Fundamenta Informaticae 69 (2006) 123–178
IOS Press

Program Transformation with Scoped Dynamic Rewrite Rules

Martin Bravenboer, Arthur van Dam, Karina Olmos and Eelco Visser[*]
Department of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht
The Netherlands
visser@acm.org

Abstract. The applicability of term rewriting to program transformation is limited by the lack of control over rule application and by the context-free nature of rewrite rules. The first problem is addressed by languages supporting user-definable rewriting strategies. The second problem is addressed by the extension of rewriting strategies with scoped dynamic rewrite rules. Dynamic rules are defined at run-time and can access variables available from their definition context. Rules defined within a rule scope are automatically retracted at the end of that scope. In this paper, we explore the design space of dynamic rules, and their application to transformation problems. The technique is formally defined by extending the operational semantics underlying the program transformation language Stratego, and illustrated by means of several program transformations in Stratego, including constant propagation, bound variable renaming, dead code elimination, function inlining, and function specialization.

1. Introduction

Program transformation is the mechanical manipulation of a program in order to improve it relative to some cost function C such that $C(P) > C(tr(P))$, i.e. the cost decreases as a result of applying the transformation [30, 29, 11]. The cost of a program can be measured in different dimensions such as performance, memory usage, understandability, flexibility, maintainability, portability, correctness, or satisfaction of requirements. Related to these goals, program transformations are applied in different settings; e.g. compiler optimizations improve performance [24] and refactoring tools aim at improving understandability [28, 14]. While transformations can be achieved by manual manipulation of programs, in general, the aim of program transformation is to increase programmer productivity by *automating*

[*]Address for correspondence: Department of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

programming tasks, thus enabling programming at a higher-level of abstraction, and increasing maintainability and re-usability of programs. Automatic application of program transformations requires their implementation in a programming language. In order to make the implementation of transformations productive such a programming language should support abstractions for the domain of program transformation.

Term rewriting [35] is an attractive formalism for expressing basic program transformations. A rewrite rule $p_1 \to p_2$ expresses that a program fragment matching the left-hand side pattern p_1 can be replaced by the instantiation of the right-hand side pattern p_2. For instance, the rewrite rule

```
|[ i + j ]| -> |[ k ]| where <add>(i,j) => k
```

expresses *constant folding* for addition, i.e. replacing an addition of two constants by their sum. Similarly, the rule

```
|[ if 0 then e1 else e2 ]| -> |[ e2 ]|
```

defines *unreachable code elimination* by reducing a conditional statement to its right branch since the left branch can never be executed. Thus, rewrite rules can directly express laws derived from the semantics of the programming language, making the verification of their correctness straightforward. A correct rule can be safely applied anywhere in a program. A set of rewrite rules can be directly operationalized by rewriting to normal form, i.e. exhaustive application of the rules to a term representing a program. If the rules are confluent and terminating, the order in which they are applied is irrelevant.

However, there are two problems associated with the application of standard term rewriting techniques to program transformation: the need to intertwine rules and strategies in order to control the application of rewrite rules and the context-free nature of rewrite rules.

Exhaustive Application of Rules Exhaustive application of all rules to the entire abstract syntax tree of a program is not adequate for most transformation problems. The system of rewrite rules expressing basic transformations is often non-confluent and/or non-terminating. An ad hoc solution that is often used is to encode control over the application of rules into the rules themselves by introducing additional function symbols. This intertwining of rules and strategies obscures the underlying program equalities, incurs a programming penalty in the form of rules that define a traversal through the abstract syntax tree, and disables the reuse of rules in different transformations.

The paradigm of programmable rewriting strategies solves the problem of control over the application of rules while maintaining the separation of rules and strategies. A strategy is a little program that makes a selection from the available rules and defines the order and position in the tree for applying the rules. Thus rules remain pure, are not intertwined with the strategy, and can be reused in multiple transformations. Support for strategies is provided by a number of transformation systems in various forms. In TAMPR [5] a transformation is organized as a sequence of canonical forms. For each canonical form a tree is normalized with respect to a subset of the rules in the specification. ELAN [4] provides non-deterministic sequential strategies. Stratego [44, 38, 41] provides generic basic traversal operators that can be used to compose a wide range of generic tree traversal schemas. See [42] for a survey of strategies in rule-based program transformation systems.

Context-free Nature of Rewrite Rules The second problem of rewriting is the context-free nature of rewrite rules. A rule has access only to the term it is transforming. However, transformation problems are often context-sensitive. For example, when inlining a function at a call site, the call is replaced by the body of the function in which the actual parameters have been substituted for the formal parameters. This requires that the formal parameters and the body of the function are known at the call site, but these are only available higher-up in the syntax tree. There are many similar problems in program transformation, including bound variable renaming, typechecking, data flow transformations such as constant propagation, common-subexpression elimination, and dead code elimination. Although the basic transformations in all these applications can be expressed by means of rewrite rules, these require contextual information.

One solution to this problem is the use of contextual rules [3, 37, 44]. A contextual rule solves the context problem by applying the transformation at the context level instead of at the location where the actual transformation takes place. A context expression `e[e']` matches or replaces an expression `e'` occurring within `e`. For instance, the following contextual rule defines the inlining of a (unary) function definition at a function call site:

```
UnfoldCall :
  |[ let function f(x) = e1 in e2[f(e3)] end ]| ->
  |[ let function f(x) = e1 in e2[let var x := e3 in e1 end] end ]|
```

The rule is applied to an abstract syntax tree that contains both the function definition and its uses. Since function calls can be nested deeply in the body of the `let` expression, a local traversal is needed to find them. When such a rule is applied as part of a complete traversal over a program, e.g., to perform inlining for all function definitions, the extra local traversal leads to quadratic complexity.

To avoid this complexity, the more common solution to this problem is to extend the traversal over the tree (be it hand-written or generic) such that it distributes the data needed by transformation rules. For example, traversal functions in ASF+SDF [7] can be declared to have an accumulation parameter in which data can be collected. Language independent definitions of operations such as bound variable renaming in Stratego [38] capture a generic tree traversal schema that takes care of distributing an environment through a tree.

The disadvantage of such solutions is that the rewriting nature of the solution is lost. Instead of a rewrite rule performing a transformation, the traversal carries along a data structure that stores the context information. The traversal code manages this data structure in order to add information at the appropriate places and retrieve it in other places. For instance, an inlining algorithm needs to maintain a table, mapping function names to their definitions. These data structures and operations are often complicated by the fact that the context information is governed by the scope and the data flow of the object program. Further complications arise when multiple kinds of context information need to be carried along. Many variations of such data structures are used in transformation systems, e.g. symbol tables in type checking, and hash tables in value numbering [24]. Representation of such data structures as terms within term rewriting has the disadvantage of the suboptimal complexity of list manipulation and inspection.

Dynamic Rules This article shows how context-sensitive rewriting can be achieved without the added complexity of local traversals and without complex data structures, by the extension of rewriting strategies with *scoped dynamic rewrite rules*. Dynamic rules are otherwise normal rewrite rules that are *defined at run-time* and that *inherit information from their definition context*. As an example, consider the following strategy definition as part of an inlining transformation:

```
DefineUnfoldCall =
  ?|[ function f(x) = e1 ]|
  ; rules( UnfoldCall : |[ f(e2) ]| -> |[ let var x := e2 in e1 end ]| )
```

The strategy `DefineUnfoldCall` matches a function definition and defines the rewrite rule `UnfoldCall`, which rewrites a call to the *specific* function *f*, as encountered in the definition, to a `let` expression binding the formal parameter *x* to the actual parameter *e2* in the body of the function *e1*. Note that the variables *f*, *x*, and *e1* are bound *in the definition context* of `UnfoldCall`. The `UnfoldCall` rule thus *defined* at the function definition site, can be *used* at all function call sites. The storage and retrieval of the context information is handled transparently by the underlying language implementation and is of no concern to the programmer.

The concept of defining rules dynamically is enriched with a number of additional concepts:

- *Multiple rules* with the same name can be defined at the same time (e.g. `UnfoldCall` rules for multiple functions).
- Rules can be *redefined* (e.g. a new definition of `UnfoldCall` for a function after its definition has been transformed).
- Rules can be *undefined* (e.g. `UnfoldCall` is undefined for recursive functions).
- The *scope* in which a rule is applicable can be limited (e.g. a specific definition of `UnfoldCall` can only be used in that part of the abstract syntax tree in which the corresponding function definition is in scope).
- *Scope labels* provide fine-grained control over the scope in which a rule is defined (e.g. the specializations of a function should be added to the scope of that function).
- Rules can be extended to rewrite to multiple right-hand sides (e.g. in partial evaluation a function definition can be rewritten to multiple specializations).
- Rule sets can be forked and later joined again with intersection or union operations, which also have fixed point variants. These operations can be used to model forking and joining in the data flow of a program (e.g. after constant propagation in the branches of an if-then-else statement the continuation of the statement should use the intersection of the propagation facts from the branches).

These concepts are combined in a natural extension of the rewriting paradigm that does not require transformation programmers to learn fundamentally new concepts. Dynamic rules are implemented in an extension of the Stratego language where they provide a single high-level abstraction for dealing with context information in a wide range of program transformations. Dynamic rules have already been proven useful in a wide range of transformations, including: the substitution in bound variable renaming [39]; the call replacement in function inlining [39]; the removal of declarations in dead code elimination [39]; the binding of variables in interpretation [12]; the representation of data flow facts in data flow optimizations, e.g. the mapping from variables to their values in constant propagation [26, 27] or the mapping from expressions to variables in common subexpression elimination; the specialization of functions in partial evaluation; the representation of type assignments in typechecking; and the memoization of instruction selections in code generation [9]. The language constructs have been carefully designed to provide a natural fit in the rewriting setting, while at the same time making an efficient implementation of the various operations possible, for instance, using hash tables for fast storage and retrieval of data.

Contribution Dynamic rules provide a small and coherent language extension that captures many specialized data structures such as symbol tables and tables for the representation of data flow facts. This high-level abstraction for program transformation is independent of any object language or kind of transformation and supports concise specification of data flow and other transformations. This in turn enables implementations of compilers and other transformation systems in significantly fewer lines of code, with all the associated benefits for productivity, understandability, and maintainability. A particular contribution of the use of dynamic rules is the combination of program analysis and program transformation in a single traversal, making it possible to achieve better results than with separate analysis and transformation stages, since the effects of transformations can be used in analysis immediately.

With respect to the earlier paper [39] that introduced dynamic rules by example, this article contributes the following. New concepts are the extension of dynamic rules with multiple right-hand sides, the application of dynamic rules only once, the scope labels that improve and generalize the earlier 'override' feature, and the intersection and union operators that model data flow splits in a transparent manner. The syntax of the various operations has been simplified and made orthogonal. In addition, this article presents a formal operational semantics of Stratego with dynamic rules. The implementation of dynamic rules is described in the technical report version of this article [8]. Finally, we illustrate the various concepts with actual Stratego code (possible due to the conciseness of the language) including several as yet unpublished applications such as common subexpression elimination, dead code elimination, and function specialization.

Outline We have aimed this article to be self contained. Therefore, the first two sections review the basics of program transformation with rewriting strategies. Section 2 reviews the representation of programs as terms, the Tiger language that will be used in examples, term rewriting, and its use in program transformation. Section 3 reviews the basics of rewriting strategies in Stratego and defines the syntax and operational semantics of the language as basis for the definition of the extension with dynamic rules.

Sections 4 through 7 introduce the concepts of dynamic rules. Each section uses example transformations to motivate the concepts before giving a formal operational semantics. As a running example an implementation of constant propagation is gradually extended. Section 4 starts with the definition of dynamic rules, the shadowing of earlier definitions, and the undefinition of rules. These ideas are illustrated with constant propagation in basic blocks. Section 5 introduces constructs for the restriction of the scope of dynamic rules and locally shadowing of earlier defined rules, which is illustrated with bound variable renaming and inlining. Scopes are further refined with labels enabling definition or redefinition of rules in earlier scopes. This is illustrated with intra-procedural constant propagation and dead function and variable declaration elimination. Section 6 extends dynamic rules with multiple right-hand sides and limited application. This is illustrated with common subexpression elimination. Section 7 describes the operations for intersection and union of rule sets to implement flow-sensitive data flow transformations. The operations are illustrated with flow-sensitive conditional constant propagation and dead code elimination. Section 8 describes a strategy for online partial evaluation as a larger example.

Section 9 discusses related work and Section 10 concludes. Appendix A provides the definition of free variables of a Stratego expression. The design of dynamic rules as described in this paper reflects the implementation of dynamic rules in Stratego/XT version 0.14 [46].

2. Program Transformation by Term Rewriting

In this section we review standard term rewriting and its application to program transformation, which requires the representation of programs, or rather their abstract syntax trees, as terms. Throughout this paper we use Tiger, the example language in the compiler construction textbook of Appel [2], to illustrate all aspects of program transformation with rewriting strategies and dynamic rules. Therefore, we start with a brief overview of Tiger.

2.1. The Tiger Language

Tiger is an imperative, first-order language with nested functions. Figure 1 presents the syntax of Tiger programs in BNF[1]. In Tiger, data is composed using arrays and records from integers and strings, but in this paper we ignore arrays and records. Integer values are processed using the standard built-in arithmetic and relational operators. Boolean values are represented by integers as in C, thus `0` represents false and all other integers represent true. Control flow is determined using the `if-then-else`, `if-then`, `while` and `for` constructs. As an inheritance from functional languages, there is no syntactic distinction between expressions (yielding a value) and statements (producing a side effect). This entails that assignments and loops can be used within ‘expressions’ using the sequence construct. A sequence of expressions `(`e_1`;...;`e_n`)` corresponds to the sequential composition of the expressions e_1 to e_n. When used as an ‘expression’, the last expression of the sequence must produce a value. Thus, `x := a + (y := x + 1; y)` is a valid assignment statement. Variables and functions in Tiger are introduced in the `let` construct. A variable or function is visible in all subsequent declarations and in the body of the `let`. Function definitions can be nested and can refer to all functions and variables in scope. The program in Figure 2 illustrates the essential aspects of the language.

Subsets of Tiger To avoid complexity that is not relevant for explaining a feature of the transformation language, many of the presented transformations are restricted to a specific subset of the Tiger language. We distinguish *basic blocks*, *function bodies*, and programs with *pure expressions*. A *basic block* is a sequence of simple statements without control flow, i.e. an expression of the form `(`x_1 `:=` e_1`; ...;` x_n `:=` e_n`)`. An example basic block is `(x := a + b + 42; y := x + y; a := x + 3)`. *Intra-procedural* transformations work on function bodies with local variable declarations in let bindings, but without nested functions. Such expressions can be considered with or without control flow. For example, the expression

```
  let  var x := a + b + 42;  var y := x + y   in   a := x + 3; x + y   end
```

[1] In actual Stratego/XT transformations, syntax definitions in SDF2 are used, mostly. The full syntax definition of Tiger in SDF2 consists of some 300 lines of code and its details are not of interest to this article.

d	$::=$	`VarDec` :	`var` x ta `:=` e	variable declaration
		`FunDecs` :	$fd_1 ... fd_n$	function definitions
fd	$::=$	`FunDec` :	`function` f`(`$farg_1$`,`...`,`$farg_n$`)` ta `=` e	function definition
$farg$	$::=$	`FArg` :	$x\ ta$	function argument
ta	$::=$	`Tp` :	`:` tp	type declaration
		`NoTp` :	ϵ	no type declaration
e	$::=$	`Var` :	x	variable
		`Str`, `Int` :	$str \mid i$	string, integer constant
		`BinOp` :	e_1 `+` e_2 | e_1 `-` e_2 | e_1 `*` e_2 | ...	arithmetic
		`RelOp` :	e_1 `<` e_2 | e_1 `>` e_2 | e_1 `=` e_2 | ...	relational
		`And`, `Or` :	e_1 `&` e_2 | e_1 `|` e_2	Boolean
		`Assign` :	x `:=` e	assignment
		`Call` :	f`(`e_1`,`...`,`e_n`)`	function call
		`Seq` :	`(`e_1`;`...`;`e_n`)`	sequence
		`If` :	`if` e_1 `then` e_2 `else` e_3	conditional
		`IfThen` :	`if` e_1 `then` e_2	conditional
		`While` :	`while` e_1 `do` e_2	while loop
		`For` :	`for` x `:=` e_1 `to` e_2 `do` e_3	for loop
		`Let` :	`let` $d_1 ... d_n$ `in` e_1`;`...`;`e_m `end`	let binding

Figure 1. Abstract syntax of a subset of Tiger.

```
let var maxnbr := -1 var count := 0
    function nextnumber() : int =
      let var number := readint()
          function setmax(number : int) = if number > maxnbr then maxnbr := number
       in count := count + 1; setmax(number); number < 0 end
 in while(nextnumber()) do ();
    print("umber of values: "); printint(count);
    print("maximum: "); printint(maxnbr) end
```

Figure 2. Example Tiger program.

is a basic block with local variables, but no control flow. The lack of separation between statements and expressions, allowing expressions such as

```
y := (if x < y then (a := x + 1; a) else x + y)
```

can complicate transformations. Programs can be transformed automatically to a form in which expressions are *pure*, i.e. side effect free and separate from statements, by lifting expressions with side effects to the statement level. Thus, the assignment expression above can be transformed to

```
if x < y then (a := x + 1; y := a) else y := x + y
```

For some transformations we will assume programs to have *pure expressions*. Note that the `if-then-else` operator can be used in pure expressions as long as all its subexpressions are pure. Naturally the

$t ::=$	str	$\equiv str\texttt{()}$	string constant
	i	$\equiv i\texttt{()}$	integer constant
	c	$\equiv c\texttt{()}$	nullary constructor application
	$c\texttt{(}t_1\texttt{,...,}t_n\texttt{)}$		n-ary constructor application $n \geq 0$
	$\texttt{(}t_1\texttt{,...,}t_n\texttt{)}$	$\equiv \texttt{Tuple(}t_1\texttt{,...,}t_n\texttt{)}$	n-ary tuple $n \geq 0$
	$\texttt{[}t_1\texttt{,...,}t_n\texttt{]}$	$\equiv \texttt{Cons(}t_1\texttt{,...,Cons(}t_n\texttt{,Nil()))}$	list

Figure 3. The Annotated Term (ATerm) Format.

simplified forms of Tiger programs can be achieved using transformations, but we will not discuss those transformations in this article.

2.2. Representing Programs as Terms

A context-free grammar for a programming language induces a tree structure for programs [1, 2], which can be used as a structured representation to transform programs. The trees induced by a context-free grammar are isomorphic to first-order terms, which are the data manipulated in term rewriting. A *first-order term* is essentially a constructor c applied to a, possibly empty, list of first-order terms t_1,...,t_n, as defined by the following grammar:

$t ::=$	$c\texttt{(}t_1\texttt{,...,}t_n\texttt{)}$	n-ary constructor application $n \geq 0$
$c ::=$	identifier $\mid str \mid i$	constructors

Constructors are identifiers, quoted strings (str), or integer constants (i). While this is the notion of terms we will use when considering the semantics of Stratego, we use a slightly enriched term format in actual Stratego programs.

The *Annotated Term (ATerm) Format* is a format for the representation and exchange of structured data [6]. The format basically corresponds to first-order terms as defined above, but provides a little syntactic sugar for common terms such as tuples and lists. The grammar in Figure 3 defines the structure of ATerms, and indicates using the equivalences how ATerms correspond to first-order terms. Note that the adjective *annotated* stems from the fact that ATerms can be annotated with terms, a feature that is not considered in this article. Stratego is indifferent to the source of the terms it transforms. These can be produced by a parser derived from an SDF2 syntax definition [36], but may be produced by any program, including for instance a YACC parser or another Stratego program.

To illustrate how programs correspond to terms, consider the constructors assigned to productions in the grammar for Tiger in Figure 1. Examples of terms over the Tiger grammar are `Var("x")` which represents the variable `x`; `Call(Var("f"),[Var("x")])`, which represents `f(x)`, the call of function `f` with argument `x`; and `Let([VarDec("x",NoTp,Int("1"))],[Var("x")])`, which represents the expression `let var x := 1 in x end`, the declaration of local variable `x` initialized to the integer constant `1`.

A *term pattern* is a term with variables, that is, a pattern is either a variable or the application $c\texttt{(}p_1\texttt{,...,}p_n\texttt{)}$ of an n-ary constructor c to term patterns p_i. To emphasize the distinction between term patterns from terms without variables, the latter are sometimes referred to as *closed terms*.

2.3. Term Rewriting

Term rewriting is a declarative paradigm for transforming terms. A rewrite system consists of a set of rewrite rules of the form $L\colon\ p_1$ `->` p_2 `where` s, consisting of a label L, term patterns p_1 and p_2, and condition s. An unconditional rule has the form $L\colon\ p_1$ `->` p_2. A condition s is some computation, involving the variables bound by the left-hand side, either checking an additional constraint for application of the rule and/or producing values to be used in the right-hand side by binding new variables. The exact nature of conditions will be discussed in Section 3. Examples of rewrite rules are the following constant folding and desugaring rules:

```
EvalBinOp : BinOp(PLUS, Int(i), Int(j)) -> Int(k) where <add>(i,j) => k
DefAnd    : And(e1, e2) -> If(e1, e2, Int("0"))
```

Note that pattern variables are typeset in italics.

A pattern p_1 matches with a term t if there is a substitution σ mapping the variables in p_1 to subterms of t such that $\sigma(p_1) \equiv t$. A rewrite rule $L\colon\ p_1$ `->` p_2 `where` s applies to a term t if the left-hand side pattern p_1 matches t with substitution σ, the condition s succeeds under σ producing an extension σ', and returns the instantiation of the right-hand side pattern p_2 with σ'. For example, the expression

```
And(Var("x"), BinOp(GT, Var("x"), Int("5")))
```

is rewritten by rule `DefAnd` to

```
If(Var("x"), BinOp(GT, Var("x"), Int("5")), Int("0"))
```

As should be apparent from this description, a rewrite rule has access only to local information, i.e. the subterms of the term to which it is applied, and thus lacks context information. This is the problem we set out to solve with dynamic rules.

The usual interpretation of a set of rewrite rules in standard rewriting engines is to compute the normal form of a term with respect to all rules, that is, exhaustively apply rules to all subterms until no rule can be applied anymore. In this interpretation, it is usually assumed or required that rule sets are confluent and terminating. That is, any order in application of the rules has the same result (confluent) and always leads to a normal form (terminating). The lack of these properties in pure rewrite rules leads to workarounds in the form of additional constructors (functions) that control the order in which transformations are applied and leads to tangling of rewrite rules and their application strategy. Programmable rewriting strategies avoid this tangling by allowing alternative strategies to be defined independently of the rewrite rules.

2.4. Concrete Syntax

We have argued above that programs can be represented as terms and that term rewrite rules can be used to manipulate programs. However, when manipulating larger program fragments, term syntax tends to become harder to understand. Exploiting the isomorphism between the trees induced by context-free grammars and terms, we can use the *concrete syntax* of the programming language to express the term patterns of rewrite rules [40]. We write $[\![\ E\ ]\!]$, with E a phrase in concrete syntax, to denote the term corresponding to E. For example, the rule

```
EvalBinOp : |[ e + 0 ]| -> |[ e ]|
EvalBinOp : |[ e + 0 ]| -> |[ e ]|
EvalBinOp : |[ i + j ]| -> |[ k ]| where <add>(i , j) => k
EvalBinOp : |[ i * j ]| -> |[ k ]| where <mul>(i , j) => k

AddAssoc  : |[ (e1 + e2) + e3 ]| -> |[ e1 + (e2 + e3) ]|

EvalIf    : |[ if 0 then e1 else e2 ]| -> |[ e2 ]|
EvalIf    : |[ if i then e1 else e2 ]| -> |[ e1 ]| where <not(eq)>(i, 0)
EvalWhile : |[ while 0 do e ]| -> |[ () ]|

EmptyLet  : |[ let in e* end ]| -> |[ (e*) ]|
LetSplit  : |[ let d1 d2 d* in e* end ]| -> |[ let d1 in let d2 d* in e* end end ]|
LetFlat1  : |[ let d in let d* in e* end end ]| -> |[ let d d* in e* end ]|

DefAnd    : |[ e1 & e2 ]| -> |[ if e1 then e2 else 0 ]|
DefOr     : |[ e1 | e2 ]| -> |[ if e1 then 1 else e2 ]|

AssignIf  : |[x := (if e1 then e2 else e3)]| -> |[if e1 then x := e2 else x := e3]|

ElimIf    : |[ if e then () else () ]| -> |[ (e; ()) ]|
ElimIf    : |[ if e1 then e2 else () ]| -> |[ if e1 then e2 ]|
ElimIf    : |[ if e1 then () else e2 ]| -> |[ if not(e1) then e2 ]|
ElimIf    : |[ if e then () ]|-> |[ (e; ()) ]|
ElimFor   : |[ for x := e1 to e2 do () ]| -> |[ (e1; e2; ()) ]|
```

Figure 4. Some rewrite rules for Tiger expressions.

denotes the following rule using abstract syntax:

```
EvalBinOp : BinOp(PLUS, e, Int("0")) -> e
```

Stratego supports the specification of transformation systems with *concrete object syntax* for arbitrary object languages [40]. In the rest of this paper, we will use concrete syntax for all terms in *example* specifications. In the semantic rules we will use the term representation, i.e. consider terms of the form $c(t_1,\ldots,t_n)$. In example programs, pattern variables will be typeset in italics. Figure 4 presents a set of rewrite rules on Tiger expressions, some of which will be referred to in later examples. Note that the names of meta-variables correspond to the non-terminals in the Tiger grammar in Figure 1; for example, e denotes a Tiger expression, x a Tiger variable, and i an integer constant.

3. Rewriting Strategies

Programmable rewriting strategies provide a mechanism for achieving control over the application of rewrite rules, while keeping rules and strategies separated and avoiding the introduction of new constructors or rules. The strategies in Stratego were inspired by the strategy language of ELAN [4], which was itself influenced by tactics in theorem provers. The specific contributions of strategies in Stratego are first class pattern matching and generic traversal based on basic traversal operators [22, 43, 44]. This

P	$::=$	$d_1 ... d_n$	program (list of definitions)
d	$::=$	$dsig$ `=` s	strategy definition
$dsig$	$::=$	f`(`$sd_1, ..., sd_n$ `|` $vd_1, ..., vd_m$`)`	definition signature
sd	$::=$	$f \mid f$`:`tp	strategy argument (with optional type)
vd	$::=$	$x \mid x$`:`tp	term argument (with optional type)
p	$::=$	$str \mid i \mid r$	string, integer, real constant
		x	term variable
		c`(`$p_1, ..., p_n$`)`	constructor application
s	$::=$	`?`p	match
		`!`p	build
		`{`$x_1, ..., x_n$`:`s`}`	term variable scope
		`let` $d_1, ..., d_n$ `in` s `end`	local definitions
		f`(`$s_1, ..., s_n$`|` $p_1, ..., p_m$`)`	call
		`id`	identity
		`fail`	failure
		s_1 `;` s_2	sequential composition
		s_1 `<` s_2 `+` s_3	guarded deterministic choice
		c`(`$s_1, ..., s_n$`)`	congruence traversal
		tr`(`s`)`	traversal to subterms
tr	$::=$	`all` | `one`	traversal operator
f	$::=$	identifier	strategy operator
x	$::=$	identifier	term variable
c	$::=$	identifier	constructor
tp	$::=$	...	type (omitted)

Figure 5. Syntax of (a subset of) core Stratego.

section reviews the syntax and semantics of basic rewriting strategies in Stratego and lays the foundation for their extension with dynamic rules in the next sections.

3.1. Syntax and Semantics

Syntax Stratego is split in a core language providing the fundamental constructs and syntactic abstractions defined in terms of those constructs. The syntax of core Stratego is presented in Figure 5. The core language is enriched with several syntactic abstractions which are presented in Figure 6 and which are reduced to the core syntax of Figure 5. The extension of Stratego with dynamic rules is introduced in Section 4. The syntax definition in Figure 5 also introduces the *meta-variables* that will be used in the operational semantics. For instance, s denotes a strategy and p a term pattern. Note that the core syntax in Figure 5 is not complete. We are omitting generic term deconstruction [38], some traversal operators, and term annotations.

d	$::=$	$dsig$ `:` p_1 `->` p_2 (`where` s)?	rule definition (with optional condition)
$dsig$	$::=$	$f(sd_1, ..., sd_n)$	definition without term arguments
		f	definition without arguments
p	$::=$	$(p_1, ..., p_n)$	tuple
		`[`$p_1, ..., p_n \mid p$`]`	list
		`[`$p_1, ..., p_n$`]`	fixed length list
s	$::=$	`if` s_1 `then` s_2 `else` s_3 `end`	conditional choice
		s_1 `<+` s_2	deterministic choice
		`where`(s)	test
		`not`(s)	negative test
		`<`s`>`p	apply to pattern
		s `=>` p	match against pattern
		f`(`$s_1, ..., s_n$`)`	call (only strategy arguments)
		f	call (no arguments)
		`rec` f`(`s`)`	recursive closure
		`{`s`}`	local scope for all free variables in s

Figure 6. Extensions (sugar) of the syntax of core Stratego.

Operational Semantics A *rewriting strategy* is a program that transforms a term or fails at doing so. In the case of success, the result is a transformed term. In the case of failure, there is no resulting term, but the state may be changed. A rewrite rule is just a strategy that applies a transformation to the root of a term. Strategies can be combined into more complex strategies by means of strategy combinators. In this section, we will give an overview of the constructs of the Stratego language and define them using a formal operational semantics. The operational semantics is an extension of the semantics presented in [43, 44], and integrates an environment and a state in the rules. The semantics should be understood as a description of the behaviour of programs, not (necessarily) as a model of the implementation.

The semantics of the core constructs is defined in terms of assertions of the form

$$D, \Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies t' \; (\Gamma', \mathcal{E}')$$

which states that the application of strategy s to subject term t in the context of strategy definitions D, state Γ, and environment $\mathcal{E}$ evaluates to the new subject term t', state Γ', and environment $\mathcal{E}'$. A *failing strategy* application is denoted by an assertion

$$D, \Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')$$

That is, the result of the application of a strategy is in the domain of terms extended with the special value $\uparrow$, denoting failure. We use $\bar{t}$ to denote a value from the extended domain of terms, while t always denotes a term, and not failure. Strategy definitions in the set D are only visible in the lexical scope of their definition and are not changed as result of strategy applications. We omit strategy definitions from semantic rules, except in Section 3.4, where we define their semantics. States Γ are used to model dynamic rules; their structure will be described in Section 4. Environments $\mathcal{E}$ model pattern variable bindings. Note that changes to the state and environment are preserved in the case of failure. The semantics of syntactic abstractions is expressed by means of equations $e_1 \equiv e_2$. We will also use such equations to illustrate some of the algebraic properties of the language constructs. Equations are universally quantified unless otherwise indicated.

3.2. Matching and Building Terms

In the previous section we described rewrite rules as operations that first match their left-hand side pattern, then evaluate their condition, and finally instantiate the right-hand side pattern. Instead of taking rewrite rules as basic actions, in Stratego the actions that make up rewrite rules are first class. That is, matching a term against a pattern and instantiating a pattern to build a new term are first class strategies. The strategy `?`p denotes matching against the pattern p, and `!`p denotes building an instantiation of the pattern p. This decomposition allows many language constructs to be defined from first principles. For instance, a rewrite rule p_1 `->` p_2 corresponds to the sequential composition `?`p_1`;!`p_2. The sequential composition of strategies will be defined formally below, but it comes down to first applying the first strategy and then the second. To define match and build precisely we need variable binding environments.

Environments A *variable binding environment* $\mathcal{E} \equiv [x_1 \mapsto \overline{t_1}, ..., x_n \mapsto \overline{t_n}]$ is a finite ordered mapping from variables to closed terms or failure. An environment can contain more than one binding for a variable x, in which case the first binding (from the left) is applicable. Thus, the *application* of an environment $\mathcal{E}$ to a variable x is defined as

$$[x_1 \mapsto \overline{t_1}, ..., x_n \mapsto \overline{t_n}](x) = \begin{cases} \overline{t_i} & \text{if } x_i \equiv x \text{ and } \forall j < i : x_j \not\equiv x \\ \uparrow & \text{if } \forall j \leq n : x_j \not\equiv x \end{cases}$$

The *loose application* $\overline{\mathcal{E}}(x)$ of an environment behaves as the identity map on unbound variables:

$$\overline{\mathcal{E}}(x) = \begin{cases} t & \text{if } \mathcal{E}(x) = t \\ x & \text{otherwise} \end{cases}$$

The application of environments can be extended to term patterns and strategies. The *strict instantiation* $\mathcal{E}(p)$ of a term pattern p with an environment $\mathcal{E}$ yields the closed term obtained by replacing each variable x in p with $\mathcal{E}(x)$, if each $\mathcal{E}(x)$ is defined, and $\uparrow$ otherwise. The *loose instantiation* $\overline{\mathcal{E}}(p)$ of a term pattern p with an environment $\mathcal{E}$ yields the term pattern obtained by replacing each variable x in p with $\overline{\mathcal{E}}(x)$. The loose instantiation $\overline{\mathcal{E}}(s)$ of a strategy expression s consists in replacing each term pattern p in s with $\overline{\mathcal{E}}(p)$.

An environment $\mathcal{E}'$ is a *refinement* of the environment $\mathcal{E}$ (notation $\mathcal{E}' \sqsupseteq \mathcal{E}$) if $\mathcal{E}'$ has the same domain as $\mathcal{E}$ and is *more defined* than $\mathcal{E}$. That is, if $\mathcal{E} = [x_1 \mapsto \overline{t_1}, ..., x_n \mapsto \overline{t_n}]$ then $\mathcal{E}' = [x_1 \mapsto \overline{t'_1}, ..., x_n \mapsto \overline{t'_n}]$ and for each i, $\mathcal{E}(x_i) = \mathcal{E}'(x_i)$ or $\mathcal{E}(x_i) = \uparrow$ and $\mathcal{E}'(x_i) = t$ for some term t. An environment $\mathcal{E}'$ is the *smallest refinement* of $\mathcal{E}$ with respect to a term pattern p (notation $\mathcal{E}' \sqsupseteq_p \mathcal{E}$), if $\mathcal{E}' \sqsupseteq \mathcal{E}$ and for all x not in p, $\mathcal{E}'(x) = \mathcal{E}(x)$. That is, the only difference between $\mathcal{E}'$ and $\mathcal{E}$ are bindings for variables in p.

The *composition* $\mathcal{E}_1\mathcal{E}_2$ of two environments $\mathcal{E}_1$ and $\mathcal{E}_2$ is equivalent to the concatenation of the two mappings.

Match The match operation $?p$ matches the subject term against the term pattern p. This involves checking that the subject term corresponds to the pattern and also involves binding the variables in the pattern to the corresponding subterms of the subject term. Matching is defined by the following rules. A strategy $?p$ applies to a term t if there is an environment $\mathcal{E}'$ that refines the current environment $\mathcal{E}$ and makes p equal to t. A match fails if there is no such environment.

$$\frac{\mathcal{E}' \sqsupseteq_p \mathcal{E} \;\;\&\;\; \mathcal{E}'(p) \equiv t}{\Gamma, \mathcal{E} \vdash \langle ?p \rangle\, t \implies t\; (\Gamma, \mathcal{E}')} \qquad \frac{\neg\exists \mathcal{E}' : (\mathcal{E}' \sqsupseteq_p \mathcal{E} \;\;\&\;\; \mathcal{E}'(p) \equiv t)}{\Gamma, \mathcal{E} \vdash \langle ?p \rangle\, t \implies \uparrow (\Gamma, \mathcal{E})}$$

Note that patterns may be non-linear or contain variables that have been used in an earlier match, but that the definition of $\sqsupseteq$ entails that such variables can only be bound to the same term they were bound to before.

Example: applying `?|[(e1 | e2) & e3]|` to the term `|[(a < b | c) & d > 10]|` succeeds since the environment `[e1 ↦ |[a < b]|, e2 ↦ |[c]|, e3 ↦ |[d > 10]|]` makes the pattern equal to the subject term.

Build The build operation $!p$ replaces the subject term with the instantiation of the pattern p using the bindings from the environment. The semantics of $!p$ is defined as follows:

$$\Gamma, \mathcal{E} \vdash \langle !p \rangle\, t \implies \mathcal{E}(p)\; (\Gamma, \mathcal{E})$$

Note that this uses the strict instantiation of p, entailing that if one of the variables in p is not bound in $\mathcal{E}$, the build fails. Example: in the presence of environment `[e1 ↦ |[a < b]|, e2 ↦ |[c]|, e3 ↦ |[d > 10]|]`, the build `!|[(e1 | e2) & e3]|` produces the term `|[(a < b | c) & d > 10]|`.

Scope Once a variable is bound, it cannot be rebound to a different term. The *scope of a variable binding* can be restricted using the $\{x_1, \ldots, x_n : s\}$ scope construct. That is, the binding to a variable x_i outside the scope $\{x_1, \ldots, x_n : s\}$ is not visible inside it, nor is the binding to x_i inside the scope visible outside it. The semantics of the scope construct is formally defined as follows:

$$\frac{\Gamma, [y_1 \mapsto \uparrow, ..., y_n \mapsto \uparrow]\mathcal{E} \vdash \langle [y_1/x_1 ... y_n/x_n]s \rangle\, t \implies \overline{t'}\; (\Gamma', [y_1 \mapsto \overline{t_1}, ..., y_n \mapsto \overline{t_n}]\mathcal{E}')(y_1 ... y_n \text{fresh})}{\Gamma, \mathcal{E} \vdash \langle \{x_1, ..., x_n : s\} \rangle\, t \implies \overline{t'}\; (\Gamma', \mathcal{E}')}$$

That is, the strategy s, with the scope variables replaced by fresh copies, is evaluated in an extended environment in which the local variables are unbound initially. After the application of the strategy the new bindings are removed from the environment and the old bindings are recovered. The convenience construct `{s}` implicitly makes all free variables in `s` local:

$$\{s\} \;\equiv\; \{x_1, ..., x_n : s\} \quad \text{with } \{x_1, ..., x_n\} \equiv \text{freevars}(s)$$

Appendix A gives a formal definition of the free variables of a strategy expression.

Example: In the strategy expression `{e1,e2,e3:?|[(e1|e2)&e3]|;!|[(e1&e3|e2&e3)]|}`, the scope of the variables `e1`, `e2`, and `e3` is restricted to the match-build sequence. Thus, this expression implements a rewrite rule that can be used multiple times. That is, each time it is applied, fresh unbound variables are used in the pattern match. As an aside, this transformation, which distributes `&` over `|`, is only valid if `e3` is a *pure* Tiger expression, since the `e3` computation is duplicated.

3.3. Strategy Combinators

Match and build are the basic operations of program transformation and can be combined using a few built-in combinators into complex transformations. The combinators can be divided into control combinators and traversal combinators. We start with the former, and come back to the latter in Section 3.5.

The control combinators basically allow composing transformations sequentially, or choosing between transformations. For programming with these combinators, it is useful to have the identity and failure strategies as unit or zero:

$$\Gamma, \mathcal{E} \vdash \langle \texttt{id} \rangle\, t \implies t\, (\Gamma, \mathcal{E}) \qquad \Gamma, \mathcal{E} \vdash \langle \texttt{fail} \rangle\, t \implies \uparrow (\Gamma, \mathcal{E})$$

The *identity* strategy `id` always succeeds and leaves its subject term unchanged. The *failure* strategy `fail` always fails.

Sequential Composition The *sequential composition* s_1 ; s_2 of strategies s_1 and s_2, first attempts to apply s_1 to the subject term. If that succeeds, it applies s_2 to the result; otherwise it fails.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies t'\, (\Gamma', \mathcal{E}') \quad \Gamma', \mathcal{E}' \vdash \langle s_2 \rangle\, t' \implies \overline{t''}\, (\Gamma'', \mathcal{E}'')}{\Gamma, \mathcal{E} \vdash \langle s_1 \texttt{ ; } s_2 \rangle\, t \implies \overline{t''}\, (\Gamma'', \mathcal{E}'')} \qquad \frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle s_1 \texttt{ ; } s_2 \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}$$

From this definition it is clear that in the sequential composition $\texttt{?}p_1 \texttt{;} \texttt{!}p_2$ of a match and a build, the bindings from the match are carried over to the build via the environment $\mathcal{E}'$ in the first rule above. The identity strategy is a unit for sequential composition, and failure a left zero, i.e.

$$\texttt{id ; } s \equiv s \qquad s \texttt{ ; id} \equiv s \qquad \texttt{fail ; } s \equiv \texttt{fail} \qquad \exists s : s \texttt{ ; fail} \not\equiv \texttt{fail}$$

However, failure is not a right zero for sequential composition because of the effects on the state that the first strategy may have.

Guarded Choice The other fundamental strategy combinator is the *guarded choice* s_1 `<` s_2 `+` s_3 of strategies s_1, s_2, and s_3. It first attempts to apply s_1 to the subject term. If that succeeds it applies s_2 to the result, but if it fails, it applies s_3 to the original subject term *and* environment.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies t'\, (\Gamma', \mathcal{E}') \quad \Gamma', \mathcal{E}' \vdash \langle s_2 \rangle\, t' \implies \overline{t''}\, (\Gamma'', \mathcal{E}'')}{\Gamma, \mathcal{E} \vdash \langle s_1 \texttt{ < } s_2 \texttt{ + } s_3 \rangle\, t \implies \overline{t''}\, (\Gamma'', \mathcal{E}'')}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}') \quad \Gamma', \mathcal{E} \vdash \langle s_3 \rangle\, t \implies \overline{t'}\, (\Gamma'', \mathcal{E}'')}{\Gamma, \mathcal{E} \vdash \langle s_1 \texttt{ < } s_2 \texttt{ + } s_3 \rangle\, t \implies \overline{t'}\, (\Gamma'', \mathcal{E}'')}$$

The following laws illustrate the choice between the branches induced by success or failure of the guard strategy:

$$\texttt{id < } s_2 \texttt{ + } s_3 \equiv s_2 \qquad \texttt{fail < } s_2 \texttt{ + } s_3 \equiv s_3$$

This might suggest that the combinator is a simple conditional choice, which it is not. Rather it is a limited backtracking combinator. The guard strategy s_1 can be a complex strategy that may fail at some point, in which case control backtracks to the s_3 strategy, which is applied to the original subject term and environment. But when the s_1 strategy succeeds, the choice is committed and control continues with s_2; no backtracking to s_3 is then possible, even if s_2 or the continuation of the expression fails.

The guarded choice operator is rarely used directly. A number of syntactic abstractions capture typical uses of the choice.

Deterministic Choice The *deterministic* or *left choice* s_1 `<+` s_2 of strategies s_1 and s_2, first attempts to apply s_1 to the subject term. Only if s_1 fails, it attempts to apply s_2 to the subject term. If s_1 and s_2 both fail, the choice fails as well. The left choice combinator is a special case of the guarded choice combinator as expressed by the first of the following equations:

$$s_1 \texttt{ <+ } s_2 \equiv s_1 \texttt{ < id + } s_2 \qquad \texttt{id <+ } s \equiv \texttt{id} \qquad \exists s : s \texttt{ <+ id} \not\equiv s \qquad \texttt{fail <+ } s \equiv s \qquad s \texttt{ <+ fail} \equiv s$$

The other equations assert that identity is a left zero, but not a right unit or zero for left choice, and that failure is a left and right unit for left choice. The inequality above indicates how left choice and identity can be used to turn a strategy that may fail into a strategy that always succeeds. Note that sequential composition does not distribute over left-choice:

$$\exists s_1, s_2, s_3 : (s_1 \texttt{ <+ } s_2) \texttt{ ; } s_3 \not\equiv (s_1 \texttt{ ; } s_3) \texttt{ <+ } (s_2 \texttt{ ; } s_3)$$

$$\exists s_1, s_2, s_3 : s_1 \texttt{ ; } (s_2 \texttt{ <+ } s_3) \not\equiv (s_1 \texttt{ ; } s_2) \texttt{ <+ } (s_1 \texttt{ ; } s_3)$$

In the first case the incorporation of s_3 in the choice may lead to failure of the left branch while s_1 by itself might succeed and commit the choice. The first case becomes an equality if s_3 is guaranteed to succeed. The second case is an equality in case s_1 has no effect on the state Γ.

The choice combinator is typically used as prioritized choice between rewrite rules; if s_1 to s_n are rewrite rules, then the strategy $s_1 \texttt{ <+ } ... \texttt{ <+ } s_n$ tries each of the rules one by one from left to right, stopping as soon as one of the rules has been applied successfully.

Testing A strategy can be used to *test* a property of a term, in which case one is not interested in the result of the transformation, but only in the fact of its success or failure. This can be achieved using the `where` and `not` combinators, which are defined as follows:

$$\texttt{not}(s) \equiv s \texttt{ < fail + id} \qquad \texttt{where}(s) \equiv \texttt{\{}x\texttt{: ?}x\texttt{; } s\texttt{; !}x\texttt{\}} \quad x \notin \text{freevars}(s)$$

The `where` combinator tests whether a strategy succeeds, and the `not` combinator tests whether a strategy fails. Both combinators restore the original term, but the effects on the state and in case of `where` on the environment are retained. The following laws hold for these combinators:

$$\texttt{where(id)} \equiv \texttt{id} \quad \texttt{where(fail)} \equiv \texttt{fail} \quad \texttt{not(fail)} \equiv \texttt{id} \quad \texttt{not(id)} \equiv \texttt{fail}$$

If-then-else-end The guarded left choice combinator is reminiscent of the conventional if-then-else construct in which the condition decides on the branch. Indeed Stratego provides an `if-then-else-end` construct defined as

$$\texttt{if } s_1 \texttt{ then } s_2 \texttt{ else } s_3 \texttt{ end} \equiv \texttt{where}(s_1) \texttt{ < } s_2 \texttt{ + } s_3$$

The condition in this construct preserves the subject term using `where`, entailing that s_2 and s_3 are applied to the original subject term.

Abstractions for Applying Strategies The first class status of pattern matching and instantiation makes it easy to define compound constructs capturing frequently occurring uses of these operations. One such use is the direct application of a strategy to a term pattern and matching the result of a strategy application to a term pattern. These are expressed using the constructs `<`s`>`p and s `=>` p, respectively, which are defined as

$$\texttt{<}s\texttt{>}p \;\equiv\; \texttt{!}p\texttt{ ; }s \qquad s\texttt{ => }p \;\equiv\; s\texttt{ ; ?}p$$

A typical example is the combined use of these constructs in a strategy expression `<add>(`i`,`j`) => `k, which is thus equivalent to `!(`i`,`j`); add; ?`k and applies the strategy `add` to the pair `(`i`,`j`)` and matches the result against the pattern k.

3.4. Strategy Definitions

With matching and building as basic operations, complex transformation strategies can be constructed using a small set of built-in strategy combinators. In order to abstract over recurring patterns in strategy expressions, *strategy definitions* can be used to create new strategy combinators.

A strategy definition f`(`f_1`,...,`f_n`|`x_1`,...,`x_m`) = `s introduces a new strategy combinator f with body s, parameterized with strategy variables f_1, ..., f_n and term variables x_1, ..., x_m. An application f`(`s_1`,...,`s_n`|`p_1`,...,`p_m`)` entails applying the body s of f with the strategy arguments s_i bound to the strategy parameter f_i and the instantiated pattern arguments p_i to the term variables x_i. To express the semantics of strategy definitions we actually need a third environment D in the semantic rules that keeps track of the available strategy definitions. Since this environment is static and just passed around unchanged in all other rules, it is omitted there.

$$\frac{D_{\text{fresh}}(f) = f\,(f_1,...,f_n\,|\,x_1,...,x_m)\;\texttt{=}\;s \quad t_1 \equiv \mathcal{E}(p_1)...t_m \equiv \mathcal{E}(p_m) \quad \mathcal{E}' \equiv [x_1 \mapsto t_1...x_m \mapsto t_m]}{\text{bind}(D,\mathcal{E})(f\,(s_1\,,\,...\,,s_n\,|\,p_1\,,\,...\,,p_m)) \equiv ([f_1\texttt{=}s_1...f_n\texttt{=}s_n],\mathcal{E}',s)}$$

$$\frac{D_{\text{fresh}}(f) = f\,(f_1,...,f_n\,|\,x_1,...,x_m)\;\texttt{=}\;s \quad \exists i : (1 \leq i \leq m \wedge \mathcal{E}(p_i) \equiv \uparrow)}{\text{bind}(D,\mathcal{E})(f\,(s_1\,,\,...\,,s_n\,|\,p_1\,,\,...\,,p_m)) \equiv \uparrow}$$

$$\frac{\text{bind}(D,\mathcal{E})(f\,(\vec{s}\,|\,\vec{p})) \equiv \uparrow}{D,\Gamma,\mathcal{E} \vdash \langle f\,(\vec{s}\,|\,\vec{p})\rangle\, t \implies \uparrow (\Gamma,\mathcal{E}'')}$$

$$\frac{\text{bind}(D,\mathcal{E})(f\,(\vec{s}\,|\,\vec{p})) \equiv (D',\mathcal{E}',s) \quad D'D,\Gamma,\mathcal{E}'\mathcal{E} \vdash \langle s\rangle\, t \implies \overline{t'}\;(\Gamma',\mathcal{E}'\mathcal{E}'')}{D,\Gamma,\mathcal{E} \vdash \langle f\,(\vec{s}\,|\,\vec{p})\rangle\, t \implies \overline{t'}\;(\Gamma',\mathcal{E}'')}$$

The auxiliary 'bind' assertion is used to bind the actual parameters of a call to the formal parameters of a strategy definition. Here $D_{\text{fresh}}(f)$ produces a fresh instance of the strategy definition of f, i.e. using unique new names $f_1,...,f_n$ and $x_1,...,x_m$ for all bound variables. In the implementation, this is of course efficiently implemented by means of stack frames. Note that the original environment $\mathcal{E}$ may change during the execution of the call since the strategy arguments s_i may refer to variables in that environment and bind them, if not already bound. However, the environment $\mathcal{E}'$ does not change since all term parameters are bound at call-time, as expressed in the definition of 'bind'.

The list of term arguments of a strategy combinator is optional and the `|` can be left out if no term arguments are present. Similarly, if the list of strategy arguments is also empty the parentheses may be omitted. Thus we have the following equivalences for definitions and calls:

$$f\texttt{(}f_1\texttt{,}\ldots\texttt{,}f_n\texttt{)} \texttt{ = } s \equiv f\texttt{(}f_1\texttt{,}\ldots\texttt{,}f_n\texttt{|)} \texttt{ = } s \qquad f \texttt{ = } s \equiv f\texttt{(|)} \texttt{ = } s$$
$$f\texttt{(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{)} \equiv f\texttt{(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{|)} \qquad f \equiv f\texttt{(|)}$$

There are no global term variables in Stratego programs. Therefore, the scope of any free term variable in a *top-level* strategy definition is the body of that definition:

$$f\texttt{(}f_1\texttt{,}\ldots\texttt{,}f_n\texttt{|}x_1\texttt{,}\ldots\texttt{,}x_m\texttt{)} \texttt{ = } s \equiv f\texttt{(}f_1\texttt{,}\ldots\texttt{,}f_n\texttt{|}x_1\texttt{,}\ldots\texttt{,}x_m\texttt{)} \texttt{ = \{}y_1\texttt{,}\ldots\texttt{,}y_j\texttt{:}s\texttt{\}}$$
$$\text{with } y_1,...,y_j = \text{freevars}(s)/\{x_1,...,x_m\}$$

Example: The following definitions define the combinator `try` that attempts to apply a strategy, but falls back to `id` when that fails, and `repeat`, which repeatedly applies a strategy until it fails:

```
try(s)    = s <+ id
repeat(s) = try(s; repeat(s))
```

Local Strategy Definitions Strategy combinators can be introduced locally using the `let-in-end` construct, which extends the definition environment while executing the body of the `let`:

$$\frac{\text{fresh}(\texttt{let}\ d_1...d_n\ \texttt{in}\ s\ \texttt{end}) = \texttt{let}\ d'_1...d'_n\ \texttt{in}\ s'\ \texttt{end} \quad [d'_1...d'_n]D,\Gamma,\mathcal{E} \vdash \langle s'\rangle\, t \implies \overline{t'}\ (\Gamma',\mathcal{E}')}{D,\Gamma,\mathcal{E} \vdash \langle \texttt{let}\ d_1...d_n\ \texttt{in}\ s\ \texttt{end}\rangle\, t \implies \overline{t'}\ (\Gamma',\mathcal{E}')}$$

Again the names of the local definitions are renamed to avoid name clashes.

Recursive Closure The *recursive closure* `rec`f`(`s`)` is syntactic sugar for a local recursive definition, i.e., `rec` f`(`s`)` is equivalent to `let` f `=` s `in` f `end`. The construct can be useful in strategy expressions to abbreviate a recursive invocation. For example, by writing `repeat(s) = rec x(try(s; x))` instead of `repeat(s) = try(s; repeat(s))`.

Rewrite Rules Now we can define rewrite rules in terms of strategies. A *labeled conditional rewrite rule* is implemented by a strategy definition that first matches the left-hand side pattern, then evaluates the condition, and finally builds the right-hand side, as is expressed by the equation:

$$dsig\ \texttt{:}\ \ p_1\ \texttt{->}\ p_2\ \texttt{where}\ s \quad\equiv\quad dsig\ \texttt{= \{}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{:?}p_1\texttt{; where(}s\texttt{); !}p_2\texttt{\}}$$
$$\text{with } x_1,...,x_j = \text{freevars}(p_1,p_2,s)/\text{vars}(dsig)$$

An unconditional rule corresponds to a conditional rule with the identity strategy as condition, i.e., $dsig$`:` p_1 `->` p_2 is equivalent to $dsig$`:` p_1 `->` p_2 `where id`. Example: the rewrite rule

```
DefAnd : |[ e1 & e2 ]| -> |[ if e1 then e2 else 0 ]|
```

corresponds to the strategy definition

```
DefAnd = {e1,e2: ?|[ e1 & e2 ]|; where(id); !|[ if e1 then e2 else 0 ]|}
```

Multiple Definitions It is sometimes useful to give a set of rules the same name. For instance the `EvalBinOp` rules in Figure 4 define multiple rules for evaluating binary arithmetic expressions. A set of definitions with the same signature are reduced to a single definition making a choice between the bodies of the definitions, i.e. $dsig$ = s_1 ... $dsig$ = s_n $\equiv$ $dsig$ = $(s_1$ <+ ... <+ $s_n)$. Note that the order in which definitions are combined is not defined. Thus, it is generally only sensible to use this method when definitions are mutually exclusive or confluent, as is the case in the example rules above.

3.5. Term Traversal

The strategy combinators just described combine strategies which apply transformation rules to the roots of their subject terms. In order to apply a rule to a subterm, it is necessary to traverse the term. Stratego defines several basic combinators which expose the direct subterms of a constructor application. These can be combined with the combinators described above to define a wide variety of complete term traversals.

Congruence Congruence combinators provide one mechanism for term traversal in Stratego. If c is an n-ary constructor, then the congruence $c(s_1, \ldots, s_n)$ is the strategy that applies only to terms of the form $c(t_1, \ldots, t_n)$, and works by applying each strategy s_i to the corresponding term t_i. For example, the congruence `Let(`s_1`,`s_2`)` transforms terms of the form `Let(`t_1`,`t_2`)` into `Let(`t'_1`,`t'_2`)`, where t'_i is the result of applying s_i to t_i. If the application of s_i to t_i fails for any i, then the application of $c(s_1, \ldots, s_n)$ to $c(t_1, \ldots, t_n)$ also fails. Congruence combinators can be defined using rewrite rules of the following form:

$c(s_1, \ldots, s_n)$: $c(x_1, \ldots, x_n)$ -> $c(y_1, \ldots, y_n)$ where <s_1>x_1 => y_1;...;<s_n>x_n => y_n

Congruences are very useful for defining traversals that are specific for some abstract syntax. For example, the following strategies define operations on lists using congruences:

```
map(s)    = [] <+ [s | map(s)]
filter(s) = [] <+ [s | filter(s)] <+ Tl; filter(s)
Tl        : [x | xs] -> xs
```

The `map` strategy applies a transformation to each element of a list, but fails when one of those applications fails. The `filter` strategy does the same, but removes elements for which the application fails.

In this article, we will use concrete syntax for congruences over Tiger constructs using the notation <s> to embed a strategy within a Tiger expression. Thus, for example, the strategy expression |[`if` <s_1> `then` <s_2> `else` <s_3>]| corresponds to congruence `If(`s_1`,`s_2`,`s_3`)` over the `if-then-else` construct and applies strategy s_1 to the condition and the strategies s_2 and s_3 to the `then` and `else` branch, respectively. We use the notation <*s> to distinguish application to a list of subterms from application to a list with a single element. For example, Tiger's `let` construct has a list of declarations and a list of expressions as direct subterms. Thus, we use |[`let` <*s_1> `in` <*s_2> `end`]| to denote `Let(`s_1`,`s_2`)`, i.e., the application of the strategies s_1 and s_2 to the lists of declarations and expressions, respectively. Compare this to |[`let` <s_1> `in` <s_2> `end`]|, which denotes `Let([`s_1`],[`s_2`])`.

All Subterms Often a traversal over an abstract syntax tree has uniform behaviour for most or even all constructors of the language. In those cases, it is attractive to use a *generic traversal* instead of spelling out the traversal for all constructors. Stratego provides basic combinators such as `all` and `one` that allow the composition of many different generic traversals. The `all`(s) combinator applies s to all direct subterms t_i of a constructor application $c(t_1, \ldots, t_n)$. It succeeds if and only if all applications to the direct subterms succeed. The resulting term is the constructor application $c(t_1', \ldots, t_n')$ where the t_i' are the results obtained by applying s to the terms t_i.

$$\frac{\Gamma_0, \mathcal{E}_0 \vdash \langle s \rangle\, t_1 \implies t_1'\ (\Gamma_1, \mathcal{E}_1) \quad \ldots \quad \Gamma_{n-1}, \mathcal{E}_{n-1} \vdash \langle s \rangle\, t_n \implies t_n'\ (\Gamma_n, \mathcal{E}_n)}{\Gamma_0, \mathcal{E}_0 \vdash \langle \mathtt{all}(s) \rangle\, c(t_1, \ldots, t_n) \implies c(t_1', \ldots, t_n')\ (\Gamma_n, \mathcal{E}_n)}$$

$$\frac{\Gamma_0, \mathcal{E}_0 \vdash \langle s \rangle\, t_1 \implies t_1'\ (\Gamma_1, \mathcal{E}_1) \quad \ldots \quad \Gamma_{i-1}, \mathcal{E}_{i-1} \vdash \langle s \rangle\, t_i \implies \uparrow (\Gamma_i, \mathcal{E}_i)}{\Gamma_0, \mathcal{E}_0 \vdash \langle \mathtt{all}(s) \rangle\, c(t_1, \ldots, t_n) \implies \uparrow (\Gamma_i, \mathcal{E}_i)}$$

Note that `id` is a zero for `all`, that `all(s)` is the identity on constants (constructor applications without subterms), and that `all(fail)` only succeeds on constants, i.e.

$$\mathtt{all(id)} \equiv \mathtt{id} \qquad \mathtt{<all(}s\mathtt{)>}c \equiv \mathtt{<id>}c \qquad \mathtt{<all(fail)>}c(t_1, ..., t_n) \equiv \mathtt{fail} \quad (\text{if } n > 0)$$

Example: Traversal Strategies Many different traversals can be composed using the `all` traversal combinator. As an example consider the following strategy definitions:

```
topdown(s)   = s; all(topdown(s))
bottomup(s)  = all(bottomup(s)); s
alltd(s)     = s <+ all(alltd(s))
downup(s)    = s; all(downup(s)); s
innermost(s) = bottomup(try(s; innermost(s)))
```

In the first definition, the strategy expression `s; all(topdown(s))` specifies that the parameter transformation `s` is first applied to the root of the current subject term. If that succeeds, the strategy is applied recursively to all direct subterms of the term, and, thereby, to all of its subterms. This definition of `topdown` captures the generic notion of a traversal that visits each subterm in pre-order. `Bottomup` defines a post-order traversal. `Alltd` applies a topdown traversal that stops as the transformation it is applying succeeds. `Downup` applies a transformation pre-order and post-order. `Innermost` is a fixed point traversal that applies a transformation exhaustively starting with innermost terms.

The examples in the following sections use traversal strategies that are partly specific for certain constructs of the language, and otherwise generic where uniform behaviour is possible.

Example: Desugaring and Constant Folding As an example of the application of these generic strategies consider the following *desugaring* transformation for Tiger that simplifies programs by defining constructs in terms of other constructs or simplifying their usage; e.g. splitting let bindings into lets with only a single binding.

```
desugar = topdown(repeat(Desugar <+ LetSplit <+ EmptyLet <+ ElimSingletonTuple))
```

The `desugar` strategy performs a topdown traversal applying a number of rules along the way. Some of these rules are shown in Figure 4. The `Desugar` strategy that is called is a composition of many rewrite rules:

```
Desugar = DefPlus <+ DefMinus <+ DefEq <+ ... <+ DefSeq1 <+ DefAnd <+ DefOr
```

Thus, a fairly elaborate transformation—touching many constructors—is defined in only a few lines of code using separately defined and reusable rewrite rules.

Another example is the following *constant folding* transformation that evaluates constant valued expressions:

```
const-fold = bottomup(try(EvalBinOp <+ EvalRelOp <+ EvalIf <+ EvalWhile <+ EvalFor))
```

It is defined as a bottom-up traversal applying various evaluation rules where possible. Again the transformation is a one-liner using separately defined rewrite rules; each of the `Eval` rules is actually a choice between several rewrite rules with the same name.

One Subterm While the `all` combinator transforms all direct subterms of a term, the `one` combinator finds a single subterm that it transforms.

$$\frac{\Gamma,\mathcal{E} \vdash \langle s\rangle\, t_1 \implies \uparrow(\Gamma_1)\ ...\ \Gamma_{i-1},\mathcal{E} \vdash \langle s\rangle\, t_{i-1} \implies \uparrow(\Gamma_{i-1})\ \ \Gamma_{i-1},\mathcal{E} \vdash \langle s\rangle\, t_i \implies t'_i\ (\Gamma_i,\mathcal{E}_i)}{\Gamma,\mathcal{E} \vdash \langle \texttt{one}(s)\rangle\, c(t_1,\ldots,t_n) \implies c(t_1,...,t_{i-1},t'_i,t_{i+1},...,t_n)\ (\Gamma_i,\mathcal{E}_i)}$$

$$\frac{\Gamma,\mathcal{E} \vdash \langle s\rangle\, t_1 \implies \uparrow(\Gamma_1,\mathcal{E}_1)\quad ...\quad \Gamma_{n-1},\mathcal{E} \vdash \langle s\rangle\, t_n \implies \uparrow(\Gamma_n,\mathcal{E}_n)}{\Gamma,\mathcal{E} \vdash \langle \texttt{one}(s)\rangle\, c(t_1,\ldots,t_n) \implies \uparrow(\Gamma_n,\mathcal{E}_n)}$$

Note that this is a backtracking combinator, in that it restores the environment after failing to apply to a subterm, and before applying to the next subterm. Some example traversal strategies composed with this combinator are:

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

The strategy `is-subterm` is an example application of `oncetd`:

```
is-subterm(|x) = oncetd(?x)
```

It traverses the subject term to find an occurrence of the term argument `x` by means of the match `?x`.

Example: Contextual Rules As an example of the expressivity of the combination of generic traversal and first class pattern matching, consider how contextual rules can be implemented in Stratego [37]. The function inlining rule

```
UnfoldCall :
  |[ let function f(x) = e1 in e2[f(e3)] end ]| ->
  |[ let function f(x) = e1 in e2[let var x := e3 in e1 end] end ]|
```

replaces a call to a function *f* by an instance of its body. This can be implemented by means of a *local traversal* in the condition of the rule:

```
UnfoldCall :
  |[ let function f(x) = e1 in e2 end ]| -> |[ let function f(x) = e1 in e2' end ]|
  where <oncetd({e3: ?|[f(e3)]|; !|[let var x := e3 in e1 end]|})> e2 => e2'
```

The `oncetd` strategy searches for one subterm of *e2* that is a call to function *f* and replaces it with the instantiation of the body. It achieves this by means of a pattern match and a build that involve variables that were bound *in the context* of the traversal, i.e. by the match of the left-hand side of the rule. The Stratego compiler supports contextual rules using the implementation scheme sketched above [44, 37] (although the syntax is currently only supported for rules using abstract syntax terms). While the idea is nice, the shortcoming of the approach is (1) that the traversal is local to the inlining rule and is initiated at the definition site, (2) that only *one* function at a time is inlined, and (3) that `oncetd` does not respect the scoping rules of the object language.

4. Defining Rules Dynamically

In the previous section we have seen how programmable rewriting strategies allow the definition of separate rewrite rules and fine-grained control over their application. The combination of first class pattern matching and traversal strategies allows context-sensitive rewriting to a certain extent in the form of contextual rules. However, the local traversal and the single binding of context variables makes contextual rules of limited use. Nonetheless, the contextual rule solution *does* contain the germ of a more general solution. The strategy expression

```
{e3: ?|[f(e3)]|; !|[let var x := e3 in e1 end]|}
```

in the implementation of the `UnfoldCall` contextual rule is a rewrite rule (a sequence of a match and a build) with some of its pattern variables, i.e., *f* and *e1*, *bound in the context* instead of being bound by the application of the match as is the case with conventional rewrite rules.

This is exactly the idea behind a *dynamic rule*. That is, a dynamic rule is a rewrite rule that inherits bindings to its pattern variables from the context in which it is defined. The difference with a contextual rule is that a dynamic rule is *named* like a normal rewrite rule and can be referred to *outside the lexical*

s	$::=$	`rules`(drd_1 ... drd_n)	dynamic rule definition
		{| $f_1, ..., f_n$: s |}	dynamic rule scope
		s_1 `/`f_1`,...,`f_n`\` s_2	fork and intersect
		s_1 `\`f_1`,...,`f_n`/` s_2	fork and union
		`/`f_1`,...,`f_n`\*` s	fix and intersect
		`\`f_1`,...,`f_n`/*` s	fix and union
drd	$::=$	$drsig$ `:` p_1 `->` p_2 (`where` s)?	dynamic rule definition
		$drsig$ `:+` p_1 `->` p_2 (`where` s)?	dynamic rule extension
		$drsig$ `:` p	dynamic identity rule definition
		$drsig$ `:-` p	dynamic rule undefinition
		f+p	label current scope
$drsig$	$::=$	sig	relative to current scope
		sig.p	relative to labeled scope
		sig+p	relative to current scope and label current scope

Figure 7. Extension of syntax of Stratego with dynamic rules.

scope of its definition. Furthermore, there can be many dynamic rules with the same name, differing in the values bound to the pattern variables of the rule. Thus, an inlining transformation can define rules dynamically when encountering function definitions and apply those rules when encountering function calls, all in the same traversal.

There are many variation points in the definition and use of dynamic rules, which have been captured in a coherent extension of the Stratego language. The syntax of this extension is presented in Figure 7. In this and the next three sections, we will define the semantics of these language constructs, and motivate their design and usage by means of example transformations. In this section, we introduce the basic concepts of dynamic rules, i.e. applying rules, defining rules dynamically, the shadowing of older rules by newer rules, and explicitly undefining rules.

4.1. Example: Constant Propagation in Basic Blocks

Constant propagation replaces uses of variables that can be determined to have constant values with those values [1, 24]. We will use this transformation as a running example, and gradually extend it to cover the whole Tiger language. We start with constant propagation in basic blocks, which should achieve a transformation such as the following:

```
(b := 1;               (b := 1;
 c := b + 3;            c := 4;
 b := b + 1;    ⇒       b := 2;
 b := z + b;            b := z + 2;
 a := b + c)            a := b + 4)
```

Here the variable `b` in the second statement is replaced by its value in the first and the resulting constant expression is folded. The assignment in the third statement redefines the value of `b` to be propagated. The assignment in the fourth statement blocks the further propagation of the constant value of `b`, but `c` does have a constant value in the fifth statement and should be replaced.

Constant folding can clearly be expressed by means of rewrite rules using the `EvalBinOp` rules from Figure 4. The replacement of a variable by a value, e.g. `b` by `1`, can also be expressed by rewriting, i.e. by a rewrite rule such as `|[ b ]| -> |[ 1 ]|`. However, such a rewrite rule cannot be applied everywhere; not to the variable in the left-hand side of an assignment, and not after a different expression has been assigned to the variable. Thus, such a rewrite rule is specific to a part of a particular program, rather than being a universally valid transformation rule. Thus, the idea of dynamic rules is to define such rules *at run-time* and only apply them to the parts of the program where they are valid.

```
prop-const =
  PropConst <+ prop-const-assign <+ (all(prop-const); try(EvalBinOp <+ EvalRelOp))

prop-const-assign =
  |[ x := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst :  |[ x ]| -> |[ e ]| )
                    else rules( PropConst :- |[ x ]| ) end
```

Figure 8. Strategy for constant propagation in basic blocks.

The `prop-const` strategy in Figure 8 implements constant propagation for basic blocks using dynamic rules. The strategy has three cases. The first is the application of the `PropConst` rule, which replaces a variable with a constant value, if it has one. In the second case an assignment statement is encountered and used to define a dynamic rule, as will be discussed below. In the final case, a generic traversal is performed and the sub-expressions are transformed with the `prop-const` transformation. After that an attempt is made to perform constant folding using some appropriate `Eval` rule.

Now the crucial part of the transformation is the `prop-const-assign` strategy, which *defines* a `PropConst` rule for each particular assignment it encounters. First, the congruence strategy `|[`*x* `:= <prop-const =>` *e*`>]|` (equivalent to `Assign(?x,prop-const => e))` is used to transform the right-hand side expression by a recursive invocation of the `prop-const` strategy, leaving the left-value untouched (to prevent the replacement of the variable in the left-value). If the right-hand side expression of the assignment is a constant, as determined by the `is-value` strategy, then a `PropConst` rule is defined that replaces an occurrence of the variable *x* from the left-hand side of the assignment by the expression *e* from the right-hand side. This is expressed by the strategy expression

```
rules( PropConst : |[ x ]| -> |[ e ]| )
```

The `rules` construct introduces several rules which inherit variable bindings for all variables occurring in the context. If the expression *e* is not a constant, then the `PropConst` rule is *undefined* for *x* with the strategy expression

```
rules( PropConst :- |[ x ]| )
```

which disables any `PropConst` rules with *x* as left-hand side.

To get an impression of events during this transformation, the middle box in the following diagram represents at each line the set of dynamic `PropConst` rules that are valid *after* transforming the statement on the same line in the left box.

```
(b := 1;            b -> 1                (b := 1;
 c := b + 3;        b -> 1, c -> 4         c := 4;
 b := b + 1;   ⇒    b -> 2, c -> 4    ⇒    b := 2;
 b := z + b;                c -> 4         b := z + 2;
 a := b + c)                c -> 4         a := b + 4)
```

The `b-` and `a-` entries indicate that the `PropConst` rule is *undefined* for these variables. Note how the definition of the rule for `b` on the third line *replaces* the previous rule for `b`. Thus, a rule can be *redefined* as well as undefined.

4.2. Semantics: Defining and Undefining Rules

For each dynamic rule L an entry in the state Γ is maintained. This entry encodes the current set of dynamic rules. In the semantic rules, we encode a set of dynamic rules as a strategy expression, since that allows us to define the behaviour of dynamic rules as concisely as possible. In our current implementation a more efficient encoding using hash tables is used [8].

Thus, *applying a dynamic rule* L entails looking up the strategy s encoding the current rule set for L and applying it.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s)}, \mathcal{E} \vdash \langle L \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}')}$$

$\Gamma_{L(s)}$ means that Γ has an entry for L and that its strategy is s. If no L rules are defined, we have $\Gamma_{L(\texttt{fail})}$.

The `rules(...)` construct is used to define dynamic rules and can contain a list of rule definitions. Such a list is equivalent to the sequential composition of the definition of the individual rules, i.e.

$$\texttt{rules}(L_1 : r_1 \ldots L_n : r_n) \;\; \equiv \;\; \texttt{rules}(L_1 : r_1) \texttt{ ; } \ldots \texttt{ ; } \texttt{rules}(L_n : r_n)$$

The definition of a single dynamic rule entails modifying the L entry in Γ.

$$\frac{s_1' \equiv \texttt{\{}\overline{\mathcal{E}}(?p_1 \texttt{ ; } \texttt{where}(s_2) \texttt{ ; } !p_2)\texttt{\}} \;\texttt{<+}\; s_1}{\Gamma_{L(s_1)}, \mathcal{E} \vdash \langle \texttt{rules}(L : p_1 \to p_2 \,\texttt{where}\, s_2)\rangle\, t \implies t\ (\Gamma_{L(s_1')}, \mathcal{E})}$$

The new strategy is a prioritized choice that first tries to apply the new rule, and if that fails the old strategy expression is applied. The new rule is specialized by substituting variables bound in the environment. This definition entails that: (1) multiple rules can be defined at the same time; (2) a definition of a rule with the same left-hand side as an earlier defined rule, shadows (or redefines) that earlier rule. Note that we require that left-hand sides of dynamic rules do not overlap in order to guarantee an efficient implementation; see [8].

Finally, rules can be *undefined*. This is achieved by inserting into the $\Gamma_{L(s)}$ strategy a test for the pattern concerned and explicitly failing when it is encountered.

$$\frac{s' \equiv \texttt{if } \texttt{\{}?\overline{\mathcal{E}}(p)\texttt{\}} \texttt{ then fail else } s \texttt{ end}}{\Gamma_{L(s)}, \mathcal{E} \vdash \langle \texttt{rules}(L \texttt{ :- } p)\rangle\, t \implies t\ (\Gamma_{L(s')}, \mathcal{E})}$$

This entails that after undefining a pattern, an attempt to rewrite a term matching that pattern will fail. For terms not matching the pattern, the search continues in the old strategy, however.

5. Dynamic Rule Scope

A useful feature of dynamic rules as defined in the previous section is that rule definitions are not constrained to a lexical scope, but are visible globally. This entails that rules are propagated *implicitly*; a transformation strategy does not need to pass around the current set of rules. Thus, a rule defined in one part of a strategy can be applied in another part, without parameter passing. In particular, this means that a transformation can be organized as a sequence of phases that pass on information through dynamic rules. For instance, one phase may define inlining rules for top-level functions, which are then used to inline function calls in subsequent phases.

As a consequence of this design, the definition of a dynamic rule permanently redefines any previous definition for the same left-hand side. Likewise, the undefinition of a dynamic rule permanently erases that definition. However, sometimes it is useful to redefine or undefine a rule only temporarily and restore the old definition after performing some local transformation. For instance, an inlining rule for a local function may redefine an inlining rule for an outer function with the same name. Thus, after traversing the subtree in which the local function is in scope, the inlining rule for the outer function should be restored. Achieving this with only rule definition and undefinition requires maintaining information about dynamic rules in strategies, which is undesirable. Instead, Stratego provides a construct for limiting the lifetime of dynamic rules. In this section, we introduce the basic *dynamic rule scope* construct $\texttt{\{|}\, L : s \,\texttt{|\}}$, and its refinement with scope labels, which provide fine-grained control over the scope in which a rule is defined. These concepts are illustrated by bound variable renaming, function inlining, constant propagation with local variables, and dead function elimination.

```
let var a : int := x
    function foo(a : int) : int =
      let var a := a + 3
          var z := 0
       in for a := a to a + 100 do
            z := z + a end
 in foo(a) end
```

⇒

```
let var a : int := x
    function foo(b : int) : int =
      let var c := b + 3
          var z := 0
      in for d := c to c + 100 do
           z := z + d end
in foo(a) end
```

Figure 9. Application of bound-variable renaming.

5.1. Example: Bound Variable Renaming

Programs can use the same name for different variables in a program. Local variables *shadow* the declaration of variables in outer blocks. The scoping rules of the language determine which variable occurrence corresponds to which variable declaration. Bound variable renaming is a transformation that gives declared variables a unique new name, producing a program in which no variable declaration shadows any other variable declaration. This may be useful to clarify the program for programmers, and for other program transformations, after bound variable renaming, two occurrences of the same identifier denote the same variable. Bound variable renaming is necessary in transformations to avoid variable capture upon substitution.

The example in Figure 9 illustrates bound variable renaming for local variables (`var`), function arguments (`function`) and loop index variables (`for`) in Tiger. The program on the left uses the identifier `a` for a number of different variables. These are renamed to new identifiers in the program on the right. The example illustrates the different binding constructs in Tiger and their scoping rules. That is, not all subterms are necessarily in the scope of a declaration. The arguments of a function definition are visible in the body of that function, and shadow all external declarations for the same name. A local variable declaration (`var`) is visible in all subsequent declarations in the same `let` and in the body of the `let`, but *not* in the initializer of the declaration. Finally, the index variable of a `for` loop is local to that loop, but the lower and upper bound of the loop are not in the scope of that declaration.

Implementation of bound variable renaming can be achieved with dynamic rules, where we define a new renaming rule for each declared variable. Thus, for each binding construct we define a transformation rule that (1) renames the variable being declared and (2) defines the renaming rule `RenameVar` to replace occurrences of the old variable with its new name. For example, for variable declarations we introduce the following rule:

```
RenameVarDec :
  |[ var x ta := e ]| -> |[ var y ta := e ]|
  where new => y; rules(RenameVar : |[ x ]| -> |[ y ]|)
```

It replaces the identifier x in the declaration with a new identifier y. The `new` primitive strategy generates a new name that is guaranteed not to occur anywhere in any term currently in memory. (See Appendix B of [8] for a discussion of the semantics of `new`.) Furthermore, an instance of `RenameVar` is defined, renaming an occurrence of x to y.

Now, when we would just perform a topdown traversal over the program applying `RenameVarDec` and `RenameVar`, as in

```
exprename = try(RenameVarDec + RenameVar); all(exprename)
exprename = rec rn(
  RenameVar
  <+ |[ var <id> <id> := <rn> ]| ; RenameVarDec
  <+ |[ let <*id> in <*id> end ]| ; {| RenameVar : all(rn) |}
  <+ |[ for <id> := <rn> to <rn> do <id> ]|
     ; {| RenameVar: RenameFor; |[ for <id> := <id> to <id> do <rn> ]| |}
  <+ |[ function <id>(<*id>) <id> = <id> ]|
     ; {| RenameVar : RenameArgs; |[ function <id>(<*id>) <id> = <rn> ]| |}
  <+ all(rn))

RenameVarDec :
  |[ var x ta := e ]| -> |[ var y ta := e ]| where <NewVar> x => y

RenameFor :
  |[ for x := e1 to e2 do e3 ]| -> |[ for y := e1 to e2 do e3 ]|
  where <NewVar> x => y

RenameArgs :
  |[ function f(x1*) ta = e ]| -> |[ function f(x2*) ta = e ]|
  where <map(FArg|[ <NewVar> : <id> ]|)> x1* => x2*

NewVar :
  x -> y where if <RenameVar> |[ x ]| then new else !x end => y
            ; rules( RenameVar : |[ x ]| -> |[ y ]| )
```

Figure 10. Bound variable renaming.

the result would not be correct, since the renaming rules for variables in inner `lets` would replace renaming rules for variables from outer `lets`. Thus, the scope of a renaming rule should be restricted to the traversal of that part of the program in which the corresponding variable is in scope. After that the old renaming rule should re-emerge.

The dynamic rule scope construct {| $L : s$ |} restricts the scope of new definitions of the dynamic rule L to the strategy s. That is, any rule defined during the execution of s is removed after s terminates. Thus, in the strategy expression `{| RenameVar : all(exprename) |}` any `RenameVar` rule defined during the traversal `all(exprename)` is undefined afterwards. This is exactly what we need in order to restrict renaming rules to the scope of the corresponding variable declarations.

Figure 10 presents the complete variable renaming transformation for Tiger. It consists of a number of rules that rename identifiers in binding constructs using the auxiliary rule `NewVar`, which defines the `RenameVar` rule. These rules are called during the traversal performed by the `exprename` strategy. This strategy uses the dynamic rule scope construct to restrict the scope of the `RenameVar` rules. For instance, the clause

```
|[ let <*id> in <*id> end ]| ; {| RenameVar : all(s) |}
```

declares that any `RenameVar` rules defined during the traversal of a `let` are restricted to that `let`. Similarly, the traversal restricts the scope of function arguments to the body of the function definition and the scope of the `for` loop index variable to the *body* of the loop. Note that the loop bound expressions

```
let function fact(n : int) : int =
      let function f(n : int, acc : int) : int =
           (while n > 0 do
              (acc := acc * n; n := n - 1)
            ; acc)
       in f(n, 1) end
 in fact(10) end
```

⇒

```
let var n : int := 10
    var b : int := n
    var acc : int := 1
 in while b > 0 do
      (acc := acc * b;
       b := b - 1);
    acc end
```

Figure 11. Example application of function inlining.

are visited before defining the renaming rule. Similarly, the initializer of a variable declaration is visited before renaming the variable declaration itself, thus ensuring that any occurrences of the identifier within the initializer are renamed first. Finally, note how the `NewVar` rule invokes the `RenameVar` rule to establish whether the identifier was already used in an enclosing scope; if `RenameVar` fails, then this is not the case and there is no need to rename the identifier.

5.2. Example: Function Inlining

Another example of the use of scoped definitions is *function inlining*. Function inlining is a transformation that replaces a function call with the body of the corresponding function, instantiating the formal parameters with the actual parameters. For example, the transformation in Figure 11 shows the result of first inlining the local function `f` in the body of `fact` and then inlining the function `fact` at its call site.

The specification in Figure 12 defines a simple inlining algorithm replacing all function calls of *inlineable* functions with their bodies. This algorithm is overly simplistic, since an actual inliner would (1) use sophisticated criteria at the definition *and* at the call site to determine whether a particular call should be inlined, and (2) would combine inlining with other transformations [31]. Nonetheless, the strategy contains the essence of inlining.

The `inline` strategy has four cases. The first case is the application of the dynamic `UnfoldCall` rule to replace a function call with an instantiation of the function body, followed by a renaming of bound variables (`exprename`), and a recursive invocation of `inline` on the inlined code. In the second case, a new scope for the `UnfoldCall` rule is entered when encountering a `let`, to ensure that functions are only inlined in their lexical scope.

The real work is done in the third case, when encountering a list of function declarations. Recall from the syntax of Tiger in Figure 1 on page 129 that a list of function declarations is a single declaration in a let binding. Furthermore, these declarations can be mutually recursive, and then have to be treated as one when inlining. The strategy `inline-fundecs` uses the congruence $[\![$`<fd*:`s`>`$]\!]$ to transform a list of function declarations. In particular, `inline-fundecs` visits the declarations three times. The first time to define an unfolding rule for each inlineable[2] function. The second time to apply the inlining transformation to the function declarations themselves. And the third time to define unfolding functions for the transformed function declarations. Furthermore, the last stage *removes* each function declaration that is being inlined; after inlining all function calls there is no need for the declarations anymore.

[2]The strategy `is-inlineable` represents some analysis to determine whether a function is inlineable. For instance, recursive functions could be excluded from inlining to prevent non-termination. Other heuristics such as the size of the function body, or its ‘atomicity’ could be added.

```
inline = inline-call <+ inline-let <+ inline-fundecs <+ all(inline)

inline-call = UnfoldCall; exprename; inline

inline-let = |[ let <*id> in <*id> end ]| ; {| UnfoldCall : all(inline) |}

inline-fundecs =
  |[ <fd*: map(is-inlineable < define-unfold + undefine-unfold)
        ; map(inline)
        ; filter(is-inlineable < define-unfold; fail + undefine-unfold)> ]|

define-unfold =
  ?|[ function f(x*) ta = e ]|
  ; rules( UnfoldCall : |[ f(a*) ]| -> |[ let d* in e end ]|
                          where <zip(bind-arg)> (x*, a*) => d* )
bind-arg :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]|

undefine-unfold =
  ?|[ function f(x*) ta = e ]|; rules( UnfoldCall :- |[ f(a*) ]|)

zip(s) : ([], []) -> []
zip(s) : ([x|xs], [y|ys]) -> [z | zs] where <s>(x,y) => z; <zip(s)>(xs, ys) => zs
```

Figure 12. Function inlining.

The `define-unfold` strategy defines a new `UnfoldCall` rule for a function *f*, replacing a call to *f* with its body inside a `let`, binding the actual parameters *a** to the formal parameters *x** as local variables. The creation of the list *d** of local variable declarations is achieved by `zip`ping together the list of formal parameters *x** and the list of actual parameters *a**, using `bind-arg` to create a declaration for each pair. If a function is not deemed inlineable, the `UnfoldCall` rule is undefined in order to shadow any inlining rules for a function with the same name in an enclosing scope.

5.3. Semantics: Scope

In the semantics of dynamic rules in the previous section, a strategy expression was used to encode the set of rules defined. In order to keep track of rules introduced in different scopes, we refine this to a list of strategy expressions $s_1|...|s_n$, where the leftmost strategy s_1 denotes the rules defined in the most recent scope. The scope construct $\{|\, L_1, ..., L_n : s \,|\}$ can restrict the scope of multiple dynamic rules L_1 to L_n, which is equivalent to nesting the scopes, as expressed by the following equation:

$$\{|\, L_1, ..., L_n : s \,|\} \equiv \{|\, L_1 : \{|\, L_2 : ... \{|\, L_n : s \,|\} ... \,|\} \,|\}$$

Therefore, we will treat only the case of a scope for a single rule. Thus, entering a new scope entails adding a new scope strategy to the list:

$$\frac{\Gamma_{L(\texttt{fail}|s_2|...|s_n)}, \mathcal{E} \vdash \langle s \rangle\, t \implies \overline{t'}\ (\Gamma'_{L(s_1|s_2|...|s_n)}, \mathcal{E}')}{\Gamma_{L(s_2|...|s_n)}, \mathcal{E} \vdash \langle \{|\, L : s \,|\} \rangle\, t \implies \overline{t'}\ (\Gamma'_{L(s_2|...|s_n)}, \mathcal{E}')}$$

Since no rules have been defined yet, the strategy for the new scope corresponds to `fail`. After application of the strategy s, the new scope is removed. The definition and undefinition of a dynamic rule modifies the strategy expression in the current scope:

$$\frac{s_1' \equiv \text{define}(drd, \mathcal{E}, s_1)}{\Gamma_{L(s_1|s_2|...|s_n)}, \mathcal{E} \vdash \langle \texttt{rules}(drd)\rangle\, t \implies t\ (\Gamma_{L(s_1'|s_2|...|s_n)}, \mathcal{E})}$$

Where the modification of the scope strategy is factored out using the semantic function 'define':

$$\text{define}(L : p_1 \to p_2\, \texttt{where}\, s_1, \mathcal{E}, s_2) \equiv \texttt{\{}\overline{\mathcal{E}}(?p_1\ \texttt{;}\ \texttt{where}(s_1)\ \texttt{;}\ !p_2)\texttt{\}} \mathrel{<\!+} s_2$$
$$\text{define}(L :\!-\, p, \mathcal{E}, s) \equiv \texttt{\{}?\overline{\mathcal{E}}(p)\ \texttt{;}\ !\bot\texttt{\}} \mathrel{<\!+} s$$

That is, undefining a rule is modeled by producing the special term $\bot$. This is necessary to distinguish failure to find any matching pattern in the current scope from finding an undefined pattern.

Applying a rule requires finding the *most recent* rule definition matching the subject term. This corresponds to the prioritized application of the strategies corresponding to the scopes, with the most recent scope having the highest priority. There are three cases to consider. First, one of the scope strategies succeeds, producing a term t' (not equal to $\bot$).

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{<\!+} ... \mathrel{<\!+} s_n\rangle\, t \implies t'\ (\Gamma', \mathcal{E}') \quad t' \not\equiv \bot}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle L\rangle\, t \implies t'\ (\Gamma', \mathcal{E}')}$$

Secondly, t matches an explicitly undefined pattern, hence, the application of the scope strategies produces $\bot$. In that case, application of the dynamic rule fails. Finally, if all of the scope strategies fail, then obviously no rule matching t was defined, and application fails as well.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{<\!+} ... \mathrel{<\!+} s_n\rangle\, t \implies \bot\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle L\rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')} \qquad \frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{<\!+} ... \mathrel{<\!+} s_n\rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle L\rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}$$

5.4. Example: Constant Propagation for Local Variables

New dynamic rules are defined in the current scope and disappear at the end of that scope. This is not always adequate. Sometimes it is necessary to redefine rules that have been defined in an earlier scope. Consider for instance, the application of constant propagation in Figure 13. Each of the variables `a`, `b`, and `c` requires different behaviour. The inner `let`-block defines a new local variable `c`. Hence, the propagation rules for `c` in the enclosing block should be shadowed within and restored at the end of the inner block. Thus, the occurrence of `c` in the last statement refers to the value of `c` before the inner block. This behaviour is obtained by using a dynamic rule scope for the traversal of `let` blocks, as follows:

```
let var a := 1 var b := 2 var c := 3
 in a := b + c;
    let var c := a + 1
     in b := b + c;
        a := a + b;
        b := z + b end;
    a := c + b + a end
```

$\Rightarrow$

```
let var a := 1 var b := 2 var c := 3
 in a := 5;
    let var c := 6
     in b := 8;
        a := 13;
        b := z + 8 end;
    a := 3 + b + 13 end
```

Figure 13. Application of constant propagation on program with local variables.

```
prop-const =
  PropConst <+ prop-const-assign <+ prop-const-let <+ prop-const-vardec
  <+ all(prop-const); try(EvalBinOp <+ EvalRelOp)

prop-const-let =
  |[ let <*id> in <*id> end ]|; {| PropConst : all(prop-const) |}

prop-const-vardec =
  |[ var x ta := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst+x :  |[ x ]| -> |[ e ]| )
                     else rules( PropConst+x :- |[ x ]| ) end

prop-const-assign =
  |[ x := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst.x :  |[ x ]| -> |[ e ]| )
                     else rules( PropConst.x :- |[ x ]| ) end
```

Figure 14. Constant propagation for basic blocks with local variables.

```
|[ let <*id> in <*id> end ]|; {| PropConst : all(prop-const) |}
```

This entails that a new local propagation rule is defined when encountering a local variable declaration or assignment, and these rules are discarded after the traversal.

On the other hand, the variables `a` and `b` are not redeclared in the inner block. Thus, assignments in the inner block affect the values of these variables in the enclosing block. That is, the occurrence of `a` in the last statement refers to the value assigned to `a` in the inner block, and since the value of `z` cannot be determined, `b` should not be replaced in the last statement. Now, the problem is that this behaviour is not supported by dynamic rule scopes as defined above. Any rules that are defined within the scope are discarded afterwards.

The solution is more fine-grained control over the scope in which a rule is defined. We achieve this by *labeling* scopes with symbolic names (terms), and referring to these labels when defining or undefining a rule. The specification in Figure 14 is an extension of the constant propagation strategy in Figure 8 for blocks with local variables. A new scope is started just as in the case of renaming with `{| PropConst : all(prop-const) |}`. In order to ensure that propagation rules are defined in the right scope, these are labeled with the names of the variables declared within that scope. Thus, when encountering a variable declaration, the rule definition

```
rules( PropConst+x : |[ x ]| -> |[ e ]| )
```

defines a constant propagation rule for the variable *x* in the *current* scope *and* labels the current scope with label *x*. This models the fact that a variable declaration introduces a new local variable. On the other hand, when encountering an assignment, a propagation rule is defined in the scope labeled with the name of the variable being assigned to:

```
rules( PropConst.x : |[ x ]| -> |[ e ]| )
```

In case the expression assigned to a variable is not a constant, the `PropConst` rule is undefined for that variable, following the same reasoning: labeling the current scope in the case of a variable declaration, and undefining in the labeled scope in the case of an assignment.

5.5. Semantics: Labeled Scopes

The definition of a dynamic rule in conjunction with labeling the current scope is a composition of those two operations:

$$\texttt{rules}(L\texttt{+}p : r) \quad \equiv \quad \texttt{rules}(L\texttt{+}p) \ \texttt{;} \ \texttt{rules}(L.p : r)$$

To model labeled scopes, each scope of a dynamic rule has a list of labels associated with it; $\Gamma.\text{labels}_{L_i}$ denotes the set of terms labeling the ith scope of Γ_L. Labeling the current scope entails adding a label to this list:

$$\frac{\mathit{lbls} \equiv [\mathcal{E}(p)|\Gamma.\text{labels}_{L_1}]}{\Gamma, \mathcal{E} \vdash \langle\texttt{rules}(L\texttt{+}p)\rangle\, t \implies t\ (\Gamma.\text{labels}_{L_1} := \mathit{lbls}, \mathcal{E})}$$

Note that labels are term patterns and are instantiated using the current variable bindings.

Defining a rule in a scope labeled with p, entails finding the first scope that is labeled with p, extending the corresponding strategy, and removing the rule from any more recent scopes.

$$\frac{\begin{array}{c}\text{label}(drd) \equiv p \quad \mathcal{E}(p) \in \Gamma.\text{labels}_{L_i} \quad \text{define}(drd, \mathcal{E}, s_i) \equiv s'_i \\ \forall_{j=1}^{i-1}(\mathcal{E}(p) \notin \Gamma.\text{labels}_{L_j} \wedge \text{remove}(drd, \mathcal{E}, s_j) \equiv s'_j)\end{array}}{\Gamma_{L(s_1|\ldots|s_{i-1}|s_i|s_{i+1}|\ldots|s_n)}, \mathcal{E} \vdash \langle\texttt{rules}(drd)\rangle\, t \implies t\ (\Gamma_{L(s'_1|\ldots|s'_{i-1}|s'_i|s_{i+1}|\ldots|s_n)}, \mathcal{E})}$$

Here 'label' denotes the label of a dynamic rule definition, where $\oplus$ abstracts over definition (`:`), undefinition (`:-`), and extension (`:+`):

$$\texttt{rules}(L \oplus r) \ \equiv \ \texttt{rules}(L.\epsilon \oplus r) \qquad \text{label}(L.p \oplus r) \ \equiv \ p$$

Note that ϵ is used to denote the current scope. That is, defining a rule without a label is equivalent to defining a rule in the scope labeled with ϵ. Since every scope has this label, this entails defining it in the current scope. Removing a rule from a scope is defined as follows:

$$\begin{array}{rcl}\text{remove}(L.p_0 : p_1 \to p_2\ \texttt{where}\ s_1, \mathcal{E}, s_2) & \equiv & \texttt{\{?}\overline{\mathcal{E}}(p_1)\texttt{\}} \ \texttt{<} \ \texttt{fail} \ \texttt{+} \ s_2 \\ \text{remove}(L.p_0 :\!\text{-}\ p_1, \mathcal{E}, s) & \equiv & \texttt{\{?}\overline{\mathcal{E}}(p_1)\texttt{\}} \ \texttt{<} \ \texttt{fail} \ \texttt{+} \ s\end{array}$$

By letting the scope strategy fail when encountering the pattern concerned, lookup will proceed in the next scope.

Finally, the semantics of dynamic rule scope needs to be redefined, since the strategies of enclosing scopes may change within the scope by rule definition relative to a label:

$$\frac{\Gamma_{L(\texttt{fail}|s_2|\ldots|s_n)}, \mathcal{E} \vdash \langle s\rangle\, t \implies \overline{t'}\ (\Gamma'_{L(s'_1|s'_2|\ldots|s'_n)}, \mathcal{E}')}{\Gamma_{L(s_2|\ldots|s_n)}, \mathcal{E} \vdash \langle\texttt{\{|}\, L : s \,\texttt{|\}}\rangle\, t \implies \overline{t'}\ (\Gamma'_{L(s'_2|\ldots|s'_n)}, \mathcal{E}')}$$

A scope label may also be assigned as part of the scope declaration, which is an abbreviation for a scope with an explicit labeling action:

$$\texttt{\{|}\, L.p : s \,\texttt{|\}} \quad \equiv \quad \texttt{\{|}\, L : \texttt{rules}(L\texttt{+}p); s \,\texttt{|\}}$$

The application of a dynamic rule is not affected by the addition of labels.

6. Extending Dynamic Rules

The examples of dynamic rewrite rules we have considered so far rewrite one left-hand side term to one right-hand side term. When defining a new rule, the old rule with the same left-hand side is discarded. In some applications it turns out to be useful to be able to rewrite to multiple right-hand sides. For example, in partial evaluation, for each function call with static arguments, a specialized function definition is generated. Thus, the original function definition should be rewritten to a list of specialized function definitions. This could be modeled by maintaining a list of terms as right-hand side. For example, the strategy for specializing function calls might have a code fragment such as the following:

```
... => fd
; <Specializations> |[ function f(x*) ta = e ]| => fd*
; rules( Specializations : |[ function f(x*) ta = e ]| -> [fd|fd*] )
```

where somehow a specialized function definition *fd* is computed. To extend the list with specializations for function *f*, the old list of specializations *fd** is retrieved by applying the `Specializations` rule, and then the `Specializations` rule for function *f* is redefined to include the new function definition. This turned out to be a frequently occurring programming pattern for dynamic rules, leading to code cluttered with list extension operations, distracting from the actual transformation being defined.

The idea of dynamic rule *extension* is to allow multiple rules with the same left-hand side and different right-hand sides. For example, using this approach, function specializations can be modeled by a dynamic rule `Specialization`, which rewrites a function definition to *a* specialization. Thus, the code fragment above is reduced to

```
... => fd
; rules( Specialization :+ |[ function f(x*) ta = e ]| -> fd )
```

which declares that the dynamic rule is *extended* with a new case for function *f* without discarding previously defined rules. A dynamic rule thus defined can be applied in various ways; producing the most recently added right-hand side, producing all right-hand sides, producing one right-hand side and then discarding it. The result is a more general model for dynamic rules in which the undefinedness of a dynamic rule corresponds to the absence of any rules. In this section, we generalize dynamic rules to support rule extension with different application modes, and we illustrate rule extension with common subexpression elimination. Another application features in the function specialization example in Section 8.

6.1. Example: Common Subexpression Elimination in Basic Blocks

Common subexpression elimination (CSE) is a transformation in which an expression is replaced with a variable containing the value of that expression as computed earlier in the program. The following application illustrates the transformation, and the main issues in its implementation:

```
(x := a + b;                    (x := a + b;
 y := a + b;                     y := x;
 z := a + c;           ⇒         z := a + c;
 a := 1;                         a := 1;
 z := (a + c) + (a + b))         z := (a + c) + (a + b))
```

```
cse = cse-assign <+ (all(cse); try(ReplaceExp))

cse-assign =
  |[ x := <cse => e> ]|
  ; where(<undefine-subexpressions> |[ x ]|)
  ; if <not(is-subterm(||[ x ]|))> |[ e ]| then
      rules(ReplaceExp : |[ e ]| -> |[ x ]|)
      ; where(<register-subexpressions(|e)> |[ x := e ]|)
    end

register-subexpressions(|e) =
  get-vars; map({y : ?|[ y ]|; rules(UsedInExp :+ |[ y ]| -> e)})

undefine-subexpressions =
  bagof-UsedInExp; map({?e; rules(ReplaceExp :- |[ e ]|)})

get-vars = collect({?|[ x ]|})
```

Figure 15. Common subexpression elimination in basic blocks.

The example shows how later occurrences of the expression `a + b` can be replaced with the variable `x`, since that variable contains the value of the expression. However, as soon as the variable `x` *or* one of the variables `a` or `b` in the expression are assigned a new value, that replacement is no longer valid. Thus, the occurrence of `a + b` in the last statement cannot be replaced, since the assignment to `a` in the preceding statement invalidates it. For the same reason, the occurrence of `a + c` in that statement cannot be replaced with `z`.

Figure 15 shows the specification of common subexpression elimination for basic blocks. The transformation is similar to constant propagation, but the rewrite rule is reversed. That is, instead of defining a rule that rewrites the variable *x* to the expression *e* when encountering an assignment |[*x* := *e*]|, a rule is defined that rewrites *e* to *x* (unless *x* occurs in *e*). The main difference between CSE and constant propagation is that it is not obvious which rules to undefine when encountering an assignment. In the case of constant propagation, an assignment |[*x* := *e*]| invalidates the propagation rule with variable *x* as left-hand side. This is directly expressed as `rules(PropConst :- |[x]|)`. However, in common subexpression elimination, an assignment |[*x* := *e*]| invalidates *all* rules that rewrite an expression *e'* containing *x* or to *x*.

The specification in Figure 15 models this by maintaining *two* dynamic rules; `ReplaceExp` rewrites expressions to the variables that contain their value, and `UsedInExp` keeps track of which expressions a variable is used in. Thus, `register-subexpressions` defines an instance of `UsedInExp` for each variable occurring in an assignment. This rule definition is *an extension*, since a variable can occur in multiple expressions. Subsequently, on encountering an assignment |[*x* := *e*]|, all `ReplaceExp` rules that rewrite an expression containing the variable *x* are undefined by `undefine-subexpressions`. This is achieved using the `bagof-UsedInExp` strategy that produces *all* expressions that `UsedInExp` rewrites *x* to.

6.2. Dependent Dynamic Rules

The approach of using an extra dynamic rule to keep track of the rules defined for another dynamic rule may seem awkward, and cumbersome to extend to a setting with local scopes and control flow (as discussed in Section 7). However, a general pattern can be recognized. The `cse` strategy is registering all dependencies of the `ReplaceExp` dynamic rule. In [27] we describe *dependent dynamic rules*, an extension of dynamic rules, which provides built-in support for registering rule dependencies and undefining rules by their dependencies. Using this approach, the `ReplaceExp` rule is defined as

```
where(<get-vars> |[ x := e ]| => deps)
; rules(ReplaceExp : |[ e ]| -> |[ x ]| depends on deps)
```

When encountering an assignment `|[y := e']|` a call to `undefine-ReplaceExp(|y)` suffices to undefine all rules depending on `y`. The dependent dynamic rule mechanism is an abstraction built on top of the dynamic rules described in this article.

6.3. Semantics: Extend Rule

To define the semantics of rule extension, the strategy encoding of a rule set needs to produce all terms that a term rewrites to. This is implemented in the semantics by having the strategies produce a list of terms. A normal rule definition adds an alternative that produces a singleton list, thus discarding all previously defined rules matching the same pattern. Undefinition of a pattern (`:-`) is modeled by a strategy producing the empty list. Finally, extension (`:+`) is defined by a strategy that builds a list with the new right-hand side as head element and any other applicable terms for applying the original strategy to produce the tail of the list.

$$\begin{aligned}
\text{define}(L.p_0 : p_1 \to p_2\,\texttt{where}\,s_1, \mathcal{E}, s_2) &\equiv \texttt{\{?}\overline{\mathcal{E}}(p_1)\texttt{;where(}\overline{\mathcal{E}}(s_1)\texttt{);![}\overline{\mathcal{E}}(p_2)\texttt{]\}} \mathrel{\texttt{<+}} s_2 \\
\text{define}(L.p_0 :\!- p_1, \mathcal{E}, s) &\equiv \texttt{\{?}\overline{\mathcal{E}}(p_1)\texttt{; []\}} \mathrel{\texttt{<+}} s \\
\text{define}(L.p_0 :\!+ p_1 \to p_2\,\texttt{where}\,s_1, \mathcal{E}, s_2) &\equiv \texttt{\{?}\overline{\mathcal{E}}(p_1)\texttt{;where(}\overline{\mathcal{E}}(s_1)\texttt{);(}s_2\texttt{<+![])=>x;![}\overline{\mathcal{E}}(p_2)\texttt{|x]\}} \\
&\quad \mathrel{\texttt{<+}} s_2 \\
&\quad \text{with } \texttt{x} \text{ a fresh variable}
\end{aligned}$$

Thus, by using the empty list `[]` to model undefinedness, there is no more need for the $\perp$ value of Section 5.

Application Normal application of a rule produces the most recent term from the applicable rule instance. Thus, if the prioritized choice of the scope strategies rewrites to a list of terms, the first one is produced:

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{\texttt{<+}} \ldots \mathrel{\texttt{<+}} s_n \rangle\, t \implies \texttt{[}t_1\texttt{,}\ldots\texttt{,}t_m\texttt{]}\ (\Gamma', \mathcal{E}')\ (m > 0)}{\Gamma_{L(s_1|\ldots|s_n)}, \mathcal{E} \vdash \langle L \rangle\, t \implies t_1\ (\Gamma', \mathcal{E}')}$$

When the scope strategies produce the empty list, rewriting for this term was explicitly undefined. When application of the scope strategies fails, no matching rule was encountered. In both cases application fails:

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{\texttt{<+}} \ldots \mathrel{\texttt{<+}} s_n \rangle\, t \implies \texttt{[]}\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\ldots|s_n)}, \mathcal{E} \vdash \langle L \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')} \qquad \frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{\texttt{<+}} \ldots \mathrel{\texttt{<+}} s_n \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\ldots|s_n)}, \mathcal{E} \vdash \langle L \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}$$

Bagof Now, the interesting use of 'extended' rules is obtaining all possible rewrites for a term. For each dynamic rule L, there is a corresponding `bagof-`L rule that produces the list of all terms t_i to which a term t rewrites with L.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{<\!+} \ldots \mathrel{<\!+} s_n \rangle\, t \implies \texttt{[}t_1\texttt{,}\ldots\texttt{,}t_m\texttt{]}\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\ldots|s_n)}, \mathcal{E} \vdash \langle \texttt{bagof-}L \rangle\, t \implies \texttt{[}t_1\texttt{,}\ldots\texttt{,}t_m\texttt{]}\ (\Gamma', \mathcal{E}')}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \mathrel{<\!+} \ldots \mathrel{<\!+} s_n \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\ldots|s_n)}, \mathcal{E} \vdash \langle \texttt{bagof-}L \rangle\, t \implies \texttt{[]}\ (\Gamma', \mathcal{E}')}$$

Note that `bagof-`L always succeeds. If there are no defined rules, the result is just the empty list.

Once Another interesting use of extended dynamic rules, is the application of a dynamic rule *just once*. That is, by applying the rule it is 'consumed' and cannot be applied again. Thus, for each dynamic rule L, there is a corresponding strategy `once-`L, which applies the first available L rule, which is then undefined:

$$\frac{\Gamma, \mathcal{E} \vdash \langle \text{once}_{L_1}(s_1) \mathrel{<\!+} \ldots \mathrel{<\!+} \text{once}_{L_n}(s_n) \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\ldots|s_n)}, \mathcal{E} \vdash \langle \texttt{once-}L \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}')}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \texttt{[}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{]}\ (\Gamma', \mathcal{E}')\ (n > 0) \quad s' \equiv \texttt{\{?}t\texttt{;}\ \texttt{![}t_2\texttt{,}\ldots\texttt{,}t_n\texttt{]\}} \mathrel{<\!+} s}{\Gamma, \mathcal{E} \vdash \langle \text{once}_{L_i}(s) \rangle\, t \implies t_1\ (\Gamma'_{L(s_1|\ldots|s_{i-1}|s'|s_{i+1}|\ldots|s_m)}, \mathcal{E}')}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \text{once}_{L_i}(s) \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')} \qquad \frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \texttt{[]}\ (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \text{once}_{L_i}(s) \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}$$

These rules are somewhat simplified, since the undefinition in the second rule is done for the *term* t, rather than for the underlying *pattern* p. This is an anomaly of the representation of dynamic rules that we have chosen for presenting the semantics.

An example of the use of this feature is to ensure that a function is unfolded at most once, which is achieved by calling the unfolding rule as `once-UnfoldCall`. When this is successfully applied to a function call, it is automatically undefined. In Section 8 we present a larger application of dynamic rules to the implementation of function specialization, which uses this feature.

7. Intersection and Union of Rule Sets

In the previous sections, we have used dynamic rules in various program transformations, including ones such as constant propagation and common-subexpression elimination where they are used to model data flow facts. However, we have only considered straight line code in these transformations so far. That is, code without conditionals or loops. In straight line code there is a single execution path. A traversal strategy follows this path and maintains data flow information along the way in the form of dynamic rewrite rules. For example, the `PropConst` rule set represents all known propagation facts at the current point in the program at any time during the constant propagation transformation. Real programs do not have a single execution path. Rather, execution forks at conditionals and iterates at loops. Thus, to model data flow facts using dynamic rules, we need to account for these phenomena. To achieve this we need to fork dynamic rule sets for use in different branches, and join them again when branches meet. These operations are captured in several strategy combinators, which provide exactly the abstractions needed to define data flow transformations for programs with structured control flow in a concise manner. In this section we define the semantics of these fork-and-join combinators, and illustrate their use with two data flow transformations, constant propagation and dead code elimination.

7.1. Example: Flow-Sensitive Conditional Constant Propagation

Thus far, we have considered constant propagation with straight line code, possibly with local variables. Figure 18 presents a *complete* specification of intra-procedural, flow-sensitive constant propagation. A particular point of interest about this specification is that it combines analysis with transformation in the same traversal, similarly to the conditional constant propagation transformation of Wegman and Zadek [47] and the approach of Lerner et al. [19]. This combination is strictly more expressive than separate analysis and transformation phases, since transformation can influence the result of analysis. For example, the transformation in Figure 16 shows how the application of constant propagation (to determine that `x` is `10` at the condition), gives rise to elimination of code, reducing the `if-then-else` construct to the `then` branch. This elimination prevents considering the `else` branch, which would interfere with the knowledge that `x` is `10`. We achieve this with the generally applicable features of strategies and dynamic rules, which are not specific to constant propagation, or even data flow transformation.

The flow-sensitive constant propagation strategy is an extension of the constant propagation strategy for straight line code. To understand the extension, we reexamine the design of the original strategy. The basic idea of the transformation is that the set of `PropConst` rules reflects the constant propagation facts valid at the currently visited program point for all executions of the program. The rule set is modified during the traversal to maintain this invariant. Thus, an assignment statement *x* `:=` *e* either redefines `PropConst` to rewrite *x* to *e* in case *e* is a constant, or undefines the rule in case *e* is not a constant. Any other rules are not affected by the assignment and remain in the rule set. Variable declarations introduce local propagation rules (or locally undefined rules) shadowing any rules in outer scope.

For straight line code, there is only one possible execution. However, for code with control flow there are multiple execution paths, for all of which the propagation invariant needs to be maintained. The issues that need to be solved in the implementation of flow-sensitive constant propagation are illustrated by the example in Figure 17. (1) Facts that are valid before a conditional should be propagated into both branches of the conditional. For example, `a` and `z` in the second branch get the value that they have *before* the conditional. (2) Within a branch, facts can be propagated as is usual in a basic block. For

```
let var x := 0 var y := 0
 in x := 10;
    while A do
      (if x = 10
       then dosomething()
       else (dosomethingelse();
             x := x + 1));
   y := x
end
```

⇒

```
let var x := 0
    var y := 0
 in x := 10;
    while A do
      dosomething();
    y := 10
end
```

Figure 16. Combination of analysis and transformation.

```
let var x := 1 var y := z
    var z := 3 var a := 4
 in x := x + z;
    a := 5;
    if y then (y := y + 5;
               z := 8)
         else (x := a + 21;
               y := x + 1;
               z := a + z);

    b := a + z;
    z := z + x end
```

⇒

x	y	z	a	b
1	-	-	-	-
1	-	3	4	-
4	-	3	4	-
4	-	3	5	-
4	-	3	5	-
4	-	8	5	-
26	-	3	5	-
26	27	3	5	-
26	27	8	5	-
-	-	8	5	-
-	-	8	5	13
-	-	8	5	13

⇒

```
let var x := 1 var y := z
    var z := 3 var a := 4
 in x := 4;
    a := 5;
    if y then (y := y + 5;
               z := 8)
         else (x := 26;
               y := 27;
               z := 8);

    b := 13;
    z := 8 + x end
```

Figure 17. Example application of flow-sensitive constant propagation. The table shows the constant values *after* transforming the statement on the same line.

example, the value of `x` can be propagated within the second branch. (3) Facts that are guaranteed to be the same after execution of any of the branches can be propagated after the conditional, but facts that are inconsistent should not be propagated. For example, the value of `a` is unchanged by both branches, so it is the same after the conditional. While `z` is changed in both branches, its value is always the same, so it can be propagated. However, the value of `x` is changed in the second branch, therefore its value cannot be propagated afterwards.

Thus, to maintain the propagation invariant (1) transformation of the two branches of a conditional should start with the same set of dynamic rules as was valid before the conditional. Hence, after propagation in one branch, the rule set should be restored to what it was *before* the conditional in order to correctly propagate in the other branch. (2) Within a branch, transformation proceeds as usual. (3) After the conditional, transformation proceeds with those rules from the rule sets for the branches that are consistent. These requirements are implemented by the s_1 /L\ s_2 strategy combinator, which applies two strategies s_1 and s_2 sequentially to the subject term, but each starts with the same rule set for L and the resulting rule sets are intersected to form the new rule set for L afterwards. The /L\ combinator is language independent, that is, it has no knowledge of what are the 'branches' that should be treated separately. Instead this notion is expressed in the argument strategies. For example, the strategy

```
|[ if <id> then <prop-const> else <id> ]|
    /PropConst\ |[ if <id> then <id> else <prop-const> ]|
```

applies the strategy `prop-const` first to the `then`-branch of the conditional and then to the `else`-branch. Afterwards, the `PropConst` rule sets from the branches are intersected to maintain only those propagation rules that are the same after both branches.

In the case of loops, one traversal is not sufficient. The propagation rules applied to the loop body should be valid for *every* iteration of the loop. The rule set applicable before the loop is not necessarily valid in every iteration. We compute a rule set that *is* valid in all iterations by repeatedly applying the propagation to the loop, taking the intersection between the rule set $\vec{s}$ before and the rule set $\vec{s'}$ after the application, until a stable rule set is achieved, i.e., such that $\vec{s} \equiv \vec{s} \cap \vec{s'}$. At each iteration we transform the *original* loop, rather than accumulating the transformations from all iterations. This is necessary since

```
prop-const = PropConst <+ prop-const-assign <+ prop-const-vardec <+ prop-const-let
           <+ prop-const-if <+ prop-const-while <+ prop-const-for
           <+ (all(prop-const); try(EvalBinOp <+ EvalRelOp))

prop-const-vardec =
  |[ var x ta := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst+x :  |[ x ]| -> |[ e ]| )
                    else rules( PropConst+x :- |[ x ]| ) end
prop-const-assign =
  |[ x := <prop-const => e> ]|
  ; if <is-value> e then rules( PropConst.x :  |[ x ]| -> |[ e ]| )
                    else rules( PropConst.x :- |[ x ]| ) end
prop-const-let =
  |[ let <*id> in <*id> end ]|; {| PropConst : all(s) |}

prop-const-if =
  |[ if <prop-const> then <id> ]|
  ; (EvalIf; s <+ (|[ if <id> then <prop-const> ]| /PropConst\ id))

prop-const-if =
  |[ if <prop-const> then <id> else <id> ]|
  ; (EvalIf; s <+ (|[ if <id> then <prop-const> else <id> ]|
                    /PropConst\ |[ if <id> then <id> else <prop-const> ]|))
prop-const-while =
  |[ while <id> do <id> ]|
  ; (|[ while <prop-const> do <id> ]|; EvalWhile
     <+ /PropConst\* |[ while <prop-const> do <prop-const> ]|)

prop-const-for =
  |[ for <id> := <prop-const> to <prop-const> do <id> ]|
  ; (EvalFor <+ /PropConst\* |[ for <id> := <id> to <id> do <prop-const> ]|)
```

Figure 18. Intra-procedural flow-sensitive constant propagation.

```
let var w := 20 var x := 20 var y := 20 var z := 10
in while SomethingUnknown() do
    (if x = 20 then w := 20 else w := 10;
     if y = 20 then x := 20 else x := 10;
     if z = 20 then y := 20 else y := 10);
   w; x; y; z end
```

```
let var w := 20 var x := 20 var y := 20 var z := 10
in while SomethingUnknown() do
    (if x = 20 then w := 20 else w := 10;
     if y = 20 then x := 20 else x := 10;
     y := 10);
   w; x; y; 10 end
```

	w	x	y	z
	20	20	20	10
1	20	20	10	10
	20	20	-	10
2	20	-	10	10
	20	-	-	10
3	-	-	10	10
	-	-	-	10
4	-	-	10	10
	-	-	-	10

Figure 19. Example showing the need for multiple iterations. The table shows the values of the variables before the loop (first row) and at the end of each iteration, before and after computing the intersection with the previous rule set.

the transformations from all but the last iteration may apply rules that are not valid in all iterations, and may thus be incorrect. This process is illustrated in Figure 19. Note that the rule set $\vec{s'}$ after the loop need not be the same as the rule set before the loop. Thus, within the loop values can still be propagated as usual. In the example, variable `y` has the value `10` after every iteration. This cannot be propagated outside the loop since it has a different value before the loop, and there is no guarantee that the loop will execute at least once.

The iteration of dynamic rule propagation with intersection is expressed using the fixed point combinator `/`L`\*` s, which repeats the application of s until no more changes in the rule set for L occur, i.e., until a fixed point is reached. For example, in the constant propagation transformation, the strategy

```
/PropConst\* ⟦ while <prop-const> do <prop-const> ⟧)
```

expresses the fixed point iteration over a `while`-loop. The specification in Figure 18 uses these intersection combinators to express constant propagation over structured control flow constructs.

Finally, the transformation is enhanced with unreachable code elimination, which gives the effect of conditional constant propagation [47, 19] as illustrated in Figure 16. This is achieved by the following strategy expression

```
⟦ if <prop-const> then <id> else <id> ⟧
; (EvalIf; prop-const <+ (⟦ if <id> then <prop-const> else <id> ⟧
                           /PropConst\ ⟦ if <id> then <id> else <prop-const> ⟧))
```

which first applies `prop-const` to the conditional. Then it tries to apply `EvalIf`, which discards one of the branches if the condition is constant, after which that branch can be transformed as normal code with an application of `prop-const`. If the conditional cannot be reduced, the intersection is invoked, instead.

7.2. Semantics: Intersection of Dynamic Rules

The semantics of the join-and-fork combinators are straightforward. The argument strategies are applied sequentially to the subject term. That is, the second strategy is applied to the result of the first. However, each strategy application uses the original set of L rules, and afterwards the intersection of the resulting rule sets is taken.

$$\frac{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies t' \; (\Gamma'_{L(\vec{s'})}, \mathcal{E}') \quad \Gamma'_{L(\vec{s})}, \mathcal{E}' \vdash \langle s_2 \rangle\, t' \implies t'' \; (\Gamma''_{L(\vec{s''})}, \mathcal{E}'')}{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle s_1 \ \texttt{/}L\texttt{\\}\ s_2 \rangle\, t \implies t'' \; (\Gamma''_{L(\vec{s'} \cap \vec{s''})}, \mathcal{E}'')}$$

$$\frac{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies \uparrow (\Gamma'_{L(\vec{s'})}, \mathcal{E}')}{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle s_1 \ \texttt{/}L\texttt{\\}\ s_2 \rangle\, t \implies \uparrow (\Gamma'_{L(\vec{s})}, \mathcal{E}')}$$

$$\frac{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies t' \; (\Gamma'_{L(\vec{s'})}, \mathcal{E}') \quad \Gamma'_{L(\vec{s})}, \mathcal{E}' \vdash \langle s_2 \rangle\, t' \implies \uparrow (\Gamma''_{L(\vec{s''})}, \mathcal{E}'')}{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle s_1 \ \texttt{/}L\texttt{\\}\ s_2 \rangle\, t \implies \uparrow (\Gamma''_{L(\vec{s})}, \mathcal{E}'')}$$

The intersection of two rule sets is the point-wise intersection of the scope strategies, and the intersection of two scope strategies corresponds to the intersection of the resulting strategy application:

$$\vec{s} \cap \vec{s'} \equiv (s_1 \cap s'_1)|...|(s_n \cap s'_n) \qquad s_1 \cap s_2 \equiv \texttt{<isect>(<}s_1\texttt{>,<}s_2\texttt{>)}$$

where `isect` is a library strategy that computes the intersection of two lists, and `union` computes the union of two lists, removing duplicate elements.

The fixed point variants of the intersection operation repeats the application of a strategy until the rule set is stable. Thus, the first semantic rule defines that the result of the application of the fixed point operation produces the result of applying the transformation, if the intersection $\vec{s} \cap \vec{s'}$ of the L rule set $\vec{s}$ before and $\vec{s'}$ after application are equal to $\vec{s}$. The second rule expresses that if this is not the case, a recursive invocation of the fixed point operation should be performed.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies t'\ (\Gamma'_{L(\vec{s'})}, \mathcal{E}') \qquad \vec{s} \equiv \vec{s} \cap \vec{s'}}{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle \texttt{/}L\texttt{\\*}\ s_1 \rangle\, t \implies t'\ (\Gamma''_{L(\vec{s})}, \mathcal{E}')}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t \implies t'\ (\Gamma'_{L(\vec{s'})}, \mathcal{E}') \quad \vec{s''} \equiv \vec{s} \cap \vec{s'} \not\equiv \vec{s} \quad \Gamma_{L(\vec{s''})}, \mathcal{E}' \vdash \langle \texttt{/}L\texttt{\\*}\ s_1 \rangle\, t \implies t''\ (\Gamma'_{L(\vec{s'''})}, \mathcal{E}'')}{\Gamma_{L(\vec{s})}, \mathcal{E} \vdash \langle \texttt{/}L\texttt{\\*}\ s_1 \rangle\, t \implies t''\ (\Gamma''_{L(\vec{s'''})}, \mathcal{E}'')}$$

Note that in the second case the recursive invocation applies to the *original* subject term t. Only the result of the last iteration is produced as result of the transformation. Thus, the transformation is applied only after a stable rule set is obtained.

We have omitted the failure rules for the fixed point operator, which recover the rule set to its original state, just like the binary operator. In fact, the fork-and-join combinators are more general, since they allow a *list* of dynamic rules over which the fork-and-join operations are performed simultaneously. The extension of the semantics to these generalized combinators is straightforward. Furthermore, Stratego provides the *union* operators $\texttt{\\}L\texttt{/}$ and $\texttt{\\}L\texttt{/*}$, the semantics of which is entirely analogous to semantics of the intersection operators with intersection replaced by union. An example of their use follows below.

7.3. Example: Dead Code Elimination

Finally, as an example of the $\texttt{\\}L\texttt{/}$ combinators for taking the union of rule sets, we present the specification of dead code elimination, a transformation that removes assignments that compute a value that is not needed. The transformation is illustrated in Figure 20. It removes an assignment statement if its left-value variable is not needed on entry to the next statement. A variable is needed if it is used in a statement, which is needed itself and follows the assignment of the variable. Thus, the neededness of a variable requires a *backwards* flow analysis. By combining this analysis with the actual transformation, i.e., removing dead assignments, all dead code can be eliminated in a single traversal.

The specification in Figure 21 defines the strategy for dead code elimination for basic blocks with control flow, but without variable declarations. The central rule of the specification is `ElimAssign`, which replaces an assignment with the empty sequence, if the variable it assigns is not needed. Neededness is indicated by the `Needed` dynamic rule. If an assignment `x:=e` *is* needed (`ElimAssign` fails to apply), the dynamic rule is *undefined* by `dce-assign` for `x`, since all subsequent uses of the variable use the value computed by the assignment. By generically traversing needed expressions, all needed variables are encountered, which are then marked as needed by `VarNeeded`. The generic traversal is specialized for sequence, if, and while statements, which should be traversed in backwards control flow order. Thus, `dce-seq` uses the `reverse-filter` strategy to apply `dce` recursively to the statements in a sequence from last to first, and to remove those statements that have been reduced to the empty sequence `()`. The `dce-if` strategy first eliminates code in the branches of an `if-then-else` statement,

```
(x := foo(b);            {c,b}        (x := foo(b);
 y := bar(h);            {x,c}
 a := c + 23;            {x,c}        a := c + 23;
 if 4 > x then           {x,a}        if not(4> x) then
   (d := b + a;          {a}
    g := 4 + y)     ⇒    {a}     ⇒
 else
   (b := 2;              {x}
    a := y + 3;          {x}
    a := 4 + x);         {x}            a := 4 + x;
 print(a))               {a}          print(a))
```

Figure 20. Example of dead code elimination. The table indicates which variables are needed *on entry* of the statement on the same line.

```
dce = VarNeeded <+ ElimAssign <+ dce-assign <+ dce-seq <+ dce-if <+ dce-while <+ all(dce)

ElimAssign : |[ x := e ]| -> |[ () ]| where <not(Needed)> |[ x ]|

ElimIf : |[ if e then () else () ]|  -> |[ (e) ]|
ElimIf : |[ if e1 then e2 else () ]| -> |[ if e1 then e2  ]|
ElimIf : |[ if e1 then () else e2 ]| -> |[ if not(e1) then e2 ]|

VarNeeded = ?|[ x ]|; rules(Needed : |[ x ]|)
dce-assign = ?|[ x := e ]|; rules(Needed :- |[ x ]|); |[ <id> := <dce> ]|
dce-seq = |[ (<* reverse-filter(dce; not(?|[ () ]|)) >) ]|

dce-if =
  (|[ if <id> then <dce> else <id> ]| \Needed/ |[ if <id> then <id> else <dce> ]|)
  ; |[ if <dce> then <id> else <id> ]|
  ; try(ElimIf)

dce-while = |[ while <id> do <id> ]|; (\Needed/* |[ while <dce> do <dce> ]|)
```

Figure 21. Intra-procedural dead code elimination.

taking the union of the variables needed in both branches, and then transforms the condition. `ElimIf` simplifies a statement if one or both branches reduced to the empty sequence. Similarly, `dce-while` computes the fixed point of the variables needed in a `while` loop.

8. Example: Function Specialization

Next we present a somewhat larger example, illustrating nested dynamic rules, the use of `once-`L dynamic rule application, and the interaction between dynamic rules.

```
function mod(x : int, y : int) : int =
  x - ((x / y) * y)
function even(x : int) : int =
  mod(x, 2) = 0
function square(x : int) : int =
  x * x
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then
    square(power(x, n / 2))
  else x * power(x, n - 1)
function main() =
  printint(power(string2int(argv(1)),5))
```

⇒

```
function square(x: int): int = x * x
function a(x: int): int = x * b(x)
function b(x: int): int = square(c(x))
function c(x: int): int = square(e(x))
function e(x: int): int = x * f(x)
function f(x: int): int = 1
function main() =
  printint(a(string2int(argv(1))))
```

Figure 22. Specialization of `power(_,5)`.

Partial evaluation is a transformation that specializes a program to its static inputs [17]. One aspect of partial evaluation is *function specialization*, the generation of a derived function definition that is specialized to some values of its parameters. Partial evaluation can be considered as an extension of constant propagation to involve functions. The example in Figure 22 illustrates partial evaluation by the specialization of the `power` function to the constant value `5` as its second argument. A specialized function `a(x)` is generated that denotes `power(x,5)`. Propagating the constant `5` in the body of the specialized function gives rise to a call `power(x,4)`, which is itself specialized to `b(x)`. This process continues by specializing all function calls that have some constant values as arguments. Function calls for which all arguments are constants, are not specialized, but are unfolded. Since all argument values are available, these calls can be completely evaluated. For example, as part of partial evaluation of the specialized body of `power(x,5)`, the call `even(n)` is instantiated to `even(5)`. By unfolding this call and all nested calls, the value of `even(5)` is computed during specialization, and can thus be used to evaluate the `if-then-else`.

To achieve partial evaluation, the specification in Figure 23 is a combination of constant propagation for local propagation of constant values, call unfolding and function specialization. The strategy has basically the same structure as the constant propagation strategy in Figure 18, with a few extra cases. The strategies `prop-const-vardec`, `prop-const-assign`, `prop-const-if`, `prop-const-while`, and `prop-const-for` are reused from Figure 18

The major part of the transformation is the strategy `declare-fundec`, which defines for each function definition three dynamic rules. First, `Specialization` is the rule used to collect specializations for the function, which is initialized to produce the function itself as specialization. Then, `UnfoldCall` is the familiar unfolding rule that replaces a call with an instantiation of the function body, but only if all arguments are constants. Finally, `SpecializeCall` generates a new function with the body of the original function with the constant actual parameters bound to the corresponding formal parameters. Before examining the latter rule in more detail, note how these rules are used in the overall strategy. The `prop-const-let` strategy transforms `let` bindings by first declaring the unfolding and specialization rules by means of a map over all declarations in `declare`, which calls `declare-fundec` for each function definition. Next, it performs constant propagation in the `let` body (recursive call to `prop-const`), which may give rise to function specializations. Finally, all function specializations are added to the `let`, by rewriting each function definition to the list of its specializations in `specialize`.

```
prop-const = PropConst <+ prop-const-assign <+ prop-const-let <+ prop-const-fundec
  <+ prop-const-funarg <+ prop-const-if <+ prop-const-while <+ prop-const-for
  <+ all(prop-const); try(EvalBinOp <+ EvalRelOp <+ unfold-call <+ SpecializeCall)

unfold-call = UnfoldCall; exprename; prop-const

prop-const-let =
  |[ let <*id> in <*id> end ]|
  ; {| PropConst, UnfoldCall, SpecializeCall, Specialization :
      |[ let <*declare> in <*prop-const> end ]|; |[ let <*specialize> in <*id> end ]| |}
  ; try( \ |[ let d* in i end ]| -> |[ i ]| \ )

declare        = map(prop-const-vardec + |[ <fd*:map(declare-fundec)> ]|)
specialize     = map(try(|[ <fd*:mapconcat(specialize-fun)> ]|))
specialize-fun = ![<once-Specialization; prop-const>|<specialize-fun>] <+ ![]

declare-fundec =
  ?|[ function f(x1*) ta = e1 ]|;
  rules(
    Specialization+f :
      |[ function f(x1*) ta = e2 ]| -> |[ function f(x1*) ta = e1 ]|

    UnfoldCall :
      |[ f(a*) ]| -> |[ let d* in e1 end ]|
      where <map(is-value)> a* // only unfold if all args static
      ; <zip(\ (FArg|[ x ta ]|, e) -> |[ var x ta := e ]|\ )> (x1*, a*) => d*

    SpecializeCall :
      |[ f(a1*) ]| -> |[ g(a2*) ]|
      where <split-static-dynamic-args> (x1*, a1*) => (d*, (x2*, a2*))
          ; new => g
          ; rules(
              Specialization.f :+ |[function f(x1*) ta = e2]| ->
                                  |[function g(x2*) ta = let d* in e1 end]|
            )
  )

split-static-dynamic-args =
  zip; partition(bind-arg-value); (not([]), unzip)

bind-arg-value : (FArg|[ x ta ]|, e) -> |[ var x ta := e ]| where <is-value> e

prop-const-fundec =
  ?|[ function f(x*) ta = e ]|; {| PropConst : all(prop-const) |}

prop-const-funarg =
  ?FArg|[ x ta ]|; rules( PropConst+x :- |[ x ]| )
```

Figure 23. A simple online partial evaluator with function specialization.

The interesting rule in the specification is `SpecializeCall`. It transforms a call *f* (*a1**) into a call to a new function *g*. The new function is a specialization of function *f* to the constant valued arguments of the call. As a side effect, the definition of the new function *g* is declared as a specialization of *f* by means of the nested dynamic rule definition for `Specialization`. By *extending* the `Specialization` rule, any previous specializations are preserved. That is, `bagof-Specialization` produces all specializations for *f*. The definition of the specialized function *g* has the following form:

```
|[ function g(x2*) ta = let d* in e1 end ]|
```

It is a combination of the original function definition for *f* and the actual parameters *a1** of the call. The `split-static-dynamic-args` strategy is used to partition the formal and actual parameters (*x1**, *a1**) into (1) a list of local variable declarations *d**, assigning the constant valued arguments to the corresponding formal parameters (produced by `bind-arg-value`); and (2) a list of formal parameters *x2** and actual parameters *a2**, corresponding to the non-constant valued arguments. Thus the specialized function *g* has the remaining parameters *x2** as formal parameters and uses the variable bindings *d** to specialize the original body *e1*. Finally, *g* (*a2**) is a call to the newly created specialized function with as parameters the non-static parameters of *f* (*a1**).

As an example, consider the specialization of the call `power(y,5)`. Splitting the argument list produces the variable declaration `|[ var n : int := 5 ]|`, and the remaining non-constant argument `y` with the formal parameter `|[ x : int ]|`. Arbitrarily choosing `a` as the name for the specialized function, `SpecializeCall` produces the specialized call `a(y)`, and extends the definition of the dynamic rule `Specialization` as follows:

```
Specialization :+ |[ function power(x : int, n : int) : int = e2 ]| ->
  |[ function a(x : int) : int =
      let var n : int := 5
       in if n = 0 then 1
          else if even(n) then square(power(x, n / 2)) else x * power(x, n - 1)
      end ]|
```

Note that this specialized function has not yet been transformed by `prop-const` itself. That is, the specialization is just a copy of the original function definition with a local variable declaration, binding the formal parameter `n` to the actual parameter `5`. No specializations have been generated for the recursive call in this specialization.

Thus, the result of transforming the body of a `let` is that all *direct* function calls have either been unfolded or specialized. The specializations have not been specialized themselves. This is done while replacing the original function definition with its specializations in the `specialize` strategy. Since the partial evaluation of the specialized functions may give rise to further specializations, it is not sufficient to apply `bagof-Specialization` once. To obtain all specializations of a function, an interaction between constant propagation and retrieving specializations is needed. For this purpose, we use `once-Specialization`, which produces one right-hand side of the dynamic rule, discarding it at the same time. Thus, `once-Specialization` produces one previously defined specialization for the function under consideration, and then deletes that specialization from the rule set. The auxiliary strategy `specialize-fun` builds a list of specializations by calling `once-Specialization` to obtain the next specialized function, partially evaluating it by calling `prop-const`, and then recursively calling `specialize-fun` to obtain further specialized functions. For example, partially evaluating the specialization of `power(_,5)` above produces the following definition

```
|[ function a(x : int) : int = let var n : int := 5 in x * b(x) end ]|
```

and gives rise to `b(x)`, a call to the specialization of `power(x,4)`. Specialization of `b` gives rise to the new function `c`, and so on. (Note that the variable declaration for `n` is dead and can be removed, which is not done by the specializer in Figure 23, but by a separate dead code removal transformation.)

To summarize, the specification in Figure 23 defines an online partial evaluator that evaluates calls with all constant arguments and specializes calls with at least one constant argument (and at least one non-constant argument). The complete specification is a bit over 100 lines including the propagation strategy for control flow, but excluding standard evaluation rules. This specification lacks memoization to prevent re-specialization of function calls with the same constant arguments. This requires another dynamic rule. This approach to partial evaluation has been incorporated in a compiler for Octave [25], a high-level language, primarily intended for numerical computations [13]. We have also developed an offline partial evaluator with separate binding-time annotation using these techniques [45].

9. Related Work

Scoped dynamic rewrite rules are a novel extension of strategic rewriting. The rewriting strategy controls not only the application of static rewrite rules, but also controls the definition, scope, and application of dynamic rewrite rules. This language extension is inspired by and/or related to previous work in several areas. Firstly, in compilers, program analysis, and program optimizers all kinds of specialized data structures, e.g. symbol tables, are used to store information about the program. Secondly, several programming languages support implicit parameters and dynamic scoping of names. Lastly, many systems provide run-time code generation, usually for adding code that is dependent on information not available until runtime.

9.1. Data Structures in Program Optimization and Compilation

Symbol Tables In program transformation systems all kinds of data structures are used for storing context-sensitive information. In particular, *symbol tables* are widely used to associate symbols in a program with information, e.g. the type of the symbol. Symbol tables are often implemented by using a hash table for efficient lookup. Since symbol tables are concerned with names, they have to handle the scoping rules of the object programming language. For example, the symbol table implementation used in Appel's Tiger compiler [2] remembers the state of a hash table in a `beginScope` and restores this information in the `endScope`. Dynamic rewrite rules lift this functionality for handling scopes to the language level by adding special purpose language constructs for scoping to the meta-language. The implementation of scopes in Stratego is efficient and general enough to handle a wide variety of scopes in programming languages.

Bit Vectors In data flow analysis *bit vectors* are used to represent facts about entities in a program. For example, in calculating the definitions that reach program points, for every program point a bit vector is constructed. The bit vector of a program point contains a character for every definition, so the length of the bit vector is the number of definitions in a procedure. To collect the required information, the control flow of the program is simulated until a fixed point is reached. Such a bit vector encoding of information is extremely compact and intersection and union of bit vectors is very efficient. Dynamic rewrite rules also have fixed point operations, union, intersection, and they can be used as predicates. However, compared to data flow analysis with bit vectors this abstraction clearly comes at a cost.

Value Numbering *Value numbering* is used in a wide range of program optimizations. Initially, it was a method used for common subexpression elimination and constant folding in basic blocks [10]. In value numbering every expression in a basic block is assigned a unique number. The goal of value numbering is to discover redundancy, which is determined by letting the value of two expressions be equal if the expressions are equal. This number is stored in a hash table, of which the keys are based on the structure of the expression. For example, for a binary operator the key is determined by its operands and the operator. Dynamic rewrite rules use a comparable method for efficient access to the rewrite rules that have been defined for a term. The rewrite rules are stored in a hash table, of which the key is based on the dynamic part of the left-hand side of the rewrite rule.

9.2. Language Independent Traversals

Language independent traversals [38] allow the implementation of certain program transformations using traversals that are independent of a specific object language. These traversals have been implemented for reoccurring program transformations, such as collecting free variables, renaming bound variables, syntactic unification of terms with object variables, and substitution of expressions for object variables. The language independent traversals are implemented using the generic traversal operators of Stratego [44] and are parameterized with strategies for handling language specific issues, such as the representation of variables and the constructs for variable binding.

Language independent traversals for program transformation often implement context-sensitive transformations. Before the introduction of scoped dynamic rewrite rules, such context-sensitive issues had to be handled by threading an environment with context information through the traversal, or alternatively, by controlling the traversal from the topmost location where the required information is available. The second option leads to repeated traversals over the abstract syntax tree of the object program and in both cases the traversals are more difficult to understand and maintain.

Scoped dynamic rewrite rules are a useful language extension for making these language independent traversals more concise. First, the threading of context-sensitive information can now be delegated to dynamic rules, which are implicitly passed to strategies. Second, scope can be controlled in a declarative way with the scoping facilities of dynamic rules. Third, program analysis and program transformation can now be combined in a single traversal. The resulting traversals are much more attractive. For example, information on constructs having local scope do not need to leave this scope. This solves the name conflicts that have to be solved in case a separate global table of information is constructed first.

The traversal functions in ASF+SDF [7] can be used to thread an environment through a traversal. ASF+SDF provides three kinds of traversal functions: transformers, accumulators and accumulating transformers. For each of these kinds, there is a fixed set of traversal functions: `bottom-up` and `top-down`, which can be configured to `break` or `continue` after a successful application. The accumulators and accumulating transformers are used to accumulate information, in case of an accumulating transformer during a transformation. The accumulated value is updated on every application of the accumulator, and the next application will then use this new value. In Stratego, context-sensitive information is represented in dynamic rewrite rules, which makes the threading of context-sensitive information more natural in the paradigm of strategic rewriting. Alternative rewritings are represented by defining several applicable rewriting rules, as opposed to expensive construction and threading of lists. Like the application of static rules is controlled by a strategy, so is the definition, scope, and application of dynamic rewrite rules under full control of user-definable traversal strategies.

9.3. Runtime Extension of Logic Programs

Dynamic rewrite rules are closely related to the extra-logical operators `assert` and `retract` in Prolog. These operators allow dynamic inspection and modification of the clause database. The predicate `assert(X)` adds the clause `X` to the rule database. If the clause is added to an existing predicate with `assert`, then the location of the new clause is implementation dependent. The `asserta` and `assertz` operators provide more control over this by letting the new clause be the first, respectively the last, clause of the predicate. The predicate `retract(X)` removes the first clause that unifies with `X`. Most Prolog implementations provide a `retractall(X)` predicate, which removes *all* clauses that unify with `X`.

Backtracking In case of backtracking a clause added by an assert is not removed from the database. The clause must be retracted by hand if this is required. Similarly, Stratego's dynamic rewrite rules are not removed on backtracking either. However, the scoping of dynamic rewrite rules makes it possible to restrict the live range of dynamic rules. This is not completely comparable to retracting clauses on backtracking, since the dynamic rules generated in this scope are removed in case of failure *and* success. Currently, there is no language construct for removing rules in case of failure, although this can easily be implemented with the dynamic rule API. Similar to backtracking over an assert predicate, clauses are not re-added to the Prolog database if backtracking occurs over a retract predicate. In Stratego, undefined rules become visible again when the scope of undefinition is left. Also in this case, there is no language construct for making the dynamic rules visible again in case of failure only.

Live Range In Prolog the live range of clauses is controlled by the retract predicate. The retract predicate requires an argument that is used for specifying the clause that must be removed. On the other hand, a dynamic rule scope automatically removes all rules that have been defined in this scope. Therefore, the dynamic rules are not removed based on their input or output. This is natural in the application domain of program transformations, since scopes typically correspond with scopes in the object language. If this scope is left, then the information collected in this scope is no longer applicable. If a scope is left, then dynamic rules that were defined outside the scope are preserved and might become visible when the scope is left.

Retract removes clauses based on their goal. In contrast, dynamic rules are undefined by their left-hand side. Although clauses and rewrite rules are different constructs, the goal of a clause is more comparable to the right-hand side of a rewrite rule. Clauses can be made undefined by a pattern of their output, whereas rewrite rules are undefined by a pattern of their input. Dynamic rules are only undefined for the current (or a given) scope. If this scope is left, then the dynamic rules that have been defined outside this scope will become visible again. Again, this is natural in the domain of program transformation.

In Prolog the location of a clause in an existing predicate can be controlled by `asserta` and `assertz`. Stratego's dynamic rewrite rules organize the defined dynamic rewrite rules in scopes, which might be labeled. This scope label can be used to define a dynamic rule in an outer scope instead of the current one. These features provide fine-grained control over the location of a newly defined rule.

Alternative Results In Prolog the ordinary collectors (e.g. `bagof`, `setof`, `findall`) can be used to get all alternative solutions. Backtracking for clauses generated at run-time by `assert` is not different from ordinary backtracking. In Stratego's dynamic rules there is a distinction between static and dynamic rewrite rules. Static rules with the same name cannot be applied to produce all alternative solutions, since they are combined by the *local-choice* operator ($<+$), which commits the choice for a strategy argument if it succeeds. In contrast, all possible results of applying a dynamic rule can be produced by means the special purpose strategy `bagof-`L, which immediately returns all results. It might be interesting to consider more programmer control over the backtracking behaviour.

9.4. Dynamic Binding

Dynamic rewrite rules are related to dynamic scoping and binding. In this section we will first review how dynamic rules implement dynamic binding, and proceed discussing differences with implementations of dynamic binding in other programming languages.

The terms dynamic scoping and dynamic binding are often used as synonyms, although they refer to different, yet related, concepts. A *binding* is an association between a name and a value. The *scope* of a name concerns its visibility, that is, the part of the program where the name can be used. *Extent* refers to the lifetime of a binding. In *static* or *lexical* scoping, the scope of a name is determined by the lexical structure of the program. A binding is available from the start of the definition construct to its end. In other words, names are evaluated in the definition environment. The name might be shadowed by a nested definition of a name, but this does not mean that the binding is really not available, since it should still be available if the control flow turns to a part of the program where the name is not shadowed by a nested binding. In *dynamic* scoping a name is visible in all execution paths that include the definition of the name. In other words, dynamic variables are evaluated in the environment of their application. Since these execution paths cannot be determined at compile-time, the binding of names must be determined at runtime, whereas the binding can be determined at compile-time if lexical scoping is used.

In most programming languages variables are lexically scoped, but dynamic binding is also used by default or at least available in many programming languages. The best-known examples are the various Lisp dialects (e.g. McCarthy's Lisp, Common Lisp, GNU Emacs Lisp), but TEX, shell scripting, and XSLT 2.0 implement dynamic binding as well. More recently several papers have reintroduced dynamic scope as a feature in strongly-typed general purpose languages, namely statically-typed Haskell-like languages with Hindley-Milner type inference [21], and Java-like languages [16].

Dynamic Rewrite Rules The scope of the name of a dynamic rewrite rule is *global*. In other words, a dynamic rewrite rule can be applied anywhere in a program. If no dynamic rules for this rule have been defined (or those that have been defined, have been undefined), then the application just fails. This is perfectly acceptable, since failure of strategy application is used for control flow in Stratego. More comparable to dynamic variables are dynamic rule scopes. The binding of dynamic variables is based on the execution path. Similarly, the dynamic rule scopes of the execution path determine the dynamic rules that can be applied. The dynamic rule scopes thus correspond to dynamic variables whose value is a set of dynamic rules. A novel aspect of dynamic rewrite rules is that a scope itself does not automatically hide the rewrite rules defined in outer scopes. Furthermore, dynamic rewrite rules defined in the current scope, but at execution paths that have already returned, are still available.

In contrast to the dynamic binding of dynamic rules, the context variables of a dynamic rewrite rule have lexical scope. That is, their value at rule definition time is stored as a closure in the dynamically defined rewrite rule. Currently, strategy definitions in the context of a dynamic rule definition are not part of its closure.

Lisp Dialects Dynamic binding first appeared as a more or less unintended feature of Lisp 1.0. Lisp 1.0 had one kind of variable, which was dynamically scoped. The unexpected behaviour of dynamic binding was soon reported, and was at first was thought to be a bug by McCarthy [23]. The analysis of this problem led to the identification of the *funarg* problem and the first implementation of closures, a representation of a function and the lexical environment in which the function is defined. In Lisp dialects that have been developed later, dynamic scoping was no longer default for variables. However, most dialects include an explicit notion of dynamically scoped variables. In Common Lisp, variables can be declared to be `special`, which indicates that they are dynamically scoped. Usually, top-level variables (globals) are special and local variables are lexical. Dynamically scoped variables can be used to give the ‘global’ variable a new value temporarily, since a new a re-definition of the global variable with a `let` only influences the execution paths that contain the `let`.

Alternatively, several Scheme implementations provide the `fluid-let` binding construct to control dynamic binding [15]. Fluid bindings are somewhat similar to special variables in Common Lisp. However, the fluid-let of Scheme does not determine the scope of a variable. Instead, it temporarily assigns a different value to variables that have already been defined in some outer scope. This binding is stored in a per-thread fluid binding association list, which is consulted when a non-local variable is evaluated. Therefore, the fluid-let can be described as a thread-local (inherited by child threads) scoped assignment construct.

Dynamically scoped variables are very useful for passing values to parts of a program to configure its behaviour, without passing loads of parameters to every function that might possibly be on an execution path to this part of the program. These variables allow values to be passed in an *implicit* way. Decades after the introduction of dynamic and lexical scoping in Lisp and Scheme, there has recently been more interest in adding dynamically scoped variables to current programming languages.

Implicit Parameters in Haskell Lewis et al. [21] have proposed implicit parameters for functional programming languages like Haskell. These implicit parameters have been implemented as extensions of Haskell in Hugs [20] and GHC. Implicit parameters can be deeply embedded in a functional definition and can be bound at some outer level without having to pass the value explicitly through all the intermediate function calls. Rather, the need for passing an implicit parameter is inferred statically. Adding implicit parameters to a statically-typed language with type inference introduces some problems, which results in some limitations. First, the approach does not allow function arguments that take implicitly parameterized arguments. Second, implicit parameters must be monomorphic. Third, implicit parameters are not allowed in the context of a class or instance declaration. Besides these restrictions that are based on static type system issues, the implicit parameters are just that: parameters. Implicit information can be only be passed to callees. There are no *implicit results*, which would allow the passing of results to callers in an implicit way.

Dynamic Variables in Imperative Languages Hanson and Proebsting [16] reintroduced dynamic scope as a feature in imperative languages. They propose a minimalistic language extension for dynamic variables, which are to be used sparingly. The use of a dynamic variable refers to the most recent setting of a dynamic variable with the same name. Dynamic variables could replace thread local variables (for example available in Java as `java.lang.ThreadLocal`), which allow separate threads to have their own, independent variables. Indeed, the dynamic variable proposal is based on a data structure for storing dynamic variables on the stack, which automatically makes the dynamic variables local to a thread. Note that there is thus a slight difference with the fluid-let in Scheme, where child threads inherit dynamic variables from their parent. Java provides a separate subclass of `ThreadLocal` called `InheritableThreadLocal`. For this class, the initial values of the thread local variables are inherited from its parent. The child thread still gets it own copy of the variable: it can set the value of the thread local variable, but the value of the parent's variable will not be modified.

XSLT's Tunnel Variables Most recently, Schadow proposed dynamically scoped variables for XSLT [33]. This proposal has been accepted for XSLT 2.0 [18], where they are now called tunnel parameters. A parameter of a template can be defined as a tunnel parameter. Tunnel parameters are recursively and implicitly passed to all templates that are called. All tunnel parameters are passed through a built-in template rule. The tunnel parameters are very similar to the dynamic variables that have been discussed before. XSLT does not allow side effects to variables and there is no way to return dynamically scoped variables implicitly. In short, the design of tunnel parameters is not very surprising, yet, it is interesting to see dynamically scoped variables live again in a pure functional language that is widely used in practice.

It would be interesting to develop a system comparable to dynamic rules for XSLT. This would then be a facility for the run-time definition of templates, where variables from the definition context can be used in the dynamically defined templates.

Domain-Specific Languages Domain-specific languages such as TEX and shell scripting make use of dynamically scoped variables that allow for easy redefinition of behaviour; for example, in TEX configuration of a document, style can be influenced by redefining macros representing parameters of the style. In shell scripting, environment variables are implicitly passed to all parts of the shell script.

9.5. Fresh O'Caml and FreshML

FreshML and Fresh O'Caml [34, 32] lift the problem of manipulating names and binding constructs to the meta language. This makes meta-programming tasks that have to consider free and bound variables much easier, since the meta language guarantees that these constructs are manipulated in a proper way. Variable binding in object languages is the focus of FreshML. Therefore, it does not provide any further facilities to deal with context-sensitive information in program transformations. Furthermore, FreshML restricts the possible ways of binding to just lexical (static) binding. Object languages with dynamic binding cannot be transformed with the variable binding facilities of FreshML.

10. Conclusion

In this paper we have presented an extension of term rewriting with the run-time definition of context-dependent rewrite rules. Dynamic rules can be used as part of the global tree traversal, thus not increasing complexity by performing additional traversals. The extension is not limited to some specific form of program representation such as control flow graphs, but can be applied in the transformation of arbitrary abstract syntax trees. The implementation of dynamic rules in Stratego has been designed to achieve the best possible efficiency of all operations such that transformations can scale to large programs.

Scoped dynamic rewrite rules solve (many of) the limitations caused by the context-free nature of rewrite rules, strengthening the separation of rules and strategies, and supporting concise and elegant specification of program transformations. This has been illustrated in this paper by the specification of several transformations, i.e., bound variable renaming, function inlining, constant propagation, common-subexpression elimination, dead function elimination, and online partial evaluation. The techniques are equally well applicable to many other program transformations.

Acknowledgments We thank Patricia Johann, Oege de Moor, and Ganesh Sittampalam for discussions on the subject of this paper. In particular, Ganesh Sittampalam's remarks during the Stratego User Days 2004 triggered the development of scope labels and change sets. The anonymous referees provided valuable feedback for improving the presentation of this article.

A. Free Variables

This appendix contains a definition of the free pattern variables in a strategy expression in core Stratego.

$$
\begin{aligned}
\text{freevars}(str) &\equiv \{\} \\
\text{freevars}(i) &\equiv \{\} \\
\text{freevars}(r) &\equiv \{\} \\
\text{freevars}(x) &\equiv \{x\} \\
\text{freevars}(c(p_1, ..., p_n)) &\equiv \text{freevars}(p_1) \cup ... \cup \text{freevars}(p_n) \\
\\
\text{freevars}(\texttt{?}p) &\equiv \text{freevars}(p) \\
\text{freevars}(\texttt{!}p) &\equiv \text{freevars}(p) \\
\text{freevars}(\texttt{\{}x_1, ..., x_n \texttt{:} s\texttt{\}}) &\equiv \text{freevars}(s) / \{x_1, ..., x_n\} \\
\text{freevars}(\texttt{let } d_1, ..., d_n \texttt{ in } s \texttt{ end}) &\equiv \text{freevars}(d_1) \cup ... \cup \text{freevars}(d_n) \cup \text{freevars}(s) \\
\text{freevars}(f\texttt{(}s_1\texttt{,...,}s_n\texttt{|}p_1\texttt{,...,}p_m\texttt{)}) &\equiv \text{freevars}(s_1) \cup ... \cup \text{freevars}(s_n) \\
&\quad \cup \text{freevars}(p_1) \cup ... \cup \text{freevars}(p_m) \\
\text{freevars}(\texttt{id}) &\equiv \{\} \\
\text{freevars}(\texttt{fail}) &\equiv \{\} \\
\text{freevars}(s_1 \texttt{ ; } s_2) &\equiv \text{freevars}(s_1) \cup \text{freevars}(s_2) \\
\text{freevars}(s_1 \texttt{ < } s_2 \texttt{ + } s_3) &\equiv \text{freevars}(s_1) \cup \text{freevars}(s_2) \cup \text{freevars}(s_3) \\
\text{freevars}(c(s_1, ..., s_n)) &\equiv \text{freevars}(s_1) \cup ... \cup \text{freevars}(s_n) \\
\text{freevars}(\texttt{all(}s\texttt{)}) &\equiv \text{freevars}(s) \\
\text{freevars}(\texttt{one(}s\texttt{)}) &\equiv \text{freevars}(s)
\end{aligned}
$$

$$
\text{freevars}(f\texttt{(}sd_1, ..., sd_n \texttt{ | } vd_1, ..., vd_m\texttt{) = } s) \equiv \text{freevars}(s) / \{vd_1, ..., vd_m\}
$$

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[4] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science,* Asilomar, Pacific Grove, CA, September 1996. Elsevier Science Publishers.

[5] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transforming system: Simplifying the development of numerical software. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 353–372. Birkhuser, 1997.

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

[7] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, 2003.

[8] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical Report UU-CS-2005-005, Department of Information and Computing Sciences, Utrecht University, 2005. Extended version of this article with appendices on implementation and performance.

[9] M. Bravenboer and E. Visser. Rewriting strategies for instruction selection. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 237–251, Copenhagen, Denmark, July 2002. Springer-Verlag.

[10] J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, April 1970.

[11] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 178–190, Portland, Oregon, USA, January 2002. ACM Press.

[12] E. Dolstra and E. Visser. Building interpreters with rewriting strategies. In M. G. J. van den Brand and R. Lmmel, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers.

[13] J. Eaton. Octave. `http://www.octave.org/`.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[15] C. Hanson. MIT/GNU Scheme reference. `http://www.gnu.org/software/mit-scheme/documentation/scheme.html`, 2003.

[16] D. R. Hanson and T. A. Proebsting. Dynamic variables. In *Programming Language Design and Implementation (PLDI'01),* Snowbird, UT, USA, June 2001. ACM.

[17] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[18] M. Kay. *XSL Transformations (XSLT) Version 2.0, W3C Working Draft 12 November 2003*. World Wide Web Consortium, November 2003. `http://www.w3.org/TR/xslt20/`.

[19] S. Lerner, D. Grove, and C. Chambers. Combining dataflow analyses and transformations. In *SIGPLAN Symposium on Principles of Programming Languages (POPL 2002)*, pages 270–282, Portland, Oregon, January 2002.

[20] J. Lewis. The hugs 98 user's guide: Implicit parameters. `http://cvs.haskell.org/Hugs/pages/users_guide/implicit-parameters.html`, 2003.

[21] J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 108–118. ACM, January 2000.

[22] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[23] J. McCarthy. History of LISP. In R. L. Wexelblat, editor, *History of Programming Languages: Proceedings of the ACM SIGPLAN Conference*, pages 173–197. Academic Press, June 1–3 1978.

[24] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[25] K. Olmos and R. Vermaas. The Stratego Octave Compiler. `http://www.octave-compiler.org`, 2003–2005.

[26] K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

[27] K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer-Verlag, April 2005.

[28] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[29] R. Paige. Future directions in program transformations. *Computing Surveys*, 28A(4), December 1996.

[30] A. Pettorossi and M. Proietti. Future directions in program transformation. *ACM Computing Surveys*, 28(4es):171–es, December 1996. Position Statement at the Workshop on Strategic Directions in Computing Research. MIT, Cambridge, MA, USA, June 14-15, 1996.

[31] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.

[32] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000.

[33] G. Schadow. Request for dynamically scoped variables in XSLT. `http://lists.w3.org/Archives/Public/xsl-editors/2002JanMar/0002.html`, 2002.

[34] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, Aug. 2003.

[35] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[36] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[37] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[38] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

[39] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[40] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[41] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[42] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Reduction Strategies in Rewriting and Programming special issue.

[43] E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, September 1998. Elsevier Science Publishers.

[44] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[45] E. Visser, M. Bravenboer, and K. Olmos. Implementation of partial evaluation in Stratego/XT. Presentation at Functional Refactoring Workshop in Canterbury, Kent, UK, February 9 2004. `http://www.cs.uu.nl/wiki/Visser/ResearchTalks`.

[46] E. Visser et al. Stratego/XT. `http://www.stratego-language.org`, 1999–2005.

[47] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:181–210, April 1991.