toolkits produce interchangeable components.
(No: you get components the size of a nuclear aircraft carrier, with internal interfaces that are too complex to duplicate.)

6. Using object-oriented programming languages helps build object-oriented systems.

5. Object-oriented software is easier to evolve.
(No: jigsaw-puzzle modularity is a serious problem.)

4. Classes are good for modularity.
(No: most worthwhile modules have more than one class.)

3. Reuse happens.
(No: you have to work too hard to make it possible. He distinguished between *use*, exploiting existing code, and *reuse*, building new code by extending existing code.)

2. Object-oriented software has fewer bugs.
(No: it has different bugs.)

1. C++ is an object-oriented programming language.

[14] From a keynote talk at the June 1993 Usenix conference by Mike Powell of Sun Labs, who is working on an object-oriented operating system in C++ (as recorded by Henry Spencer in the July 1993 issue of ";login:" magazine, whose remarks are in parentheses.)

Aspect-Oriented Programming

Aspect-Oriented Programming[15] is intended to facilitate coding of *cross-cutting* aspects, i.e. those aspects that cut across a typical decomposition of an application in terms of object classes.

[15]Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, "Aspect-Oriented Programming", in ECOOP '97, vol. 1241 of LNCS, Springer-Verlag, June 1997.

Aspect Weaving

In Aspect-Oriented Programming, an *aspect weaver* combines program fragments from separate aspects of a design at *join points*.

Glisp

The Glisp (Generic Lisp) language is based on Lisp and is compiled into Lisp. Its features include:

- Lisp-like language with types
- Recursive in-line expansion of OOP-like code.
- Specialization of generic procedures through *views*
- Partial evaluation
- Graphical programming tools
- Translation into programming language of choice

Data Structure Descriptions

A *structure description* is:

- A *basic type*:

  ```
  INTEGER REAL NUMBER STRING
  SYMBOL BOOLEAN ANYTHING
  ```

- A named structure description:

  ```
  (<name> <sd>)
  ```

- A composite structure description:

  ```
  (CONS               <sd1> <sd2>)
  (LIST               <sd1> ... <sdn>)
  (SYMBOL (PROPLIST (<name1> <sd1>) ...))
  (ATOMOBJECT        (<name1> <sd1>) ...)
  (LISTOBJECT        (<name1> <sd1>) ...)
  (CRECORD <name>    (<name1> <sd1>) ...)
  ```

- A group structure description:

  ```
  (LISTOF  <sd>)
  (ARRAYOF <sd>)
  ```

Reference to Substructures

A substructure or property is referenced by its name, in a form similar to that of a Lisp function call.

```
(glispobjects
  (circle (list (center vector)
                (radius real))
    prop  ((diameter (radius * 2))
           (area     (pi * radius ^ 2)) ) )
 )

(gldefun t1 ((c circle)) (radius c))

(glcp 't1)
  result type: REAL
  (LAMBDA (C) (CADR C))

(gldefun t2 ((c circle)) (area c))

(glcp 't2)
  result type: REAL
  (LAMBDA (C) (* 3.14159265 (EXPT (CADR C) 2)))
```

glispobjects

Data structures and messages are defined by the form:

```
(glispobjects
  (<name>  <structure>
     prop  ( <messages> )
     adj   ( <messages> )
     isa   ( <messages> )
     msg   ( <messages> )
    supers ( <super-classes> )
    views  ( (<view-name> <view-type>) ...)
  )
  ... )
```

The class name and structure are necessary. The other items are optional, in any order.

Conventionally, `prop`s are properties of the object itself, i.e. have only the object itself as argument and do not have side-effects. `adj` and `isa` are boolean predicates on the object itself. `msg` are general messages, i.e. may have additional arguments and side-effects.

Message List

A message list has the following format:

```
  (<selector>  <response>  <options>)
```

The `<selector>` is a symbol that is the name of a message. `<response>` is the way in which the message is interpreted:

```
  ( <expression> )               (radius * 2)
  ( <code> )                     ((sqrt area))
  (glambda (<args>)              (glambda (self)
    <code>)                        (radius self))
  <function-name>                mymsgfn
```

`<options>` may include:

```
RESULT <type>
MESSAGE T
OPEN T
SPECIALIZE T
ARGTYPES (<type1> ... <typen>)
```

Language Features

- An expression may appear on the left-hand side of an assignment:

```
( x ^ 2  = 2.0 )

(gldefun t3 ((c circle))
  ((area c) = 100))

(glcp 't3)
   result type: INTEGER
   (LAMBDA (C)
     (SETF (CADR C) 5.641895835477626)
     100)
```

- Creation of objects uses the `A` construct:

```
(a circle with
   center = (a vector with x = 10 y = 10)
   radius = 4.5)
```

For Loop

The Glisp compiler can compile loops over sets of data, including lists, trees, and any type for which iterator macros are defined.

A `for` loop can include `when` *predicate* and verbs `do`, `collect`, `sum`, `average`, `stats`.

```
(gldefun t11 ((grades (listof integer)))
  (for x in grades average x))

>(glcp 't11)

result type: INTEGER
(LAMBDA (GRADES)
  (LET (SUM VAL (N 0))
    (DOLIST (X GRADES)
      (SETQ VAL X)
      (IF (ZEROP N) (SETQ SUM VAL) (INCF SUM VAL))
      (INCF N))
    (IF (PLUSP N) (/ SUM N) 0)))

>(t11 '(90 95 87 100))

93
```

Inheritance

Methods can be inherited from superclasses.

```
(dcircle (listobject (start    vector)
                     (diameter real))
prop    ((radius (diameter / 2)))
supers  (circle))

(gldefun t5 ((d dcircle)) (area d))

result type: REAL
(LAMBDA (D)
  (* 0.7853981633974483
     (EXPT (CADDR D) 2)))

(gldefun t6 ((d dcircle)) ((area d) = 100))

(LAMBDA (D)
  (SETF (CADDR D) 11.283791670955125)
  100)
```

An inherited method is compiled in the context of the inheriting object type; for example, when the `area` is inherited by a `dcircle`, it is complied in the context of a `dcircle`.

Operator Overloading

Operators can be overloaded by defining them as messages.

```
(vector (list (x integer)
              (y integer))
  msg  ((+ vectorplus open t
                      argtypes (vector))))

(gldefun vectorplus ((v1 vector) (v2 vector))
  (a (typeof v1) with x = (x v1) + (x v2)
                      y = (y v1) + (y v2)))
```

The use of **`typeof`** lets the code use a type derived from an argument.

```
(gldefun t7 ((u vector) (v vector)) u + v)

result type: VECTOR
(LAMBDA (U V) (LIST (+ (CAR U) (CAR V))
                    (+ (CADR U) (CADR V))))
```

Recursive Expansion

In adding a `vector` of `vector`s, the function `vectorplus` is expanded three times:

```
(vofv (list (x vector)
            (y vector))
  supers  (vector))

(gldefun t8 ((u vofv) (v vofv)) u + v)

result type: VOFV
(LAMBDA (U V)
  (LIST
    (LET ((GLVAR35435 (CAR U))
          (GLVAR35436 (CAR V)))
      (LIST (+ (CAR GLVAR35435)
               (CAR GLVAR35436))
            (+ (CADR GLVAR35435)
               (CADR GLVAR35436))))
    (LET ((GLVAR35437 (CADR U))
          (GLVAR35438 (CADR V)))
      (LIST (+ (CAR GLVAR35437)
               (CAR GLVAR35438))
            (+ (CADR GLVAR35437)
               (CADR GLVAR35438))))))
```

Writing Generic Procedures

If a procedure is to be generic, *i.e.* reusable for a variety of data representations, it must make minimal assumptions about its data:

> Any *assumption* in a generic procedure becomes a *requirement* when the procedure is reused.

- The language syntax must not reflect the form of data.
- The generic procedure must not make assumptions about whether data is stored or computed.
- The generic procedure must not make assumptions about numeric type (*e.g.* `real` or `integer`) or units of measurement.
- There must be structure-independent ways to *write* and *create* data, as well as to read it.

Generics and Views

- *Views* describe how user data types implement features of abstract types.
- *Generic algorithms* are compiled with respect to views.
- The result is specialized versions of the generics that operate directly on application data.

Views

A *view* is a wrapper type that makes a concrete type appear to be an abstract type.

For example, we might want to view a `planet` as being a `circle`.

There may be multiple views of a concrete type, including multiple views as the same abstract type.

- A view is purely *gedanken*; it does not change the concrete data. Views allow a concrete type to be *thought of* as another type without *being* that type.
- A view type has the concrete type as its stored form.
- A view type defines properties that express what the abstract type expects in terms of what the concrete type has.
- Other data of the concrete type are hidden.
- The abstract type is a superclass of the view type.
- Procedures of the abstract type can be inherited and specialized through the view.

Computation as Simulation

A useful view is to think of all computation as simulation. *cf.*: *Isomorphism of semigroups* [16]

> Given two semigroups $G_1 = [S, \circ]$ and $G_2 = [T, *]$, an invertible function $\varphi : S \rightarrow T$ is said to be an *isomorphism* between G_1 and G_2 if, for every a and b in S,
> $$\varphi(a \circ b) = \varphi(a) * \varphi(b)$$

from which: $a \circ b = \varphi^{-1}(\varphi(a) * \varphi(b))$

[16]Preparata, F. P. and Yeh, R. T., *Introduction to Discrete Structures*, Addison-Wesley, 1973, p. 129.

Views as Isomorphisms

Requirements of Views

- An abstract type is defined as an abstract record containing a set of *basis variables*.
- Generic procedures are written in terms of the basis variables.
- A view type emulates the abstract record using the concrete record as storage:
  - *storage property*: after a value is stored into a basis variable, a read of the basis variable returns the same value.
  - *independence property*: storing into one basis variable does not change the values of others.
- A store into a basis variable may require updating several concrete fields.
- Slight inaccuracy in emulation may be allowed (e.g., floating point round-off error in representing (x, y) using (r, θ)).

Specifying a View

A view type has the format:

```
(glispobjects
 (<name> (<gensym> <concrete>)
   prop  ( (<abstract-prop>  <concrete-prop>)
           ... )
   supers (<abstract>) )
)
```

where the parts are:

`<name>`	name of the view type
`<gensym>`	unique name, e.g. a `gensym`
`<concrete>`	the concrete type
`<abstract-prop>`	property name needed by abstract type
`<concrete-prop>`	definition of property in terms of concrete type, referred to using `<gensym>`
`<abstract>`	the abstract type

Example: Textual View

```
(glispobjects

 (pizza (list (topping string) (size symbol))
   views ((circle pizza-as-circle)) )

 (pizza-as-circle (z753 pizza)
   prop  ((radius
            ((if ((size z753) == 'large)
                 9
                 6)) ) )
  supers (circle) )
 )

(gldefun t12 ((p pizza)) (area (circle p)))

>(glcp 't12)
result type: REAL
(LAMBDA (P)
  (IF (EQ (CADR P) 'LARGE)
      254.46900494077323
      113.09733552923255))
```

Note that the view name, `circle`, is used as a type-change operator to change to the view type.

A Textual View

```
(gldefun t9 ((pc planet-orbit-as-circle))
  (area pc))

(LAMBDA (PC)
  (* PI (EXPT (GET PC 'ORBIT-RADIUS) 2)))
```

Pipe Example

```
(pipe (listobject (inside-diameter real)
                  (outside-diameter real)
                  (length real)
                  (material material)
                  (nxt (^ pipe)))
  views ((inside-circle  circle
              (radius (inside-diameter / 2)))
         (outside-circle circle
              (radius (outside-diameter / 2))) )
  prop  ((cross-section
            (  (area (outside-circle self))
             - (area (inside-circle self)))))
  adj   ((floats
           ( (area (outside-circle self))
              > (cross-section self)
                 * (density (material self))))) )

(gldefun t10 ((p pipe)) (floats p))

result type: BOOLEAN
(LAMBDA (P)
  (> (EXPT (CADDR P) 2)
     (* (- (EXPT (CADDR P) 2) (EXPT (CADR P) 2))
        (GET (FIFTH P) 'DENSITY))))
```

Notes on Pipe Example

The `pipe` example has several interesting features:

- The `pipe` is viewed as a `circle` in two different ways.
- No additional data or runtime computation is needed for the views.
- The expression of properties of the pipe is clear and natural.
- In-line expansion of properties through views makes the generated code efficient.

Making Views

- Views may be complex: an example `line-segment` view is 67 lines of code.
- Views may be error-prone if written by hand: storage and independence properties must be maintained.
- Difficulty of coding view types detracts from benefit of reuse.
- Automated assistance for making views is needed.

Views can be created in several ways:

- Textual specification, by hand.
- Graphical specification: click on corresponding parts of menus or diagrams.
- `VIEWAS`: (semi-) automatic generation of a view, *e.g.*, as a `linked-list`.
- `MKV`: (semi-) automatic generation of a mathematical view, based on diagram connections.

Data Structure Views: VIEWAS

The VIEWAS system makes a view of a concrete type as a data structure type, such as a `linked-list` or `avl-tree`.

- VIEWAS makes correspondences between abstract and concrete records
  - Pointers
  - Values, e.g. field to sort on
  - Choices, e.g. `sort-direction`
  - Groups of fields, e.g. *contents* is everything except the link pointer.
- Type filtering and propagation of choices reduce input from user; simple views can be made automatically.

Example: Sorted Linked List

```
(part (name string)
      (size integer)
      (next (^ part)))
```

- The `link` pointer is inferred to be the `next` field: the only possibility allowed by type filtering.
- The `sort-value` is selected as `name` by the user.
- The `sort-direction` is selected as `ascending`.

The user only needs to make two choices from menus; then any procedure defined for `sorted-linked-list` can be specialized for the user's data.

Standard data structure templates can easily be instantiated with a given contents.

Views by Graphical Correspondences

Views can be specified by graphical correspondences between user data and a diagram of an abstract object.

Example: View a Christmas tree as a cone.

```
>(mkv 'cone 'xmas-tree)
```

```
(gldefun tg ((x xmas-tree))
  (side-area (cone x)))

(LAMBDA (X)
  (* 3.1415926535897931
     (* (FIFTH X)
        (SQRT (+ (EXPT (FIFTH X) 2)
                 (EXPT (CADDR X) 2))))))
```

Mathematical Views: MKV

- User makes correspondences between a diagram of the abstract type and a menu of fields and computed properties of the concrete type.
- Buttons on the diagram correspond to variables of the abstract type
- Equations associated with abstract type are solved by symbolic algebra
- A button is removed from diagram when solved; this prevents inconsistent specification.
- Symbolic algebra is used to make a view that maintains storage and independence properties.
- An instance of concrete type can be created from a set of basis variables.
- Data translation is possible through views as a common abstract type:

Line Segment

A line segment could be represented in many ways:

- two end points
- one end point, length, theta
- one end point, slope, delta-x

The figure presents variable buttons to allow nearly any representation to be specified easily.

Example of Line Segment View

```
(mkv 'line-segment 'ls1)
```

```
(gldefun tb ((l ls1) (p consv))
  (leftof-distance (line-segment l) p))

result type: REAL
(LAMBDA (L P)
  (- (* (COS (CADDDR L))
        (- (CDR P) (CADR L)))
     (* (SIN (CADDDR L))
        (- (CAR P)
           (- (FIFTH L)
              (* (CADDR L) (COS (CADDDR L))))))))
```

Line Segment Data Conversion

```
(mkv 'line-segment 'ls2)
```

```
(gleqns-transfer-by-view 'ls2 'ls1)

(LAMBDA (VAR-LS1)
  (LET ((VAR-LS1-VIEW VAR-LS1))
    (LIST 'LS2
          (- (FIFTH VAR-LS1-VIEW)
             (* (CADDR VAR-LS1-VIEW)
                (COS (CADDDR VAR-LS1-VIEW))))
          (FIFTH VAR-LS1-VIEW)
          (- 1.570796 (CADDDR VAR-LS1-VIEW))
          (+ (CADR VAR-LS1-VIEW)
             (* (CADDR VAR-LS1-VIEW)
                (SIN (CADDDR VAR-LS1-VIEW)))))))
```

Advantages of Graphical Correspondences

- Diagrams are self-documenting.
- Diagrams represent concepts that the user already knows.
- The interface is fast and easy to use.
- Reusing procedures through views is less error-prone than doing algebra by hand.
- Application data does not have to conform to a library program. The library program is converted automatically.
- User data could be used with a remote program over a network if the user and author of the program each make a view of their data as a common abstract type:

This allows compatibility without conformity.

MKV: Incremental Equation Solving

Equations are solved incrementally as each variable becomes defined by correspondence to application data.

- A set of simple, redundant equations is used.
- A simple equation solver is used.
- Buttons for variables that can be solved are removed from the diagram, preventing redundant specification.
- Variables are solved as soon as possible, which tends to give efficient code.

Basis Variables and Equations

```
(setf (get 'line-segment 'basis-vars)
      '(p1x p1y p2x p2y))

(setf (get 'line-segment 'equations)
      '((= p1     (tuple (x p1x) (y p1y)))
        (= p1x    (x p1))
        (= p1y    (y p1))
        (= p2     (tuple (x p2x) (y p2y)))
        (= p2x    (x p2))
        (= p2y    (y p2))
        (= deltax (- p2x p1x))
        (= deltay (- p2y p1y))
        (= slope  (/ deltay (float deltax)))
        (= slope  (tan theta))
        (= slope  (/ 1.0 (tan phi)))
        (= length (sqrt (+ (expt deltax 2)
                           (expt deltay 2))))
        (= theta  (atan deltay deltax))
        (= phi    (- (/ pi 2.0) theta))
        (= phi    (atan deltax deltay))
        (= deltay (* length (sin theta)))
        (= deltax (* length (cos theta)))
        (= deltay (* length (cos phi)))
        (= deltax (* length (sin phi))) ) )
```

Equation Solving

When a variable is defined, equations are examined:

- If an equation contains exactly one unsolved variable, it is solved for that variable.
- If an equation contains no unsolved variables, the equation is deleted.
- If a component of a `tuple` is solved, the `tuple` equation is deleted.
- If an equation contains a deleted `tuple` variable, the equation is deleted.

These steps are repeated until no further variables are solved. A list of solved and deleted variables is returned.

Basis Variable and Equations

```
1. Enter var-defined, var = P1Y
  2c. deleting tuple (= P1 (TUPLE (X P1X) (Y P1Y)))
  2d. deleting eqn   (= P1X (X P1))
  2d. deleting eqn   (= P1Y (Y P1))
4. exit, vars (P1 P1Y)
1. Enter var-defined, var = LENGTH
4. exit, vars (LENGTH)
1. Enter var-defined, var = THETA
  2a. solved eqn     (= SLOPE (TAN THETA))
  2a. solved eqn     (= PHI (- (/ PI 2.0) THETA))
  2a. solved eqn     (= DELTAY (* LENGTH (SIN THETA)))
  2a. solved eqn     (= DELTAX (* LENGTH (COS THETA)))
  3.  repeating step 2.
  2a. solved eqn     (= DELTAY (- P2Y P1Y))
          giving     (= P2Y (+ DELTAY P1Y))
  2b. deleting eqn   (= SLOPE (/ DELTAY (FLOAT DELTAX)))
  2b. deleting eqn   (= SLOPE (/ 1.0 (TAN PHI)))
  2b. deleting eqn   (= LENGTH (SQRT (+ (EXPT DELTAX 2)
                                        (EXPT DELTAY 2))))
  2b. deleting eqn   (= THETA (ATAN DELTAY DELTAX))
  2b. deleting eqn   (= PHI (ATAN DELTAX DELTAY))
  2b. deleting eqn   (= DELTAY (* LENGTH (COS PHI)))
  2b. deleting eqn   (= DELTAX (* LENGTH (SIN PHI)))
  3.  repeating step 2.
  2c. deleting tuple (= P2 (TUPLE (X P2X) (Y P2Y)))
  2d. deleting eqn   (= P2X (X P2))
  2d. deleting eqn   (= P2Y (Y P2))
4. exit, vars (P2 P2Y DELTAX DELTAY PHI SLOPE THETA)
1. Enter var-defined, var = P2X
  2a. solved eqn     (= DELTAX (- P2X P1X))
          giving     (= P1X (- P2X DELTAX))
  3.  repeating step 2.
4. exit, vars (P1X P2X)
```

Variable Dependency Graph

A set of solved equations defines an acyclic variable dependency graph:

Reuse through View

```
(gldefun line-segment-leftof-distance
  ((ls line-segment) (p vector))
    (let ( (dx (deltax ls)) (dy (deltay ls)))
      ( ( dx * ( (y p) - (p1y ls) )
        - dy * ( (x p) - (p1x ls) ) )
        / (sqrt dx ^ 2 + dy ^ 2) ) ))

(gldefun t3 ((l ls1) (p consv))
  (leftof-distance (line-segment l) p))

(LAMBDA (L P)
  (LET ((DX (* (THIRD L) (COS (FOURTH L))))
        (DY (* (THIRD L) (SIN (FOURTH L)))))
    (/ (- (* DX (- (CDR P) (SECOND L)))
          (* DY (- (FIRST P)
                   (- (FIFTH L)
                      (* (THIRD L) (COS (FOURTH L)))))))
       (SQRT (+ (EXPT DX 2) (EXPT DY 2))))))

float lsdist (l, p)
  CLS1 l; CVECTOR p;
  {   float dx, dy;
      dx = l->size * cos(l->angle);
      dy = l->size * sin(l->angle);
      return (dx * (p->y - l->low)
               - dy * (p->x - (l->right
                       - l->size * cos(l->angle))))
             / sqrt(square(dx) + square(dy));  }
```

Storing Basis Variables

A procedure is needed to "store" each basis variable into the concrete data structure. The "stored" variable must not affect the value of any other basis variable.

- *Transfer variables* are variables of the abstract type that directly correspond to concrete data fields.
- Create a set of *basis equations* by running the **`var-defined`** procedure for each basis variable.
- The set **`xfers`** is computed: the subset of transfer variables that depend on the basis variable to be stored.
- The set **`dep`** is computed: the subset of the basis variables that some member of **`xfers`** depends on.
- Create a procedure that computes **`dep`** (minus the variable to be stored) from the concrete data structure, then computes each member of **`xfers`** and stores it.

Data Translation through Views

Given two application types, each of which can be viewed as the same abstract type, a procedure to translate between them can be created.

- Compute each transfer variable of the goal type from the source type.
- Use the GLISP `A` function to create the new data.

```
(transfer-by-view 'ls2 'ls1)

(GLAMBDA ((VAR-LS1 LS1))
  (LET ((VAR-LS1-VIEW (LINE-SEGMENT VAR-LS1)))
    (A LS2 LEFT  (P1X VAR-LS1-VIEW)
           RIGHT (P2X VAR-LS1-VIEW)
           ANGLE (PHI VAR-LS1-VIEW)
           UP    (P2Y VAR-LS1-VIEW))))

(LAMBDA (VAR-LS1)
 (LET ((VAR-LS1-VIEW VAR-LS1))
  (LIST 'LS2
   (- (FIFTH VAR-LS1-VIEW)
      (* (THIRD VAR-LS1-VIEW) (COS (FOURTH VAR-LS1-VIEW))))
   (FIFTH VAR-LS1-VIEW)
   (- 1.5707963267948966 (FOURTH VAR-LS1-VIEW))
   (+ (* (THIRD VAR-LS1-VIEW) (SIN (FOURTH VAR-LS1-VIEW)))
      (SECOND VAR-LS1-VIEW)))))
```

VIP: Programming by Graphical Connections

Related techniques make it possible to write scientific programs and solve physics problems.

- The initial diagram consists of boxes that represent input data and an `output` box.
- Physical laws and geometric models may be selected by menu and added to the picture.
- Connections may be made between diagram buttons and data.
- A program is generated from the diagram by symbolic data flow:
  - Initially, input data and constants are defined.
  - When a data item becomes defined, its value is propagated into boxes to which it is connected.
  - When a value is propagated into a box, equations associated with the box are examined to see whether they can be solved for other variables.
  - Code is generated as a side-effect of the data flow.
- Units of measurement are converted automatically.

Calculation of the Mass of the Sun

```
result type: (UNITS REAL KILOGRAM)

(LAMBDA () 1.9660057055546021E30)
```

Aircraft Position from Radar Data

```
(vip
  '((TIME-DIFF         (UNITS INTEGER (* 100 NANOSECOND)))
    (AIRCRAFT-ALTITUDE (UNITS INTEGER (* 10 FOOT)))
    (RADAR-ALTITUDE    (UNITS INTEGER (* 10 FOOT)))
    (RADAR-ANGLE       (UNITS INTEGER (/ (* 2 PI RADIANS)
                                          4096)))
    (RADAR-UTM         UTM-VECTOR))
```

Aircraft Position Program

```
(LAMBDA (TIME-DIFF: (UNITS INTEGER (* 100 NANOSECOND))
         AIRCRAFT-ALTITUDE: (UNITS INTEGER (* 10 FOOT))
         RADAR-ALTITUDE: (UNITS INTEGER (* 10 FOOT))
         RADAR-ANGLE: (UNITS INTEGER (/ (* 2 PI RADIANS)
                                        4096))
         RADAR-UTM:UTM-CVECTOR)
  (LET (OUT7 OUTPUT D3 OUT8 X5 Y3 X6 RELPOS:UTM-CVECTOR)
    (OUT7 = (- AIRCRAFT-ALTITUDE RADAR-ALTITUDE))
    (D3 = (* '(Q 2.997925E8 (/ M S)) TIME-DIFF))
    (OUT8 = (/ D3 2))
    (X5 = (SQRT (- (EXPT OUT8 2) (EXPT OUT7 2))))
    (Y3 = (* X5 (SIN RADAR-ANGLE)))
    (X6 = (* X5 (COS RADAR-ANGLE)))
    (RELPOS = (A UTM-CVECTOR NORTH Y3 EAST X6))
    (OUTPUT = (+ RELPOS RADAR-UTM))
    OUTPUT))
```

Aircraft Position Program in C

```
CUTM  *tqc (time_diff, aircraft_altitude, radar_altitude,
            radar_angle, radar_utm)
  long time_diff, aircraft_altitude, radar_altitude,
       radar_angle;
  CUTM  *radar_utm;
  {
    long out7;
    CUTM  *output;
    float d3, out8, x5, y3, x6;
    CUTM  *relpos,  *glvar4796;
    out7 = aircraft_altitude - radar_altitude;
    d3 = 2.997925E8 * time_diff;
    out8 = d3 / 2;
    x5 = sqrt(square(out8) - 9.290303999999988E14
                             * lsquare(out7));
    y3 = x5 * sin(0.0015339807878856412 * radar_angle);
    x6 = x5 * cos(0.0015339807878856412 * radar_angle);
    relpos = (CUTM*) malloc(sizeof(CUTM));
    relpos->north = 1.0000000000000001E-7 * y3;
    relpos->east = 1.0000000000000001E-7 * x6;
    glvar4796 = (CUTM*) malloc(sizeof(CUTM));
    glvar4796->east = relpos->east + radar_utm->east;
    glvar4796->north = relpos->north + radar_utm->north;
    output = glvar4796;
    return output;
  }
```

Language Translation

Specialized procedures can be delivered in several languages: Lisp, C, C++, Java, Pascal.

- Lisp is equivalent to abstract syntax trees used by compilers
- Data structures in other languages can be specified to GLISP
- Simulated data structures in Lisp can be used for rapid prototyping
- Program transformation:
  - Transform Lisp features, e.g. returning a value from `if` statement
  - Patterns transform Lisp idioms to target idioms
- Syntactic transformation: names, types, patterns
- Printing program produces formatted, readable code.

Automatic Programming Server

The Automatic Programming Server operates over the Web and writes specialized procedures for the user.

- User describes concrete data types
- User makes views showing how concrete types correspond to abstract types known to the system
- User selects desired procedures
- Procedures are specialized, translated to target language, and delivered as a source code file.
- Data conversion programs can also be generated.
- Example: `avl-tree`: 200 lines of generated code in less than a minute of user time.

Automatic Programming Server

- User defines a `part` data structure:

```
(part (name string)
      (size integer)
      (next (^ part)))
```

- User makes a view as a sorted linked list
- User chooses desired procedures

APS: Example

GPS: Component Programming

A natural extension to the Automatic Programming Server, which makes individual specialized components, is to make whole programs by connecting components.

Graphical Programming System (GPS) allows composition of programs involving iteration, accumulators, and data structures. The goal is to produce programs similar to those written by humans.

Producing a program from generic components requires many view types with complex property definitions. A key feature of GPS is propagation and inference of types and properties to reduce the amount of user input and assist the user in making choices.

A Simple Program

Given as input a list of sublists, each of which has the format `(key n)`, sum the `n` for each `key`.

```
(alsum '((a 3) (b 2) (c 1) (a 5) (c 7)))

    =  ((C 8) (B 2) (A 8))
```

```
(defun alsum (lst)
  (let ((alist nil) entry)
   (dolist (item (identity lst))
     (setq entry
       (or (assoc (car item) alist)
           (progn (push (list (car item) 0 ) alist)
                  (assoc (car item) alist))))
     (incf (cadr entry) (cadr item) ) )
   alist))
```

Different fonts are used for code associated with different abstract components. Clearly, code of the components is thoroughly mixed in this program.

Program Components

This small program combines several generic components:

- Iterate-Accumulate: Iterate through a sequence of items, accumulating some aspect of the items.
- Find-Update: Find an item in an indexing structure using its key value, then update the record with information from the item.
- dolist: an iterator for linked lists in Lisp.
- `alist`: association list, a kind of database structure.
- **Sum**: a kind of accumulator.

In addition, there are several interfaces of *glue code*: how to get the sequence from the input, how to get the key from the item, what aspect of the item to accumulate.

Substitution Test

A *substitution test* is useful to determine what the generic components of a program are:

> Can the program be modified by a substituting a different component for an existing one, without changing the other components?

In this example, we could produce different but related programs by substitution of components:

- If the input were an array rather than a linked list, a different iterator would be used.
- A different database structure could be used, e.g. an AVL tree rather than an Alist.
- The product of the integers could be accumulated rather than their sum.
- By changing the glue code, we could change the key and accumulated value, e.g. to accumulate symbols associated with each numeric value.

Generic Software Components

We claim that to be effectively reusable, software components must be generic. Otherwise:

- Each decision that is hard-coded into a component becomes a requirement on the use of that component.
- There will be a combinatoric number of components based on combinations of decisions.
- It will be hard to find the right component among the many available.
- Learning the requirements of a component becomes a burden on the user.
- Conforming to the requirements reduces the benefit of reuse.
- Changes will require changes of components.

Instead, there should be *only one* generic version of each component, which should serve for all uses in all languages.

Generic Component: Views

A simple generic component such as `sum` involves several types, mappings, and a constant:

- T_{acc}, accumulator type. `sum` is defined for any type that defines a `+` operator, including `integer`, `real`, `boolean`, `string`, and any application type that defines `+`.
- Constant $C_0 : T_{acc}$, initial value of the accumulator. C_0 defaults to the `zero` value of the type T_{acc}; it is possible to specify a `sum` with a non-zero initial value (e.g. a taxi meter).
- T_{rec}, accumulator record type. A component such as `sum` never "owns" its storage; instead, the storage is owned by some outside component and passed to `sum` as a parameter.
- A mapping $M_{acc} : T_{rec} \rightarrow T_{acc}$ that specifies how to obtain the data for this instance of `sum` from data of type T_{rec}. This mapping must allow both reading and writing of the accumulator value. There could be multiple instances of `sum` data in a T_{rec}, e.g., it might be desired to sum both cost and shipping weight at the same time.

Generic Component: Views ...

- View type V_{rec}, which wraps T_{rec} and defines M_{acc}, C_0, and the name of V_{item}. V_{rec} lists `myadder` as a superclass so that it inherits generics that do the work of `sum`.
- T_{item}, type of the item from with the summand is derived.
- A mapping $M_{item} : T_{item} \rightarrow T_{acc}$ that specifies how to obtain the data to be summed from each item of type T_{item}.
- A predicate $P_{item} : T_{item} \rightarrow \{T, F\}$ that determines whether an item should be summed. P_{item} defaults to T.
- View type V_{item}, which wraps T_{item} and defines M_{item} and P_{item}.

Clearly, it is necessary to produce most of this automatically, or the component will be too hard to reuse.

Program Representation

A program is represented as a network of *component instances*. A *component specification* describes a class of instances.

Components are connected via *interfaces*, each with multiple parameters, often bi-directional.

A component specification contains:

- Descriptions of interfaces of the component
- Descriptions of types used within the component
- Descriptions of properties, such as the `summand` of `sum`
- Specifications for making views from the completed component instance.

Component Specification

```
(defcspec
 '(sum   abbrev add
    interfaces ((accout accumulator offers
                    ((item item in)
                     (acc  sumd  out)
                     (outt conttype out))))
         types ((item anything t)
                (conttype (hasop +) nil)
                (sumd (sum conttype) myadder))
         props
          ((test           (prop item boolean)
                                  () true)
           (summand        (prop item conttype))
           (initial-value  (constant conttype) ()
                             (zero (a conttype)))
           (dataview       (viewtype item))
           (accum          (choice sumd sum)) )
       otspecs ((item (test summand))
                (sumd (initial-value dataview
                          accum)))  ))
```

Inference and Propagation

Inference of types and properties occurs within component instances, guided by component specs. Most properties are inferred automatically; in some cases, the user is asked via menu.

When a type or value that is part of an interface is inferred, it is propagated across the interface to the other component, where it may cause additional inferences. *cf.* constraint propagation in Waltz filtering and MOLGEN.

Type Propagation Example

Assume that the `sum` is the last component added; the following steps occur:

- When the summand is selected, the type `integer` becomes the accumulator type for the `sum`.
- The `sum` sends its accumulator type to `find-update`.
- `find-update` sends its accumulator type to `alist`.
- `alist` infers its element type (a combination of key and accumulator) and its data structure type (a list of elements) and sends its data structure type back to `find-update`.
- `find-update` sends the data structure type to `iter-acc` as its accumulator type.
- `iter-acc` will declare a variable to hold the alist.

Modifying Properties

Properties such as the `test` or `summand` can be specified in several ways:

- by specifying the name of a function to compute the property
- by entering a bit of Glisp code, e.g.

  ```
  (> (size self) 3)
  ```

  using `self` to refer to the item being tested
- by entering a graphical specification of a function using VIP

Partial Evaluation

- Evaluation of `cast` and `funcall` when the type and "function" arguments are constant, as described above.
- Evaluation of operations on constants.
- Simplification of `if` statements when the test value is known.
- Unrolling of `for` loops when the loop is over a constant list. This is used for accumulators which have the arity `*`; it has the effect of turning a loop over the declarative set of accumulators into a sequence of straight-line code that updates each accumulator individually.

Updating a `sum`

```
(glambda (self item)
  (if (test (cast item (dataview self)))
      ((accum self) _+
         (summand (cast item (dataview self))))))
```

Generic Iterate-Accumulate

```
(for x in (seq arg)
  (for ac in (accfields arg)
    (update (funcall ac acc) x)))
```

C Example

```
float gpfn2 (struct assemblyc *arg)
  {
    float acc;
    acc = 0.0;
    {
      struct partc *ptr2; struct partc *x;
      struct partc *ptrnext;
      ptr2 = NULL;
      ptr2 = arg->ptr;
      while ( ptr2 != NULL )
          {
            ptrnext = ptr2->next;
            x = ptr2;
            if (x->size > 2)
              acc += x->weight;
            ptr2 = ptrnext;
          }
    }
    return acc;
  }   /* gpfn2 */
```

Software Reuse

Approaches to automatic programming:

- **Power-based:** derive program from a small specification. Typically, this approach uses a small number of axioms and deep inference.
- **Knowledge-based:** reuse knowledge that is stored in the form of programs, equations, facts, rules, etc. Typically, this approach uses a large amount of knowledge and shallow inference.

The power-based approach hasn't gotten very far. That leaves reuse.

Krueger's Survey of Software Reuse [17]

Krueger emphasizes *cognitive distance*: the difference between the problem and the expression of the problem in the programming formalism. Cognitive distance is a rough estimate of the intellectual effort required to use an approach.

Emphasis on cognitive distance recognizes an important fact:

Programming is an activity performed by humans.

Cognitive distance is reduced in two ways:

1. *Abstractions* in the reuse technique make it easier to go from a concept of the system to representation in the reuse technique.
2. *Automation* reduces the effort to get from the abstract representation to executable code.

[17]**Krueger, Charles W.**, "Software Reuse", *ACM Computing Surveys*, vol. 24, no. 2 (June 1992), pp. 131-184.

Software Reuse

- Proposed in print in 1968 by McIlroy: *Mass Produced Software Components*, NATO Software Engineering Conference.
- It is still not standard practice in software construction.
- There is renewed interest in how reuse can be made effective.

The term *reuse* applies to software construction. Repeated execution or making distribution copies of the same software do not count as reuse.

Evaluation of Reuse Approaches

An approach to reuse can be characterized along several dimensions:

1. **Abstraction:** Abstraction is essential: without it, the artifact to be reused would have to match the new problem exactly.
2. **Selection:** How does the user locate the desired artifact?
3. **Specialization:** If an artifact is abstract, it must be specialized for the application. Specialization can be done by parameterization, transformation, or constraints.
4. **Integration:** How can reusable artifacts be integrated into a complete system? An *integration framework* such as a *module interconnection language* may be provided.

Abstraction

> "Successful application of a reuse technique to a software engineering technology is inexorably tied to raising the level of abstraction for that technology."

Every software abstraction has two levels:

1. *Specification*
2. *Realization*

If there are multiple levels, the realization of one level may be the specification of the next lower level.

An abstraction has several parts:

1. *hidden part:* details of the realization that are not visible in the specification
2. *variable part:* the part of the specification that can be varied
3. *fixed part:* the part of the specification that is fixed

Minimizing Cognitive Distance

For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.

- Use fixed and variable abstractions that are both succinct and expressive.
- Maximize the hidden part of abstractions.
- Automate the translation from specification to realization.

High-level Languages

- Perhaps the most successful reuse technology
- Factor of 5 speedup in programming [Brooks 75].
- Abstraction: instruction patterns (*e.g.*, loops, **`if-then-else`**, array references).
- Selection: memorize the programming language.
- Specialization: fill in the slots in the instruction pattern.
- Integration: link editor
- Cognitive distance: often large. System design must precede coding; low level of abstraction.

Design and Code Scavenging

Scavenging: copy as much as possible from existing design or code, modify it for current application.

- CS education is design scavenging! Programmer knows a library of reusable designs.
- Reuse works better if code is designed for reuse; but it is hard to get people to do this.
- Abstraction: designs or source code fragments represent informal concepts.
- Selection: informal. Programmer must remember or find reusable code.
- Specialization: manually edit code. This may introduce errors; cannot “reuse” validation.
- Integration: modify code and/or context of use.
- Cognitive distance: may be large unless reusing one’s own code.

For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.

Source Code Components

Off-the-shelf components used as building blocks for larger software.

- Examples: `sort` verb in COBOL, library functions such as `sqrt`, Lisp system functions, IMSL library.
- Booch components (based on abstract data types) are used somewhat.
- Inheritance in object-oriented programming can be considered to be in this category.
- Abstraction: must provide specifications (aside from code) of what the components do.
- Selection: Manuals not so good for selection; need good indexes and automation.
- Specialization: Edit code or parameterize, *e.g.* by specifying a “contents” type. Problem: how much parameterization is possible?
- Integration: Name conflicts may be a problem. Unix pipes are another form of integration.
- Cognitive distance: small if user already understands the concept (*e.g.*, matrix inversion).

Software Schemas

- Similar to reuse of code, but more abstract.
- A wider range of things can be parameterized: code, data types, etc.
- Abstraction: abstracts code, data structures.
- Selection: there may be machine help in finding possibly applicable modules. Must then prove preconditions.
- Specialization: substituting language constructs into parameterized parts of schema, or choosing options.
- Integration: done in implementation language, or by composing schemas.
- Cognitive distance: abstraction allows description of *what* rather than *how*. However, formal description of even simple abstractions is difficult.

Application Generators

- Similar to compilers
- Input specifications are high-level, special-purpose abstractions from a narrow application domain.
- Code expansion ratio can be high.
- Examples: database generators, report generators, compiler generators.
- Abstraction: reuse the system architecture, major subsystems, data structures and algorithms.
- Selection: Choose an application generator for application domain.
- Specialization: generated from the input specification.
- Integration: none if single system is generated. In terms of domain abstractions (*e.g.*, tokens passed between LEX and YACC) otherwise.
- Cognitive distance: Low when fit to application domain is good.

Very-high-level Languages

- Also called *executable specification languages*.
- May sacrifice speed for generality ("slow is beautiful").
- "Very high level" components include sets, tuples, maps, history.
- Abstraction: sets or constraints
- Selection: select appropriate VHLL, language constructs.
- Specialization: replace high-level constructs with implementations. If human-guided, can provide good performance.
- Integration: functional (side-effect-free) languages simplify integration.
- Cognitive distance: low if the problem fits the abstractions provided.

Transformational Systems

- Software described in a high-level description language.
- Transformations enhance efficiency and move toward executable implementation, without changing semantics.
- Separation of “what” from “how”.
- Abstraction: prototypes, development histories, transformations
- Selection: expert system rules may help selection.
- Specialization: done by transformation
- Integration: analogous to functional composition.

Software Architectures

- Large-grain software frameworks for global structure of software system.
- Examples: database, compiler (*e.g.* Draco), blackboard architecture, user interface architecture.
- Abstraction: from application domains
- Selection: similar to software libraries
- Specialization: source-to-source transformations, component refinements
- Integration: integrates components into a single framework.
- Cognitive distance: small when used appropriately.

Krueger’s Truisms of Software Reuse

1. For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.
2. For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.
3. To select an artifact for reuse, you must know what it does.
4. To reuse a software artifact effectively, you must be able to “find it” faster than you could “build it.”

Refine

Refine is a "wide-spectrum" language: it includes sets, mappings, relations, predicates, enumerations, state transformation sequences, and both declarative and procedural statements.

A high-level program in **Refine** is refined into executable code by applying transformation rules.

Optimization: *e.g.*, move a predicate outside a quantifier. For example, if p does not depend on x,

$$\forall x[p \wedge q \Rightarrow r] \quad \rightarrow \quad p \Rightarrow \forall x[q \Rightarrow r]$$

Cf. moving invariant computations out of loops.

Refinement: move code closer to executable form, *e.g.*, convert a bounded quantifier into a loop, or implement a set by means of a data structure.

Form of Rules

Rules can be specified in abbreviated form as $P \rightarrow Q$, or more formally:

$$\forall x_1 ... x_n [P(s) \Rightarrow Q(succ(s))]$$

Example:
$class(a) = set \quad \wedge \quad element(a) = x$
$\rightarrow \quad class(a) = mapping \quad \wedge \quad domain(a) = x \quad \wedge \quad range(a) = boolean$

or $a : `set\ of\ x' \quad \rightarrow \quad a : `mapping\ from\ x\ to\ boolean'$

Possible implementations of a set:

Operations $\in \quad \cap \quad \cup \quad \forall \quad \exists$ will be different for each possible implementation.

Refinements: Logic to Code

- Conjuncts with no unknown variables become tests.
- Conjuncts with one unknown variable become computations to find the values(s) of the variable.
- Implications become `if` statements.
- Substitutions become variable bindings.
- Bounded quantifiers become enumerations.

KIDS

"KIDS [Kestrel Interactive Development System] is basically a program transformation system – one applies a sequence of consistency-preserving transformations to an initial specification and achieves a correct and hopefully efficient program."[18]

Program development in KIDS follows these steps:

- Develop a *domain theory*:
  - types
  - functions
  - distributive and monotonicity laws
- Create a *specification* of the program.
- Apply a *design tactic*: divide and conquer, global search (binary search, backtrack, branch-and-bound), local search (hill-climbing).
- Apply *optimization*: simplification, partial evaluation, finite differencing.
- Apply data type *refinements*.
- *Compile* to executable form.

[18]Smith, Douglas R., "KIDS: A Semiautomatic Program Development System", *IEEE Trans. on Software Engineering*, vol. 16, no. 9, Sept. 1990, pp. 1024-1043.

REFINE: Sets

- Type declaration: $S : set(Nat)$
- Literal sets: $\{1, 2, 4\}$ $\{2..5\}$
- Set former: $\{f(x) \mid P(x)\}$
- Comparison predicates: $= \neq \in \notin \subseteq$
- Reduction: $reduce(op, S)$
- Element addition, deletion: $S + x \quad S - x$

REFINE: Sequences

- Type declaration: $A : seq(integer)$
- Empty sequence: $[\,]$
- $empty(A) : A = [\,]$
- Element: $A(i)$
- Comparison predicates: $= \neq \in \notin$
- $domain(A) = 1..length(A)$
- $range(A) = \{A(i) \mid i \in domain(A)\}$
- $length(A)$
- $first(A)$: $A(1)$
- $rest(A)$
- $append(A, x)$: insert x at the end of A.
- $concat(A, B)$: concatenate sequences A and B.

KIDS Specifications

A *specification* is a quadruple: $F \ = < D, R, I, O >$

D	Input type: *domain*
R	Output type: *range*
$I : D \rightarrow boolean$	Input condition: *assumptions*
$O : D \times R \rightarrow boolean$	Output condition: *feasible solutions*

We seek a constructive proof of the theorem:

$$\forall_{x \in D} \; I(x) \;\Rightarrow\; \exists_{z \in R} \, O(x, z)$$

In a program-like form, a specification can be written:

```
**function** $F(x : D) : set(R)$
    **where** $I(x)$
    **returns** $\{z \mid O(x, z)\}$
    $= Body$
```

Body is code that can be executed to compute F. A spec is *consistent* if:

$$\forall_{x \in D} \; I(x) \;\Rightarrow\; F(x) = \{z \mid O(x, z)\}$$

Directed Inference

Directed inference transforms a *source* formula into a *target* formula.

Conditional patterns are used to perform inference of several kinds:

$$C \;\Rightarrow\; (s \rightarrow t)$$

where C is a condition, s is source, t is target, and $\rightarrow$ is one of the directions:

$\Rightarrow$	forward inference
$\Leftarrow$	backward inference
$=$	simplification
$\geq$	deriving a lower bound
$\leq$	deriving an upper bound

Directed inference uses logic to help *construct* a program, not just to prove it correct.

The programmer selects an expression and the kind of operation to be performed. Problem: *replaying* a derivation if the spec is changed.

Mappings

A mapping $f: M \rightarrow S$ can be classified as:

injective	1-1	$f(a) = f(b) \Leftrightarrow a = b$
surjective	onto	$Range(f) = S$
bijective	1-1 and onto	

$$\begin{aligned} &injective(M : seq(integer),\ S : set(integer)) : boolean \\ &= range(M) \subseteq S \\ &\quad \wedge \forall(i, j)(i \in domain(M) \ \wedge \ j \in domain(M) \\ &\qquad\quad \wedge i \neq j \ \rightarrow \ M(i) \neq M(j)) \end{aligned}$$

$$\begin{aligned} &bijective(M : seq(integer), S : set(integer)) : boolean \\ &= injective(M, S) \wedge range(M) = S \end{aligned}$$

Example: k Queens

The goal is to put k queens on a $k \times k$ chess board so that no two queens can attack each other.

A solution is represented as a sequence $assign$ where $assign(i)$ is the row of the queen in the i column: $[3, 1, 4, 2]$

$$\begin{aligned}&no\_two\_queens\_per\_up\_diagonal(S : seq(integer)) : boolean\\ &= \forall_{i,j}(i \in domain(S) \wedge j \in domain(S) \wedge i \neq j\\ &\quad \Rightarrow (S(i) - i \neq S(j) - j))\end{aligned}$$

$$\begin{aligned}&no\_two\_queens\_per\_down\_diagonal(S : seq(integer)) : boolear\\ &= \forall_{i,j}(i \in domain(S) \wedge j \in domain(S) \wedge i \neq j\\ &\quad \Rightarrow (S(i) + i \neq S(j) + j))\end{aligned}$$

Comment: These formulations are *programming*:

- there could be bugs in the domain theory, just as there could be bugs in the corresponding programs.
- skill is needed to formulate the problem in a way that can be turned into an efficient program.

Distributive Laws

$$\forall_S(injective([\,], S) = true)$$

$$\begin{aligned}&\forall_{W,a,S}(injective(append(W, a), S)\\&\quad = (injective(W, S) \wedge a \in S \wedge a \notin range(W)))\end{aligned}$$

$$\begin{aligned}&\forall_{W1,W2,S}(injective(concat(W1, W2), S)\\&\quad = (injective(W1, S) \wedge injective(W2, S)\\&\quad \wedge range(W1) \cap range(W2) = \{\}))\end{aligned}$$

Example: k Queens

```
function $Queens(k : integer) : set(seq(integer))$
  where $1 <= k$
  returns
    $\{assign \mid bijective(assign, \{1..k\})$
     $\wedge\ no\_two\_queens\_per\_up\_diagonal(assign)$
     $\wedge\ no\_two\_queens\_per\_down\_diagonal(assign)$
    $\}$
```

Global Search

Global search can be considered to be a tree search:

- Nodes of the tree are *intensional descriptions* of sets of solutions. Symbols with a hat, e.g. $\hat{r}$, refer to such intensional descriptions.
- Branches correspond to splitting sets into subsets.
- Filters prune set descriptions that contain no solutions.

Global Search Theory

Global search theory is based on operations:

- **Split:** split a set of candidate solutions
- **Extract:** extract solutions from a set
- **Filter:** eliminate sets with no solutions.

Good performance depends on a good filter.

Ideal filter:

$$\exists(z : R)(Satisfies(z, \hat{r}) \wedge O(x, z))$$

Necessary filter: Φ

$$\exists(z : R)(Satisfies(z, \hat{r}) \wedge O(x, z)) \Rightarrow \Phi(x, \hat{r})$$

Any necessary filter Φ will do (even True). Skill is needed to make a good necessary filter Φ. A stronger Φ can prune more branches, but may be more expensive to compute.

Global Search Theory

A global search theory can be converted to a program as follows, where Φ is a necessary filter

Queens as Global Search

The Queens problem can be mapped to the global search theory as follows:

Initial Queens Program

Simplification

Simplification allows elimination of conditions such as:
if $injective([\,], \{1..k\}) \wedge ...$

Context-Dependent Simplification

Part of the specification can be simplified based on context, e.g. output conditions that unify with input conditions.

Partial Evaluation

Partial evaluation and unfolding (inlining) function definitions results in:

Finite Differencing

Finite differencing replaces expensive operations with incremental computations.

$i \notin range(part\_sol) \wedge i \in \{1..k\}$ becomes
$i \in setdiff(\{1..k\}, range(part\_sol))$

Finite differencing replaces the $setdiff$ operation with a new variable, $unoccupied\_rows$.

Algorithm after Finite Differencing

Case Analysis

Case analysis recognizes that the two cases of this algorithm are disjoint and replaces them with an **if** statement.

Data Type Refinement

Data type refinement replaces sets with data structures. Selecting the right data structures has a large effect on efficiency.

Results

Optimizations performed using KIDS improved performance dramatically: from 3600 seconds to less than one second.

The user made 16 high-level design decisions, taking 15 minutes for derivation.

KIDS has been used to derive programs for job scheduling, enumerating cyclic difference sets, graph coloring, bin packing, binary search, vertex covers of a graph, linear programming, maximum segment sum, sorting.

Goguen: Theory

```
theory POSET is
   types ELT
   functions <=: ELT ELT -> BOOLEAN
   vars E1 E2 E3 : ELT
      axioms (E1 <= E1)
             (E1 <= E3 if E1 <= E2 AND E2 <= E3)
             (E1 =  E2 if E1 <= E2 AND E2 <= E1)
end POSET
```

‘View’ as used by Goguen

A *view* of an entity **A** as a theory **T** consists of a mapping from the types of **T** to the types of **A** and a mapping from the operations of **T** to the operations of **A** that preserves arity (the list of argument types), value type (if any), and operation attributes such as **assoc**, **comm**, and **id:** (if any) such that the translation of every axiom in **T** is satisfied by **A**.

```
view NATD :: POSET => NATURAL is
   types (ELT => NATURAL)
   ops   (<=  => DIVIDES)
end NATD
```

Deductive Composition of Astronomical Software from Subroutine Libraries [19] [20]

Amphion: Compose programs from a subroutine library, based on a graphical specification, using deduction.

SPICE: subroutine library for solar-system geometry.

- Various systems of time: ephemeris time, spacecraft clock time, etc.
- Various frames of reference
- Light does not travel instantaneously over astronomical distances

Example task: observe the position of a moon of a nearby planet to determine position of the spacecraft.

[19] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood: "Deductive Composition of Astronomical Software from Subroutine Libraries", *Proc. 12th Int. Conf on Automated Deduction (CADE'94)*, Nancy (France), June 994, *LNAI 814*, Springer Verlag, pp. 341-355.

[20] Steve Roach and Jeffrey Van Baalen, "Experience Report on Automated Procedure Construction for Deductive Synthesis", *Proc. Automated Software Engineering Conf.*, Sept. 2002, pp. 69-78.

Difficulty of Programming

- Subroutines may not be well documented
- User must understand documentation
- Many subroutines: takes time to become familiar with the collection
- User may rewrite subroutine rather than reusing it
- User might make mistakes, e.g. wrong type of units of argument

Constructive proof: given a theorem $\forall x \exists y\ P(x, y)$, prove it by constructing a y that satisfies the theorem.

During program synthesis, *witnesses* are constructed for existential terms. The witnesses correspond to subroutines in the SPICE library (*concrete* terms).

$specification \rightarrow theorem \rightarrow proof \rightarrow program$

Domain Theory

A *domain theory* provides a logical language for the application domain:

- Time: *time* is *abstract*, but has several *concrete* representations (ephemeris time, UTC, spacecraft clock time)
- Points, rays, planes
- Photon travel
- Events (space-time points)
- Celestial bodies, e.g. Saturn
- Axioms that relate abstract and concrete terms.

$$\forall tc\ (=\ \begin{array}{l}(absctt\ UTC\ tc)\\ (absctt\ Ephemeris\ (UTC2Ephemeris\ tc)))\end{array}$$

$$\forall tc\ (=\ \begin{array}{l}(absctt\ Ephemeris\ tc)\\ (absctt\ UTC\ (Ephemeris2UTC\ tc)))\end{array}$$

absctt abstracts from a time system and time coordinate to an abstract time. These axioms specify what the conversion functions such as *Ephemeris2UTC* do.

Representation conversions are combinatorially explosive because they can loop.

Events

An *event* is a combination of a point in space and a time.

lightlike? is a relation that says that a ray of light leaving one event will arrive at the other event.

Given the location and time of the origin ($event_1$), the function *recd* computes the time at the destination ($event_2$) such that ($lightlike?\ event_1\ event_2$). The function *sent* computes the time of sending in $event_1$ to match the time at $event_2$.

```
∀ onid dnid ets
  (lightlike?
    (obj&time2event (absid onid)
                (absctt Ephemeris ets))
    (obj&time2event (absid dnid)
                (absctt Ephemeris (recd onid dnid ets))
```

SNARK Theorem Prover

The SNARK theorem prover has the following features:

- Resolution
- Paramodulation for handling equality
- Mathematical induction (results in recursive programs)
- Manna and Waldinger's *deductive tableaux* framework
- Proofs can be restricted to be constructive.

Axioms

Two kinds of axioms are potentially combinatorically explosive:

- Representation conversion, e.g. convert between UTC-Calendar time (string "YYYY MMM DD HH:SS") and Ephemeris time (a double float)
- $lightlike$? axioms

Both allow many possible conversion paths, and perhaps infinite loops.

Astronomical Domain

About 200 axioms are used to describe the domain of astronomy.

- *lightlike*?(e_1, e_2) holds if a photon could leave the event (position and time) e_1 and arrive at event e_2.
- *ephemeris-object-and-time-to-event* yields an event corresponding to the position of a given astronomical object (planet or spacecraft) at a given time.
- *a-sent*(o, d, ta) computes the time a photon must leave object o to arrive at destination d at time ta.
- Axiom *lightlike?-of-a-sent*:

```
(all (o d ta)
  (lightlike?
    (ephemeris-object-and-time-to-event
      o (a-sent o d ta))
    (ephemeris-object-and-time-to-event d ta)))
```

  - `o` = origin
  - `d` = destination
  - `ta` = time of arrival

Problem Difficulty

The program out-performs human experts and significantly out-performs non-experts:

- Expert who knows subroutine library: 30 minutes
- Non-expert: several days
- Program: 3 minutes

Time to construct specification:

- Expert: a few minutes
- Non-expert: 30 minutes

Where is the shadow of Io on Jupiter?

Dotted lines indicate photon motion connections, i.e. *lightlike*?.

Shadow of Io Theorem

```
(all (time-voyager-2-c)
 (find (shadow-point-c)
  (exists
   (time-sun sun-spacetime-loc time-io io-spacetime-loc
    time-jupiter jupiter-spacetime-loc time-voyager-2
    voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
    ray-sun-to-io)
  (and
   (= ray-sun-to-io
      (two-points-to-ray
        (event-to-position sun-spacetime-loc)
        (event-to-position io-spacetime-loc)))
   (= jupiter-ellipsoid
      (body-and-time-to-ellipsoid jupiter time-jupiter))
   (= shadow-point
      (intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
   (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
   (lightlike? io-spacetime-loc jupiter-spacetime-loc)
   (lightlike? sun-spacetime-loc io-spacetime-loc)
   (= voyager-2-spacetime-loc
      (ephemeris-object-and-time-to-event voyager-2 time-voyager-2))
   (= jupiter-spacetime-loc
      (ephemeris-object-and-time-to-event jupiter time-jupiter))
   (= io-spacetime-loc
      (ephemeris-object-and-time-to-event io time-io))
   (= sun-spacetime-loc
      (ephemeris-object-and-time-to-event sun time-sun))
   (= shadow-point (abs (coords-to-point j2000) shadow-point-c))
   (= time-voyager-2
      (abs ephemeris-time-to-time time-voyager-2-c))))))
```

Shadow of Io Program

```
SUBROUTINE SHADOW ( TIMEVO, SHADOW )
DOUBLE PRECISION TIMEVO          ...
INTEGER JUPITE
PARAMETER (JUPITE = 599)         ...
DOUBLE PRECISION RADJUP ( 3 )    ...
CALL BODVAR ( JUPITE, 'RADII', DMY0, RADJUP )
TJUPIT = SENT ( JUPITE, VOYGR2, TIMEVO )
CALL FINDPV ( JUPITE, TJUPIT, PJUPIT, DMY20 )
CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
TIO = SENT ( IO, JUPITE, TJUPIT )
CALL FINDPV ( IO, TIO, PIO, DMY30 )
TSUN = SENT ( SUN, 10, TIO )
CALL FINDPV ( SUN, TSUN, PSUN, DMY40 )
CALL VSUB ( PIO, PSUN, DPSPI )
CALL VSUB ( PSUN, PJUPIT, DPJPS )
CALL MXV ( MJUPIT, DPSPI, XDPSPI )
CALL MXV ( MJUPIT, DPJPS, XDPJPS )
CALL SURFPT ( XDPJPS, XDPSPI, RADJUP ( 1 ),
              RADJUP, RADJUP ( 3 ), P, DMY90 )
CALL VSUB ( P, PJUPIT, DPJUPP )
CALL MTXV ( MJUPIT, DPJUPP, SHADOW )
END
```

Theorem Proving

SNARK uses term rewriting based on a *recursive path ordering*:

- Replace abstract, non-computable terms with concrete, computable ones.

```
(all (onid dnid eta)
  (= (a-sent (abs naif-id-to-body onid)
             (abs naif-id-to-body dnid)
             (abs ephemeris-time-to-time eta))
    (abs ephemeris-time-to-time
         (sent onid dnid eta))))
```

SNARK uses an agenda for ordering formulas to be proved. Agenda-ordering strategy: give priority to $lightlike?(c, v)$ involving a constant c and variable v.

Performance of the prover was a continuing problem and required hand-tuning to keep proofs from taking too long.

Result of Ordering

Ordering in the theorem prover produces the strategy:

- First find space-time location of all bodies
- Replace abstract function symbols with concrete ones (library subroutines).

Set-of-Support

Mathematical proof usually involves deep proofs with few axioms.

The astronomical domain involves *shallow proofs* with *many axioms*. Set-of-support focuses on goals rather than axioms.

Problem: combination of restrictions loses completeness.

- Set-of-support
- Recursive path ordering
- Constructiveness

They added redundant axioms as a stopgap measure.

Performance

Overall system performance is a win:

- Easy to use, even by novices.
- Easier to revise a stored specification than to make a new one.
- Easy to expand axiom set for new subroutines.

Most programs produced are 2-3 pages of Fortran, consisting mainly of declarations and subroutine calls. There are no `if` statements or loops.

Applications of Amphion

- Amphion/NAIF: solar system geometry
- Amphion/CFD: computational fluid dynamics
- Amphion/TOT: space shuttle navigation

Software Synthesis in Mechanical CAD [21]

Synthesis: *What to do* $\rightarrow$ *How to do it*

Geometric Constraint Satisfaction:

- Given: a set of geometric bodies or *geoms*
- Given: *constraints* among the bodies
- Find: *configuration* of the bodies (position, orientation, dimension) that satisfies all the constraints.

Solutions to this problem are needed in:

- Constraint-based sketching and design
- Geometric modeling for CAD (computer-aided design)
- Kinematics analysis of robots and other mechanisms
- Describing mechanical assemblies

[21]Sanjay Bhansali and Tim J. Hoar, "Automated Software Synthesis: An Application in Mechanical CAD", *IEEE Trans. Software Engineering*, vol. 24, no. 10 (Oct. 1998), pp. 848-862.

Degrees of Freedom

The approach is to incrementally solve the problem using operators that change a *degree of freedom*:

- *preserve* existing constraints
- *achieve* a new constraint

Solvers are specialist programs that modify a geom to achieve a constraint.

A solver to solve multiple constraints simultaneously would be quite complex; many solvers are needed.

Entities and Constraints

Entities:

- Line: through-point, direction
- Line-segment: end1, end2, length, direction
- Point: x, y, z
- Circle: center, axis, radius

Constraints: e.g. Line-segment:

- $InvariantPoint(ls, p_{ls}, p_g)$
- $InvariantDirection(ls, v)$
- $InvariantLength(ls, d)$
- $1dConstrainedPoint(ls, p_{ls}, locus_1)$
- $2dConstrainedPoint(ls, p_{ls}, locus_2)$
- $FixedDistanceFromPoint(ls, p, d)$

Example

Existing constraints:

- LS touches line L
- LS is tangent to circle C

Goal constraints:

- LS is oriented in direction D

Approach

- Generate logic to solve constraints, ignoring exceptional cases. This produces *plan fragments*.
- Elaborate plan fragments to handle over-constrained or under-constrained situations.
- Generate mathematical support routines.

State: $S \; = \; \langle E, \; C \rangle$: entities, constraints

Operator: $op_i : \; S_i \rightarrow S_{i+1}$

The goal is to find a sequence of operators that satisfies all constraints.

Skeletal Plan Fragment Generation

Search is used to find a sequence of operators to achieve the set of constraints.

Intersect two sets of operators:

- Operators that *preserve* an existing constraint:
  $$Preserve(1dConstrainedpoint(ls,p,loc)) = \{scale(ls,p,d),\ rotate(ls,p,\theta),\ translate(ls,vector(p,arbitraryPt(loc)))\}$$
- Operators that *achieve* an unsatisfied constraint

Choose one operator from the intersection, repeat (i.e. search). Heuristic: largest subsets are tried first.

This process is repeated until all constraints have been solved, or no further progress is possible.

Constraint Matching

Intersection of sets of operations is done by unification.

In addition to standard unification, an attempt is made to unify two terms using operation-matching rules; there are 18 rules, e.g.:

- Unify overlapping numerical intervals:
  $[L_1, U_1] \cup [L_2, U_2] \ = \ [max(L_1, L_2), min(U_1, U_2)]$
- Intersection of two loci, e.g. unification of a point along one line and a point along another line is a point at the intersection of the lines.

Example

- Existing: InvariantLength(LS, D), 1dConstrained-Point(LS.END1, L)
- Goal: FixedDistanceFromPoint(LS,P,R)

Search produces two plans:

- Rotate the line segment until it is tangent to the circle
- Translate the line segment until it touches the circle

Example ...

Plan Fragment Elaboration

Skeletal plans ignore exceptions. Skeletal plan fragments are elaborated to:

- Eliminate plans that may not produce a solution
- Use *principle of least motion* to find the lowest-cost solution.

Terms that appear in operations are mapped to (hand-written) functions.

If statements are introduced for possibly over-constrained terms; **loops** are introduced to select the best solution for under-constrained terms.

Results

About 80% of generated programs were correct. Still, the project was a success:

- Task complexity: these solvers are complex; doing them by hand is costly, error-prone.
- Combinatorial effects: a small number of basic concepts are combined in many ways.
- Similarity of specifications: specs can be written succinctly at a high level of abstraction.
- Hybrid reuse strategy: support modules written manually.
- Requirement of consistency: human-written routines might have more variability.
- Incremental approach: early successes were extended.