Language Independent Traversals for Program Transformation

Eelco Visser
Department of Information and Computing Sciences, Universiteit Utrecht
http://www.cs.uu.nl/~visser/
visser@acm.org

May 2000

SUMMARY

Many language processing operations have a generic underlying algorithm. However, these generic algorithms either have to be implemented specifically for the language under consideration or the language needs to be encoded in a generic format that the generic algorithm works on. Stratego is a language for program transformation that supports both specific and generic views of data types.

A Stratego program defines a transformation on first-order ground terms. Transformation rules define single transformation steps. Transformation rules are combined into transformation *strategies* by means of combinators that determine where and in what order rules are applied. These combinators include: primitives for traversal to the direct subterms of a node, allowing the definition of many kinds of full term traversals; full control over recursion in traversals; patterns as first-class citizens; generic term construction and deconstruction.

These features create a setting in which it is possible to combine generic traversal with data type specific pattern matching, and separating logic (transformation, pattern matching) from control (traversal). This makes it possible to give language independent descriptions of language processing operations that can be instantiated to a specific language by providing the patterns of the relevant constructs. These generic algorithms only touch relevant constructors and do not need to know the entire datatype, making the algorithms insensitive to changes in the abstract syntax that do not affect the constructors relevant to the operation.

Stratego is currently implemented by compilation to C code. All constructs of the language are implemented directly, i.e., the compiled program is as large as the specification, in contrast to approaches that rely on preprocessing or program generation which may have a scaling problem when dealing with large languages.

The approach to generic programming in Stratego is illustrated by means of several examples including free variable extraction, bound variable renaming, substitution and syntactic unification.

In: J. Jeuring (editor) Proceedings of the Workshop on Generic Programming (WGP2000), Ponte de Lima, Portugal, July 2000. Technical Report, Department of Information and Computing Sciences, Universiteit Utrecht.

REPRESENTATION IMPEDES GENERIC IMPLEMENTATION OF LANGUAGE PROCESSING ALGORITHMS

Many language processing operations have a generic underlying algorithm. However, these generic algorithms either have to be implemented specifically for the language under consideration or the language needs to be encoded in a generic format that the generic algorithm works on. Stratego [2, 17, 18] is a language for program transformation that supports both specific and generic views of data types.

LANGUAGE INDEPENDENT PROGRAM TRANSFORMATION?

Program transformation systems are usually developed for transformations of programs in a single language. After all transforming a C++ program is quite different from transforming a Haskell program. However, many components of transformation systems are independent of the particular language under consideration. For example, consider the following tasks: collecting the free variables in an expression, renaming the bound variables in an expression, flattening the import structure of a program module, finding the modules imported by a module, computing the call graph of a program, unifying two expressions.

For most languages, each of these tasks is an instance of the same generic algorithm. In practice, however, one needs to spell out a new instance of the algorithm for each language since the intermediate representations are different. For example, consider the fragment of the abstract syntax of C++ in Figure 1 and a fragment of an intermediate representation for a functional language in Figure 2. Then consider the task of renaming variables for each of these languages. At a high enough level of abstraction these tasks come down to the same thing: For each binding construct create new names for the bound variables and substitute all bound occurrences of these variables by a new name. This requires traversing the program to find binding constructs, renaming their variables, finding variable occurrences and relating them to their renamings.

SPECIFIC VS GENERIC REPRESENTATION

On the whole there are two approaches to tackle the problem. (1) Adapt the generic algorithm to work on the specific data type under consideration. (2) Adapt the data type so that the generic algorithm can work with it.

In the first approach a language is represented by a specific data type as in Figures 1 and 2. Implementing the scheme for variable renaming requires writing out a traversal that is specific for the language under consideration and that touches all constructors of the language. The advantage of this approach is that transformations specific for the language can be expressed directly. The disadvantage is the lack of genericity. This can be mitigated by generating instantiations of generic functionality, e.g., [5, 4, 10]. However, this can result in large programs with associated scaling problems and is inflexible since the generator needs to be reprogrammed for each new generic algorithm.

In the second approach, programs are translated to a data type that is suited especially for the algorithm under consideration, e.g., a generic intermediate representation. This might not fit well with other, language specific, operations that one wants to carry out. Another variant of this approach is the encoding of programs in a universal data type such as XML [19] or ATerm [3] that make generic operations easier. The advantage of this approach is that generic algorithms can be expressed concisely. The disadvantage is that language specific operations have to be encoded in the universal data type as well.

STRATEGO PROVIDES GENERIC AND SPECIFIC VIEWS OF REPRESENTATION

Stratego [2, 17, 18] is a language for automatic program transformation that combines both a specific *and* a generic view of programs in a light-weight manner. This is achieved by providing transformation rules that work on a specific data type and basic building blocks for composing generic traversals over programs that apply transformation or compute analyses. This approach combines the advantages of the approaches discussed above without their disadvantages.

The Stratego library [16] contains a large number of language independent program transformation and analysis algorithms. In this paper we explain the building blocks of Stratego programs by means of several small examples and illustrate how these can be used for generic programming by means of several examples: extracting free variables, renaming bound variables, safe substitution under variable bindings and syntactic unification.

```
signature
  constructors
    Program : List(SD) -> Program
    VarDecl : Name * Type * List(Spec) * Expr -> SD
    FunDecl : Name * ArgList * Type * List(Spec)
                             * Qualifier * CInit * SD -> SD
    Block   : List(SD) -> SD
    If      : Expr * SD -> SD
    Var     : Name -> Expr
    App     : Expr * List(Expr) -> Expr
    IdName  : String -> Name
```

Figure 1: Fragment of an abstract syntax definition for representation of C++ programs.

```
signature
  constructors
    Var    : String * Type -> Exp
    App    : Exp * List(Exp) -> Exp
    If     : Exp * Exp * Exp -> Exp
    Abs    : String * Type * Exp -> Exp
    Let    : String * Type * Exp * Exp -> Exp
    Letrec : List(Fdec) * Exp -> Exp
    Fdec   : String * Type * List(String) * Exp -> Fdec
```

Figure 2: Fragment of the signature of a typed abstract syntax for a functional language.

CONTROLLING THE APPLICATION OF TRANSFORMATION RULES

A Stratego program defines a transformation on first-order ground terms. Transformation rules define single transformation steps. Transformation rules are combined into transformation *strategies* by means of combinators that determine where and in what order rules are applied.

FIRST-ORDER TERMS

Programs are represented by means of abstract syntax trees or *terms*. A term is a structure of the form `C(t1,...,tn)`, where the `ti` are terms. In case `n=0` the parentheses can be dropped, i.e., `C` is equivalent to `C()`. Lists of the form `[t1,...,tn|t]` are abbreviations for terms `Cons(t1,...,Cons(tn,t))`. The list `[t1,...,tn]` is a special case denoting `[t1,...,tn|[]]`, where `[]` denotes the term `Nil`. Term tuples of the form `(t1,...,tn)` abbreviate terms `TCons(t1,...,TCons(tn,TNil))`. Signatures (e.g., Figures 1 and 2) describe subsets of the universal set of terms.

A *strategy* is a program that transforms a term to another term or fails to do so, in which case there is no result of transformation.

TRANSFORMATION RULES

Transformation rules are strategies that perform a single transformation step. A rule of the form `L : t1 -> t2 where s` defines a strategy `L` that transforms a term that matches with the pattern `t1` into the instantiation of the pattern `t2` if the condition `s` (a strategy expression) is satisfied, i.e., when it succeeds to apply. The notation `<s> t` denotes the application of strategy `s` to term `t`. Figure 3 gives some examples of transformation rules for the functional language.

SEQUENTIAL NON-DETERMINISTIC PROGRAMMING

Strategies can be combined into new strategies by means of sequential composition `s1;s2` (first apply `s1` then `s2`), non-deterministic choice `s1+s2` (apply `s1` or `s2`), left choice `s1<+s2` (first try `s1`, if that fails apply `s2`), negation `not(s)` (succeeds if `s` fails) and recursion `rec x(s)` (recursively call the strategy when applying `x`). The identity strategy `id` always succeeds and does not transform the subject term.

A strategy definition `f(x1,...,xn) = s` defines a new strategy operator `f` with `n` strategies as arguments. Note that a strategy definition does not mention the term to which the strategy is applied, but combines the parameter strategies into a new strategy. Figure 4 defines several flavours of repeated application of a strategy.

TERM TRAVERSAL

The strategies discussed above apply transformations at the root of a term. In order to achieve transformations of subterms, some form of term traversal is needed. The strategy `all(s)` applies the strategy `s` to each direct subterm of a term. For example, `<all(s)>F(A,B)` corresponds to `F(<s> A, <s> B)`. The strategy `one(s)` applies `s` to exactly one direct subterm of a term. For example,

`<one(s)>F(A,B)` result in either `F(A, <s> B)` or `F(<s> A, B)`.

Figure 5 defines several generic traversals by means of recursion, sequential composition or left choice, and the primitive term traversal operators `all` and `one`. Figure 6 shows how generic control strategies can be used to apply selected transformation rules.

```
rules
  Alpha : e -> <lrename> e
  Beta  : App(Abs(x, t, e1), e2) -> <lsubs> ([(x,e2)], e1)
  LBeta : Let(x, t, e1, e2) -> <lsubs> ([(x, e1)], e2)
  Eta   : Abs(x, t, App(e, Var(x, t))) -> e
          where <not(in)>(x, <lvars> e)
  Dead  : Let(x, t, e1, e2) -> e2 where <not(in)>(x, <lvars> e2)
```

Figure 3: Some transformation rules for a typed functional language. The strategies `lrename` (rename bound variables), `lsubs` (substitute expressions for variables), `lvars` (extract free variables) and `in` (check occurrence in term) are all instantations of generic strategies.

```
strategies
  try(s)             = s <+ id
  repeat(s, c)       = rec x(s; x <+ c)
  repeat(s)          = repeat(s, id)
  repeat-until(s, c) = rec x(s; (c <+ x))
  for(i, c, s)       = i; repeat-until(s, c)
  while(c, s)        = rec x(try(c; s; x))
  do-while(s, c)     = rec x(s; try(c; x))
```

Figure 4: Several kinds of iteration strategies defined by means of sequential composition, choice and recursion. The parameter `s` denotes the body of the loop and `c` the stop condition. Note that the success of the application of the body `s` contributes to the control flow. For example, `repeat(id)` will not terminate.

```
strategies
  topdown(s) = rec x(s; all(x))    bottomup(s) = rec x(all(x); s)
  downup(s)  = rec x(s; all(x); s)
  oncetd(s)  = rec x(s <+ one(x))  alltd(s) = rec x(s <+ all(x))
```

Figure 5: Several flavours of one-pass traversals over terms. The first three apply a strategy to all subterms in different orders, `oncetd` finds a single application of `s` somewhere in a term and `alltd` applies a strategy along a frontier.

```
strategies
  remove-dead = bottomup(repeat(Eta + Dead))
```

Figure 6: A transformation on functional expresssions that removes dead code by traversing the expression bottom up and repeatedly trying to apply the rules `Eta` or `Dead` at each subterm.

BASIC BUILDING BLOCKS OF TRANSFORMATION RULES

> Transformation rules are not the smallest particles of program transformation. Rules are composed of pattern matching and pattern instantiation. Making these atomic actions available at the language level opens many possibilities for genericity. Generic term construction and deconstruction are other atomic actions that allow generic processing.

MATCHING AND INSTANTIATING PATTERNS

Transformation rules can be considered as strategies that first match a pattern (a term with variables), then check the condition and finally build an instantation of the pattern on the right-hand side. That is, given the operations `?t` for matching a pattern `t` and `!t` for instantiating a pattern `t`, a rule `t1 -> t2` can be defined as a strategy `?t1;!t2`. Since patterns contain variables, their scope needs to be restricted. The construct `{x1,...,xn:s}` restricts the scope of the variables `xi` to the strategy `s`. Using these primitives, a rule of the form `L : t1 -> t2 where s` is just syntactic sugar for a strategy definition `L = {x1,...,xn: ?t1; s; !t2}`, where the `xi` are the variables appearing in the rule. The condition of a rule is just a strategy that needs to succeed in order for the rule to apply.

Making pattern matching and instantiation first-class citizens allows the easy definition of other useful constructs. If it is not necessary to make a rule reusable by giving it a name one can use an anonymous rule of the form `\t1 -> t2 where s\`, which denotes the strategy `{x1,...,xn: ?t1; s; !t2}`, where the `xi` are the variables of the *left-hand side* `t1`. Other syntactic sugar that is defined in this manner is `?t <= s` and `s => t`, both denoting `s; ?t`, and `<s> t` denoting `!t; s`.

It is also possible to directly program with match and build strategies. Figure 7 shows several possibilities for using match and build in traversals.

GENERIC TERM CONSTRUCTION AND DECONSTRUCTION

Using pattern matching it is possible to get at the direct subterms of a constructor application for a specific constructor. For some operations we are not interested in the constructor but just in combining the (transformed) subterms in some manner. The pattern `c#(xs)` matches any term of the form `C(t1,...,tn)` and binds `"C"` (the name of `C`) to `c` and `[t1,...,tn]` to `xs`.

Figure 8 shows how generic term deconstruction can be used in the generic definition of an algorithm for the collection of subterms.

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

  Inline : Let(x, t, e1, e2) -> Let(x, t, e1, e2')
           where <oncetd(?Var(x,t); <lrename> e1)> e2 => e2'

  get-imports :
    Module(m, b) -> <collect(\Import(x) -> (m, x)\)> b

strategies
  vars = collect(?Var(_,_))
```

Figure 7: Several examples of using match and build strategies.

The strategy `in` succeeds if `a` occurs in `t`.

The strategy `Inline` replaces an occurrence of `Var(x)` by the expression `e1`. Note that this is a non-linear match, i.e., the pattern `?Var(x,t)` only matches terms for which variable name and type are the same as in the declaration in the left-hand side of the rule.

The strategy `get-imports` collects all imports from a module and returns a list of pairs of the name of the importing module and the imported module. This strategy assumes a strategy `collect(s)` that collects subterms for which strategy `s` succeeds.

Using the same `collect` strategy, the strategy `vars` collects all subterms that match the pattern `Var(_,_)`, where '_' is a wildcard for pattern matching.

```
rules
  collect-kids(s) : _#(xs) -> <foldr(![], union, s)> xs

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

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

Figure 8: A generic strategy for collecting subterms from a term.

The strategy `collect(s)` yields the set of sub-terms (represented as a list) for which `s` succeeds. It tries to apply `s` to the subject term. If that succeeds a subterm has been found and it is placed in a singleton list. Otherwise the collection continues in the direct subterms (the kids). Note that `;` binds stronger than `<+`.

The `collect-kids` strategy deconstructs the subject term and takes the union of the subterms found in the direct subterms. This is achieved by means of a fold right over the list of direct subterms, where the subterms of the elements are collected by means of the recursive call to the recursion variable `x` of `collect`, which is passed to `collect-kids`.

COLLECTING FREE VARIABLES

Free variable extraction is governed by the shape of variables and variable binding constructs. Other elements of the abstract syntax are irrelevant for the algorithm.

Collecting the free variables of an expression is a common operation in program transformation. One application is closure conversion in which the free variables of a nested function definition are turned into explicit parameters. Figure 9 shows a transformation rule for closure conversion in a functional language, which makes use of the strategy `lvars` for extracting the free variables from a function definition.

Collecting the free variables of an expression first of all requires the declaration of the shape of variables. The strategy `collect(?Var(_,_))`, which collects all subterms of an expression that match the pattern `Var(_,_)`, does almost what we want, but not quite. Figure 10 defines the rule `LVars` that maps variables to a list containing the pair of the variable name and its type.

In addition to collecting the variables, variables that are bound should be removed. That is, we also need to know what the binding constructs are and which variables they bind. This requires the declaration of the binding constructs of the language. Figure 10 defines the rules `LBnd` that map each binding construct to a list of pairs of variable name and type for the variables that are bound by the construct. Note that multiple rules (definitions) with the same name are equivalent with the non-deterministic choice of the bodies of the rules (definitions).

Given the shape of variables and the bound variables we can define the collection of free variables. Figure 10 defines free variable collection in terms of `LVars` and `LBnd` by means of the generic strategy `free-vars`.

Figure 11 defines the generic strategy `free-vars` as a variant of `collect`. In addition to computing the free variables of the direct-subterms with the strategy `collect-kids(x)`, the bound variables are computed with the parameter strategy `boundvars`. The difference between the free variables and the bound variables determines the free variables for a node.

This model for free variable extraction works well for representations in which the scope of each binding construct ranges over all its subterms. However, this is not always the case. The `Let` construct for instance does usually not bind its variable in the expression assigned to the variable it defines. For example, in the expression `Let("x",t,Var("x",t),e)`, the sub-expression `Var("x",t)` refers to a definition outside the scope of this `Let`, i.e., is free in the expression.

```
rules
CloseConv :
  Letrec([Fdec(g,Arrow(ts,t),xs,e)], e') ->
  Letrec([Fdec(g',Arrow(<conc>(ts,ts'),t),<conc>(xs,ys),e)], e'')
  where ?g' <= new;
        ?vs <= <lvars> Fdec(g,Arrow(ts,t),xs,e);
        ?ys' <= <map(\(y,t) -> Var(y,t)\)> vs;
        ?(ys,ts') <= <unzip(id)> vs;
        ?e'' <= <topdown(try({as:?App(g, as);
                                  !App(g', <conc>(as, ys'))}))> e'
```

Figure 9: A transformation rule for closure conversion. The function declaration for `g` is replaced with an new function declaration for `g'` (a new name) that takes as extra arguments the free variables of the body of `g`. All calls to `g(as)` are replaced by calls to `g'(as,ys')`.

```
strategies
  lvars = free-vars(LVars, LBnd)
rules
  LVars : Var(x,t) -> [(x,t)]

  LBnd  : Abs(x, t, e) -> [(x,t)]
  LBnd  : Let(x, t, e1, e2) -> [(x,t)]
  LBnd  : Letrec(fdecs, e) ->
            <map(\Fdec(f,t,xs,e) -> (f,t)\)> fdecs
  LBnd  : Fdec(f, Arrow(ts,t), xs, e) -> <zip(id)> (xs,ts)
```

Figure 10: Collecting free variables from functional expressions. The strategy `lvars` is the strategy for extracting free variables. The rule `LVars` maps a variable to the pair of its name and type. The rules `LBnd` map binding constructs to a list of the variables they bind.

```
strategies
  free-vars(getvars, boundvars) =
    rec x(getvars
          <+ split(collect-kids(x), boundvars <+ ![]); diff)
rules
  split(f, g) : x -> (<f> x, <g> x)
```

Figure 11: A generic algorithm for collecting free variables with the assumption that all binding constructs declare bindings that range over all subterms. The strategy `diff` computes the difference between a pair of lists.

REFINING THE DEFINITION OF FREE VARIABLE COLLECTION

In addition to determining the names of bound variables it is necessary to determine the arguments in which these variables are binding. Furthermore, the assumption that collection can stop at variables does not hold for representations that declare variable names and contain other variable holding constructs as subterms. To specify a refinement of a generic traversal Stratego provides congruence operators, which allow the specification of traversal for specific constructors.

BINDING POSITIONS

If variables are not bound in all direct subterms of a binding constructs, the model of free variable extraction needs to be refined. Binding positions can be declared by indicating for each binding construct the arguments in which variables are bound, the arguments in which variables are not bound and the arguments that do not contain variables at all and can thus be ignored.

Figure 12 defines the strategy `LBoundIn` with parameters `bnd` for positions where variables are binding, `ubnd` for positions where variables are not binding and `ignore` for positions where no variables occur. It declares binding positions for abstraction, let, letrec and function declaration by means of *congruence* operators.

SPECIFIC TERM TRAVERSAL WITH CONGRUENCES

The generic traversal operators `all` provides generic traversal through arbitrary constructors. Sometimes it is necessary to refine the traversal of specific constructors. For each constructor `C : t1 * ... * tn -> t` declared in a signature there is a corresponding strategy operator `C(s1,...,sn)` (the congruence over `C`) that transforms a term `C(t1,...,tn)` to `C(<s1>t1,...,<sn>tn)`.

Figure 12 uses such congruence operators to direct traversal. For example, the pattern `Let(ignore,ignore,ubnd,bnd)` declares that variables are not bound in the third and are bound in the fourth argument. The declaration of binding positions `LBoundIn` can be passed to the refined version of `free-vars`, which is defined in Figure 13.

The computation of the free variables of the direct subterms is split into three cases. In the first case, if a term is a variable it is returned. In the second case, if the current node is a binding construct (`boundvars` succeeds), the `boundin` strategy is used to differentiate between binding and non-binding positions. At the binding positions the difference between the free variables and the bound variables is computed, at the unbound positions all free variables are computed and the other positions are ignored by building the empty list. In the third case, if the term is neither a variable nor a binding construct the free variables of its subterms are collected.

VARIABLES CONTAINING VARIABLES

The model in the extraction methods described above implies that variables are leaves of abstract syntax trees and do not contain variables themselves. This is not always appropriate. For example, consider an abstract syntax with an application declared as in Figure 14. Here variables in function and argument positions have a different shape and in addition an application contains terms that are variables. Figure 15 defines an alternative free variable collection strategy that does not stop when encountering a variable, but instead takes the union of the variables at a node and the variables contained in its subterms.

```
strategies
  lvars = free-vars(LVars, LBnd, LBoundIn)
  LBoundIn(bnd, ubnd, ignore) =
    Abs(ignore, ignore, bnd)
  + Let(ignore, ignore, ubnd, bnd)
  + Letrec(bnd, bnd)
  + Fdec(ignore, ignore, ignore, bnd)
```

Figure 12: Declaration of binding positions by means of congruence operators.

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

Figure 13: Algorithm for collecting free variables that takes binding positions into account.

```
signature
  constructors
    App : String * SimpleExp -> Exp
    Var : String -> SimpleExp
rules
  LVars : App(f, es) -> [f]
  LVars : Var(x) -> [x]
strategies
  lvars = free-vars2(LVars,LBnd)
```

Figure 14: Abstract syntax representation in which variables are not leaves and extraction of variable names from expressions.

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

Figure 15: Algorithm for collecting free variables that takes variables in subterms of variables into account. A variant of this algorithm taking into account binding positions can be created analogously to Figure 13

RENAMING BOUND VARIABLES

> Renaming of bound variables depends on the shape of variables and the shape of binding constructs. For binding constructs, in addition to determining what variables are bound and in which arguments they are binding, it is necessary to declare where new variables should be pasted. In order to keep track of renamings it is also required to distribute an environment along with the renaming traversal.

Renaming of bound variables is used to prevent name clashes between variables, for example as a preparation for inlining to prevent free variable capture.

PASTING NEW VARIABLE NAMES

The same considerations as for free variable extraction hold. A renaming algorithm depends on the shape of variables and on the shape of binding constructs.

A new aspect is that the renamed variables should be pasted back into the binding construct. This is the inverse of extracting variables bound by a binding construct, i.e., replace the current binding variables by their renamings. For example, replace `Abs("x",t,e)` by `Abs("y",t,e)`. Since extraction of bound variables from a binding construct is an arbitrary strategy we cannot automatically reverse it. Therefore, an additional strategy is needed that replaces binding variables by new names.

Figure 16 defines the strategy `LPaste` with parameter `nwvars`. Using congruences it declares how the new variables should be pasted back into the binding constructs. The parameter strategy `nwvars` builds a list of new variables names. Using the strategies `IsLvar`, `LBnd`, `LBoundIn` and `LPaste`, the strategy `rename` composes a renaming strategy for functional expressions.

DISTRIBUTING ENVIRONMENTS

Figure 17 defines the generic algorithm for renaming bound variables. It is defined by means of the `env-alltd` traversal (Figure 18), which distributes an environment through a traversal. The environment for renaming is a list of pairs `(x,y)` with `x` a reference to the old variable and `y` the name of the new variable.

The `rename` strategy is defined by means of the rules `RnVar`, `RnBinding` and `DistBinding`. `RnVar` renames a variable by looking up its name in the environment. `RnBinding` creates new variable names for the variables of a binding construct and pastes the new names back into the term. It also creates an extended environment mapping the old variables to the new variables. `DistBinding` applies the renaming to the binding and non-binding positions of the binding construct using the same `boundin` strategy used for free variable extraction. At the binding positions the extended environment is used, at the non-binding positions the old environment is used.

```
strategies
  lrename = rename(IsLvar, LBnd, LBoundIn, LPaste)
rules
  IsLvar(s) : Var(x,t) -> Var(<s>(x,t), t)
  Hd : [x|xs] -> x
strategies
  LPaste(nwvars) =
    Abs(nwvars; Hd, id, id)
  + Let(nwvars; Hd, id, id, id)
  + Letrec(split(id, nwvars);
             zip(\(Fdec(f, t, xs, e), g) -> Fdec(g, t, xs, e)\)
          ,id)
  + Fdec(id, id, nwvars, id)
```

Figure 16: Instantiation of a generic renaming algorithm and declaration of pasting new variables in binding constructs.

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

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

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

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

Figure 17: Generic algorithm for renaming bound variables. The strategy `new` creates a new unique string.

```
rules
  dist(s) : (t, env) -> <all(\x -> <s>(x,env)\)> t
strategies
  env-alltd(s) = rec x(s <+ dist(x))
```

Figure 18: Generic traversals that distributes an environment.

SUBSTITUTING TERMS FOR VARIABLES

Substitution of expressions for variables in a language with variable bindings requires renaming of bound variables to prevent free variable capture and can be expressed as a generalization of the generic bound variable renaming algorithm.

Substitution of expressions for variables is another common operation in program transformation. One of the problems with substitution is the danger of variable capture when substituting under a binding. The solution is to rename bound variables while descending into the expression tree.

Figure 20 defines a generic definition of this solution as a generalization of the generic variable renaming strategy. Figure 19 shows how this algorithm can be instantiated.

```
strategies

  lsubs = substitute(IsLvar, LBnd, LBoundIn, LPaste)
```

Figure 19: Instantiation of a generic algorithm for substitution for the functional language.

```
strategies

  substitute(isvar, bndvars, boundin, paste) =
    ?(sbs, t); !(t, []);
    rec x(env-alltd(RnVar(isvar)
                    <+ SubsVar(isvar, !sbs)
                    <+ RnBinding(bndvars, paste);
                       DistBinding(boundin, x)))

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

Figure 20: Generic algorithm for substituting expressions for variables under variable bindings. Capture of free variables in the substituted expressions and substituting for bound variables is prevented by renaming all variables on the way.

UNIFYING TERMS WITH OBJECT VARIABLES

Syntactic unification depends on the shape of variables. For all other constructs the only requirement is that their structure is the same. This can be expressed generically.

Syntactic unification is another example of a problem that can be solved by means of a generic, language independent algorithm that is instantiated with the shape of a language construct. Figure 21 shows the definition of a unification algorithm by instantiation of a generic algorithm. Figure 22 presents the definition of a generic unification algorithm. The algorithm does not take variable bindings into account, nor does it unify modulo equations.

```
signature
  sorts Types
  constructors
    TVar  : String              -> Type
    Arrow : Type * Type         -> Type
    Prod  : Type * Type         -> Type
    Tcons : String * List(Type) -> Type
strategies
  tp-unify = unify(TVar(id))
```

Figure 21: Instantiation of generic unification algorithm for polymorphic types.

```
strategies
  unify(isvar) =
    for(\ pairs -> (pairs, []) \
       ,\ ([], sbs) -> sbs \
       ,UfIdem + UfVar(isvar) + UfSwap(isvar) <+ UfDecompose)
rules
  UfIdem : ([(x,x) | ps], sbs) -> (ps, sbs)

  UfVar(isvar) :
        ([(x,y) | ps], sbs) -> (ps', [(x, y) | sbs''])
        where <isvar> x; <not(in)>(x,y);
              ?(sbs'', ps') <= <subs(isvar,![(x,y)])> (sbs, ps)

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

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

Figure 22: Generic definition of syntactic unification. The algorithm manipulates a list of pairs of terms to be unified and builds a substitution. Note that the decomposition rule uses generic term deconstruction `f#(xs)`.

COMPARING STRATEGO GENERICITY TO OTHER FRAMEWORKS

Traversal through data structures is a common problem in all programming languages. Stratego provides primitives for concisely describing a wide range of traversals.

TRAVERSAL SCHEMATA

In object-oriented programming the visitor design patttern [6] is used to separate the methods making up a traversal over (objects in) a set of classes from those classes. Visiting the actual classes is achieved through a hook (the `accept` method) that calls back the visitor. The genericity of this method is resctricted because the set of classes participating in the traversal is fixed and because the control of each traversal has to be spelled out. The generic visitors of Palsberg and Jay [12] overcome the problem of a fixed set of classes and default control by definining generic traversal through an object by means of reflection. In absence of a visit method for a class the generic behaviour is to traverse the fields of the object. Visit methods that override the default method still have to define the continuation of the traversal.

Attribute grammars (e.g., [14]) define propagation of attribute values through a tree. This requires defining for each attribute and for each node how the value is derived from other attribute values. Although some tools support copy rules that fill in undeclared propagations, traversals are not first-class citizens; it is not possible to define generic traversal patterns in the language itself.

Functional languages support the polymorphic definition of traversals over data types, but these have to be implemented for each data type separately. Wallace and Runciman [19] discuss two possible approaches for XML processing in Haskell: generic combinators working on a universal data type for XML, and type specific operations working on a Haskell data type generated from a DTD. They note that the direct programming style of the latter and the genericity of the former are difficult to combine.

GENERATION OF TRAVERSALS

In the absence of programmable traversals, it is possible to generate traversals. For example, Van den Brand et al. [5, 4] describe the generation of traversals for ASF+SDF from a grammar; Kort et al. [10] describe the generation of several kinds of morhphisms in Haskell from a grammar; The Java Tree Builder [1] generates parse tree classes and a default visitor for parse trees from grammars. The adaptive object-oriented programming approach [11, 13] is close in spirit to Stratego. It defines traversal through the object structure by means of a regular expression over paths. Code to be executed during the the traversal can be specified in wrappers. Different from Stratego is that traversals are not first-class citizens; it is not possible in AOP to define a generic traversal schema, which can be instantiated for a specific data type. Generation depends on the availability of the entire signature before generation and is rather rigid in that incorporation of a new traversal scheme requires reprogramming the generator. Generation can also lead to scaling problems [10].

Section: Conclusion

In polytypic functional programming in PolyP [9] and in in the generalization proposed by Hinze [8, 7], generic functions are defined on a universal data type and conversion functions between regular data and the universal data type are provided. Polytypic functions can be implemented by implementing the conversion or by generating specializations as needed in a program. Only the conversion functions from and to the universal data type are fixed and can be derived from the data type declaration automatically.

TYPING STRATEGIES

Stratego requires the declaration of constructors in signatures for those constructors that are used in strategies. Occurrences of constructors are checked against the signature. No further type system is imposed on specifications at this point, because there are several options to choose from. In one approach, the type system allows only type preserving transformations. A type system for this approach is straightforward to realize; since strategies do not have a type changing effect, only rules have to be checked. However, the restriction to type preserving transformations would not allow the kinds of specifications discussed in this paper. Transformations that are not type preserving in combination with the generic traversal of terms lead to a more difficult typing problem for which no satisfactory type system has been found yet.

Another non-standard feature of Stratego with respect to typing is that incomplete signatures are supported. This entails that subject terms can contain constructors that are not known in the specification. This allows for concise specifications and programs that are insensitive to irrelevant changes in the signature of subject terms. On the other hand, with complete signatures it would be possible to shortcut generic traversals based on type information.

Type systems are useful for catching run-time errors statically and for catching logical mistakes by checking for internal consistency. The former class of errors does not occur in Stratego because failure is integrated in the language. (A crash is caused by a bug in the implementation, not by a mistake from the user.) For the latter class of errors it is not yet clear what kind of checking can be provided. This is an area of research.

INPUT AND OUTPUT

Stratego programs can read and write terms represented in the annotated term (ATerm) format [3], i.e., simple first-order prefix terms. The ATerm format has a textual and a binary representation. In the latter maximal sharing of sub-terms is maintained. A parser (written in YACC, SDF2, Happy, etc.) can be connected to a transformer by having it output its (abstract) syntax tree in the form of an ATerm. Pretty-printers can be connected by having them read syntax trees represented as ATerms.

ACKNOWLEDGEMENTS

The generic algorithms in this paper were first developed for application in a formalization of the optimizer for RML in collaboration with Andrew Tolmach. He also provided feedback on an earlier version of this paper. Patrik Jansson and an anonymous referee commented on an earlier version of this paper.

BIBLIOGRAPHY

[1] www.cs.purdue.edu/jtb/index.html. The Java Tree Builder homepage.

[2] www.stratego-language.org.

[3] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[4] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In *Proc. 4th Working Conference on Reverse Engineering*, pages 144–155, 1997.

[5] Mark G. J. van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[7] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, September 1999. The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28.

[8] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, 3(4):159–180, 1999.

[9] Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997*, pages 470–482. New York, 1997.

[10] Jan Kort, Ralf Lämmel, and Joost Visser. Functional transformation systems. CWI, Amsterdam, March 2000.

[11] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, pages 94–101, May 1994.

[12] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[13] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, 1995.

[14] Joao Saraiva. *Functional Attribute Grammars*. PhD thesis, Universiteit Utrecht, 1999.

[15] J. R. Tracey, D. E. Rugh, and W. S. Starkey. Sequential thematic organization of publications (stop): How to achieve coherence in proposals and reports. *The Journal of Computer Documentation*, 23(3):4–68, August 1999.

[16] Eelco Visser. The Stratego Library. Technical report, Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 2000.

[17] Eelco Visser. The Stratego Reference Manual. Technical report, Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 2000.

[18] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).

[19] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *4'th International Conference on Functional Programming (ICFP 99)*, Paris, September 1999. ACM Press.

ABOUT THIS DOCUMENT

The format of this document is inspired by the STOP method [15], which advocates *sequential thematic organization of publications*, i.e., organization of a document as a series of topics each described in a two page spread. The author is interested in both positive and negative feedback on the readability of this document. The document was typeset using the `stop-report` document class for LaTeX.