The Stratego Library

Eelco Visser

STRATEGO

For Stratego version 0.4.18

www.stratego-language.org

Department of Information and Computing Sciences
Universiteit Utrecht
September 29, 2000

Address:
Department of Computer Science
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
email: visser@acm.org
http://www.cs.uu.nl/~visser/

The Stratego Library

Eelco Visser

September 29, 2000

Summary

Stratego is a language for the specification of program transformation systems based on the paradigm of rewriting strategies. Programmable rewriting strategies support the separation of transformation rules and the strategies that apply them to programs, thus encouraging the reuse of rules over many applications. The Stratego library is a collection of generic, language independent transformation rules and strategies that abstract over common patterns in transformation systems.

This document documents all modules in the library. For a tutorial on the usage of Stratego see [2]. The Stratego reference manual [1] explains the constructs of the language. Other publications on Stratego can be found on the Stratego web page:

http://www.cs.uu.nl/~visser/stratego/

CONTENTS

Chapter: Introduction
Section: The Entire Library

LIB

All modules in the library are imported in `lib`.

```
module lib
imports

 (* sequential control *)

        conditional
        iteration

 (* traversal *)

        simple-traversal
        fixpoint-traversal
        env-traversal

 (* data types *)

        list
        tuple
        option
        string
        term
        int-list
        integers
        reals

 (* system *)

        io
        file
        memo
        options
        parse-options
        exec
        tables

 (* object variables *)

        substitution
        rename
        unification
        free-variables
```

CONDITIONAL

```
module conditional
strategies

  try(s) = s <+ id

  if(c, b) = try(c; b)

  if(c, b1, b2) = (c; b1) <+ b2

  ior(a, b) = (a; try(b)) <+ b

  eq = {x: ?(x, x)}

  FAIL = {x: ?x; ?(x, x)}
```

ITERATION

`repeat(s,c)` repeats `s` as long as possible and finishes with `c`

`repeat1(s,c)` applies `s` at least once.

```
module iteration
strategies

  repeat(s, c)       = rec x(s; x <+ c)
  repeat(s)          = repeat(s, id)

  repeat1(s, c)      = rec x(s; (x <+ c))
  repeat1(s)         = repeat1(s, id)

  repeat-until(s, c) = rec x(s; (c <+ x))

  while(c, s)        = rec x(try(c; s; x))
  do-while(s, c)     = rec x(s; try(c; x))

  while-not(c, s)    = rec x(c <+ s; x)

  for(i, c, s)       = i; repeat-until(s, c)
```

Chapter: Terms
Section: Term Traversal

SIMPLE-TRAVERSAL: ONE PASS TRAVERSAL

> The primitive term traversal operators of Stratego (all, some, one) can be combined with the other control operators in a wide variety of ways to define full term traversals. This module defines a collection of the most common generic one-pass traversals over terms.

```
module simple-traversal
imports conditional
strategies
```

Term traversals can be categorized into classes according to how much of the term they traverse and to which parts of the term they modify.

EVERYWHERE

The most general class of traversals visits every node of a term and applies a transformation to it. The following operators define traversals that apply a strategy `s` to all nodes of a term.

```
  topdown(s)        = rec x(s; all(x))
  bottomup(s)       = rec x(all(x); s)
  downup(s)         = rec x(s; all(x); s)
  downup(s1, s2)    = rec x(s1; all(x); s2)

  downup2(s1, s2)   = rec x(s1; all(x); s2)
```

The traversals above go through all constructors. If it is not necessary to traverse the entire tree, the following versions of the traversals can be used. They are parameterized with a strategy operator `stop` that

```
  topdown(s, stop)      = rec x(s; (stop(x) <+ all(x)))
  bottomup(s, stop)     = rec x((stop(x) <+ all(x)); s)
  downup(s, stop)       = rec x(s; (stop(x) <+ all(x)); s)
  downup2(s1, s2, stop) = rec x(s1; (stop(x) <+ all(x)); s2)
```

The strategy `don't-stop` is a unit for these traversals, i.e., `topdown(s)` is equivalent to `topdown(s,don't-stop)`.

```
  don't-stop(s) = fail
```

ALONG A SPINE

A spine of a term is a chain of nodes from the root to some subterm. `spinetd` goes down one spine and applies `s` along the way to each node on the spine. The traversal stops when `s` fails for all children of a node.

```
spinetd(s) = rec x(s; try(one(x)))
spinebu(s) = rec x(try(one(x)); s)

spinetd'(s) = rec x(s; (one(x) + all(fail)))
spinebu'(s) = rec x((one(x) + all(fail)); s)
```

ALONG ALL SPINES

Apply `s` everywhere along al spines where `s` applies.

```
somespinetd(s) = rec x(s; try(some(x)))
somespinebu(s) = rec x(try(some(x)); s)

spinetd'(s) = rec x(s; (one(x) + all(fail)))
spinebu'(s) = rec x((one(x) + all(fail)); s)
```

ONCE

Apply s at one position. One s application has to succeed.

```
oncetd(s) = rec x(s <+ one(x))
oncebu(s) = rec x(one(x) <+ s)

oncetd-stop(s, stop) = rec x(s <+ not(stop); one(x))
```

AT LEAST ONCE

Apply s at some positions, but at least one. As soon as one is found, searching is stopped, i.e., in the top-down case searching in subtrees is stopped, in bottom-up case, searching in upper spine is stopped.

```
sometd(s) = rec x(s <+ some(x))
somebu(s) = rec x(some(x) <+ s)
```

FRONTIER

Find all topmost applications of `s`;

```
alltd(s)           = rec x(s <+ all(x))
alldownup2(s1, s2) = rec x((s1 <+ all(x)); s2)

downup2'-obsolete(s1, s2) = rec x((s1 <+ all(x)); s2)
```

LEAVES

```
leaves(s, is-leaf, skip) =
```

```
    rec x((is-leaf; s) <+ skip(x) <+ all(x))

  leaves(s, is-leaf) =
    rec x((is-leaf; s) <+ all(x))

  is-leaf = not(one(id))
```

MANY

Find as many applications as possible, but at least one.

```
  manybu(s) = rec x(some(x); try(s) <+ s)
  manytd(s) = rec x(s; all(try(x)) <+ some(x))


  somedownup(s) = rec x(s ; all(x) ; try(s) <+ some(x) ; try(s))


  breadthfirst(s) = rec x(all(s); all(x))
```

Chapter: Terms
Section: Term Traversal

FIXPOINT-TRAVERSAL

A collection of strategies that keeps traversing a term until no more applications of some strategy to the nodes can be found.

```
module fixpoint-traversal
strategies

  reduce(s)     = repeat(rec x(some(x) + s))

  outermost(s)  = repeat(oncetd(s))

  innermost'(s) = repeat(oncebu(s))

  innermost(s)  = rec x(all(x); (s; x <+ id))

  innermost''(s) = {mark: where(new => mark);
                          rec x(@?(NF, mark) <+
                                all(x); (s; x <+ @!(NF, mark)))}

  pseudo-innermost3(s) =
        rec x(all(x); rec y(try(s; all(all(all(y); y); y); y)))

signature
  sorts Mark
  constructors
    NF : Mark
```

ENV-TRAVERSAL

```
module env-traversal
rules

  dist(s) : (t, env) -> <all(\x -> <s>(x,env)\)> t

  d(s) : Pair(t, env) -> <s> t

  t(s) : Pair(t, env) -> Pair(<s>t, env)

  coll(s) : f#(xs) -> (f#(ys), zs)
            where <unzip(s)> xs => (ys, zs)

strategies

  env-alltd(s) = rec x(s <+ dist(x))

  env-topdown(s, skip) = rec x(s; (skip(x) <+ dist(x)))

  env-bottomup(s) = rec x(split(dist(x), Snd); s)

  thread-alltd(s) = rec x(s <+ thread(x))
```

Chapter: Terms
Section: Term Construction and Deconstruction

TERM

Some primitives for the manipulation of terms.

<mkterm> (f, [t1,...,tn]) builds the constructor application f(t1,...,tn)

<explode-term> f(t1,...,tn) is the inverse of mkterm and produces (f, [t1,...,tn])

Note: the primitive strategies mkterm and explode-term have been turned into language constructs. The pattern f#(xs) denotes the decomposition of a term into its function symbol f and its *list* of arguments xs. This pattern can be used in matching ?f#(xs) and building !f#(xs) terms (so also in left- and right-hand sides of rules) and also as a congruence s1#(s2).

<address-lt> (t1, t2) compares the address of two terms and succeeds if the address of the first is smaller than the address of the second. This predicate induces a total ordering on terms and can be used to sort terms. Note that this relation is valid in one session (but what happens after rehashing), but not necessarily between two sessions.

<address> t replaces t with its address (an integer). This can be used to obtain a unique symbolic reference to a term.

```
module term
strategies

  mkterm       = prim("_ST_mkterm")
  explode-term = prim("_ST_explode_term")
  address-lt   = prim("_ST_address_lt")
  address      = prim("_ST_address")
```

SHARE

The ATerm library preserves maximal sharing of subterms through hash-consing. This sharing is not directly available to the user of an ATerm. For some applications it is necessary to make the implicit sharing in terms explicit in the form of a let construct in which all occurences of a shared subterm are replaced by a symbolic pointer (variable).

```
module share
imports list-set term
```

The strategy `share` defined in this module achieves such an explicit sharing for arbitrary terms. The approach used by the strategy is to first turn the term into its underlying graph and then inlining those subterms that are not shared (only occur once) or that cannot be shared in this way (upto the needs of an application).

```
strategies

  share(mkvar, always, mklet) =
        graph(mkvar);
        inline-graph(always, mklet)
```

The graph of a term is obtained by turning each node `F(t1,...,tn)` into an edge `(a, F(a1,...,an))`, where `a` is the address of the node and the `ai` are the addresses of its direct subterms. The `mkvar` parameter is used to embed the address in some constructor. (If `mkvar` is `id`, nothing is done.)

```
strategies

  edge(mkvar)      = split(address; mkvar, all(address; mkvar))
  list-edge(mkvar) = split(address; mkvar, map(address; mkvar))

  graph(mkvar) =
    rec x(is-list; split(list-edge(mkvar), map(x); unions); MkCons
              <+ split(edge(mkvar), Kids; map(x); unions); MkCons)
```

The first edge in the graph is the root of the tree. By definition it is never shared. The graph can be turned into one big let-expression with the root as its body. That is what the first line of the definition of `inline-graph` accomplishes.

Subsequently, nodes that are not shared, i.e., a pointer to which only occurs once, can be inlined. Some nodes may always have to be inlined (for application specific reasons). The shape of such nodes is specified by the parameter `always`. Edges that cannot be inlined are turned into a let-binding the form of which is determined by the parameter `mklet`.

After all graph edges have either been inlined or turned into let-bindings the, now empty, `GraphLet` is discarded and replaced by its body.

```
signature
  constructors
    GraphLet : List(Product([Int, Term])) * Term -> Term

strategies

  inline-graph(always, mklet) =
        \ Cons((a, t), graph) -> GraphLet(graph, t) \ ;
        repeat(
          inline; (GraphLet(Cons((id,always),id),id) + dead) <+
          dead <+
          dont-inline(mklet));
        \ GraphLet([], t) -> t \


rules

  inline :
        GraphLet(Cons((a, skel), graph), t[a]) ->
        GraphLet(Cons((a, skel), graph), t[skel])

  dead :
        GraphLet(Cons((a, skel), graph), t) ->
        GraphLet(graph, t)
        where <not(in)> (a, t)

  dont-inline(mklet) :
        GraphLet(Cons((a, skel), graph), t) ->
        GraphLet(graph, <mklet>(a, skel, t))
```

INTEGERS

```
module integers
strategies

  is-int = prim("_ST_is_int")

  minus = prim("_ST_minus")
  plus  = prim("_ST_plus")

  add   = prim("_ST_add")
  subt  = prim("_ST_subt")
  mul   = prim("_ST_mul")
  div   = prim("_ST_div")
  mod   = prim("_ST_mod")

  geq   = prim("_ST_geq")
  gt    = prim("_ST_gt")
  lt    = not(geq)
  leq   = not(gt)

  max   = prim("_ST_max")
  min   = prim("_ST_min")

  int   = prim("_ST_int")
```

Chapter: Built-in Data Types
Section: Numbers

REALS

```
module integers
strategies

  is-real = prim("_ST_is_real")

  cos   = prim("_ST_cos")
  sin   = prim("_ST_sin")
  sqrt  = prim("_ST_sqrt")
```

Chapter: Built-in Data Types
Section: Numbers

INT-LIST

```
module int-list
imports list integers
strategies

  sum = foldr(!0, add)

  average = split(sum, length); div

  list-min = list-accum(min)

  list-max = list-accum(max)

  list-accum(s) = split(Tl, Hd); foldl(s)

  add-lists = list-accum(zip(add <+ !""))

  averages =
  { len: where(length => len);
    add-lists;
    map(try(sect(div, !len)))
  }

  round-list = map(test(sect(leq, !100)) <+ int)
```

STRING

This module defines some operations on strings, including conversions to and from numbers.

```
module string
imports list conditional iteration

rules

  sect(op, arg) : x -> <op> (x, <arg>())

strategies

  new = prim("_ST_new")

  is-string = prim("_ST_is_string")

  implode-string = prim("_ST_implode_string")
  explode-string = prim("_ST_explode_string")

  conc-strings = (explode-string, explode-string); conc; implode-string

  concat-strings = map(explode-string); concat; implode-string

  int-to-string =
    rec x(split(sect(mod, !10); sect(add, !48), sect(div, !10); int);
          (id, ?0; ![] <+ x); MkCons );
    reverse;
    implode-string

  string-to-int =
    explode-string;
    split(!0, id);
    repeat(S2I2);
    S2I1

  escape =
    explode-string;
    rec x(Escape; [id, id | x] <+ [id | x] <+ []);
    implode-string

rules

  Escape : [34 | cs] -> [92, 34 | cs]
  Escape : [92 | cs] -> [92, 92 | cs]

  S2I1 : (n, [])         -> n

  S2I2 : (n, [m|ms]) -> (<add>(<mul>(10, n), <subt>(m, 48)), ms)
```

```
         where <geq>(m, 48); <leq>(m, 57)

  S2D0 : (n, [46|ys])  -> (n, 10, ys)

  S2D1 : (n, f, []) -> n

  S2D2 : (n, f, [m|ms]) ->
         (<add>(n, <div>(<subt>(m, 48), f)), <mul>(f, 10), ms)
         where <geq>(m, 48); <leq>(m, 57)

strategies

  string-to-num =
    explode-string;
    split(!0, id);
    repeat(S2I2);
    ( S2I1
    + S2D0;
      repeat(S2D2);
      S2D1
    )

strategies

  lower-case =
    explode-string;
    map(lc);
    implode-string

   lc = try(where(sect(geq, !65)); where(sect(leq, !90)); sect(add, !32))

rules

  SplitInit : x -> ([], [], x)

  SplitExit :
        (xs, cs, []) ->
        <reverse> [<reverse; implode-string> cs|xs]

  SplitNext :
        (xs, cs, [32|ys]) ->
        (Cons(<reverse; implode-string> cs, xs), [], ys)

  SplitNext :
        (xs, cs, [y|ys]) -> (xs, [y|cs], ys)
        where <not(eq)> (y, 32)

strategies

  split-at-space =
```

```
    explode-string;
    SplitInit;
    rec x(SplitExit <+ SplitNext; x)

  basename =
    explode-string;
    try(rec x([id|x] <+ ?[46 | _]; ![]));
    implode-string

  basename(ext) =
    explode-string;
    try(rec x([id|x] <+ [46 | ext]; ![]));
    implode-string

  guarantee-extension(ext) =
        basename;
        split(id, <ext>());
        add-extension

rules

  add-extension : (name, ext) -> <concat-strings> [name, ".", ext]
```

Chapter: Built-in Data Types
Section: Tables

TABLES

An interface to the ATerm table facility. Note that these primitives work by side-effect.

`<create-table> name` creates a table with name `name`, which can be any term.

`<destroy-table> name` destroys it.

`<table-put> (name, key, value)` associates `value` with `key` in the `name` table.

`<table-get> (name, key)` yields the value associated to `key` or fails.

`<table-remove> (name, key)` removes the entry for `key` from table `name`.

`<table-keys> name` produces the list of keys of table `name`.

```
module tables
strategies

  create-table  = prim("_ST_create_table")
  destroy-table = prim("_ST_destroy_table")
  table-put     = prim("_ST_table_put")
  table-get     = prim("_ST_table_get")
  table-remove  = prim("_ST_table_remove")
  table-keys    = prim("_ST_table_keys")

  table-getlist =
        ?name; table-keys; map(\ x -> (x, <table-get> (name, x))\ )

  table-putlist =
        ?(name, list); <map({x,y: ?(x, y); <table-put> (name, x, y)})> list
```

MEMO

The memo operator makes a strategy into a memoizing strategy that looks up the term to be transformed in a memo table and only computes the transformation if the term is not found.

`<memo-init> tbl` creates a new memo table and `<memo-purge> tbl` destroys it.

`<memo(tbl, s)> t` first looks up the term `t` in the memo table. If present the association in the table is produced, else the result of `<s> t` is computed and stored in the table.

```
module memo
imports tables

strategies

  memo-init  = create-table

  memo-purge = destroy-table

rules

  memo(name, s) :
        t -> t'
        where (<table-get> (<name>(), t) => t')
              <+ (<s> t => t'; <table-put> (<name>(), t, t'))
```

Chapter: Standard Data Types
Section: Options

OPTION

```
module option
signature
  sorts Option(a)
  constructors
    None : Option(a)
    Some : a -> Option(a)
strategies

  option(s) = None + Some(s)
```

Chapter: Standard Data Types
Section: Tuples

TUPLE

tindex: get the nth element of a tuple

tmap: apply a strategy to each element of a tuple

Tuple Concat: concatenate the lists in a tuple of lists, where the concatenation strategy s is a parameter.

```
module tuple
imports list-cons
signature
  sorts Prod(ListType)
  constructors
    TNil  : Prod([])
    TCons : a * Prod(lt) -> Prod(Cons(a, lt))

    Pair  : a * b -> Prod([a,b])
rules
  Fst   : TCons(x, tp) -> x
  Snd   : TCons(x, TCons(y, tp)) -> y
  Third : TCons(x, TCons(y, TCons(z, tp))) -> z
  TInd1 : (1, TCons(x, tp)) -> x
  TInd2 : (n, TCons(x, tp)) -> (<minus> (n, 1), tp)

  Dupl  : x -> (x, x)

  split(f, g)     : x -> (<f> x, <g> x)
  split3(f, g, h) : x -> (<f> x, <g> x, <h> x)

  Swap : (x, y) -> (y, x)

  Thd : TCons(x, xs) -> x
  Ttl : TCons(x, xs) -> xs

strategies

  tindex = rec x(TInd1 <+ TInd2 ; x)

  tmap(s) = rec x(TNil + TCons(s, x))

  tconcat(s) = rec y(<<TNil -> Nil>>
                  + {x, xs: <<TCons(x, xs) -> (x, <y> xs)>>} ; s)

  tconcat'(s1, s2) =
    rec y(TNil; s1
          + {x, xs: <<TCons(x, xs) -> (x, <y> xs)>>}; s2)

  at_tsuffix(s) = rec x(s <+ TCons(id, x))
```

```
tcata(s1, s2) = rec y(TNil; s1 <+
                      {x, xs: <<TCons(x, xs) -> (x, <y> xs)>>}; s2)
```

Chapter: Standard Data Types
Section: Lists

LIST

```
module list
imports simple-traversal
        tuple
        list-cons
        list-basic
        list-index
        list-zip
        list-sort
        list-set
        list-lookup
        list-misc
        list-filter
```

LIST-CONS

Lists are represented by means of the constructors `Nil` and `Cons`.

```
module list-cons
signature
  sorts List(Type)
  constructors
    Nil  : List(a)
    Cons : a * List(a) -> List(a)
```

Chapter: Standard Data Types
Section: Lists

LIST-BASIC

Basic functionality on lists.

Map: Apply strategy to each element of a list

Length of a list

Fetch: Find first list element for which s succeeds

At tail: apply a strategy to the tail of a list

At suffix: apply a strategy to some suffix of a list

```
module list-basic
imports list-cons
rules
  Hd     : [x | l] -> x
  Tl     : [x | l] -> l
  Last   : [x] -> x
  MkCons : (x, []) -> [x]
  MkCons : (x, [y| z]) -> [x, y | z]
  MkSingleton : x -> [x]
strategies

  is-list = [] + [id| id]

  map(s) = rec x([] + [s| x])

  list(s) = rec x([] + [s| x])

  list-some(s) =
    rec x([s| id]; [id| list(try(s))] <+ [id| x])

  list-some-filter(s) =
    rec x([s| id]; [id| filter(s)] <+ [id| x]; Tl)

  length = rec x([]; !0 + Tl; x; \n -> <add> (n, 1)\ )

  fetch(s) = rec x([s | id] <+ [id | x])

  // split-fetch, splits a list in two at the point
  // where the argument strategy succeeds.

  split-fetch(s) =
    at_suffix([s|id];[id|?tl];![]); split(id, !tl)

  at_tail(s) = [id | s]

  at_end(s) = rec x([id | x] + []; s)
```

```
  at_suffix(s) = rec x(s <+ [id | x])

  at_last(s) =
    obsolete(!"at_last -> at-last");
    rec x([id]; s <+ [id | x])

  at-last(s) = rec x([id]; s <+ [id | x])

  listbu(s)       = rec x(([] + [id| x]); s)
  listtd(s)       = rec x(s; ([] + [id| x]))
  listdu(s)       = rec x(s; ([] + [id| x]); s)
  listdu2(s1, s2) = rec x(s1; ([] + [id| x]); s2)

rules

  RevInit : xs -> (xs, [])
  Rev     : ([x| xs], ys) -> (xs, [x| ys])
  RevExit : ([], ys) -> ys

strategies

  reverse = RevInit; repeat(Rev); RevExit

rules

  UptoInit : i -> (i, [])
  UptoExit : (i, xs) -> xs where <lt> (i, 0)
  UptoStep : (i, xs) -> (<minus> (i, 1), [i| xs])

strategies

  upto = UptoInit; rec x(UptoExit <+ UptoStep; x)

rules

  conc : (l1, l2) -> <at_end(!l2)> l1
  //Concat(x) : [l | ls] -> <at_end(<x> ls)> l

strategies

  concat = rec x([] + \ [l | ls] -> <at_end(<x> ls)> l\ )

rules

  Sep(s) : [x| xs] -> [<s>(), x | xs]

strategies

  separate-by(s) =
    [] + [id| rec x([] + [id| x]; Sep(s))]
```

Chapter: Standard Data Types
Section: Lists

LIST-INDEX

```
module list-index
imports list-cons simple-traversal
rules

  Ind1   : (1, [x | xs]) -> x
  Ind2   : (n, [x | xs]) -> (<minus> (n, 1), xs) where <geq> (n, 2)

  Gind0  : (x, ys) -> (1, x, ys)
  Gind1  : (n, x, [x | xs]) -> n
  Gind2  : (n, y, [x | xs]) -> (<plus> (n, 1), y, xs)

strategies

(* Index: Get the n-th element of a list *)

  index = repeat(Ind2) ; Ind1

(* Get-index: get index of element in list *)

   get_index = Gind0 ; rec x(Gind1 <+ Gind2 ; x)
   get-index = Gind0 ; rec x(Gind1 <+ Gind2 ; x)
```

LIST-LOOKUP

Lookup: find value associated to key

Find first element of a list to which a strategy s applies

An alternative formulation of lookup is the following. The advantage over lookup is that it does not construct intermediate pairs.

```
module list-lookup
imports list-basic
rules

  Look1 : (x, [(x, y)|_]) -> y
  Look2 : (x, [_|xs]) -> (x, xs)

  Look1'(keyeq) : (x, [y|_]) -> y where <keyeq> (x, y)

strategies

   lookup = rec x(Look1 <+ Look2 ; x)

   getfirst(s) = rec x(Hd; s <+ Tl; x)

   lookup' = {x, xs: ?(x, xs) ; <getfirst({y:<<(x, y) -> y>>})> xs}

   lookup(keyeq) = rec x((Look1'(keyeq) <+ Look2; x))
```

Chapter: Standard Data Types
Section: Lists

LIST-MISC

```
module list-misc
imports list-cons list-basic
strategies

  member = (?x, fetch(?x))

rules

  FoldR1   : [x, y] -> (x, y)
  FoldR    : [x | xs] -> (x, xs)
  TFoldR   : TCons(x, xs) -> (x, xs)

  FoldL(s) : ([x | xs], y) -> (xs, <s> (x, y))
  FoldL(s) : ([], y) -> y

  lsplit(f, g) : x -> [<f> x, <g> x]

strategies

  foldr1(s1,s2) = rec x([id]; s1 <+ FoldR; (id, x); s2)

  foldr1(s) = rec x((FoldR1 <+ FoldR; (id, x)); s)

  foldr(s1, s2)     = rec x([]; s1 + FoldR; (id, x); s2)

  (* foldr(s1, s2, f)  = rec x([]; s1 + FoldR; (f, x); s2) *)

  foldr(s1, s2, f)  = rec x([]; s1 + \ [y|ys] -> (<f>y, <x>ys) \ ; s2)

  tfoldr(s1, s2)    = rec x(TNil; s1 + TFoldR; (id, x); s2)

  foldl(s) = repeat(FoldL(s))

  mapfoldr(s1, s2, s3) =
    rec x([]; s1 <+ [s2|x]; \ [a|b]->(a,b)\; s3)

  mapfoldr1(s1, s2, s3) =
    rec x([id]; s1 <+ [s2|x]; \ [a|b]->(a,b)\; s3)

  last = rec x(Last <+ Tl; x)

  init = at-last(Tl)

  copy = for(\ (n,t) -> (n,t,[]) \
           ,\ (0,t,ts) -> ts \
           ,\ (n,t,ts) -> (<subt>(n,1), t, [t|ts]) where <geq>(n,1) \ )
```

```
copy(s) = for(\ (n,t) -> (n,t,[]) \
          ,\ (0,t,ts) -> ts \
          ,\ (n,t,ts) -> (<subt>(n,1), t, [<s> t|ts]) where <geq>(n,1) \ )
```

Chapter: Standard Data Types
Section: Lists

LIST-SET

Lists can be used to represent sets of terms. Using the notion of sets we can define the collection of a set of subterms corresponding to some criterion.

```
module list-set
imports list-basic

rules

  HdMember(mklst) : Cons(x, xs) -> xs where mklst; fetch(?x)

  HdMember'(eq, mklst) :
     Cons(x, xs) -> xs
     where mklst; fetch(\y -> <eq> (x, y)\)
```

Union: Concatenation of two lists, only those elements in the first list are added that are not already in the second list.

```
rules

  union : (l1, l2) ->
          <rec x([]; !l2 <+ HdMember(!l2); x <+ [id | x])> l1

strategies

  unions = foldr(![], union)
```

Diff: Difference of two lists

```
rules

  diff : (l1, l2) ->
         <rec x([] <+ HdMember(!l2); x <+ [id | x])> l1

rules

  diff'(eq) :
        (l1, l2) ->
        <rec x([] <+ HdMember'(eq, !l2); x <+ [id | x])> l1

strategies

  diff(eq) = diff'(eq)
```

Intersection is defined in terms of difference.

```
rules

  isect : (l1, l2) -> <diff> (l1, <diff> (l1, l2))
```

COLLECTION

Strategy `collect(s)` produces a collection of all *outermost* subterms for which `s` succeeds.

Strategy `collect-split(f, g)` reduces terms with `f` and extracts information with `g` resulting in a pair `(t, xs)` of a reduced term and the list of extracted subterms.

```
imports tuple

rules

  foldr-kids(nul,sum)    : _#(xs) -> <foldr(nul,sum)> xs

  foldr-kids(zero,sum,s) : _#(xs) -> <foldr(zero, sum, s)> xs

strategies

  collect-kids(s) = foldr-kids(![],union,s)

  collect(s) =
    rec x(s; \y -> [y]\
          <+ collect-kids(x))

  collect(s, skip) =
    rec x(s; \y -> [y]\
          <+ skip(x,![]); collect-kids(id)
          <+ collect-kids(x))

  bu-collect(s) =
        rec x(some(x); collect-kids([s|id] <+ ![])
              <+ s; \y -> [y]\ )
        <+ ![]

  collect-split(f, g) =
        rec x((is-string + is-int); split(try(f), g <+ ![])
              <+ CollectSplit(x, f, g))

(*
  CollectSplit(s, f, g) =
    coll(s) => (t, xs);
    <split(try(f), g <+ ![])> t => (t', ys);
    !(t', <union>(ys, <unions> xs))
*)
```

```
rules

  CollectSplit(s, f, g) :
        f#(as) -> (t, <union> (ys, <unions> xs))
        where <list(s); unzip> as => (bs, xs);
              <split(try(f), g <+ ![])> f#(bs) => (t, ys)
```

OCCURENCE COUNTING

```
strategies

  nrofoccs(s) = rec count(s; !1 <+ foldr-kids(0,add,count))

  twicetd(s) = oncetd(explode-term;
                      (id, at_suffix(Cons(oncetd(s), oncetd(s))));
                      mkterm)

  atmostonce(s) = not(twicetd(s))

  atmostonce'(s) = {n : nrofoccs(s) => n; <leq> (n, 1)}
```

Chapter: Standard Data Types
Section: Lists

LIST-SORT

Sorting

```
module list-sort
imports list-basic
rules

  SortL(s) : Cons(x, Cons(y, l)) -> Cons(y, Cons(x, l))
             where <s> (x, y)

  LSort(s) : Cons(x, l) -> Cons(y, Cons(x, l'))
             where <at_suffix(<<Cons(y, ys) -> ys where <s> (x, y)>>)> l => l'

  Uniq : Cons(x, l) -> l
         where <fetch(?x)> l

  LMerge(s) : Cons(x, l) -> Cons(z, l')
              where <at_suffix(<<Cons(y, ys) -> ys where <s> (x, y) => z>>)> l => l'


strategies

  sort-list(s) = try(rec x((s <+ Cons(id, x)) ; try(x)))

  isort-list(s) = try(rec x((Cons(id, x) <+ s) ; try(x)))

  jsort-list(s) = try(rec x(Cons(id, x) <+ s; try(x)))

  uniq = listbu(repeat(Uniq))
```

LIST-ZIP

Zipping two lists into a list of pairs is a useful operation in many situations. There are many variants of zipping, for instance in the way lists of unequal length are treated. This module defines a collection of zip-like strategies for lists based on one underlying control scheme.

```
module list-zip
imports list-cons
imports tuple
rules

  Zip1   : ([],[]) -> []
  Zip1a' : ([],_) -> []
  Zip1b' : (_,[]) -> []
  Zip1c  : ([],x) -> x
  Zip1c' : (x,[]) -> x
  Zip1d  : ([],[_|_]) -> []
  Zip2   : ([x|xs],[y|ys]) -> ((x, y), (xs, ys))

  LZip2  : ([x|xs], y) -> ((x, y), (xs, y))
  RZip2  : (x, [y|ys]) -> ((x, y), (x, ys))

  Zip3   : (x, xs) -> [x|xs]

  UnZip1 : [] -> ([], [])
  UnZip2 : ((x, y), (xs, ys)) -> ([x|xs], [y|ys])
  UnZip3 : [x |  xs] -> (x, xs)

  NZip00 : xs -> (0, xs)
  NZip01 : xs -> (1, xs)
  NZip1  : (n, []) -> []
  NZip2  : (n, [y|ys]) -> ((n, y), (<plus> (n, 1), ys))
  NZip3  : (x, xs) -> [x| xs]

  TZip1   : (TNil, TNil) -> TNil
  TZip2   : (TCons(x, xs), TCons(y, ys)) -> ((x, y), (xs, ys))
  TZip3   : (x, xs) -> TCons(x, xs)

  cart(s) : (xs, ys) ->
            <map(\x -> <map(\y -> <s>(x, y)\ )> ys\ ); foldr(![], union)> xs

  Skip(s) : ([x|xs], ys) -> (x, (xs, ys))

strategies

  genzip(a, b, c, s) = rec x(a + b; (s, x); c)

  zip(s)  = genzip(Zip1,   Zip2,   Zip3,   s)
  zip'(s) = genzip(Zip1a' <+ Zip1b',   Zip2,   Zip3,   s)
```

```
  zipl(s) = genzip(Zip1a',   Zip2,   Zip3,   s)
  zipr(s) = genzip(Zip1b',   Zip2,   Zip3,   s)

  rest-zip(s) =
    genzip((?([],_) + ?(_,[])); ?(tla, tlb); ![], Zip2, Zip3, s);
    \ pairs -> (tla, tlb, pairs) \

  unzip    = genzip(UnZip1, UnZip3, UnZip2, id)
  unzip(s) = genzip(UnZip1, UnZip3, UnZip2, s)

  nzip0(s) = NZip00 ; genzip(NZip1,  NZip2,  NZip3,  s)
  nzip(s)  = NZip01 ; genzip(NZip1,  NZip2,  NZip3,  s)
  tzip(s)  = genzip(TZip1,   TZip2,   TZip3,   s)

  lzip(s)  = genzip(Zip1a', LZip2, Zip3, s)
  rzip(s)  = genzip(Zip1b', RZip2, Zip3, s)


  zipFetch(s) = rec x(Zip2; ((s, id) <+ (id, x)))
  lzipFetch(s) = rec x(LZip2; ((s, id) <+ (id, x)))
  rzipFetch(s) = rec x(RZip2; ((s, id) <+ (id, x)))

  zipPad(s, padding) =
    rec x(Zip1 + Zip2; (s, x); Zip3 +
          ([], [id|id]); (![<padding>()|[]], id); x +
          ([id|id], []); (id, ![<padding>()|[]]); x)

  zip-tail = rec x(Zip1c + (Tl, Tl); x)
  zipl-tail-match(s) = rec x(Zip1c + Zip2; (s, id); Snd; x)
  zipr-tail-match(s) = rec x(Zip1c' + Zip2; (s, id); Snd; x)

  zip-skip(pred, s) =
        rec x(Zip1 + (Skip(pred); (id, x) <+ Zip2; (s, x)); Zip3)

strategies

  tuple-zip(s) =
    rec x(split(tmap(Hd), tmap(Tl)); (s, x); Zip3
          <+ tmap([]); ![])

  tuple-unzip(s) =
    rec x(split(map(Thd), map(Ttl)); (s, x); \ (a,b) -> TCons(a,b) \
          <+ map(()); !())
```

LIST-FILTER

```
module list-filter
imports list
strategies

  filter(s) = rec x(Nil + (Cons(s, x) <+ Tl; x))

  filter-gen(pred, cont) =
        rec x(Nil + (pred; cont(x)) <+ Tl; x)

  (* filter(s) = filter-gen(Cons(s,id), at_tail) *)

  skip1(s) = at_tail(s)
  skip2(s) = at_tail(at_tail(s))

  filter-option-args(flag) = filter-gen(Cons(flag,id);Tl, skip1)

  filter-options(flag) = filter-gen(Cons(flag,id), skip2)
```

Chapter: Standard Data Types
Section: Binary Trees

BIN-TREE

Binary trees are represented by the constructors `EmptyNode` and `BinNode`.

```
module bin-tree
signature
  sorts BinTree(a)
  constructors
    EmptyNode : BinTree(a)
    BinNode   : a * BinTree(a) * BinTree(a) -> BinTree(a)
```

BIN-TREE-SET

Binary trees are efficient representations of sets of elements in terms of lookup. Based on the unique addresses of terms under hash-consing, efficient representations of sets of terms can be made with binary terms.

```
module bin-tree-set
imports bin-tree
strategies

  branch(mky, lt, gt, et) =
        BinNode(where(\x -> <eq>(x, <mky>())\), id, id) <+
        BinNode(where(\x -> <address-lt>(<mky>(), x)\), lt, id) <+
        BinNode(where(\x -> <address-lt>(x, <mky>())\), id, gt) <+
        EmptyNode; et

rules

  bin-add :
        (a, t) ->
        <rec x(branch(!a, x, x, !BinNode(a, EmptyNode, EmptyNode)))> t

  Merge(s) :
        (EmptyNode, x) -> x

  Merge(s) :
        (x, EmptyNode) -> x

  Merge(s) :
        (BinNode(x, l1, r1), BinNode(x, l2, r2)) ->
        BinNode(x, <s> (l1, l2), <s> (r1, r2))

  Merge(s) :
        (BinNode(x, l1, EmptyNode), BinNode(y, l2, r2)) ->
        BinNode(y, <s> (BinNode(x, l1, EmptyNode), l2), r2)
        where <address-lt> (x, y)

  Merge(s) :
        (BinNode(x, EmptyNode, r1), BinNode(y, l2, r2)) ->
        BinNode(y, l2, <s> (BinNode(x, EmptyNode, r1), r2))
        where <address-lt> (y, x)

  Merge(s) :
        (BinNode(x, l1, r1), BinNode(y, l2, r2)) ->
        <s> (r1, BinNode(y, <s> (BinNode(x, l1, EmptyNode), l2), r2))
        where <address-lt> (x, y)

  Merge(s) :
        (BinNode(x, l1, r1), BinNode(y, l2, r2)) ->
        <s> (r2, BinNode(x, <s> (BinNode(y, l2, EmptyNode), l1), r1))
```

```
        where <address-lt> (y, x)

strategies

  merge = rec x(Merge(x))

  mkbinset = foldr(!EmptyNode, bin-add)
```

Chapter: Generic Algorithms
Section: Object Variables

FREE-VARIABLES

Extraction of free variables from an expression is governed by the shape of variables and the shape of variable bindings.

```
module free-variables
imports list
```

PARAMETERS OF FREE VARIABLE EXTRACTION

the following aspects determine the extraction of free variables from expressions

- shape of variables
- variables bound by a binding construct
- arguments of the binding constructs where variables are bound

In addition can variable constructs contain other variables, or in other words, whether variables are leaves or non-leaves.

VARIABLES ARE LEAVES

In the first style of free variable extraction, variables are leaves of abstract syntax trees.

Free variables of a term; The first argument s1 is a strategy that transforms variables into lists of variables, e.g., `Var(x) -> [x]`; The second argument s2 is a strategy that maps binding constructs to the list of bound variables, e.g., `Scope(xs, s) -> xs`;

```
strategies

  free-vars(getvars, boundvars) =
    rec x(getvars
          <+ split(collect-kids(x), boundvars <+ ![]); diff)

  free-vars(getvars, boundvars, boundin) =
    rec x(getvars
          <+ {vs: where(?vs <= boundvars);
                  boundin(split(x, !vs); diff, x, ![])};
             collect-kids(id)
          <+ collect-kids(x))

(* // if we had strategy abstraction /\(x1,...,xn) -> s

  free-vars(getvars, boundvars, boundin) =
    collect(getvars
           ,/\ (x,nil) ->
```

```
            {vs: where(?vs <= boundvars);
                 boundin(split(x, !vs); diff, x, nil)})
*)

  free-vars(getvars, boundvars, boundin, eq) =
    rec x(getvars
          <+ {vs: where(?vs <= boundvars);
                  boundin(split(x, !vs); diff(eq), x, ![])};
             collect-kids(id)
          <+ collect-kids(x))
```

VARIABLES ARE NOT LEAFS

In a more complicated style of free variable extraction, variables are not leaves of abstract syntax trees, but can contain subterms that again contain variables.

```
strategies

  free-vars2(getvars, boundvars) =
    rec x(split(getvars <+ ![],
                split(collect-kids(x), boundvars <+ ![]); diff);
          union)

  free-vars2(getvars, boundvars, boundin) =
    rec x(split(getvars <+ ![],
                ({vs: where(?vs <= boundvars);
                      boundin(split(x, !vs); diff, x, ![])};
                 collect-kids(id)
                 <+ collect-kids(x)));
          union)

  free-vars2(getvars, boundvars, boundin, eq) =
    rec x(split(getvars <+ ![]
               ,{vs: where(?vs <= boundvars);
                     boundin(split(x, !vs); diff(eq), x, ![]);
                     collect-kids(id)}
                <+ collect-kids(x)
               ); union)
```

Chapter: Generic Algorithms
Section: Object Variables

RENAME: RENAMING BOUND VARIABLES

Renaming of bound variables is determined by the shape of variables and binding constructs. Three generic strategies are defined that cater for different complexities of binding constructs.

Variable binding constructs protect variables from clashing with variables in other parts of a program when their names are the same. To prevent the introduction of name clashes during program transformation it can be useful to give all variable bindings a unique name. This module defines three generic strategies for bound variable renaming all based on the same idea, but dealing with increasingly complex variable binding models.

Renaming depends *only* on the shape of variable bindings and variable occurences. Other language constructs are irrelevant.

In the generic strategies the following assumptions about binding constructs are made: (1) There is a subtree that covers the scope in which the variables are bound. (2) variables are atomic, i.e., do not contain subterms that are variables or binding constructs.

Approach: indicate shape of variable occurences and variable binders

The strategy `rename(isvar, mkvar, bnd)` renames all bound variables in a term to fresh variables;

Parameters:

isvar: Succeeds if applied to a variable

newvar: Takes a string and builds a variable

bnd: Maps a binding construct to the list of bound variables

apply(a, b, c): reconstruct the binding construct with fresh variables;

- a should be applied to the subterm containing the variable(s)

- b should be applied to the subterms in which the variables are bound

- c should be applied to the subterms in which the variables are not bound;

```
module rename
imports simple-traversal tuple list env-traversal
rules

  RnVar(isvar) :
    (t, env) -> <isvar(split(id, !env); lookup)> t

  RnBinding(bndvrs) :
    (t, env1) -> (t, env1, env2)
    where <bndvrs> t => xs; map(new) => ys;
          <conc>(<zip(id)>(xs,ys), env1) => env2
```

```
  DistBinding(s) :
    (t, env1, env2) -> <all( \x -> <s>(x, env2)\ )> t

strategies

  // renaming bound variables assuming that variables are bound
  // in all subterms of a binding construct
  // variable declarations in binding constructs are assumed to
  // have the same shape as variable uses

  rename(isvar, bndvars) =
    \ t -> (t, []) \ ;
    rec x(env-alltd(RnVar(isvar)
                    <+ RnBinding(bndvars);
                       DistBinding(x)))

rules

  DistBinding(s, boundin) :
    (t, env1, env2) -> <boundin(\x -> <s>(x, env2)\
                                ,\x -> <s>(x, env1)\
                                ,id)> t

strategies

  // renaming while making a distinction between subterms
  // in which the variables are bound or not
  // variables at binding sites are assumed to have
  // the same shape as other variable occurences

  rename(isvar, bndvars, boundin) =
    \ t -> (t, []) \ ;
    rec x(env-alltd(RnVar(isvar)
                    <+ RnBinding(bndvars);
                       DistBinding(x, boundin)))

rules

  RnBinding(bndvrs, paste) :
    (t, env1) -> (<paste(!ys)> t, env1, env2)
    where <bndvrs> t => xs; map(new) => ys;
          <conc>(<zip(id)>(xs,ys), env1) => env2


strategies

  rename(isvar, bndvars, boundin, paste) =
    \ t -> (t, []) \ ;
    rec x(env-alltd(RnVar(isvar)
```

```
                    <+ RnBinding(bndvars, paste);
                       DistBinding(x, boundin)))
```

```
signature
  sorts Exp
  constructors
    Abs : String * Exp -> Exp
    Var : String -> Exp
rules

  Bnd : Abs(x, e) -> [x]

strategies

  ern_apply(nwvars, bndvars, ubndvars) =
        Abs(nwvars; Hd, bndvars)

  erename = rename''(Var, Bnd, ern_apply)
```

Figure 1: Example: Untyped lambda calculus

```
signature
  sorts Exp
  constructors
    Abs    : String * Type * Exp -> Exp
    Var    : String * Type -> Exp
    Letrec : List(Fdec) * Exp -> Exp
    Fdec   : String * Type * Exp -> Fdec
rules

  Bnd : Abs(x, t, e) -> [x]
  Bnd : Letrec(fdecs, e) -> <map(Name)> fdecs

  Name : Fdec(f, t, e) -> f

strategies

  is-var(s) = Var(s, id)

  ern_apply(nwvars, bndvars, ubndvars) =
        Abs(nwvars; Hd, bndvars) +
        Letrec(split(id,nwvars); zip(f(bndvars)), bndvars)

  f(bndvars) : (Fdec(f, t, e), g) -> Fdec(g, t, <bndvars> e)

  erename = rename''(is-var, Bnd, ern_apply)
```

Figure 2: Example: Typed lambda calculus

SUBSTITUTION

Substituting terms for variables depends mainly on the shape of variables. This module implements several generic strategies for different styles of parallel substitution, including ones that rename bound variables to prevent name capture.

A substitution is a mapping from variables to terms. Given a substitution the strategy `substitute(...)` traverses a term and replaces variables in the domain of the mapping by their associated term. The strategy can be applied in two ways; (1) to a pair of a substitution and a term `<substitute(...)> (sbs, t)` and (2) to a triple of a list of variables, a list of (equal length) of terms and a term `<substitute(...)> (xs, ts, t)`. This entails that the type of `substitute(...)` is either `Prod([List(Prod([a,b])),b]) -> b` or `Prod([List(a),List(b),b]) -> b`, with `a` the type of variables and `b` the type of terms.

There are four versions of the substitution strategy that depend on two factors; (1) renaming of bound variables in terms substituted for variables (2) renaming of bound variables in the context of substituted variables.

All versions are parameterized with a strategy `isvar` recognizing variables and mapping them to a substitution key, which can be the entire variable structure or just its name. That is, `isvar` should have type `b -> a`.

The substitution strategy can be parameterized with a variable renaming strategy `ren` (of type `b -> b`) that will be applied to each term after it is substituted for a variable. This can be used to ensure that all bound variables are unique throughout an abstract syntax tree and thus prevent free variable capture.

A better way to ensure that free variables are not captured when substituting under bindings requires renaming the bound variables in the context of the variables that are substituted for. This is achieved by combining the generic bound variable renaming techniques from module `rename` with replacing a variable by a term. For this purpose there are two variants of the substitution strategy that are parameterized with strategies indicating shape of variables, the bound variables, the arguments that they are binding in and a replacement strategy. See module `rename` for an explanation of these parameters.

```
module subs
imports simple-traversal tuple list rename

strategies

  // substitutions accept two types of input
  // 1) a pair of a substitution (is list of pairs) and a term
  // 2) a triple of a list of variables, a list of terms that should
  //    replace them, and a term

  subs-args =
    ?(sbs, t) <+ \ (xs, ys, t) -> (<zip(id)> (xs, ys), t) \
```

```
rules

  // replacing a variable with its value in the substitution

  SubsVar(isvar, mksbs) :
    t -> <lookup> (x, sbs)
    where <isvar> t => x; mksbs => sbs

strategies

  // substitute variables, no regard for variable bindings, and
  // rename bound variables in substituted terms

  substitute(isvar, ren) =
    subs-args => (sbs,t); !t;
    alltd(SubsVar(isvar, !sbs); ren)

  // substitute variables, no regard for variable bindings

  substitute(isvar) =
    substitute(isvar, id)

  // substitute all variables, rename bound variables on the way down,
  // and rename the bound variables in the terms that are substituted
  // for variables using the renaming strategy ren

  substitute(isvar, varshape, bndvars, boundin, paste, ren) =
    subs-args => (sbs,t); !(t, []);
    rec x(env-alltd(RnVar(varshape)
                    <+ Fst; SubsVar(isvar, !sbs); ren
                    <+ RnBinding(bndvars, paste);
                       DistBinding(x,boundin)))

  // substitute variables and rename bound variables encountered
  // on the way to prevent variable capture, don't rename
  // substituted terms

  substitute(isvar, varshape, bndvars, boundin, paste) =
    substitute(isvar, varshape, bndvars, boundin, paste, id)
```

OBSOLETE SUBSTITUTION STRATEGIES

The following definitions are obsolete and should be replaced with uses of the strategies above.

```
strategies

  subs'(isvar, mklst) =
    obsolete(!"subs'/2");
    subs(isvar, mklst)

  subs(isvar) =
    obsolete(!"subs/1");
    //subs-args;
    //\(sbs, t) -> <subs(isvar, !sbs)> t \
    substitute(isvar)

  subs(isvar, mklst) =
    obsolete(!"subs/2");
    split(mklst, id); substitute(isvar)
    //where(mklst => lst);
    //alltd(isvar; {z: \ x -> z where <fetch(?(x, z))> lst\ })

  subs_proper(isvar, ren) =
    obsolete(!"subs_proper/2");
    substitute(isvar, ren)
    //?(xs, ts, t) ;
    //  <zip(id)> (xs, ts) => lst ;
    //  <alltd(isvar; {x, z:<<x -> <ren> z where <fetch(?(x, z))> lst >>})> t

  subs_proper'(isvar, ren, mklst) =
    obsolete(!"subs_proper'/2");
    split(mklst, id); substitute(isvar, ren)
    //where(mklst => lst);
    //alltd(isvar ; {x, z:<<x -> <ren> z where <fetch(?(x, z))> lst >>})
```

UNIFICATION

Syntactic unification, no variable bindings are taken into account.

```
module unification
imports list term substitution
```

```
<unify(isvar)> [(t1,t2),(t3,t4),...] => [(x1,p1),(x2,p2),...]
```

The strategy `unify` unifies a list of pairs of terms and creates the most general unifier for them. The strategy is parameterized by a strategy `isvar` that determines the shape of variables. The result is a list of pairs `(x1,p1)`, where `x1` is a term for which `isvar` succeeds and `p1` is the term it should be substituted with to unify the terms.

```
strategies

  equal =
    for(id ,[], UfIdem <+ UfDecompose)

rules

  UfIdem :
    [(x,x) | ps] -> ps

  UfDecompose :
    [(f#(xs), f#(ys)) | ps] -> <conc>(<zip(id)>(xs, ys), ps)

strategies

  diff =
    for(\ ps -> ([],ps) \ , (id,[]), (id, UfIdem <+ UfDecompose) <+ UfShift)

rules

  UfShift :
    (ps1, [p | ps2]) -> ([p | ps1], ps2)

strategies

  unify(isvar) =
    for(\ pairs -> (pairs, []) \
       ,\ ([], sbs) -> sbs \
       ,(UfIdem, id) <+ UfVar(isvar) + UfSwap(isvar) <+ (UfDecompose, id))

rules

  UfVar(isvar) :
        ([(x,y) | ps], sbs) -> (ps', [(x, y) | sbs''])
```

```
        where <isvar> x; <not(in)>(x,y);
              ?(sbs'', ps') <= <substitute(isvar)> ([(x,y)], (sbs, ps))

  UfSwap(isvar) :
        ([(x,y) | ps], sbs) -> ([(y,x) | ps], sbs)
        where <not(isvar)> x; <isvar> y

rules

  // Test occurrence of a in b

  in : (a, t) -> <oncetd(?a)> t

strategies

  equal(fltr) =
    for(id ,[], UfIdem <+ try([(fltr,fltr)|id]); UfDecompose)
```

Chapter: Generic Algorithms
Section: Graphs

PACK-GRAPH

> The strategy 'graph-nodes' is a generic algorithm for mapping a graph to a collection of nodes reachable from a given root node. The algorithm is parameterized with the following notions: 'get-node' maps a node name and a graph to the node itself, 'out-edges' maps a node to the names of its out edges, 'add-node' that adds a name and its corresponding node to a collection of nodes.

```
module pack
imports string list
```

Basic idea: configuration of the form `(todo, done, files)`

keep adding files corresponding to the names in `todo` until empty

`<worklist(get, next, add)> (root, source, target)` produces a target to which all things reachable from `root` via the `next` relation are `add`ed. The things are obtained via `<get>(name,source)`.

```
     get-node  :: name * graph -> node
     out-edges :: node -> List(name)
     add-node  :: name * node * nodes -> nodes
```

```
rules

  GnInit : (root, graph, nodes) -> ([root], [root], graph, nodes, [])

  GnNext(get-node, out-edges, add-node) :
        ([name | todo], done, graph, nodes, undef) ->
        (<conc> (todo', todo), <conc> (todo', done),
         graph, <add-node> (name, node, nodes), undef)
        where ?node  <= <get-node> (name, graph);
              ?names <= <out-edges> node;
              ?todo' <= <diff> (names, done)

  GnUndefined :
        ([name | todo], done, graph, nodes, undef) ->
        (todo, done, graph, nodes, [name | undef])

  GnExit : ([], done, graph, nodes, undef) -> (nodes, undef)

strategies

  graph-nodes-undef(get-node, out-edges, add-node) =
        for(GnInit, GnExit, GnNext(get-node, out-edges, add-node)
                            <+ GnUndefined)

  graph-nodes(get-node, out-edges, add-node) =
```

```
graph-nodes-undef(get-node, out-edges, add-node);
\ (nodes, undef) -> nodes \
```

Chapter: Generic Algorithms
Section: Graphs

PACK-MODULES

CHANGES (by Joost Visser) * Strategy pack-modules(pack, dep-base) now takes two additonal options: -dep target -nodep The first one specifies the maketarget and basename of the dependency file that is created. If this option is not passed, the argument dep-base is used instead. Finally, the -nodep option can be used to disable this and prevent any dependency file to be created. Note that -dep takes precendence over -nodep. * An additional strategy pack-modules(pack) was added that behaves like pack-modules, except no default for the dependency file base name needs to be specified. Hence, only the command line options are relevant.

```
module pack-modules
imports options pack-graph

strategies

  pack-module-options =
    parse-options( io-options
                 + ArgOption("-I"  , \x -> Include(x)\ )
                 + Option   ("-nodep",     !NoDependency )
                 + ArgOption("-dep", \x -> Dependency(x)\ )
                 )

  pack-modules(pack)
    = pack-modules(pack,fail)
      <+ <fatal-error> ["  packing failed"]

  pack-modules(pack, dep-base) =
    ?options <= pack-module-options;
    try(need-help(pack-modules-usage));
    list(try( ?Program(prog)
            + ?Input(in) + ?Output(out) + ?Binary(bin)
            + ?Dependency(dep)
            ));
    ?path    <= filter( \Include(p)->p\ );
    ?infile  <= (!in <+ !stdin);
    ?outfile <= (!out <+ !stdout);
    ?(files, spec) <= <pack(!path)> infile;
    <!bin; WriteToBinaryFile <+ WriteToTextFile> (outfile, spec);
    try(?depfile <= (!dep
                    <+ (not(<option-defined(?NoDependency())> options);
                       !outfile));
        <create-dep-file(dep-base)> (depfile, files)
    );
    dtime => time;
    <printnl>(stderr, [prog, " (", time, " secs)"])

  pack-modules-usage =
    option-defined(?Program(prog));
```

```
    <printnl> (stderr,
               ["usage : ", prog,
                " [-S] [-I dir] [-i file]",
                " [-o file] [-b] [-s] [--help|-h|-?]",
                " [-dep target | -nodep]" ]);
    <exit> 1

rules

  create-dep-file(dep-base) :
        (outfile, files) -> (outfile, files)
        where
          <dep-base> outfile => out;
          <add-extension; open-file> (out, "dep") => dep;
          <printnl>(dep, <separate-by(!" ")> [out, ":" | files])

signature
  constructors
    NoDependency :          Option
    Dependency : String -> Option
```

Chapter: Generic Algorithms
Section: Graphs

PACK: PACKING AND FLATTENING MODULES

> Module systems allow the definition of a program to be split into separate units stored in separate files. For languages that do not support separate compilation (such as grammar formalisms) these separate units need to be combined to create the whole program. This module defines generic strategies for packing a set of modules reachable from a root module and for flattening a set of modules into a single program.

ANALYSIS

Aspects of module packing and flattening

1. finding the module associated with the module name
2. doing something with the module, i.e., adding it to the result
3. finding the imports in a module
4. keeping track of which modules have already been inlined

```
module pack
imports string pack-graph
```

PACKING

Packing a module consists of collecting all modules into a single file.

```
rules

  PackInit : root -> (root, (), [])

strategies

  pack(parser, imp) =
        PackInit;
        graph-nodes(Fst; parser, get-imports(imp), \ (n,x,xs) -> [x|xs] \ )

  get-imports(imp) =
        collect(imp);
        concat
```

UNPACKING

Unpacking is the reverse of packing, i.e., writing each module in a list of modules to a separate file.

```
rules
```

```
  WriteMod(getname, write, ext) :
    mod -> <write>(<add-extension>(<getname>mod, <ext>()), mod)

strategies

  unpack(wrapper, getname, ext) =
        wrapper(WriteMod(getname, WriteToTextFile, ext))
```

FLATTEN

`<flatten> (root, mods)` produces a flattened version of the root module.

```
strategies

  flatten(imp, nameeq, getcontent) =
        FlattenInit;
        graph-nodes(lookup(nameeq),
                  get-imports(imp),
                  Ttl; (getcontent, id); conc)

rules

  FlattenInit : (root, mods) -> (root, mods, [])
```

Chapter: System Interface
Section: Input/Output

IO: INPUT AND OUTPUT

A transformation system needs to input terms to transform and output transformed terms. Instead of providing a single fixed mechanism for IO, this module defines primitives for file input and output of terms and strings. These primitives can be used in a variety of ways to define customized IO.

A compiled Stratego specification applies the strategy main to the command line options that it gets. When interpreting these it will probably be necessary to read in a term from file and later write the transformed term back to (another) file. This module provides the primitives for doing file input and output. Module options defines strategies to parse and analyze the command line options.

```
module io
imports list string tuple integers time exec
```

A file can be a string or one of the terms stdin, stdout, stderr.

```
signature
  sorts File
  constructors
    stdin  : File
    stdout : File
    stderr : File
```

<ReadFromFile> file reads the term in file. The file needs to be in textual or binary ATerm format.

<WriteToTextFile> (file, term) writes term to file in textual ATerm format.

<WriteToBinaryFile> (file, term) writes term to file in BAF format.

<print> (file, [t1,...,tn]) prints the terms ti to file. If ti is a string it is printed without quotes, otherwise it is printed as a term. printnl has the same behaviour, but also prints a newline after tn.

Before printing to a file the file should be opened using <open-file> filename, which truncates the file, or creates it if it doesn't exist. To append to a file, open the file with <append-file> filename. The file is created if it doesn't exist.

```
strategies

  print            = prim("_ST_print")
  printnl          = prim("_ST_printnl")
  printascii       = prim("_ST_printascii")

  file-exists      = prim("_ST_file_exists")
```

Chapter: System Interface
Section: Input/Output

```
  open-file         = prim("_ST_open_file")
  append-file       = prim("_ST_append_file")
  close-file        = prim("_ST_close_file")

  ReadFromFile      = prim("_ST_ReadFromFile")
  WriteToBinaryFile = prim("_ST_WriteToBinaryFile")
  WriteToTextFile   = prim("_ST_WriteToTextFile")
```

The primitive `print-stack` prints the top n elements of the stack if applied as `<print-stack>` n or the entire stack if applied to a non-integer term.

```
  print-stack       = prim("_ST_PrintStack")
```

The strategy `debug` prints the current term to `stderr` without changing it. This is a useful strategy for debugging specifications (hence its name).

```
strategies

  open(file) = file; ReadFromFile

  save(file) = split(file, id); WriteToTextFile

  debug      = where(split(!stderr, \x -> [x]\); printnl)
  debug(msg) = where(split(!stderr, \x -> [<msg>(),x]\); printnl)

  say(msg) = where(msg; debug)

  debug-stdout(msg) = where(split(!stdout, \x -> [<msg>(),x]\); printnl)

  trace(msg,s) =
    debug(msg); (s; debug(!"succeeded: ") <+ debug(!"failed: "))

  error = where(split(!stderr, id); printnl)

  fatal-error = where(error; <exit> 1)

  printchar   = where(split(!stdout, \x -> [x]\ ); printascii)

  printstring = where(split(!stdout, \x -> [x]\ ); print)

  print-strings-nl(out) = where(split(out, id); printnl)

  obsolete(msg) = where(<debug(msg)> ": obsolete library strategy")
```

The operator `stdio` implements a simple user-interface for transformers. A term is read from standard input, transformed with parameter strategy `s` and then written to standard output. If the transformation failed the text `rewriting failed` is written to standard error.

Chapter: System Interface
Section: Input/Output

```
strategies

  stdio(s) = <ReadFromFile> stdin;
             s;
             split(!stdout, id); WriteToTextFile
          <+ <fatal-error> ["** rewriting failed"]
```

A variant of this strategy provides a pair of the command-line options and the input file to the strategy.

```
strategies

  stdioO(s) = split(id, <ReadFromFile> stdin);
              s;
              split(!stdout, id); WriteToTextFile
           <+ <fatal-error> ["** rewriting failed"]
```

Chapter: System Interface
Section: Input/Output

FILE

```
module file
imports io

rules

  find-in-path :
    (file, path) ->
    <rec x(\ [dir|_] -> <concat-strings; file-exists> [dir,"/",file] \
           <+ Tl; x)> path

strategies

  find-in-path(mkpath) =
        file-exists
        <+ split(id, mkpath); find-in-path
        <+ \ x -> <fatal-error> ["no such file: ", x] \

  find-file(mkpath, ext) =
        guarantee-extension(ext);
        find-in-path(mkpath)
```

OPTIONS

```
module options
imports io parse-options
signature
  constructors
    Silent      : Option
    Verbose     : Option
    Version     : Option
    Input       : String -> Option
    Output      : String -> Option
    Binary      : Option
    Statistics  : Option
    Help        : Option
    Runtime     : Real   -> Option
    DeclVersion : String -> Option
```

The operator `iowrap` defines a default wrapper around a strategy that handles processing of options and reading and writing of terms from and to files.

```
strategies

  iowrap(strat) = iowrapO((id, strat), fail)

  iowrapO(strat, extra-options) =
        (parse-options(extra-options <+ io-options));
        (need-help
         <+ input-file';
            apply-strategy(strat);
            output-file';
            report-success
         <+ report-failure
        )

  iowrapNoOutput(strat, extra-options) =
        (parse-options(extra-options <+ io-options));
        (need-help
         <+ input-file';
            apply-strategy(strat);
            report-success
         <+ report-failure
        )
```

Handling of options

```
strategies
```

```
io-options =
      Option("-S",           !Silent())
      + Option("--silent",    !Silent())
      + Option("--verbose",   !Verbose())
      + Option("-v",          !Version())
      + Option("--version",   !Version())
      + ArgOption("@version",  \x -> DeclVersion(x)\ )
      + ArgOption("-i",       \x -> Input(x)\ )
      + ArgOption("--input",  \x -> Input(x)\ )
      + ArgOption("-o",       \x -> Output(x)\ )
      + ArgOption("--output", \x -> Output(x)\ )
      + Option("-b",          !Binary())
      + Option("-s",          !Statistics())
      + Option("--help",      !Help())
      + Option("-h",          !Help())
      + Option("-?",          !Help())

usage' =
       where(option-defined(?Program(prog));
             <printnl>
             (stderr,
              ["usage : ", prog,
               " [-S] [-i file] [-o file] [-b] [-s] [--help|-h|-?]"]);
             <exit> 1)

need-help =
      option-defined(Help + Undefined(id));
      usage'

need-help(u) =
      option-defined(Help + Undefined(id) + Version); u
```

Input, strategy application and output

```
input-file' =
       where((option-defined(?Input(infile)) <+ !stdin => infile));
       split(id, <ReadFromFile> infile)

apply-strategy(strat) =
       where(dtime);
       strat;
       where(dtime => runtime);
       \(options, trm) -> ([Runtime(runtime) | options], trm)\

output-file' =
       where((option-defined(?Output(outfile)) <+ !stdout => outfile, id));
       (id, split(!outfile, id));
       ((option-defined(?Binary()), WriteToBinaryFile)
        <+ (id, WriteToTextFile))
```

```
report-success =
       where(try((not(option-defined(?Silent())), id);
                  (option-defined(?Runtime(runtime)), id);
                  (option-defined(?Program(prog)), id);
                  <printnl> (stderr,
                              [prog, " (", runtime, " secs)"])));
       <exit> 0

report-failure =
      <printnl> (stderr, ["rewriting failed"]);
      <exit> 1
```

Chapter: System Interface
Section: External Processes

EXEC

Transformation systems often consist of multiple components, e.g., parsers, pretty-printers and several actual transformation components. To glue these components together this module defines the process control primitive `call`.

The strategy `<call> (prog, args)` executes the program with name `prog` and passes the list of arguments `args` to it.

The strategy `<transform-file(s, suf)> (base, ext)` reads in the term from file "base.ext", applies strategy `s` to it and write the result to the concatenation of the strings `(base, <suf>())`.

```
module exec.r
strategies

  exit       = prim("_ST_exit")
  call       = prim("_ST_call")
  call-noisy = prim("_ST_call_noisy")

  get-pid    = prim("_ST_get_pid")

  rm-files = ?files; where(<call> ("rm", ["-f" | files]))

  pipe(c, suf2) = pipe'(c, suf2, ![])

  pipe'(c, suf2, args) =
    where(conc-strings => in);
    (id, suf2);
    where(conc-strings => out);
    // where(<debug(!"calling : ")> [<c>(), in, out]);
    where(<call> (<c>(), <conc> (<args>(), ["-i", in, "-o", out])))

  transform-file(s, suf) =
    where(conc-strings => in);
    (id, suf);
    where(conc-strings => out);
    where(<apply-to-file(s)> (in, out))

  apply-to-file(s) =
    ?(in, out);
    where(<ReadFromFile; s> in => trm);
    where(<WriteToBinaryFile> (out, trm))

  copy-file(s, new-base, new-suf) =
    ?(base, suf);
    (new-base, new-suf);
    ?(nbase, nsuf);
    where(<apply-to-file(s)>
            (<conc-strings> (base, suf),
```

```
<conc-strings> (nbase, nsuf)))
```

Chapter: System Interface
Section: Time

TIME

The primitive `dtime` returns the CPU time in seconds since the last call to `dtime`. Can be used to time strategies.

```
module time
strategies

  dtime = prim("_ST_dtime")
```

Chapter: Component Interfaces
Section: Pretty-Printing

ABOX

It is often desirable to format a (program) text in various output formats such as plain text, LaTeX, or HTML. This module provides an abstract-syntax interface to Merijn de Jonge's Box format for generic pretty-printing. The `gpp` package provides formatters that translate boxes to ASCII text, HTML and LaTeX. This interface can be used to format a program by transforming its abstract syntax tree to a Box term and then formatting that with one of the formatters from `gpp`.

KERNEL

The basic constructor for boxes is the `S` operator that creates a string box. The box `S("string")` denotes the literal text `string`.

The fundamental constructors for composing boxes from other boxes are `H` that places boxes horizontally next to each other and `V` that places boxes vertically above each other. The box `H([],[S("a"),S("b")])` denotes the text `a b`, the box `V([],[S("a"),S("b")])` denotes the text

```
a
b
```

The constructor `HV` is a less rigid combination of horizontal and vertical composition. It places boxes horizontally as long as there is space and then continues placing boxes on the next line. The constructor `ALT` chooses the argument box which fits best.

```
module abox
imports list-cons tuple
signature
  sorts Box
  constructors
    S     : String -> Box
    H     : List(S-Option) * List(Box) -> Box
    V     : List(S-Option) * List(Box) -> Box
    HV    : List(S-Option) * List(Box) -> Box
    ALT   : Box * Box -> Box
```

SPACE OPTIONS

The first argument of the Box operators above is a list of space options that determine the horizontal or vertical spacing between the elements composed by the operator. An option `SOpt(HS,1)` determines a horizontal spacing of 1 between boxes. If no space option is specified a default spacing is applied. For `H` the default is a horizontal spacing of 1 and for `V` the default is a horizontal spacing of 0.

```
    SOpt  : Space-Symbol * Int -> S-Option
```

```
SOptb : Space-Symbol * Box -> SOptb
VS    : Space-Symbol
HS    : Space-Symbol
IS    : Space-Symbol
```

ALIGNMENTS

The alignment operator can be used to produce tables.

```
A     : List(A-Option) * List(S-Option) * List(Box) -> Box
R     : List(S-Option) * List(Box) -> Box
AL    : List(S-Option) -> A-Option
AC    : List(S-Option) -> A-Option
AR    : List(S-Option) -> A-Option
```

FONTS

The box constructor `FBOX` can be used to declare the font to be used in the argument box. The scope of the font declaration reaches until the enclosed font constructors. The parameters `KW`, `VAR`, `NUM` and `MATH` declare abstract fonts for the categories keyword, variable, number and mathematical expression.

```
FBOX  : Font-Operator * Box -> Box
F     : List(F-Option) -> Font-Operator

FInt  : Font-Param * Int -> F-Option
FFID  : Font-Param * FID -> F-Option
FN    : Font-Param
FM    : Font-Param
SE    : Font-Param
SH    : Font-Param
SZ    : Font-Param
CL    : Font-Param

KW    : Font-Operator
VAR   : Font-Operator
NUM   : Font-Operator
MATH  : Font-Operator
```

CROSSREFERENCES

The constructors `LBL` and `REF` define labels and crossreferences to these labels, respetively.

```
LBL   : String * Box -> Box
REF   : String * Box -> Box

C     : List(S-Option) * List(Box) -> Box
```

```
L     : Box * Box -> Box
LINT  : Int * Box -> Box
```

PRETTY-PRINT TABLES

```
Arg   : Int       -> Box
Arg2  : Int * Int -> Box
```

ABOX-EXT: EXTENSION TO ABOX INTERFACE

This module provides abstractions on top of the ABox interface defined in module `abox`.

```
module abox-ext
imports abox
```

OVERLAYS

The following overlays define abbreviations of frequently used constructs.

```
overlays

  EmptyBox = H([],[])

  H0(xs)  = H([SOpt(HS(),0)],xs)
  H1(xs)  = H([],xs)
  H2(xs)  = H([SOpt(HS(),2)],xs)

  V0(xs)  = V([SOpt(VS(),0)],xs)
  V1(xs)  = V([SOpt(VS(),1)],xs)

  HV1(xs) = HV([],xs)

  Keyword(x) = FBOX(KW(), x)
  Parens(x)  = H0([S("("), x, S(")")])
  Indent(x)  = H0([S("  "), x])
```

CONSTRUCTOR STRATEGIES

```
rules

  MkS : x -> S(x)
  MkParens : x -> Parens(x)
```

SEPARATOR LISTS

The following strategies define various ways to format lists with separators.

(*** These need to be cleaned up ***)

```
strategies

  sep-list(s1, s2) = map(s1); separate-by(!S(<s2>()))

  hpref-sep-list(s1, s2, s3) =
        map(s1);
```

```
        ([] + [ \ x -> H1([<s2>(), x]) \ | hpref(s3)])

  hpref(s) = map(\ x -> H1([<s>(), x]) \)

  hpost-sep-list(s1, s2) =
    rec y([] + [s1] <+ [\ x -> H0([<s1> x, <s2>()]) \ | y])

  semicolons = hpost-sep-list(id,!S(";"))

  presemicolons(s) = hpref-sep-list(id, s, !S(";"))
  prebars(s)       = hpref-sep-list(id, s, !S("|"))

  commas = sep-list(id,!",")
  post-commas = hpost-sep-list(id,!S(","))

rules

  HPost(s) : [x | xs] -> [H0([x, <s>()]) | xs]

  Quote : x -> H0([S(quote), x, S(quote)])
          where <implode-string> [34] => quote

  Quote' : x -> H0([S(quote), x, S(quote)])
          where <implode-string> [92, 34] => quote

  CommaList(s) : x -> H0(<sep-list(s,!", ")> x)

  MkParens : x -> Parens(x)
```

UGLY-PRINT

```
module ugly-print
imports abox-ext
strategies

  ugly-print =
  rec x(try(
        UP-Int
        <+ UP-Str
        <+ UP-Cnst
        <+ UP-Lst(x)
        <+ UP-App(x)
  ))

rules

  UP-Cnst :
    f#([]) -> S(f)

  UP-App(s) :
    f#(xs) -> H0([S(f), Parens(V0(<map(s); post-commas> xs))])

  UP-Str :
    x -> <Quote> S(x) where <is-string> x

  UP-Int :
    x -> S(<int-to-string> x) where <is-int> x

  UP-Lst(s) :
    [] -> S("[]")

  UP-Lst(s) :
    l @ [x | xs] -> H0([S("["), V0(<map(s); post-commas> l), S("]")])
```

BIBLIOGRAPHY

[1] Eelco Visser. *The Stratego Reference Manual*. Institute of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[2] Eelco Visser. *The Stratego Tutorial*. Institute of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 1999.