7
Strategic Graph Rewriting

This chapter introduces GraphStratego, a prototype extension to the Stratego programming language for rewriting on terms with references. The main motivation behind GraphStratego is to extend the Stratego language with support for the notion of cycles in its program model. Many abstract program models require support for circular structures. Perhaps the most commonly used are the various flow graphs, such as for control- and data-flow, employed in compilers and program analysers. These models are often language independent.

GraphStratego improves the language independence for transformations in the sense that the abstract program models mentioned above may now be captured natively and processed using new language constructs. The extended language allows succinct formulations of transformation programs which traverse and rewrite graph-like models.

This chapter is a verbatim reprint of the paper *Strategic Graph Rewriting: Transforming and Traversing Terms with References*[KV06] written with Eelco Visser, with the exception of some minimal formatting changes.

7.1 Abstract

Some transformations and many analyses on programs are either difficult or unnatural to express using terms. In particular, analyses that involve type contexts, call- or control flow graphs are not easily captured in term rewriting systems. In this paper, we describe an extension to the System S term rewriting system that adds references. We show how references are used for graph rewriting, how we can express more transformations with graph-like structures using only local matching, and how references give a representation that is more natural for structures that are inherently graph-like. Furthermore, we discuss trade-offs of this extension, such as changed traversal termination and unexpected impact of reference rebinding.

7.2 Introduction

Strategic programming is a powerful technique for program analysis and transformation that offers the separation of local data transformations from data traversal logic. The technique is independent of language paradigm and underlying data structure, but is perhaps most frequently found in functional and term rewriting languages. Much of its power comes from the ability to define complex traversal strategies from a small set of traversal combinators. Traversal strategies are often used to traverse terms.

Terms, when implemented as maximally shared, directed acyclic graphs, have many desirable properties for representing abstract syntax trees (ASTs) as used in program transformation. In the ATerm model [vdBdJKO00], maximal sharing of subterms means that all occurrences of a term are represented by the same node in memory. Consequently copying of terms is achieved by copying pointers, and term equality entails pointer comparison. The model ensures *persistence*; modifying a term means creating a *new* term, the old term is still present. This makes it easy to support backtracking. Destructive updates of term references are not permitted, which allows efficient memory usage.

A consequence of the DAG representation is that terms cannot have explicit backlinks, i.e. *references* to arbitrary subterms elsewhere in the same term. Such references to other parts of the AST, including an ancestor of a term, are useful for expressing the results of semantic analyses as local information for rewrite rules. Turning ASTs into terms with references can turn some transformation problems from global-to-local into local-to-local. By adding explicit references in the terms, we arrive at a variation of term graphs [Plu01]. The added expressivity gained from term references allows us to encode graphs rather succinctly, and therefore also express structures that are inherently graph-like more naturally, such as high-level program models and many intermediate compiler representations, including call-, control flow- and type graphs, thus setting the stage for implementing transformations such as constant and copy propagation, type checking and various static analyses. We conserve the ability to express program transformations using local rewrite rules together with generic traversal strategies, obtaining a form of strategic graph rewriting. By adding explicit references, we also give up some of the desirable properties of terms mentioned above, as the references allow destructive updates, change the termination criteria for traversals, alter the matching behavior of rewrite rules and exhibit side effects due to reference rebinding.

In this paper, we explore the design of a strategic rewriting language on terms with references, and discuss the tradeoffs found in this design space. We will show that we can arrive at a practical and useful variation of term graphs that allows us to express graph algorithms and rewriting problems rather precisely.

The paper is organized as follows: In Section 7.3, we introduce basic term rewriting with Stratego, show new primitives for manipulating references and how existing constructs extend to handle terms with references. In Section 7.4, we show how the new language features are used to compute various common graph representations for programs from ASTs. In Section 7.5, we implement two basic graph algorithms: depth first search and strongly connected components, and their application to finding sets of mutually recursive functions. We also discuss an implementation of lazy graph loading. In Section 7.6, we discuss notable aspects of our implementation. In Section 7.7, we discuss related work. In Section 7.8, we discuss design tradeoffs and future work. We conclude in Section 7.9.

7.3 Extending Term Rewriting Strategies to Term Graphs

We now present the extension of the strategic term rewriting language Stratego with term references. First, we give an overview of the basic concepts of Stratego. Next, we extend terms to term graphs, i.e. terms with references, by introducing primitives for references. Finally, we discuss rewrite rules and generic traversals on term graphs.

7.3.1 Term Rewriting Strategies

The Stratego program transformation language [BKVV06] is an implementation of the System S [VB98] core language for term rewriting. We will not discuss all the features of System S and Stratego in this paper, but restrict ourselves mainly to conditional rewrite rules, strategies, traversals, and scoped, dynamic rules.

Terms A term t is an application $c(t_1, \ldots, t_n)$ of a constructor c to zero or more terms t_i. There are some special forms of terms such as lists ($[t_1, \ldots, t_n]$) and integer constants, but these are essentially sugar for constructor terms. A term pattern is a term p with variables x, written on the form $p(x)$.

Rewrite Rules A conditional rewrite rule, R : $p_l(x)$ -> $p_r(x)$ `where` c, is a rule named R that transforms the left-hand side pattern p_l to an instantiation of the right-hand side pattern p_r if the condition c holds. The following rule can be used to simplify addition expressions.

```
Simplify: Add(Int(x), Int(y)) -> z where <add> (x, y) => z
```

When applied to the term `Add(Int(1),Int(2))`, it will execute the condition `<add> (x,y)`, which is a strategy expression for computing the sum of two integers. The resulting term, 3, will be bound to the variable `z` using the operator => as assignment.

Strategies Strategies are used to implement rewriting algorithms. A *strategy* is built from primitive traversals (`one(s)`, `all(s)`, `some(s)`), combinators (`s1 <+ s2` (left choice), `s1 ; s2` (strategy composition)), identity (`id`), failure (`fail`) and invocations of rewrite rules or other strategies. The following strategy will attempt to apply the rule `Simplify` to all subterms of the current term. If `Simplify` fails, either because the left hand side pattern does not match, or because the condition does not hold, the strategy `id` will be applied instead. `id` always succeeds, and returns the identity (i.e. same term).

```
try-simplify = all(Simplify <+ id)
```

When applied to `Sub(Add(Int(1),Int(2)),Int(4))`, the first subterm of `Sub` will be rewritten to `3`, but `Simplify` will fail for the second subterm (`Int(4)`), and `id` will be applied instead. The end result is the term `Sub(Int(3),Int(4))`.

Strategy Expression	Meaning
`!`$p(x)$	*(build)* Instantiate the term pattern $p(x)$ and make it the current term
`?`$p(x)$	*(match)* Match the term pattern $p(x)$ against the current term
s_0 `<+` s_1	*(left choice)* Apply s_0. If s_0 fails, roll back, then apply s_1
s_0 `;` s_1	*(composition)* Apply s_0, then apply s_1. Fail if either s_0 or s_1 fails
`rec` $x(s(x))$	*(recursion)* Strategy x may be called in s for recursive application
`id, fail`	*(identity, failure)* Always succeeds/fails. Current term is not modified
`one(`s`)`	Apply s to one direct subterm of the current term
`some(`s`)`	Apply s to as many direct subterms of the current term as possible
`all(`s`)`	Apply s to all direct subterms of the current subterm
`\`$p_l(x)$ `->` $p_r(x)$`\`	Anonymous rewrite rule from term pattern $p_l(x)$ to $p_r(x)$
`?`x`@`$p(y)$	Equivalent to `?`x `;` `?`$p(y)$; bind current term to x then match $p(y)$
`<`s`>` $p(x)$	Equivalent to `!`$p(x)$ `;` s; build $p(x)$ then apply s
s `=>` $p(x)$	Equivalent to s `;` `?`$p(x)$; match $p(x)$ on result of s

Generic Traversal Strategies The primitive traversals `one(s)`, `some(s)` and `all(s)` can be composed using the strategy combinators to obtain *generic traversal strategies*, which are used to control the order of rewrite rule applications throughout a term.

```
topdown(s)  = s ; all(topdown(s))
bottomup(s) = all(bottomup(s)) ; s
```

The strategy `topdown(s)` will apply the strategy `s` to the current term before recursively applying itself to `all` the subterms of the new current term. `bottomup(s)` works similarly: `s` is applied to all subterms before the current term (with freshly rewritten subterms) is processed by `s`.

Build and Match Pattern matching is also available independently of rewrite rules by the match operator strategy, written `?`. The expression `?Add(Int(`x`),Int(`y`))` when applied to the term `Add(Int(1),Int(2))`, will bind `x` to 1 and `y` to 2. The inverse operator of match is build, written `!`, which will instantiate a pattern. Given the previous bindings of `x` and `y`, `!Add(Int(`*x*`),Int(`*y*`))` will instantiate to `Add(Int(1),Int(2))`. Figures 7.1(a), and 7.1(b) show simple ground terms and their corresponding ATerms.

7.3.2 References

Thus far, the language we have presented only operates on plain terms. We now extend terms with *references*. That is, in addition to a constructor application, a term can now also be a term reference r. A reference can be thought of as a pointer to another term. It is a special kind of term that our language extension knows how to distinguish from other terms (refer to Section 7.6 for implementation details).

We conceptually extend the language with three new operators for operating on references: *create new reference*, *dereference*, and *bind reference*. The creation of a new reference produces a fresh, unbound reference with a unique identifier. The identifier is used to compare references with each other. Like terms, references may be passed around as parameters, copied, matched, and assigned to variables. Additionally, references may be bound. Binding a term to a reference is similar to binding a value to a variable. After binding, a reference may be dereferenced. A deference will produce the term which was previously bound. These conceptual operators are available inside pattern expressions in three different forms, giving us *term graph patterns*.

Build and bind: `!`r˜$p(x)$ will instantiate the term pattern $p(x)$ and bind the resulting term to a new, unique reference which will be bound to the variable r. If the variable r is already bound to a reference, a new reference will not be created. Instead, the reference of r will be rebound to the new term.

Bind or match: `?`r˜$p(x)$ matches a reference r bound to a term that matches the term pattern $p(x)$. More specifically, to succeed, this expression must be applied to a reference r', r' must have the same reference identifier as r, and r' must be bound to a term which matches the term pattern $p(x)$. If the variable r is unbound, r will be bound to r' before the matching starts.

Dereference: `^r` will dereference the reference r, i.e., if `r` is bound to the term `t`, `^r` will produce `t`. With `r` bound to `t`, `?^r` is equivalent to `?t` and `!^r` is equivalent to `!t`.

With these operations, we can instantiate the term graph in Figure 7.1(c): `!r~Int(2);` `!Sub(r,r)`. Or more succinctly as one term graph pattern: `!Sub(r~Int(2),r)`. In the following we use some idioms: When we need a reference, but do not yet have its term, we use the expression `!r~()` to create a new reference bound to the "dummy" term `()`. If we want to match a reference, but do not care what it is bound to, we write `?r~_`.

General Graphs Term references may also be used to construct more general graphs, such as those shown in Figures 7.1(d) and 7.1(e). When constructing mutually dependent graphs, such as the `f()` and `g()` nodes in Figure 7.1(d), term graph construction must always be split into two stages. First, one half must be built with an unbound reference, e.g. `!f~FunDef("f",[g~()])`, then the graph is connected when the other half is built: `!g~FunDef("g",[f])`. Note the use of the idiom `g~()` to break cycles.

7.3.3 Rewrite Rules and References

Conditional rewrite rules on term graphs, R: $g_l(x)$ `->` $g_r(x)$ `where` c, mirror rewrite rules on terms. $g_l(x)$ and $g_r(x)$ are term graph patterns, as described previously and c is the rule condition. `Simplify` can now be reformulated to work on term graphs:

```
Simplify: r0~Add(r1~Int(x~_), r2~Int(y~_)) -> r0~Int(r3)
  where !r3~Int(r4~<add> (^x,^y))
```

Rewrite rules on term graphs will not maintain maximal sharing unless the programmer takes explicit care. This leads to differences in the equality checking of term graphs compared to equality checking of term. For efficiency, the built-in comparison of term graphs only exists in a "shallow" form, i.e. identity checking: Two terms with references are equal iff all subterms are structurally equal, and all references have the same identity. This means that terms may now in fact be structurally equal, but differences in their reference identities will prevent the shallow equality test from uncovering this.

Rebinding of References For terms, maximal sharing and constant time equality checking is always guaranteed by the ATerm library. When matching a regular variable against a term, the pointer to that term gets copied when it is used in a build. If the original term is later modified, copy-on-write is performed behind the scenes to ensure referential transparency. For term graphs, this is no longer the case, as references may be rebound at any time. Consider the graph building expression `!Sub(r~Int(2),r) => a ; !r~Int(3)`. Here, we assign the graph from Figure 7.1(c) to the variable `a`, but subsequently change the binding of the references contained in the term graph of `a`. Effectively, this will change the value of `a` after `a` was bound. This may seem dangerous, as it opens up for problems related to lack of referential transparency. Certainly, these issues must be managed, but it is important to note that the binding of terms to references is always done explicitly. It is not possible to retroactively create a reference to a subterm of another term. E.g., if the term `Sub(Int(2),Int(2))` from Figure 7.1(a) is bound to the variable `v`, it can never change, as it does not contain any references.

If side effects are unwanted, in the sense that references in the term of an already bound variable should never change, assignments of term graphs should be coupled

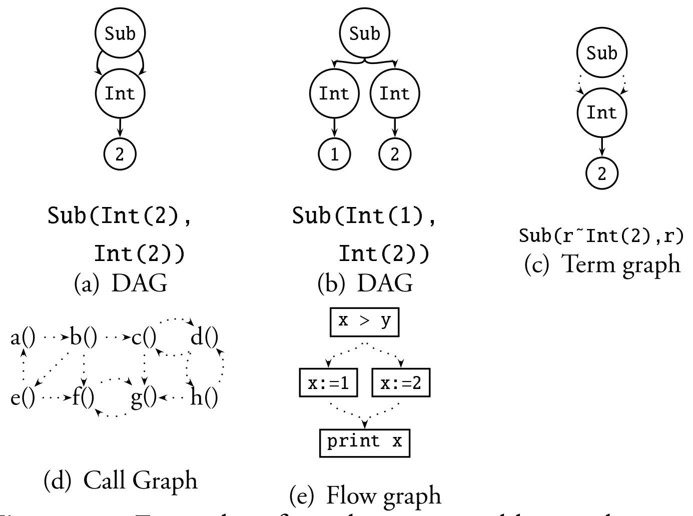

Figure 7.1: Examples of graphs supported by our language. References are shown as stippled edges.

with a call to `duprefs`, e.g. `!r~Int(2); !Sub(r,r); duprefs => a; !r~Int(3)`. Here, the references in the term graph `Sub(r,r)` will be replaced with new, unique references before the assignment, so that subsequent rebindings cannot affect the value of a.

7.3.4 Term Graph Traversal

We will now show how generic traversals are adapted to work on term graphs through an example of term graph normalization. Our goal is to use `Simplify`, shown earlier, to simplify the term graph `Sub(r~Add(r'~Int(1),r'), r)`. Specifically, we only want to simplify each referenced term once. This illustrates how "side-effects" can be used beneficially, and how term graph rewriting can be more efficient than term rewriting: we only need to consider identical terms once, because we can recognize them by their reference identifiers. This argument is only valid once a proper term graph has been constructed, however. Our current implementation makes no attempt at maintaining such term graph properties globally during arbitrary rewriting sequences.

Phased Traversals To manage termination of graph traversals, we introduce a concept of *phases*. Phases are used to ensure that each reference is only visited once, so that loops in the graph do not result in non-termination. We do this by introducing a new primitive strategy, `phase(s)`, and new variants of the primitive traversal operators: `wone(s)`, `wsome(s)` and `wall(s)`.

When applied to a reference r, `wall(s)` will first dereference r, obtaining the term t, then apply s to all subterms of t. The resulting term is rebound to r. If any subterm of t is also a reference, it will be dereferenced before s is applied, and rebound afterwards. `wone(s)` and `wsome(s)` are similar.

`phase(s)` will internally instantiate a new, globally unique marker and then apply `s` to the current term. Any invocations of `wone`, `wsome` and `wall` in `s` will take the marker into consideration. As each reference is dereferenced during the traversal, using `wone`, `wsome` or `wall`, the marker is placed on the reference. Any subsequent attempt at dereferencing will not yield a term result, thus making the reference untraversable. When the phase is exited, all markers for that phase are removed. It is possible to nest phases. The inner phase will instantiate a new, unique marker and can revisit references already visited by its enclosing phase.

During a phased traversal, it is sometimes necessary to control whether the dereferences due to matching or the ^ operator should be marked with the current phase marker or not. This can be controlled by using `wrap-ref(s)`, which dereferences, applies `s`, then rebinds, irrespective of any phase markers. Analogously, `wrap-phase-ref(s)` can be used to only visit unmarked references, and mark the reference after a visit.

Reference Expression	Meaning
`!`r~$p(x)$	*(Build and bind)* Instantiate term pattern $p(x)$ and bind result to r.
`?`r~$p(x)$	*(Bind or match)* See text in Section 7.3.2.
^r	*(Dereference)* Look up term for r. Fail iff r is unbound.
`duprefs`	Replace all references in the current term with new, unique ones
`phase(`s`)`	For traversals done by s, visit each reference at most once
`wrap-ref(`s`)`	Apply s to reference and rebind, irrespective of phase
`wrap-phase-ref(`s`)`	Apply s to reference and rebind, while respecting phase

Generic Graph Traversals The following traversal strategies are adaptations of the generic traversals for terms. They use phase markers to avoid visiting the same reference more than once. These strategies use `wall` and will rebind references they encounter to rewritten terms.

```
wtopdown(s)    = phase(rec x(s ; wall(x)))
wbottomup(s)   = phase(rec x(wall(x) ; s))
wdownup(s1,s2) = phase(rec x(s1; wall(x); s2))
```

Term Graph Normalization With the phasing and generic graph traversals in place, we can now express term graph normalization simply and precisely as:

```
simplify = wbottomup(try(Simplify))
```

When applied to the term `Sub(r~Add(Int(1),Int(2)), r)`, the result of `simplify` is `Sub(0~Int(3), 0~)`. Here, `0~t` indicates that the reference 0 is bound to the term `t`. Bindings of references are only shown once, the first time they are encountered.

For more general rule sets, it may be necessary to exhaustively apply rules using a fixed-point strategy. The following strategy follows the same definition pattern as `innermost` for plain terms [JV01, JV03]:

```
  winnermost(s) = phase(rec x(wall(x); try(s; x)))
```

The difference is that the `phase` mechanism ensures that a node in a term graph (where all subterms are references) is visited only once. Thus the strategy performs a bottom-up traversal and tries to apply the normalization strategy `s` to each node. If that succeeds, the result is transformed by a recursive call to itself. This would entail a complete bottom-up traversal of the resulting term. However, the subterms that have already been visited, i.e., normalized, will not be visited again. This property ensures efficient implementation of the strategy, a result that was obtained in the term case only through a specialization of the strategy to its argument rules [JV01, JV03].

7.4 From Terms to Term Graphs

In this section, we describe how ASTs can be turned into various types of graphs commonly found in compilers, such as use-def chains, call graphs and flow graphs. First, however, we turn our attention to the problem of computing term graphs from terms with maximal sharing. This is done using dynamic rules.

Dynamic Rules A dynamic rule S is a rewrite rule which is defined and possibly undefined at runtime, see [BKVV06]. The expression `rules(`S`: t -> r)` creates a new rule in the rule set for S. The scope operator `{| S :` s `|}` introduces a new scope for the rule set S around the strategy s. Changes (additions, removals) to the rule set S done by the strategy s are undone after s finishes (both in case of failure and success of s). Sometimes, multiple rules in a rule set S may match. To get the results of all matching rules in S, we can use `bagof-S`.

Computing Term Graphs The following strategy implements a top down traversal with a memoization scheme to efficiently construct term graphs from terms. For each term it encounters, the strategy checks if this term has been memoized in the dynamic rule S. If so, the term is replaced with its corresponding reference. If not, all its subterms are replaced with references recursively, then a new reference is created and recorded in S.

```
term-graph = {|S: rec x(S<+all(term-graph); ?t; !r~t; rules(S: t->r)|}
```

Applied to `A(A(B),A(A(B),A(B)))`, we get `3˜A(1˜A(0˜B),2˜A(1˜,1˜))`.

7.4.1 Use-Def Chains

The use-def chain is a data representation found in most compilers for recording links from the use of a variable to its closest definition or assignment. Such data flow information is the basis for many program optimizations, in particular constant and copy propagation. A variable is said to be *used* when its value is read; it is said to be *defined* when it is assigned to, either at its declaration or by a later assignment statement. Def-use chains are links from the definition of a variable to all its uses. The algorithm we present below will record both def-use and use-def chains.

```
use-def       = {| Use, Def: def-to-use ; use-to-def |}
def-to-use    = Var <+ VarRef <+ Assign <+ If <+ wall(def-to-use)
use-to-def    = topdown(try(add-ref-to-var <+ add-ref-to-assign))
new-def(|v,r) = rules(Def : v -> r)
add-use(|d,u) = rules(Use :+ d -> u)

add-ref-to-var    = ?r~Var(v,e,_); !r~Var(v,e, Uses(<bagof-Use> r))
add-ref-to-assign = ?r~Assign(v,e,_); !r~Assign(v,e,Uses(<bagof-Use> r))

If: If(c,t,e) -> If(c',t',e') where <def-to-use> c => c'
  ; <def-to-use> t => t' \Def/ <def-to-use> e => e'

Var: Var(v, e) -> r where <def-to-use> e => x
  ; !r~Var(v,x,Uses([])); new-def(|v, r)

VarRef: VarRef(v) -> r where <bagof-Def> v => defs
  ; !r~VarRef(v, Defs(defs)); <map(add-use(|<id>, r))> defs

Assign: Assign(v, e) -> r where <def-to-use> e => x
  ; !r~Assign(v, x, Uses([])); new-def(|v,r)
```

The `use-def` algorithm assumes the existence of the following term constructors: `If`, for `if` constructs, `Var`, for variable definitions, `VarRef` for variable (de)references and `Assign` for assignments. It is divided into two parts, `def-to-use` and `use-to-def`. For `def-to-use`: If a definition of a variable is seen, i.e. an `Assign` or `Var` term, this term is replaced with a reference to itself, and a mapping from the variable name to the reference is recorded in the dynamic rule `Def` using `new-def`. This is done in the `Var` and `Assign` rules. When a variable use is subsequently seen, it is also replaced by a term to itself by the `VarRef` rule. Its name is looked up in the `Def` rule, and references to the closest definitions are added using `bagof-Def`, see `VarRef`. The `Use` rule is updated to record the reference to this use, using `add-use`. Special care must be taken in the case of control constructs. We only show the case for `If`. Here, one rule set is computed for each branch, and the rule sets are joined afterwards, using the rule set union operator, `\Def/`. This ensures that new definitions from both branches are kept.

For `use-to-def`: In this pass, each `FunDef` is updated to contain references to the uses recorded by the previous pass, in the `Use` rule. When applied to a term for the program

```
var x := 0; if (x > 0) { x := 1 + x } else { x := 2 + x } ; print x
```

we get (read `Asn` as `Assign` and `VRef` as `VarRef`):

```
Block([~5,If(Int(0),Block([~1]),Block([~3])),Print(~0)])
~0 = VRef("x",Defs([~3,~1]))  ~1 = Asn("x",Add(Int(1),~2),Uses([~0]))
~2 = VRef("x",Defs([~5]))     ~3 = Asn("x",Add(Int(2),~4),Uses([~0]))
~4 = VRef("x",Defs([~5]))     ~5 = Var("x",Int("10"),Uses([~4,~2]))
```

7.4.2 Call Graphs

Another common program representation in modern compilers is the call graph. It records the interrelationships between the functions of a program, i.e. which functions call which, and is used for various static analyses such as reachability analysis, optimizations such as dead code removal and by documentation generation tools. The following code transforms an AST into a call graph by introducing references from all call sites (`Call` terms) to the corresponding function definition (`FunDef`) terms, and by adding a reference from the `FunDef` being called (callee) to the `FunDef` of the calling function (caller). Figure 7.1(d) illustrates the forward direction.

```
compute-call-graph   = {| FunLookup:  add-refs ; add-call-markers |}
introduce-references = topdown(try(AddFunRef))
with-fundefs(s)      = Program(map(s), id)
register-fun         = ?r; ?r~FunDef(_,_,_,_); rules(CurFun: _ -> r)

AddFunRef: x@FunDef(n,_,_,_) -> r where !r~x; rules(FunLookup: n -> r)

add-call-markers = {| CalledBy, CurFun:
    with-fundefs(wdownup(try(register-fun), try(AddCallRef)))
  ; with-fundefs(wrap-ref(AddCalledByRef)) |}

AddCallRef: Call(n, xs) -> Call(n, xs, r)
where <FunLookup> n => r ; CurFun => z ; rules(CalledBy :+ n -> z)

AddCalledByRef: FunDef(n,a,t,b) -> FunDef(n,a,t,ns,b)
where <bagof-CalledBy> n => ns
```

Three dynamic rules are used in this algorithm. `FunLookup` is used to map names of functions to their corresponding `FunDef`. `CurFun` is used to keep a reference to the `FunDef` we are currently inside. `CalledBy` is used to accumulate a set of callees for a given function name.

The algorithm works as follows. First, we replace every `FunDef` n node with a reference to n, and record the function name in the `FunLookup` rule set. This is done by `add-refs`. Second, we do a downup traversal, where the current function is kept in the dynamic rule `CurFun` on the way down. On the way up, we add a reference to the destination `FunDef` *f* for any `Call` encountered, and register the current function in the `CalledBy` set for *f*. This is the first part of `add-call-markers`. Third, we place the callee sets collected in the `CalledBy` dynamic rule set on the corresponding `FunDefs`, finally obtaining a bidirectional call graph.

7.4.3 Flow Graphs

Flow graphs are used to represent the control flow of a program, analogously to the way use-def chains represent data flow. Flow graphs, along with use-def chains, are at the heart of many flow-sensitive optimizations, such as constant folding, loop optimization, and jump threading. We can compute a flow graph from the various statements in the AST as follows. In `If(c,t,e)`, flow goes from the condition *c* to both branches, *t* and *e*, which in turn go to the successor block of `if`. In `While(c,b)`, flow goes from the condition *c* to the body *b*, and from *c* to the successor block. The body *b* always flows back to the condition *c*. All other statements correspond to basic blocks: the flow from one statement goes directly to the successor block.

We show a three pass algorithm, `ast-to-flow-graph`. First, we do rewrites of control flow constructs locally, as described above, with the `MarkControlFlow` rule set. In the case of `If` and `While`, temporary `FlowT` blocks are inserted with dummy references, since the successor block is not known locally yet. Second, AST statement blocks are split into basic blocks, with `SplitBlocks`. Each non-control statement is rewritten to a Flow block. Third, the `FlowT` blocks are connected to the `Flow` blocks produced in (2), and rewritten to `Flow`, resulting in a flow graph, as seen in Figure 7.1(e).

```
ast-to-flow-graph = bottomup(try(MarkControlFlow))
  ; bottomup(try(SplitBlocks))
  ; wbottomup(try(\ FlowT(x,y) -> Flow(x,y) \))

SplitBlocks: Block(xs) -> r where
  <map(?FlowT(_,_) <+ {r: \ t -> Flow(t, r) where !r~() \})> xs => xs'
; foldr(\ (f@Flow(t1, n), t2) -> f where !n~t2 \
     <+ \ (f@FlowT(t1, n), t2) -> f where !n~t2 \ |None)
; <Hd> xs' => hd ; !r~hd

MarkControlFlow: If(cond, thn, els) -> FlowT(If(cond', thn', els'), next)
where !next~(); ; !thn'~Flow(thn,[next])
  ; !els'~Flow(els,[next]); !cond'~Flow(cond,[thn', els'])

MarkControlFlow: While(cond, body) -> FlowT(r, next)
```

```
where !body'~(); !next~(); !cond'~Flow(cond, [next, body'])
  ; !body~Flow(body, [cond']); !r~While(cond', body')
```

7.5 Graph Algorithms and Applications

In this section, we show how some basic graph algorithms can be implemented using the reference mechanism we have described in Section 7.3.

Depth First Search Our depth first search implementation works on graphs where each node is a term. The algorithm takes two parameters, `l` and `es`. `es` will be used to compute the outgoing edges from each node. `dfs` keeps track of the current depth during visits. On a visit to a node, the strategy `l` will be called with the current depth value as parameter, so that it can be used to compute the label for the current node, or for other transformations.

```
dfs(l : a * a -> a, es)     = phase(wall(dfs(l, es | 0)))
dfs(l : a * a -> a, es | n) =
     wrap-phase-ref(where(es => edges)
  ; where(l(|n) => label)
  ; where(<wall(dfs(l, es | <inc> n))> edges) ; !label)
```

The traditional depth first search, as described in for example [CLR97], is applied initially to the set V of a graph $G = (V, E)$. We get the same behavior by applying `dfs` to a list of references to all nodes in the graph. We will demonstrate the use of the `dfs` strategy next, when we discuss strongly connected components.

Strongly Connected Components The basic algorithm for strongly connected components (SCC) is also described in [CLR97], and consists of four steps: First, call `DFS(G)` to compute finishing times `f[u]` for each vertex `u`. Second, compute the transposed graph `GT=transpose(G)`. Third, call `DFS(GT)`, but in the main loop of `DFS`, consider the vertices in order of decreasing `f[u]`. Fourth, produce as output the vertices of each tree in the `DFS` forest formed in point 3 as a separate strongly connected component.

In our implementation of SCC, shown below, we avoid actual graph transposition by requiring one strategy, `es` for computing forward edges from a node, and another, `res`, for computing reverse edges. We also combine the third and fourth step by using a modified `dfs`, called, `dfs-collect`, which collects each set of SCCs into a list during the third step.

```
dfs-collect(l : a * a -> a, es) =
  phase(all({|C: dfs-collect(l,es|0) ; bagof-C|}))
```

```
dfs-collect(l : a * a -> a, es | n) = ?r~_
  ; wrap-phase-ref(where(es => edges) ; where(l(|r) => label)
  ; where(<all(dfs-collect(l, es | <inc> n))> edges) ; !label)

sort-fundefs =
  sort-list(LSort(where((?r;!^r; FinishTime,?r';!^r'; FinishTime); gt)))
collect-components(|r) = rules(C :+ _ -> r)
inc-time = (Time <+ !0) => n ; where(inc => n'; rules(Time: _ -> n'))
time-count(|n) = ?x; where(inc-time => n'); rules(FinishTime: x -> n')

scc(l : a * a -> a, es, res) = {|FinishTime, Time:
    dfs(l, es)
  ; sort-fundefs
  ; dfs-collect(collect-components, res)
  ; filter(not(?[])) |}
```

The current time is maintained in the `Time` dynamic rule, and the finishing time in `FinishTime`. Our `scc` should normally be called with the `time-count` strategy as its first argument, but the user is free to adapt this.

7.5.1 Finding Mutually Recursive Functions

Suppose we want to use SCC to compute sets of mutually recursive functions. Then, each node in the graph is a function f. The outgoing edges of f are references to the functions *called by* f. The incoming edges of f are references to the functions *calling* f. This graph is what was computed by `call-graph`, discussed in Section 7.4.2. The following strategies may used for edge computations.

```
calls-as-outbound    = collect(\ Call(_,_,x) -> x \)
calledby-as-outbound = collect(\ FunDef(_,_,_,x,_) -> x \) ; concat
```

Applying `scc(time-count, calls-as-outbound, calledby-as-outbound)` to a list of references to all functions in a program, say Figure 7.1(d), will produce the cliques (a,b,e), (f,g) and (c,d,h).

7.5.2 Lazy Graph Loading

Instead of binding terms to references, strategies may be bound instead. When a reference r with the strategy s bound to it is dereferenced, s is invoked, and the resulting term is taken as the term value for r. We call this an *active reference* since it has a strategy (i.e., function) attached to it that is activated and executed upon dereference. Active references are useful for term (graph) rewriting of larger terms, especially when doing sparse analyses on larger bodies of program code. With active references, terms for programs can be loaded as skeletons. For example, all bodies of functions or classes may be left out, and be parsed and loaded, or even generated, on demand.

7.6 Implementation

We have implemented a prototype of the language extension described in this paper. Our implementation is a conservative extension to the existing Stratego infrastructure: Every valid Stratego program retains its behavior and terms without references are still represented entirely as ATerms. References are introduced as a special kind of term, `Ref`, and we have modified the language implementation to recognize and treat terms of this type specially. `Ref`s are closely related to pointers, as found in C, and to references, as found in Java. Unlike pointers and Java references, a Stratego `Ref` always starts out as bound. It may subsequently be rebound to another term. The bindings from references to terms are maintained in a global table, or more precisely, in a global, dynamic rule set. When a new reference is bound, a new rule is added to the set. When an existing reference is rebound, its corresponding rule is changed. Using dynamic rule sets aids in implementing backtracking behavior. For left- and guarded choice, references rebound or introduced by a failed strategy should be backtracked before the program proceeds. This is implemented in our compiler by rewriting every left choice operator to

```
start-ref-cs ; s1 ; commit-ref-cs <+ discard-ref-cs ; s2
```

Here, `start-ref-cs` will make a change set for the global rule set. If `s1` succeeds, the change set is committed and changes are kept. If `s1` fails, all changes to the reference rule set by `s1` are undone.

Managing the revisitation of references in term graphs is crucial for ensuring termination. The `wrap-phase-ref` mentioned earlier is responsible for this.

```
wrap-phase-ref(s) = ?r@Ref(_) < seen-before < id
+ where(<phase-deref> r; s; bind-ref(|r)) + s
```

`wrap-phase-ref` is implemented using guarded choice $s_1 < s_2 + s_3$, which works as follows. If s_1 succeeds, s_2 will be applied to the resulting term. If it fails, s_3 will be applied to the initial term. If `wrap-phase-ref(s)` is applied to term, `s` is applied and we are done. When at a reference r, we first use `seen-before` to check if we have seen r before. If so, we ignore `s` (by applying `id`, then returning). If not, r is dereferenced and marked, using `phase-deref`, `s` is applied to its term, and r is rebound by `bind-ref`.

Using `wrap-phase-ref`, we can now implement new traversal primitives on references. Let us consider `wall(s)`:

```
wall(s) = is-ref
  < wrap-phase-ref(all(wrap-phase-ref(s))) + all(wrap-phase-ref(s))
```

If we are not at a reference, `all` will be applied to the current term and any references it has as direct subterms will be marked. If we are at a reference, we will mark, transform then rebind it. The markers used by `wrap-phase-ref` can be managed using `phase(s)`, given next:

```
phase(s) = where(local-phase-ctr => pc; inc-phase-ctrs)
  ; start-seen-cs; s; discard-seen-cs
  ; where(restore-local-phase-ctr(|pc))
```

`phase(s)` works as follows: Before *s* is applied, a new, unique, internal phase marker is produced using `local-phase-ctr`, then the counter is increased, preparing for the next invocation. `start-seen-cs` enters a new "scope" for this marker. The counter is maintained in a dynamic rule defined inside `increase-phase-ctrs`, and is later used by `phase-deref` and `seen-before`. Once *s* completes, all markers will be discarded.

Our implementation has not yet been tuned for performance. While we have used Stratego's dynamic rules for implementation convenience, we only rely on the ability of dynamic rules to provide hash tables with change sets. In the current implementation, reference lookup time is linear in the depth of choices on the stack. A more efficient implementation of hash tables with change sets is likely to improve performance.

7.7 Related Work

Term graph rewriting theory is an active field. For an introduction and summary, see [Plu01]. A calculus for rewriting on cyclic term graphs has been proposed by Bertolissi [Ber05]. Many systems exist for general graph rewriting, such as PROGRES [Sch04] and FUJABA [NNZ04]. Few term graph rewriting systems for practical applications exist. HOPS [Kah99] and Clean [PvE98] are a notable exceptions. Claessen and Sands [CS99] describe an extension to the Haskell language which adds references with equality tests. Their goal is to better describe circuits, which are graph structures with cycles, in a purely functional language. Their references are immutable once created, unlike ours, making rewriting more difficult express. Lämmel et al [LVV03] discusses how strategic programming relates to adaptive programming, a technique found in aspect-oriented systems for traversing object structures. They show how traversal strategies may be implemented for cyclic structures, such as graphs, by keeping record of visited nodes. Our phased traversals expand upon this by allowing nested, overlapping traversals and fine-grained control of visitation marking. Our implementation shares some features with monadic programming. Monads are sometimes described as patterns for using functions to transmit state without mutation, and are described by Wadler [Wad92]. In our implementation, dynamic rules are the functions used to transmit the state, namely the internal graph references. Some functional languages, such as Clean [PvE98], are implemented as rewriting systems with implicit term graphs. Functions in Clean are graph rewrite rules on the underlying term graph. In our language, the choice between term and term graph rewriting and their corresponding tradeoffs is not fixed, but rather left to the programmer. An important goal for our language extension is to better capture graph-like program representations, and to offer convenient transformation capabilities for these. Many excellent and general graph libraries exist, and we are not aiming to replace these.

To the best of our knowledge, no other term graph rewriting system supports strategic term graph rewriting, using rewriting strategies and generic traversals.

7.8 Discussion and Further Work

The construction of use-def chains, call- and flow graphs shows how global-to-local problems are now local-to-local, as the remote context is available locally for rules to match on. The addition of references for this purpose also introduces the problem of traversal non-termination in the presence of cycles. We have shown how this can be managed by phases. Another issue of term references is the unexpected impact of reference rebinding, in the loss of referential transparency. The code `!Sub(r~Int(2)) => v ; !r~Int(3)` will alter the value of `v` after it is bound. Judicious use of `duprefs` can control this. Comparison of term graphs is currently done using weak equality; i.e., comparison references in terms is done based on identity, not structure, which allows constant time comparison. Deep comparison is available through the library, and is linear in the size of the term graphs. The pattern-based language constructs introduced in this paper for reference manipulation came about after trying to program with only the primitive operators create reference, bind reference and dereference. While these primitives are still at the heart of the implementation, the notation presented in this paper make them more convenient to use. Further exploration of the design space is warranted. One attractive extension is matching modulo references, which allows term patterns to be matched directly on terms with references, by implicitly visiting references during matching.

7.9 Conclusion

We have presented the design and implementation of an extension to the Stratego term rewriting language for rewriting on terms with references, and demonstrated its practical application through the construction of several common graph-based program representations found in compilers. The contributions of this paper include the introduction of language abstractions for dealing with references within a rule-based term rewriting language, a demonstration of how term matching, building and rewrite rules can be combined with term references, how benefits of generic term traversal can be kept by using phased traversals to deal with non-termination due to cyclic graphs, and how backtracking can be combined with destructive graph updates to retain the strategic programming flavor of Stratego. We showed how our language can be used to implement some basic graph algorithms and how these can be applied to graph-based program representations. We discussed design tradeoffs related to introducing references in terms, including traversal termination and impact of reference binding.