

Program Transformation with Scoped Dynamic Rewrite Rules

Martin Bravenboer, Arthur van Dam, Karina Olmos and Eelco Visser

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, their application to transformation problems, and their implementation. 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 [28, 27, 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 [26, 13]. While transformations can be achieved by manual manipulation of programs, in general, the aim of program transformation is to increase programmer productivity by *automating* 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 [33] is an attractive formalism for expressing basic program transformations. A rewrite rule $p_1 \rightarrow p_2$ 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

$$[[i + j]] \rightarrow [[k]] \text{ where } \langle \text{add} \rangle(i, j) \Rightarrow k$$

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

$$[[\text{if } 0 \text{ then } e1 \text{ else } e2]] \rightarrow [[e2]]$$

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 strategy 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 [6] 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 [5] provides non-deterministic sequential strategies. Stratego [42, 35, 39] provides generic basic traversal operators that can be used to compose a wide range of generic tree traversal schemas. See [37, 40] 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 only access 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, constant and copy propagation, 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 [4, 34, 42]. Contextual rules combine the context and the local transformation by using a local traversal that applies a rule, reusing information from the context. For instance, the following contextual rule defines the inlining of a 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 it. When such a rule is applied as part of a complete traversal over a program, 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 [8] 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 [35] 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. 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. For instance, an inlining algorithm needs to maintain a table mapping function names to their formal parameters and bodies, which needs to be entered at the definition site and retrieved at call sites. 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.

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. 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 fixpoint 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, and have been used for the substitution in bound variable renaming [36]; the call replacement in function inlining [36]; the removal of declarations in dead code elimination [36]; 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 [25] or the mapping from expressions to variables in common subexpression elimination; the specialization of functions in partial evaluation; the representation of type assignments in type checking; and the memoization of instruction selections in code generation [9]. The language constructs have been designed carefully to provide a natural fit in the rewriting setting, while at same time making 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. The use of dynamic rules enables the combination of program analysis and program transformation in a single traversal which makes it possible to achieve better results, since the effects of transformations can be used in analysis immediately.

With respect to the earlier paper [36] 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. This paper presents a formal operational semantics of Stratego with dynamic rules and describes its implementation. 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 a formal syntax definition 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 of the sections 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 and function specialization. Section 7 closes the description of features with operations for computing with sets of dynamic rules. In particular, operations for intersection and union of rule sets to implement data-flow transformations. The operations are illustrated with constant propagation and dead code elimination in the presence of structured control-flow constructs.

Section 8 gives an overview of the issues in the implementation of dynamic rules. Section 9 discusses some benchmarks of the dynamic rule implementation. Sections 10, 11 and 12 discuss other applications, related and future work. Section 13 concludes.

The mechanism of dynamic rules as described in this paper reflects the implementation of dynamic rules in Stratego/XT version 0.10 available from www.stratego-language.org.

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 [3], 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 are 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. The Boolean operators `&` and `|` are defined as short-circuit operators in terms of `if-then-else`. The Boolean negation operator is represented as call of the built-in function `not`. Control flow is determined using the `if-then-else`, `if-then`, `while` and `for` constructs. As an inheritance from functional languages, there is no 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 ($e_1; \dots; e_n$) 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 following program illustrates the essential aspects of the language. The program reads integers until a negative one is encountered and then prints the number of integers read and their maximum,

```
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("number of values: "); printint(count);
            print("maximum: "); printint(maxnbr)
        end
```

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.

¹In 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 : $\text{var } x \text{ ta} := e$	variable declaration
		FunDecs : fd^*	function definitions
		TypeDecs : td^*	type definitions
fd	$::=$	FunDec : $\text{function } f(farg^*) \text{ ta} = e$	function definition
$farg$	$::=$	FArg : $x \text{ ta}$	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 : $e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \dots$	arithmetic
		RelOp : $e_1 < e_2 \mid e_1 > e_2 \mid e_1 = e_2 \mid \dots$	relational
		And, Or : $e_1 \& e_2 \mid e_1 \mid e_2$	Boolean
		Assign : $x := e$	assignment
		Call : $f(e^*)$	function call
		Seq : (e^*)	sequence
		If : $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$	conditional
		IfThen : $\text{if } e_1 \text{ then } e_2$	conditional
		While : $\text{while } e_1 \text{ do } e_2$	while loop
		For : $\text{for } x := e_1 \text{ to } e_2 \text{ do } e_3$	for loop
		Let : $\text{let } d^* \text{ in } e^* \text{ end}$	let binding

Figure 1. Abstract syntax of Tiger (incomplete).

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 $(x_1 := e_1; \dots; x_n := e_n)$. 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 // return value
end
```

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 automatically transformed to a form in which expressions are side-effect free and separate from statements by lifting expressions with side-effects to the statement level. Thus the assignment expression above would 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 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, 3], 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. We will use terms of the following form:

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

Terms of this form correspond to *ATerms* [7], the data exchange format used to exchange program representations between Stratego programs. Stratego is indifferent to the source of the terms it transforms. These can be produced by a parser derived from an SDF2 syntax definition, but may be produced by any program, including for instance a YACC parser or another Stratego program.

When considering the semantics of Stratego we will use the observation that terms according to the grammar above are isomorphic to terms according to the following grammar:

$t ::= c(t_1, \dots, t_n)$	n -ary tuple $n \geq 0$
$c ::= \text{identifier} \mid str \mid i \mid \text{Tuple} \mid \text{Cons} \mid \text{Nil}$	constructors

That is, a first-order term is essentially a constructor c applied to a, possibly empty, list of terms t_1, \dots, t_n . The other cases can be reduced to such constructor applications as indicated by the equivalences in the first grammar.

To illustrate how programs correspond to terms consider the constructors assigned to productions in the BNF grammar for Tiger in Figure 1. Examples of terms over the Tiger signature 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.

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 $L: p_1 \rightarrow p_2$ where s , consisting of a label L , term patterns p_1 and p_2 , and condition s . An unconditional rule has the form $L: p_1 \rightarrow p_2$. A *term pattern* is a term with variables, that is, a pattern is either a variable or the application $c(p_1, \dots, p_n)$ of an n -ary constructor c to term patterns p_i . Examples of rewrite rules are the following constant folding and desugaring rules:

```
EvalBinOp : BinOp(PLUS, e, Int("0")) -> e
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 $L: p_1 \rightarrow p_2$ where s applies to a term t if the left-hand side pattern p_1 matches t with substitution σ and the condition s holds under 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("0")))
```

As should be apparent from this description, a rewrite rule has only access 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 that rules 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 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. Thus the `EvalBinOp` rule above becomes

```
EvalBinOp : [| e + 0 |] -> [| e |]
```

where the phrase `|[e + 0]|` denotes the term `BinOp(PLUS, e, Int("0"))`, i.e. the corresponding abstract syntax term according to the grammar. Stratego supports the specification of transformation systems with *concrete object syntax* for arbitrary object languages [38]. In the rest of this paper we will

```

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)

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 ]]
ElimWhile : [[ while e do () ]] -> [[ e ]]
ElimFor   : [[ for x := e1 to e2 do () ]] -> [[ (e1; e2) ]]

```

Figure 2. Some rewrite rules for Tiger expressions.

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 $c(t_1, \dots, t_n)$. In example programs, pattern variables will be typeset in italics. Figure 2 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 BNF 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 strategy separate and avoiding the introduction of new constructors or rules. The strategies in Stratego were inspired by the strategy language of Elan [5], 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, 41, 42]. This 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.

P	$::= d^*$	program (list of definitions)
d	$::= dsig = s$	strategy definition
$dsig$	$::= f(sd_1, \dots, sd_n \mid vd_1, \dots, vd_m)$	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
	$c(p_1, \dots, p_n)$	constructor application
s	$::= ?p$	match
	$!p$	build
	$\{x_1, \dots, x_n : s\}$	term variable scope
	$\text{let } d_1, \dots, d_n \text{ in } s \text{ end}$	local definitions
	$f(s_1, \dots, s_n \mid p_1, \dots, p_m)$	call
	id	identity
	fail	failure
	$s_1 ; s_2$	sequential composition
	$s_1 < s_2 + s_3$	guarded deterministic choice
	$c(s_1, \dots, s_n)$	congruence traversal
	$tr(s)$	traversal to subterms
tr	$::= \text{all} \mid \text{one} \mid \text{some}$	traversal operator
f	$::= \text{identifier}$	strategy operator
x	$::= \text{identifier}$	term variable
c	$::= \text{identifier}$	constructor
tp	$::= \dots$	type (undefined)

Figure 3. Syntax of core Stratego.

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 3 using BNF. The core language is enriched with several syntactic abstractions which are presented in Figure 4. The extension of Stratego with dynamic rules is introduced in the next section. The syntax definition in Figure 3 also introduces the *meta-variables* that will be used in the operational semantics. For instance, s denotes a strategy and p a term pattern.

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,

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

Figure 4. Extensions (sugar) of the syntax of core Stratego.

but the state may be changed. Rewrite rules are just strategies that apply transformations to the roots of terms. 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 [41, 42], and integrates environments and state in the rules. The semantics should be understood as a description of the behaviour of programs, not (necessarily) a model of the implementation. The semantics of the core constructs is defined in terms of assertions of the form

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

which state that the application of strategy s to subject term t in the context of 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

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

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 indicate that the result may be either a term or failure. States Γ are used to model dynamic rules; their structure will be described in the next section. Environments \mathcal{E} model

pattern variable bindings. 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.

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 the 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 term pattern p , and $!p$ denotes building an instantiation of the term pattern p . This decomposition allows a host of language features to be defined from first principles. For instance, an unconditional rewrite rule $p_1 \rightarrow p_2$ corresponds to the sequential composition $?p_1; !p_2$. The sequential composition of strategies will be defined formally below, but comes down to first applying the first strategy and then the second. To define match and build precisely we need the notion of environment.

Environments A *variable binding environment* $\mathcal{E} \equiv [x_1 \mapsto \bar{t}_1, \dots, x_n \mapsto \bar{t}_n]$ 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

$$[x_1 \mapsto \bar{t}_1, \dots, x_n \mapsto \bar{t}_n](x) = \begin{cases} \bar{t}_i & \text{if } x_i \equiv x \text{ and } \forall j < i : x_j \neq x \\ \uparrow & \text{if } \forall j \leq n : x_j \neq x \end{cases}$$

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

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

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* $\bar{\mathcal{E}}(p)$ of a term pattern p with an environment \mathcal{E} yields the term pattern obtained by replacing each variable x in p with $\bar{\mathcal{E}}(x)$. The loose instantiation $\bar{\mathcal{E}}(s)$ of a strategy expression s consists in replacing each term pattern p in s with $\bar{\mathcal{E}}(p)$.

An environment \mathcal{E}' is a *refinement* of 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 $\mathcal{E} = [x_1 \mapsto \bar{t}_1, \dots, x_n \mapsto \bar{t}_n]$ then $\mathcal{E}' = [x_1 \mapsto \bar{t}'_1, \dots, x_n \mapsto \bar{t}'_n]$ 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 if $\mathcal{E}'(x) = \mathcal{E}(x)$ if x does not occur 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 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.

$$\frac{\mathcal{E}' \supseteq_p \mathcal{E} \wedge \mathcal{E}'(p) \equiv t}{\Gamma, \mathcal{E} \vdash \langle ?p \rangle t \Longrightarrow t(\Gamma, \mathcal{E}')} \quad \frac{\neg \exists \mathcal{E}' \supseteq_p \mathcal{E} \wedge \mathcal{E}'(p) \equiv t}{\Gamma, \mathcal{E} \vdash \langle ?p \rangle t \Longrightarrow \uparrow(\Gamma, \mathcal{E}'')}$$

As example of the match operation consider the following: applying $?[(e1 \mid e2) \& e3]$ against the term $[(a < b \mid c) \& d > 10]$ succeeds since the environment $[e1 \mapsto [a < b], e2 \mapsto [c], e3 \mapsto [d > 10]]$ 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:

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

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

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 $\{x_1, \dots, x_n : s\}$ scope construct. That is, the binding to a variable x_i outside the scope $\{x_1, \dots, x_n : s\}$ 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:

$$\frac{\Gamma, [x_1 \mapsto \uparrow, \dots, x_n \mapsto \uparrow] \mathcal{E} \vdash \langle s \rangle t \Longrightarrow \bar{t}'(\Gamma', [x_1 \mapsto \bar{t}_1, \dots, x_n \mapsto \bar{t}_n] \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \{x_1, \dots, x_n : s\} \rangle t \Longrightarrow \bar{t}'(\Gamma', \mathcal{E}'')}$$

That is, the strategy s is evaluated in an extended environment in which the local variables are unbound initially. After application of the strategy the bindings are removed from the environment. The convenience construct $\{s\}$ implicitly makes all free variables in s local.

$$\{s\} \equiv \{x_1, \dots, x_n : s\} \quad \text{if } \{x_1, \dots, x_n\} \equiv \text{freevars}(s)$$

Example: In the following strategy expression, the scope of the variables $e1$, $e2$, and $e3$ is restricted to the match-build sequence:

$$\{e1, e2, e3 : ?[(e1 \mid e2) \& e3]; ![(e1 \& e3 \mid e2 \& e3)]\}$$

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 that distributes $\&$ over \mid 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 at the end of this section.

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:

$$\Gamma, \mathcal{E} \vdash \langle \text{id} \rangle t \Longrightarrow t (\Gamma, \mathcal{E}) \quad \Gamma, \mathcal{E} \vdash \langle \text{fail} \rangle t \Longrightarrow \uparrow (\Gamma, \mathcal{E})$$

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.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle t \Longrightarrow t' (\Gamma', \mathcal{E}') \quad \Gamma', \mathcal{E}' \vdash \langle s_2 \rangle t' \Longrightarrow \bar{t}'' (\Gamma'', \mathcal{E}'')}{\Gamma, \mathcal{E} \vdash \langle s_1 ; s_2 \rangle t \Longrightarrow \bar{t}'' (\Gamma'', \mathcal{E}'')} \quad \frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle s_1 ; s_2 \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}'')}$$

From this definition it is clear that in the sequential composition $?p_1 ; !p_2$ 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.

$$\text{id} ; s \equiv s \quad s ; \text{id} \equiv s \quad \text{fail} ; s \equiv \text{fail} \quad s ; \text{fail} \not\equiv \text{fail}$$

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

Deterministic Choice The *left deterministic choice* $s_1 \leftarrow s_2$ 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.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle t \Longrightarrow t' (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle s_1 \leftarrow s_2 \rangle t \Longrightarrow t' (\Gamma', \mathcal{E}')} \quad \frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}') \quad \Gamma', \mathcal{E}' \vdash \langle s_2 \rangle t \Longrightarrow \bar{t}'' (\Gamma'', \mathcal{E}'')}{\Gamma, \mathcal{E} \vdash \langle s_1 \leftarrow s_2 \rangle t \Longrightarrow \bar{t}'' (\Gamma'', \mathcal{E}'')}$$

The left choice is thus a limited backtracking combinator; the first argument strategy can be a complex strategy that may fail at some point, in which case it backtracks to the second argument strategy. However, if the first argument strategy succeeds, the choice is committed and no backtracking to the second strategy is possible. Note that on failure of s_1 the environment is reset to the environment before the choice, but that the state is maintained. Identity is a left zero, but not a right unit or zero for left choice, failure is a left and right unit for left choice, i.e.

$$\text{id} \leftarrow s \equiv \text{id} \quad s \leftarrow \text{id} \not\equiv s \quad \text{fail} \leftarrow s \equiv s \quad s \leftarrow \text{fail} \equiv s$$

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. This is captured in the definition

$\text{try}(s) = s \triangleleft \text{id}$

which defines the combinator try that tries to apply a strategy, but falls back to id when that fails.

Note that the following distribution laws do not hold. The first inequality is fundamental since the incorporation of s_3 within the choice may lead to failure of the left branch while s_1 by itself might succeed and commit the choice. The second inequality is in fact an equality in case s_1 has no effects on the state Γ .

$$(s_1 \triangleleft s_2); s_3 \not\equiv (s_1; s_3) \triangleleft (s_2; s_3) \quad s_1; (s_2 \triangleleft s_3) \not\equiv (s_1; s_2) \triangleleft (s_1; s_3)$$

The choice combinator is typically used as prioritized choice between rewrite rules; if s_1 to s_n are rewrite rules, then the strategy $s_1 \triangleleft \dots \triangleleft s_n$ tries each of the rules one by one from left to right, stopping as soon as one of the rules has succeeded.

Non-Deterministic Choice The *non-deterministic choice* $s_1 + s_2$ of strategies s_1 and s_2 attempts to apply either s_1 or s_2 to the subject term, but in an unspecified order. It succeeds if either s_1 or s_2 succeeds, and fails otherwise. The non-deterministic choice in the order of evaluation is made *at compile-time* by applying one of the following equivalences:

$$s_1 + s_2 \equiv s_1 \triangleleft s_2 \quad \text{or} \quad s_1 + s_2 \equiv s_2 \triangleleft s_1$$

The non-deterministic choice combinator is used in Stratego to indicate that the order of applying the branches does not matter. This is for instance the case in the choice between a set of mutually exclusive rewrite rules. The combinator is also needed to model the modular composition of rewrite rules in which no order of evaluation is specified.

Guarded Choice The left choice combinator is in fact a special case of the *guarded left deterministic choice* $s_1 < s_2 + s_3$, which decides on basis of the success or failure of s_1 which branch to take; it is defined as:

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

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

This definition gives rise to the following laws. The last one indicates the relation with left-choice:

$$\text{id} < s_1 + s_2 \equiv s_1 \quad \text{fail} < s_1 + s_2 \equiv s_2 \quad s_1 < \text{fail} + s_2 \equiv s_1; \text{fail} \quad s_1 < \text{id} + s_2 \equiv s_1 \triangleleft s_2$$

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. 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. Formally, we have

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle t \Longrightarrow t' (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \text{where}(s) \rangle t \Longrightarrow t (\Gamma', \mathcal{E}')} \quad \frac{\Gamma, \mathcal{E} \vdash \langle s \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \text{where}(s) \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E})}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle t \Longrightarrow t' (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \text{not}(s) \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E})} \quad \frac{\Gamma, \mathcal{E} \vdash \langle s \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \text{not}(s) \rangle t \Longrightarrow t (\Gamma', \mathcal{E})}$$

In fact these combinators can be expressed in terms of the combinators above as expressed by the following laws:

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

The following laws hold for these combinators:

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

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

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

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 $\langle s \rangle p$ and $s \Rightarrow p$, respectively, which are defined as

$$\langle s \rangle p \equiv !p; s \quad s \Rightarrow p \equiv s; ?p$$

A typical example is the combined use of these constructs in a strategy expression $\langle \text{add} \rangle (i, j) \Rightarrow k$, which applies the strategy `add` to the pair (i, j) and matches the result against the pattern `k`.

In the core language, term patterns are composed solely as terms with variables. In some cases it is useful to apply a strategy to one of the subterms of a pattern. For instance, in the build strategy $! \text{Int}(\langle \text{add} \rangle (i, j))$, an `Int` term is constructed with as argument the sum of `i` and `j`. This syntax is implemented by lifting such strategy applications out of the term pattern to the surrounding strategy level as defined by the following equations:

$$!p_1[\langle s \rangle p_2] \equiv \{x: \text{where}(\langle s \rangle p_2 \Rightarrow x); !p_1[x]\} \quad ?p_1[\langle s \rangle p_2] \equiv \{x: ?p_1[x]; !x; \langle s \rangle p_2\}$$

Another use of applying a strategy within a pattern is to ‘wrap’ a pattern around a term, or to ‘project’ a sub-term from a term. This can be achieved using the $\langle s \rangle$ application in a term, defined as

$$!p[\langle s \rangle] \equiv \{x: \text{where}(s \Rightarrow x); !p[x]\} \quad ?p[\langle s \rangle] \equiv \{x: ?p[x]; \langle s \rangle x\}$$

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 $f(f_1, \dots, f_n | x_1, \dots, x_m) = s$ introduces a new strategy combinator f with body s parameterized with strategy variables f_1, \dots, f_n and term variables x_1, \dots, x_m . An application $f(s_1, \dots, s_n | p_1, \dots, p_m)$ 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 just passed around unchanged in all other rules it is omitted there.

$$\begin{array}{c} D_{\text{fresh}}(f) = (f(f_1, \dots, f_n | x_1, \dots, x_m) = s) \\ \mathcal{E}' \equiv [x_1 \mapsto \mathcal{E}(p_1) \dots x_m \mapsto \mathcal{E}(p_m)] \\ \frac{[f_1=s_1 \dots f_n=s_n]D, \Gamma, \mathcal{E}' \mathcal{E} \vdash \langle s \rangle t \Longrightarrow \bar{t}' (\Gamma', \mathcal{E}'')}{D, \Gamma, \mathcal{E} \vdash \langle f(s_1, \dots, s_n | p_1, \dots, p_m) \rangle t \Longrightarrow \bar{t}' (\Gamma', \mathcal{E}'')} \end{array}$$

Here $D_{\text{fresh}}(f)$ produces a fresh instance of the strategy definition of f , i.e. using unique new names 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.

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 empty the parentheses may be omitted. Thus we have the following equivalences for definitions and calls:

$$\begin{array}{l} f(f_1, \dots, f_n) = s \equiv f(f_1, \dots, f_n |) = s \quad f = s \equiv f(|) = s \\ f(s_1, \dots, s_n) \equiv f(s_1, \dots, s_n |) \quad f \equiv f(|) \end{array}$$

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

$$\begin{array}{l} f(f_1, \dots, f_n | x_1, \dots, x_m) = s \equiv f(f_1, \dots, f_n | x_1, \dots, x_m) = \{y_1, \dots, y_j : s\} \\ \text{with } y_1, \dots, y_j = \text{freevars}(s) / \{x_1, \dots, x_m\} \end{array}$$

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`:

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

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

Recursive Closure The *recursive closure* $\text{rec } f(s)$ is sugar for a local recursive definition:

$$\text{rec } f(s) \equiv \text{let } f = s \text{ in } f \text{ end}$$

The construct can be useful in strategy expressions to abbreviate a recursive invocation. For example, by writing $\text{repeat}(s) = \text{rec } x(s; x)$ instead of $\text{repeat}(s) = s; \text{repeat}(s)$.

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:

$$dsig : p_1 \rightarrow p_2 \text{ where } s \equiv dsig = \{x_1, \dots, x_n : ?p_1; \text{where}(s); !p_2\}$$

$$\text{with } x_1, \dots, x_j = \text{freevars}(p_1, p_2, s) / \text{vars}(dsig)$$

An unconditional rule corresponds to a conditional rule with the identity strategy as condition:

$$dsig : p_1 \rightarrow p_2 \equiv dsig : p_1 \rightarrow p_2 \text{ where id}$$

Example: the rewrite rule

$$\text{DefAnd} : \llbracket e_1 \ \& \ e_2 \rrbracket \rightarrow \llbracket \text{if } e_1 \text{ then } e_2 \text{ else } 0 \rrbracket$$

corresponds to the strategy definition

$$\text{DefAnd} = \{e_1, e_2 : ?\llbracket e_1 \ \& \ e_2 \rrbracket; \text{where}(\text{id}); !\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } 0 \rrbracket\}$$

Multiple Definitions It is sometimes useful to give a set of rules the same name. For instance the EvalBinOp rules in Figure 2 define multiple rules for evaluating binary arithmetic expressions. A set of definitions with the same signature are reduced to a single definition consisting of the non-deterministic choice of the bodies of the definitions, i.e.

$$dsig = s_1 \ \dots \ dsig = s_n \equiv dsig = (s_1 + \dots + s_n)$$

For example, the EvalBinOp rules from Figure 2 reduce to a single definition

$$\text{EvalBinOp} = \{e : ?\llbracket e + 0 \rrbracket; !\llbracket e \rrbracket\}$$

$$+ \{e, j, k : ?\llbracket i + j \rrbracket; \text{where}(\langle \text{add} \rangle(i, j) \Rightarrow k); !\llbracket k \rrbracket\}$$

$$+ \{i, j, k : ?\llbracket i * j \rrbracket; \text{where}(\langle \text{mul} \rangle(i, j) \Rightarrow k); !\llbracket k \rrbracket\}$$

Note here that the use of the non-deterministic choice operator + entails that there is no order in which the alternative rules are tried in this composition.

3.5. Generic 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 at an internal site of a term (i.e. to a subterm), it is necessary to traverse the term. Stratego defines several primitive 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 $c(s_1, \dots, s_n)$ is the strategy that applies only to terms of the form $c(t_1, \dots, t_n)$, and works by applying each strategy s_i to the corresponding term t_i . For example, the congruence $\text{Let}(s_1, s_2)$ transforms terms of the form $\text{Let}(t_1, t_2)$ into $\text{Let}(t'_1, t'_2)$, 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 $c(s_1, \dots, s_n)$ to $c(t_1, \dots, t_n)$ also fails.

$$\frac{\Gamma, \mathcal{E} \vdash \langle s_1 \rangle t_1 \Longrightarrow t'_1(\Gamma_1, \mathcal{E}_1) \quad \dots \quad \Gamma_{n-1}, \mathcal{E}_{n-1} \vdash \langle s_n \rangle t_n \Longrightarrow t'_n(\Gamma_n, \mathcal{E}_n)}{\Gamma, \mathcal{E} \vdash \langle c(s_1, \dots, s_n) \rangle c(t_1, \dots, t_n) \Longrightarrow c(t'_1, \dots, t'_n)(\Gamma_n, \mathcal{E}_n)}$$

$$\frac{c \neq c'}{\Gamma, \mathcal{E} \vdash \langle c(s_1, \dots, s_n) \rangle c'(t_1, \dots, t_n) \Longrightarrow \uparrow(\Gamma_n, \mathcal{E}_n)}$$

$$\Gamma, \mathcal{E} \vdash \langle s_1 \rangle t_1 \Longrightarrow t'_1(\Gamma_1, \mathcal{E}_1)$$

$$\dots$$

$$\Gamma_{i-2}, \mathcal{E}_{i-2} \vdash \langle s_{i-1} \rangle t_{i-1} \Longrightarrow t'_{i-1}(\Gamma_{i-1}, \mathcal{E}_{i-1})$$

$$\Gamma_{i-1}, \mathcal{E}_{i-1} \vdash \langle s_i \rangle t_i \Longrightarrow \uparrow(\Gamma_i, \mathcal{E}_i)$$

$$\frac{}{\Gamma, \mathcal{E} \vdash \langle c(s_1, \dots, s_n) \rangle c(t_1, \dots, t_n) \Longrightarrow \uparrow(\Gamma_i, \mathcal{E}_i)}$$

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)] <+ ?[_ | <filter(s)>]
```

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 $\langle s \rangle$ to embed a strategy within a Tiger expression. Thus, for example, the strategy expression

```
[[if <s> then <id> else <id>]]
```

is a congruence over the `if-then-else` construct and applies strategy `s` to the condition and the identity strategy to the branches.

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 combinator `all(s)` applies `s` to all direct subterms t_i of a constructor application $c(t_1, \dots, t_n)$. It succeeds if and only if all applications to the direct subterms succeed. The resulting term is the

constructor application $c(t'_1, \dots, t'_n)$ where the t'_i are the results obtained by applying s to the terms t_i .

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle t_1 \Longrightarrow t'_1 (\Gamma_1, \mathcal{E}_1) \quad \dots \quad \Gamma_{n-1}, \mathcal{E}_{n-1} \vdash \langle s \rangle t_n \Longrightarrow t'_n (\Gamma_n, \mathcal{E}_n)}{\Gamma, \mathcal{E} \vdash \langle \mathbf{all}(s) \rangle c(t_1, \dots, t_n) \Longrightarrow c(t'_1, \dots, t'_n) (\Gamma_n, \mathcal{E}_n)}$$

$$\Gamma, \mathcal{E} \vdash \langle s \rangle t_1 \Longrightarrow t'_1 (\Gamma_1, \mathcal{E}_1)$$

$$\dots$$

$$\Gamma_{i-2}, \mathcal{E}_{i-2} \vdash \langle s \rangle t_{i-1} \Longrightarrow t'_{i-1} (\Gamma_{i-1}, \mathcal{E}_{i-1})$$

$$\Gamma_{i-1}, \mathcal{E}_{i-1} \vdash \langle s \rangle t_i \Longrightarrow \uparrow (\Gamma_i, \mathcal{E}_i)$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle \mathbf{all}(s) \rangle c(t_1, \dots, t_n) \Longrightarrow \uparrow (\Gamma_i, \mathcal{E}_i)}$$

Note that \mathbf{id} is a zero for \mathbf{all} , that $\mathbf{all}(s)$ is the identity on constants (constructor applications without children), and that $\mathbf{all}(\mathbf{fail})$ only succeeds only on constants, i.e.

$$\mathbf{all}(\mathbf{id}) \equiv \mathbf{id} \quad \langle \mathbf{all}(s) \rangle c \equiv \mathbf{id} \quad \langle \mathbf{all}(\mathbf{fail}) \rangle c \equiv \mathbf{id}$$

$$\langle \mathbf{all}(\mathbf{fail}) \rangle c(t_1, \dots, t_n) \equiv \mathbf{fail}(\text{if } n > 0)$$

Example: Traversal Strategies Many different traversals can be composed using the \mathbf{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 $s; \mathbf{all}(\mathbf{topdown}(s))$ 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 $\mathbf{topdown}$ captures the generic notion of a traversal that visits each sub-term in pre-order. $\mathbf{bottomup}$ defines a post-order traversal. \mathbf{alltd} applies a $\mathbf{topdown}$ traversal that stops as the transformation it is applying succeeds. \mathbf{downup} applies a transformation pre-order and post-order. $\mathbf{innermost}$ is a fixpoint traversal that applies a transformation exhaustively starting with innermost terms.

The examples in the next sections use traversals that are partly specific for the abstract syntax of Tiger, and partly generic, i.e. for those parts of the Tiger syntax where uniform behaviour is required.

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 in lets with only a single binding.

```

desugar = topdown(repeat(Desugar + LetSplit + EmptyLet + ElimSingletonTuple)
; try(DeQuote))

```

The $\mathbf{desugar}$ strategy performs a $\mathbf{topdown}$ traversal applying a number of rules along the way. (Some of these rules are shown in Figure 2.) The $\mathbf{Desugar}$ strategy that it calls is actually a large composition of rewrite rules:

```
Desugar = DefUmin + DefTimes + DefPower + DefDiv + DefPlus + DefMinus
  + DefEq + DefNeq + DefGt + DefLt + DefGeq + DefLeq
  + DefSeq1 + SumSplit + DefSum + ProdSplit + DefProd + 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 strategies is actually a choice between several rewrite rules.

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

$$\frac{\begin{array}{c} \Gamma, \mathcal{E} \vdash \langle s \rangle t_1 \implies \uparrow (\Gamma_1, \mathcal{E}_1) \\ \dots \\ \Gamma_{i-1}, \mathcal{E} \vdash \langle s \rangle t_{i-1} \implies \uparrow (\Gamma_{i-1}, \mathcal{E}_{i-1}) \\ \Gamma_{i-1}, \mathcal{E} \vdash \langle s \rangle t_i \implies t'_i (\Gamma_i, \mathcal{E}_i) \end{array}}{\Gamma, \mathcal{E} \vdash \langle \text{one}(s) \rangle c(t_1, \dots, t_n) \implies c(t_1, \dots, t_{i-1}, t'_i, t_{i+1}, \dots, t_n) (\Gamma_i, \mathcal{E}_i)}$$

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

Some example traversal strategies composed with this combinator are:

```
oncetd(s) = s <+ one(oncetd(s))
oncebu(s) = one(x) <+ oncebu(s)
spinetd(s) = s; try(one(spinetd(s)))
```

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

```
is-subterm = ?(x, e); where(<oncetd(?x)> e)
```

It matches against a pair of terms and traverses the second term to find an occurrence of the first by means of the match ?x in which the x is bound by the previous match.

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 [34]. 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 <onced({e3: ?[[f(e3)]]; ![[let var x := e3 in e1 end]]})> e2 => e2'

```

The `onced` 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 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. Although 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, and (2) that only *one* function at a time is inlined.

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 `InlineFun` contextual rule is a rewrite rule (a sequence of a match and a build) with some of its pattern variables *bound in the context* instead of being bound by the application of the match as is the case with ‘normal’ 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 contextual rules is that a dynamic rule is *named* like a normal labeled rewrite rule and can be referred to *outside the lexical 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 5. In this and the next three sections we will define the semantics of these language constructs and illustrate 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 [24]. We will use this transformation as a running example, and gradually extend it to

s	$::=$	<code>rules (drd₁ ... drd_n)</code>	dynamic rule definition
		<code>{f₁, ..., f_n : s }</code>	dynamic rule scope
		<code>s₁ / f₁, ..., f_n \ s₂</code>	fork and intersect
		<code>s₁ \ f₁, ..., f_n / s₂</code>	fork and union
		<code>/ f₁, ..., f_n * s</code>	fix and intersect
		<code>\ f₁, ..., f_n /* s</code>	fix and union
drd	$::=$	<code>drsig : p₁ -> p₂ (where s)?</code>	dynamic rule definition
		<code>drsig :+ p₁ -> p₂ (where s)?</code>	dynamic rule extension
		<code>drsig : p</code>	dynamic identity rule definition
		<code>drsig :- p</code>	dynamic rule undefinition
		<code>drsig'</code>	label current scope
$drsig$	$::=$	<code>sig</code>	relative to current scope
		<code>sig.p</code>	relative to labeled scope
$drsig'$	$::=$	<code>sig+p</code>	relative to current scope and label current scope

Figure 5. Extension of syntax of Stratego with dynamic rules.

cover the whole language. We start with constant propagation in basic blocks, which should achieve a transformation such as the following:

<pre>(b := 1; c := b + 3; b := b + 1; b := z + b; a := b + c)</pre>	⇒	<pre>(b := 1; c := 4; b := 2; b := z + 2; a := b + 4)</pre>
---	---	---

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 2. 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.

The `prop-const` strategy in Figure 6 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 when an assignment is encountered, a congruence strategy is used to transform the right-hand side expression by a recursive invocation of the `prop-const` strategy, but leaves the lvalue untouched (to prevent the replacement of the variable

```

prop-const =
  PropConst
  ⇐ [[ <id> := <prop-const> ]]; AssignPropConst
  ⇐ all(prop-const); try(EvalBinOp ⇐ EvalRelOp)

AssignPropConst =
  ?[[ x := e ]]
  ; if <is-value> e then
    rules( PropConst : [[ x ]] -> [[ e ]] )
  else
    rules( PropConst :- [[ x ]] )
  end

```

Figure 6. Strategy for constant propagation in basic blocks.

in the lvalue). After transforming the expression, a propagation rule for the assignment is defined—to be further 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 `AssignPropConst` strategy, which *defines* a `PropConst` rule for each particular assignment it encounters. 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

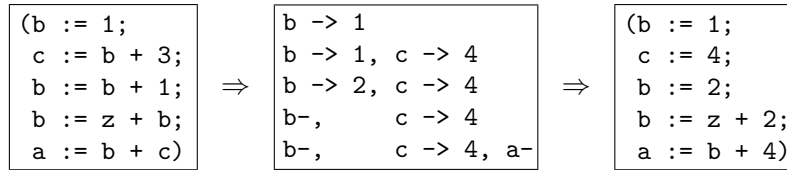
```
rules( PropConst : [[ x ]] -> [[ e ]] )
```

The `rules` construct introduces a list of rules which inherit variable bindings for all variables occurring in the context. If the expression 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.



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. For the semantic description of the constructs that is of no concern at this point, however. In Section 8 we return to implementation issues.

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

$$\frac{\Gamma, \emptyset \vdash \langle s \rangle t \Longrightarrow \bar{t}' (\Gamma', \mathcal{E}')}{\Gamma_{L(s)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow \bar{t}' (\Gamma', \mathcal{E})}$$

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

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.

$$\text{rules}(L_1 : r_1 \dots L_n : r_n) \equiv \text{rules}(L_1 : r_1) ; \dots ; \text{rules}(L_n : r_n)$$

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

$$\frac{s'_1 \equiv \{\mathcal{E}(\text{? } p_1 ; \text{where}(s_2) ; ! p_2)\} \Leftarrow s_1}{\Gamma_{L(s_1)}, \mathcal{E} \vdash \langle \text{rules}(L : p_1 \rightarrow p_2 \text{ where } s_2) \rangle t \Longrightarrow t (\Gamma_{L(s'_1)}, \mathcal{E})}$$

The new strategy is a prioritized choice that first tries to apply the new rule, only 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, as long as their left-hand sides do not overlap; (2) a definition of a rule with the same left-hand side as an earlier defined rule, shadows (or redefines) that earlier rule

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.

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

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 nice 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 $\{L : s\}$, 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.

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, which can then assume that two occurrences of the same identifier denote the same variable. Bound variable renaming is necessary in transformations to avoid variable capture upon substitution.

The following example 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.

<pre> 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 end </pre>	⇒	<pre> 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 end </pre>
--	---	--

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 function definition are visible in the body of that function, and shadow all external declarations. 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. 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)
```

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 $\{[\text{RenameVar} : \text{all}(\text{exprename})]\}$ any `RenameVar` rules defined during the traversal `all(exprename)` are undefined afterwards. This is exactly what we need in order to restrict renaming rules to the scope of the corresponding variable declarations.

Figure 7 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 actually 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` rule. 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, 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 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 this is not the case, and then there is no need to rename the identifier.

```

exprename = rec rn(
  RenameVar
  ⇐ [[ let <id> in <id> end ]]; { | RenameVar : all(rn) | }
  ⇐ [[ var <id> <id> := <rn> ]]; RenameVarDec
  ⇐ [[ 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 7. Bound variable renaming.

5.2. Example: Function Inlining

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

<pre> 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 </pre>	⇒	<pre> let var n : int := 10 var b : int := n var acc : int := 1 in while b > 0 do (acc := acc * b_0; b := b - 1); acc end </pre>
--	---	---

The specification in Figure 8 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* the call site to determine whether a particular call should

```

inline =
  ([ let <*id> in <*id> end ]; { | UnfoldCall : all(inline) |}
  ← Declare1; [ [ <fd*:inline> ] ]; Declare2
  ← all(inline)
  ); try(UnfoldCall; exprename)

Declare1 =
  [ [ <fd* : map(is-inlineable < DeclareFun + UnDeclareFun)> ] ]

Declare2 =
  [ [ <fd* : filter(is-inlineable < DeclareFun; fail + UnDeclareFun)> ] ]

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

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

```

Figure 8. Function inlining.

be inlined, and (2) would combine inlining with other transformations [29]. Nonetheless, the strategy contains the essence of inlining.

When encountering a `let`, a new scope for the `UnfoldCall` rule is entered. When encountering a block of function definitions, an inlining rule is declared for each inlineable² function using a `map` over the list of definitions. After applying the inlining transformation to the function definitions themselves, thus inlining any inlineable calls, the inlining rules are refreshed, thus reflecting the optimized function bodies. Furthermore, the `Declare2` strategy *removes* each function that is being inlined; after inlining all function calls there is no need for the definitions anymore.

The `DeclareFun` 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. If a function is not deemed inlineable, the `UnfoldCall` rule for it 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

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

list of strategy expressions $s_1 | \dots | s_n$, where the leftmost strategy s_1 denotes the rules defined in the most recent scope. The scope construct $\{ | L_1, \dots, L_n : s | \}$ 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:

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

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:

$$\frac{\Gamma_{L(\text{fail}|s_2|\dots|s_n)}, \mathcal{E} \vdash \langle s \rangle t \Longrightarrow \bar{t}' (\Gamma'_{L(s_1|s_2|\dots|s_n)}, \mathcal{E}')}{\Gamma_{L(s_2|\dots|s_n)}, \mathcal{E} \vdash \langle \{ | L : s | \} \rangle t \Longrightarrow \bar{t}' (\Gamma'_{L(s_2|\dots|s_n)}, \mathcal{E}')}$$

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

$$\frac{s'_1 \equiv \text{define}(drd, \mathcal{E}, s_1)}{\Gamma_{L(s_1|s_2|\dots|s_n)}, \mathcal{E} \vdash \langle \text{rules}(drd) \rangle t \Longrightarrow t (\Gamma_{L(s'_1|s_2|\dots|s_n)}, \mathcal{E})}$$

where the modification of the scope strategy is factored out using the semantic function ‘define’, which is defined as

$$\begin{aligned} \text{define}(L : p_1 \rightarrow p_2 \text{ where } s_1, \mathcal{E}, s_2) &\equiv \{ \mathcal{E}(\ ? p_1 ; s_1 ; ! p_2) \} \leftarrow s_2 \\ \text{define}(L :- p_1, \mathcal{E}, s) &\equiv \{ ? \mathcal{E}(p) ; ! \perp \} \leftarrow s \end{aligned}$$

That is, undefining a rule is modeled by producing the special term \perp . 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 \perp):

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow \dots \leftarrow s_n \rangle t \Longrightarrow t' (\Gamma', \mathcal{E}') \ t' \neq \perp}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow t' (\Gamma', \mathcal{E})}$$

Secondly, t matches an explicitly undefined pattern, hence, the application of the scope strategies produces \perp . 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.

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow \dots \leftarrow s_n \rangle t \Longrightarrow \perp (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E})} \quad \frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow \dots \leftarrow s_n \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E})}$$

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 following application of constant propagation:

<pre> 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 </pre>	⇒	<pre> 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 </pre>
---	---	---

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:

$$\llbracket \text{let } \langle *id \rangle \text{ in } \langle *id \rangle \text{ end} \rrbracket ; \{ \llbracket \text{PropConst} : \text{all}(\text{prop-const}) \rrbracket \}$$

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 9 is an extension of the constant propagation strategy in Figure 6 for blocks with local variables. A new scope is started just like in the case of renaming with $\{ \llbracket \text{PropConst} : \text{all}(\text{prop-const}) \rrbracket \}$. 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

$$\text{rules}(\text{PropConst} + x : \llbracket x \rrbracket \rightarrow \llbracket e \rrbracket)$$

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:

```

prop-const =
  PropConst
  ⇐ [[ <id> := <prop-const> ]]; AssignPropConst
  ⇐ [[ let <*id> in <*id> end ]]; { PropConst : all(prop-const) }
  ⇐ all(prop-const); try(DeclarePropConst ⇐ EvalBinOp ⇐ EvalRelOp)

DeclarePropConst =
  ?[[ var x ta := e ]]
  ; if <is-value> e
  then rules( PropConst+x : [[ x ]] -> [[ e ]] )
  else rules( PropConst+x :- [[ x ]] ) end

AssignPropConst =
  ?[[ x := e ]]
  ; if <is-value> e
  then rules( PropConst.x : [[ x ]] -> [[ e ]] )
  else rules( PropConst.x :- [[ x ]] ) end

```

Figure 9. Constant propagation for basic blocks with local variables.

```
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 in fact a composition of those two operations:

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

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 i th scope of Γ_L . Labeling the current scope entails adding a label to this list:

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

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

Defining a rule in a scope labeled with p , entails finding the first scope, which is labeled with p , and extending the corresponding strategy:

$$\frac{\text{label}(drd) \equiv p \quad \mathcal{E}(p) \in \Gamma.\text{labels}_{L_i} \quad \forall_{j=1}^{i-1} (\mathcal{E}(p) \notin \Gamma.\text{labels}_{L_j}) \quad \text{define}(drd, \mathcal{E}, s_i) \equiv s'_i}{\Gamma_{L(s_1 | \dots | s_{i-1} | s_i | s_{i+1} | \dots | s_n)}, \mathcal{E} \vdash \langle \text{rules}(drd) \rangle t \Longrightarrow t(\Gamma_{L(s_1 | \dots | s_{i-1} | s'_i | s_{i+1} | \dots | s_n)}, \mathcal{E})}$$

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

$$\text{label}(L.p \oplus r) \equiv p \quad \text{label}(L \oplus r) \equiv \epsilon$$

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.

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:

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

Note that a scope label may also be assigned as part of the scope declaration. This is just an abbreviation for a scope with an explicit labeling action:

$$\{L.p:s\} \equiv \{L:\text{rules}(L+p);s\}$$

The application of a dynamic rule is not affected by the addition of labels, but it should be noted that only inner rules are visible. That is, after the definition of a rule relative to labeled scope, that rule may not be visible since it could be shadowed by a previously defined rule in the current scope.

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 rules 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 illustrate rule extension in two example transformations: common-subexpression elimination and function specialization.

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 expressions as computed earlier in the program. The following application illustrates the transformation, and the main issues in its implementation:

<pre>(x := a + b; y := a + b; z := a + c; a := 1; z := (a + c) + (a + b))</pre>	⇒	<pre>(x := a + b; y := x; z := a + c; a := 1; z := (a + c) + (a + b))</pre>
---	---	---

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 10 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 .

The specification in Figure 10 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 expression e . 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 `kill-subexpressions`. This is achieved using the `bagof-UsedInExp` strategy that produces *all* expressions that `UsedInExp` rewrites x to.

```

cse = ([ [ <id> := <cse> ] ] <- all(cse)); try(AssignCSE <- ReplaceExp)

AssignCSE =
  ? [ [ x := e ] ]
  ; where(<kill-subexpressions> [ [ x ] ])
  ; if <is-subterm>([ [ x ] ], [ [ e ] ]) then
    id // do not propagate
  else
    rules(ReplaceExp : [ [ e ] ] -> [ [ x ] ])
    ; where(<register-subexpressions(|e)> [ [ x := e ] ])
  end

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

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

```

Figure 10. Common subexpression elimination in basic blocks.

6.2. 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 to the strategy that produces a singleton list, thus discarding all previously defined rules matching the same pattern. Undefinedness 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 from applying the original strategy in the tail.

$$\text{define}(L : p_1 \rightarrow p_2 \text{ where } s_1, \mathcal{E}, s_2) \equiv \{\mathcal{E}(?p_1; s_1; ! [p_2])\} \leftarrow s_2$$

$$\text{define}(L :- p_1, \mathcal{E}, s) \equiv \mathcal{E}(?p); ! [] \leftarrow s$$

$$\text{define}(L :+ p_1 \rightarrow p_2 \text{ where } s_1, \mathcal{E}, s_2) \equiv \{\mathcal{E}(p_1); \mathcal{E}(s_1); ! [\mathcal{E}(p_2) | \leftarrow s_2 \leftarrow ! [] >]\} \leftarrow s_2$$

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:

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow \dots \leftarrow s_n \rangle t \Longrightarrow [t_1, \dots, t_m] (\Gamma', \mathcal{E}') \ (m > 0)}{\Gamma_{L(s_1 | \dots | s_n)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow t_1 (\Gamma', \mathcal{E})}$$

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:

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \Leftarrow \dots \Leftarrow s_n \rangle t \Longrightarrow [] (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E})} \quad \frac{\Gamma, \emptyset \vdash \langle s_1 \Leftarrow \dots \Leftarrow s_n \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle L \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E})}$$

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 .

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \Leftarrow \dots \Leftarrow s_n \rangle t \Longrightarrow [t_1, \dots, t_m] (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle \text{bagof-}L \rangle t \Longrightarrow [t_1, \dots, t_m] (\Gamma', \mathcal{E})}$$

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \Leftarrow \dots \Leftarrow s_n \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle \text{bagof-}L \rangle t \Longrightarrow [] (\Gamma', \mathcal{E})}$$

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

Once Another interesting use of dynamic rules that is possible because of the design above, is the application of a dynamically defined 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. For example, to ensure that a function is unfolded at most once, the unfolding rule is called as once-UnfoldCall. When this is successfully applied to a function call, it is automatically undefined. The semantic rules look a bit complicated, but comes down to finding the first alternative that matches the subject term and then removing it.

$$\frac{\Gamma, \emptyset \vdash \langle \text{once}_{L_1}(s_1) \Leftarrow \dots \Leftarrow \text{once}_{L_n}(s_n) \rangle t \Longrightarrow \bar{t}' (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|\dots|s_n)}, \mathcal{E} \vdash \langle \text{once-}L \rangle t \Longrightarrow \bar{t}' (\Gamma', \mathcal{E})}$$

$$\frac{\begin{array}{l} s \equiv \{ ? p_1; s'_1 \} \Leftarrow \dots \Leftarrow \{ ? p_{j-1}; s'_{j-1} \} \Leftarrow \{ ? p_j; s'_j \} \Leftarrow \{ ? p_{j+1}; s'_{j+1} \} \Leftarrow \dots \Leftarrow \{ ? p_k; s'_k \} \\ \forall_{l=1}^{j-1} : \Gamma, \emptyset \vdash \langle \{ ? p_l; s'_l \} \rangle t \Longrightarrow \uparrow (\Gamma', \mathcal{E}') \quad \Gamma, \emptyset \vdash \langle \{ ? p_j; s'_j \} \rangle t \Longrightarrow [t_1, \dots, t_n] (\Gamma', \mathcal{E}') \\ s' \equiv \{ ? p_1; s'_1 \} \Leftarrow \dots \Leftarrow \{ ? p_{j-1}; s'_{j-1} \} \Leftarrow \{ ? p_{j+1}; s'_{j+1} \} \Leftarrow \dots \Leftarrow \{ ? p_k; s'_k \} \end{array}}{\Gamma, \mathcal{E} \vdash \langle \text{once}_{L_i}(s) \rangle t \Longrightarrow t_1 (\Gamma'_{L(s_1|\dots|s_{i-1}|s'_{i+1}|\dots|s_m)}, \mathcal{E})}$$

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

6.3. 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.

Partial evaluation is a transformation that specializes a program to its static inputs [18]. 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 an extension of constant propagation to involve functions. Figure 11 presents an extension of the constant propagation transformation with call unfolding and specialization. Parts of the specification are not presented in Figure 11;

```

prop-const =
  PropConst
  <+ [[ <id> := <prop-const> ]]; AssignPropConst
  <+ prop-let <+ prop-fun <+ DeclareFunArg
  <+ prop-control(prop-const)
  <+ all(prop-const)
    ; try(EvalBinOp <+ EvalRelOp
          <+ UnfoldCall; exprename; prop-const <+ SpecializeCall)

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

DeclareFunDec =
  ?[[ 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*

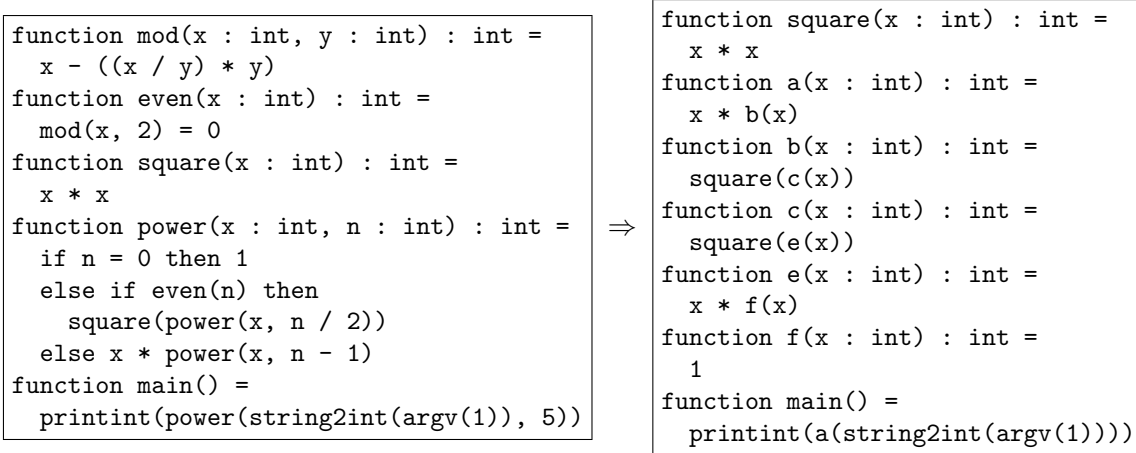
    SpecializeCall :
      [[ f(a1*) ]] -> [[ g(a2*) ]]
      where <split-static-dynamic-args> (x1*, a1*) => (d*, (x2*, a2*))
      ; new => g
      ; rules(
        Specialization.f :+
          [[function f(x1*) ta = e2]] -> [[function g(x2*) ta = let d* in e1 end]]
      )
  )

split-static-dynamic-args =
  zip; partition(\ (FArg[[ x ta ]], e) -> [[ var x ta := e ]] where <is-value> e \ )
  ; not(([],id)); (id, unzip)

prop-fun      = ?[[ function f(x*) ta = e ]]; {| PropConst : all(prop-const) |}
DeclareFunArg = ?FArg[[ x ta ]]; rules( PropConst+x :- [[ x ]] )

```

Figure 11. A simple online partial evaluator with function specialization.

Figure 12. Specialization of `power(.,5)`.

the strategies `AssignPropConst` and `DeclarePropConst` were already defined in Figure 9, the strategy `prop-control` defines propagation through control structures and is defined in the next section.

The example in Figure 12 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 this transformation, the specification in Figure 11 is a combination of constant propagation for local propagation of constant values, and function unfolding and specialization. The strategy has basically the same structure as the constant propagation strategy in Figure 9, with a few extra cases. The major part of the transformation is the strategy `DeclareFunDec`, 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 or inlining rule that replaces a call with an instantiation of the function body. 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-let` strategy treats `let` bindings by first declaring the unfolding and specialization rules with a map over all declarations. Next, it performs constant propagation in the `let` body, which should give rise to function specializations. Finally, all function specializations are added to the `let`, by extending each function with its specializations using `specialize-fun`.

The interesting rule in this specification is `SpecializeCall`. It transforms a call to function f into a call to a new function g . The new function is a specialization of f to the constant valued arguments

of the call $f(a1^*)$. As a side effect, the definition of the new function g is declared as a specialization of f . By *extending* the Specialization rule, any previous specializations are preserved. The `split-static-dynamic-args` strategy is used to partition the combined list of formal and actual parameters into (1) a list of local variable declarations d^* , assigning the constant valued arguments to the corresponding formal parameters; and (2) a list of formal parameters x^* and actual parameters $a2^*$, corresponding to the non-constant valued arguments. These are used to produce (1) the specialized function g , which has the remaining parameters $x2^*$ as formal parameters and uses the variable bindings d^* to specialize the original body $e1$, and (2) the call $g(a2^*)$ to the specialized function.

As an example, consider the specialization of `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 `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 subjected to constant propagation itself. Doing this reduces the function to

```
[[ function a(x : int) : int = let var n : int := 5 in x * power(x, 4) end ]]
```

The variable declaration for `n` is dead and can be removed, which is not done by the specializer in Figure 11, but by a separate dead code removal transformation. The partial evaluation of the new function `a` gives rise to the call to `power(x,4)` as the next candidate for specialization.

The partial evaluation of the specialized function is not performed by `SpecializeCall`, but in the `specialize` strategy, which replaces function definitions with their specializations. This could be done with the `bagof-Specialization` strategy to obtain all specializations defined so far. However, the partial evaluation of the specialized functions may give rise to further specializations. Thus, 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, partial evaluating it by calling `prop-const`, and then recursively calling `specialize-fun` to obtain further specialized functions.

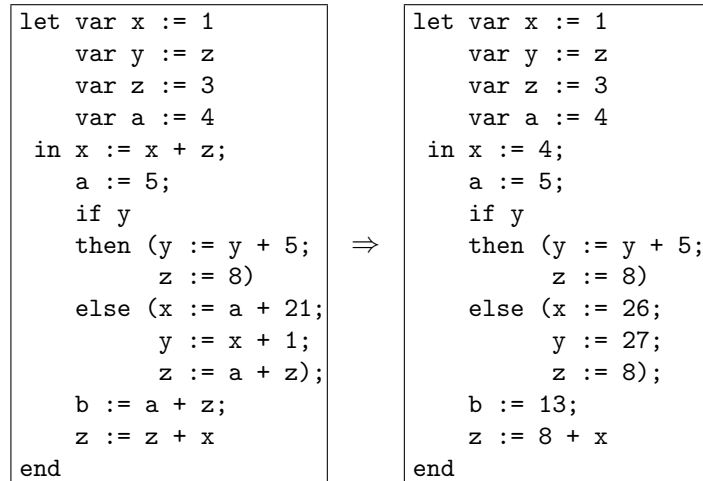
To summarize, the specification in Figure 11 defines an online partial evaluator that evaluates calls with all constant arguments, and specializes calls with at least one constant argument (and at least one non-constant argument). The complete specification is a bit over 100 lines including propagation 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. We have also developed an offline partial evaluator with separate binding-time annotation in the same style.

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 at any time during the constant propagation transformation represents all known propagation facts at the current point in the program. 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: Constant Propagation with Control Flow

Thus far, we have considered constant propagation with straight line code, possibly with local variables. Figure 13 presents a *complete* specification of intra-procedural constant propagation with structured control-flow constructs. The issues that need to be solved are illustrated by the following example:



(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 usual in a basic block. For 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 is the same after the conditional. While z is changed in both branches, its value is always the same, so it can be propagated. The value of x is changed in the second branch, therefore its value cannot be propagated afterwards.

```

prop-const =
  PropConst
  <& [[ <id> := <s> ]]; AssignPropConst
  <& [[ let <*id> in <*id> end ]]; { | PropConst : all(s) | }
  <& prop-control(prop-const)
  <& all(prop-const); try(DeclarePropConst <& EvalBinOp <& EvalRelOp)

DeclarePropConst =
  ?[[ var x ta := e ]]
  ; if <is-value> e
  then rules( PropConst+x : [[ x ]] -> [[ e ]] )
  else rules( PropConst+x :- [[ x ]] ) end

AssignPropConst =
  ?[[ x := e ]]
  ; if <is-value> e
  then rules( PropConst.x : [[ x ]] -> [[ e ]] )
  else rules( PropConst.x :- [[ x ]] ) end

prop-control(s) =
  [[ if <s> then <id> ]]
  ; (EvalIf; s <& ([[ if <id> then <s> ]] /PropConst\ id))

prop-control(s) =
  [[ if <s> then <id> else <id> ]]
  ; (EvalIf; s
  <& ([[ if <id> then <s> else <id> ]] /PropConst\ [[ if <id> then <id> else <s> ]]))

prop-control(s) =
  [[ while <id> do <id> ]]
  ; ([[ while <s> do <id> ]]; EvalWhile
  <& /PropConst\* [[ while <s> do <s> ]])

prop-control(s) =
  [[ for <id> := <s> to <s> do <id> ]]
  ; (EvalFor <& /PropConst\* [[ for <id> := <id> to <id> do <s> ]])

```

Figure 13. Intra-procedural constant propagation for expressions with local variables and structured control constructs.

From these requirements we can conclude that (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 propagation 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 $s_1 /L \setminus s_2$ 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 $/L \setminus$ com-

binator 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 expression

```
[[ if <id> then <s> else <id> ]] /PropConst\ [[ if <id> then <id> else <s> ]]
```

applies a strategy s 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.

For loops the story is basically the same. However, there may be any number of iterations. Therefore, the application of the transformation to the loop is iterated until a stable rule set is obtained. This is expressed using the fixpoint combinator $/L\^* s$, which repeats the application of s until no more changes in the rule set L occur. For example, in the constant propagation transformation, the strategy

```
/PropConst\^* [[ while <s> do <s> ]]
```

expresses the fixpoint iteration over a while-loop.

The specification in Figure 13 uses these intersection combinators to express constant propagation over structured control-flow constructs. Furthermore, the transformation combines constant propagation with unreachable code elimination, which is a powerful combination. For example, the following example

<pre>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</pre>	⇒	<pre>let var x := 0 var y := 0 in x := 10; while A do dosomething(); y := 10 end</pre>
---	---	--

shows how the application of constant propagation (to determine that x is 10 at the condition), gives rise to elimination of code, which would otherwise interfere with the knowledge that x is 10. In the transformation this is achieved by first evaluating the condition of the conditional, and then trying to evaluate it. That is, the strategy expression

```
[[ if <s> then <id> else <id> ]]
; (EvalIf; s <- ([ [ if <id> then <s> else <id> ] ]
  /PropConst\ [ [ if <id> then <id> else <s> ] ]))
```

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

7.2. Semantics: Intersection and Union 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.

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

The intersection/union of two rule sets is the point-wise intersection/union of the scope strategies.

$$\vec{s} \cap \vec{s}' \equiv (s_1 \cap s'_1) | \dots | (s_n \cap s'_n) \quad \vec{s} \cup \vec{s}' \equiv (s_1 \cup s'_1) | \dots | (s_n \cup s'_n)$$

The intersection/union of two scope strategies corresponds to the intersection/union of the resulting strategy application

$$s_1 \cap s_2 \equiv \langle \text{isect} \rangle (\langle s_1 \rangle, \langle s_2 \rangle) \quad s_1 \cup s_2 \equiv \langle \text{union} \rangle (\langle s_1 \rangle, \langle s_2 \rangle)$$

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

Fixpoint Combinators The fixpoint variants of the intersection and union operations repeat the application of a strategy until the rule set is stable. Thus, the first rules define that the result of the application of the fixpoint operation produces the result of applying the transformation, if the L rule set before and after are the same. The third and second rules express that if this is not the case, a recursive invocation of the fixpoint should be performed.

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

Note that 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.

```

elim-code =
  VarNeeded <+ elim-assignment(elim-code)
  <+ elim-seq(elim-code) <+ elim-if(elim-code) <+ elim-while(elim-code)
  <+ all(elim-code)

VarNeeded =
  ?[[ x ]]; rules( Needed : [[ x ]] )

elim-assignment(s) =
  ?[[ x := e ]]
  ; if <Needed> [[ x ]] then
    rules( Needed :- [[ x ]] ); [[ <id:id> := <s> ]]
  else
    ![[ ( ) ]]
  end

elim-seq(s) =
  [[ (<*reverse-filter(s; not(?[[ ( ) ]]))>) ]]

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

elim-while(s) =
  [[ while <id> do <id> ]]; (\Needed/* [[ while <s> do <s> ]]); try(ElimWhile)

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 ]]

```

Figure 14. Intra-procedural dead code elimination.

7.3. Example: Dead Code Elimination

Finally, as an example of the union combinator, we present the specification of dead code elimination, a transformation that removes assignments, computing a value that is not needed. A difference with previous transformations we have seen thus far is that it performs a *backward* flow analysis. The specification in Figure 14 defines the strategy for dead code elimination. For each variable, the dynamic rule `Needed` indicates whether its value is needed. If we encounter a variable that means it is needed and a rule is generated that records this fact. Otherwise the strategy should ensure that we never visit variables that are not needed. An assignment is transformed to the empty sequence `()` if the value it computes is not needed. Otherwise, the variables in the right-hand side of the assignment are marked as needed by a recursive traversal. The other non-standard aspects of the transformation are the backwards traversal over the expressions in a sequence and visiting the condition of the `if-then-else` after visiting the branches.

8. Implementation

The implementation of dynamic rewrite rules has to cater for large numbers of dynamically defined rewrite rules. This imposes some challenges on the implementation. Firstly, the implementation cannot just follow the method of dynamic rule composition of the formal semantics. In the semantics, all rules are combined by choices. In the implementation, this would imply that a potentially very large number of rewrite rules has to be applied to find a successful application of a rewrite rule. Secondly, dynamic construction of a single strategy from dynamic rule definitions would imply interpretation of the complete composed strategy.

For this reason, an efficient implementation of dynamic rewrite rules must lookup the dynamic rules that could be applied successfully, without actually applying them. The solution is to store the dynamically defined rules in such a way that all possible applicable rules can be accessed with just a key that is efficiently derived from the term to which the dynamic rule is applied.

8.1. Static and Dynamic Patterns

Dynamically defined rewrite rules vary at one point: the terms to which the context variables are bound when the dynamic rule is defined. For example, consider

```
?[ var  $x$   $ta$  :=  $n$  ]
; rules( PropConst : |[  $x$  ] -> |[  $n$  ] )
```

The dynamic PropConst rules will vary in the actual value of the context variable x and n . If the PropConst rule is applied to a variable, then the dynamic rule that might (and in this case will) be applicable can be identified by the name of the variable, which should correspond to x .

Therefore, dynamic rules are stored in a hash table, where the keys represent the *dynamic* aspects of the left-hand side pattern of a dynamic rule. To this end, the left-hand side pattern is split in two parts: a *dynamic pattern* and a *static pattern*. Dynamic rule definitions are stored in a hash table with as key the dynamic pattern. Upon application of the rewrite rule, the dynamic pattern corresponding to the subject term can be reconstructed and with this key all the possibly applicable rules can be found in constant time. The static pattern ignores the dynamic aspects of a rewrite rule, i.e. the context-bound variables.

What then are these dynamic and static patterns? In both cases, some parts of the left-hand side pattern are replaced with wildcards. For a dynamic pattern, all *non-context-bound* variables are replaced by wildcards and the context-bound variables are replaced by their actual value in the context of definition. For a static pattern, all *context-bound* variables are replaced by wildcards. For example, consider the following fragment

```
?[ function  $f(x1*)$   $ta$  =  $e1$  ]
; rules( UnfoldCall : Call( $f$ ,  $a*$ ) -> ... )
```

If this fragment is applied to a function with the name `power`, then the dynamic pattern of the rule that is then defined is `Call("power", _)`, since f is bound to "power" and $a*$ is not a context variable. Note that the dynamic pattern contains no variables whatsoever; it only contains the context value for f and a wildcard for $a*$, and it is exactly this pattern that will be used as a key in the hash table. For the static pattern, the context variables are just replaced by wildcards. Hence, the static pattern is `Call(_, $a*$)`.

Note that if the patterns are combined, then they exactly fit each other. The combined pattern is `Call("power", a*)`. This is the actual pattern of dynamically defined rewrite rule and this pattern is used in the formal semantics of dynamics rewrite rules.

8.2. Dynamic Rule Application

Dynamic rule application is performed in three steps. First, the static pattern is applied to the subject term. This application is very efficient, since the static pattern is known at compile-time and can therefore be compiled as efficient as an ordinary static pattern match. Next, the dynamic pattern is reconstructed from the subject term. The hash table of dynamically defined rules is then consulted to find possible applicable rules for this dynamic pattern. In the third step, the dynamically defined rules that have been found in the hash table are applied. This involves the execution of the `where` clause and the instantiation of the pattern at the right-hand side of the dynamic rule.

Overlapping Static Patterns Without having revealed the details of the compilation of dynamics yet, it is already clear that pattern matching of the left-hand side of dynamic rules is performed in two steps. Unfortunately, this two step application introduces a restriction to the kind of patterns that can be used at the left-hand side of a dynamic rule. Luckily, this restriction does not affect most real program transformations. Hence, all programs in this article satisfy the restriction. What is this restriction then? The restriction is related to the way static patterns are applied. If dynamic rules are defined at different places in the code, i.e. there are more than one `rules(L : ...)` for a dynamic rule name L , then there are also more than one static patterns for this rule L . These static patterns are combined with the non-deterministic choice operator (+). This choice operator does not respect the definition order of the dynamically defined rules. If more than one of the static patterns for rule L is applicable, then the order in which the dynamic rules have been defined is not respected. Hence, static patterns must exclude each other to preserve this order. In other words, overlapping static patterns are not allowed. For example, the patterns (x, y) and (y, x) , where x is a context variable only, do overlap.

Closure of Dynamic Rule Thus far, the actual way the dynamically defined rewrite rules are stored in the hash table has not been discussed. Since dynamic rules have lexical scope, context-bound variables can be used in the dynamic rule, but they can be applied when this lexical scope has already been left. To keep dynamic rule definitions available for application in an entirely different context as they were generated, the closure of a dynamic rule is stored at definition time. This closure contains all values for context variables used in the dynamic rule definition and a pointer to the code that must be executed. As explained before, this closure is stored in the hash table, indexed by dynamic pattern of the rewrite rule. The closure contains only the values of the context variables that occur in the `where` clause and the right-hand side pattern, but not in the left-hand side of the rewrite rule. The value of the context variables that occur only in the left-hand side are already available at application time, since they refer to subterms of the term where the rule is applied to.

The pointer to the code that is to be executed needs some more explanation. The Stratego compiler lifts the dynamic rule definition (i.e. within `rules(...)`) out of its context up to a top-level static rewrite rule. Each dynamic rule definition is assigned a unique stamp, which is used in both the lifted static rule (at compile-time), and as a pointer in the closure, which is constructed at run-time upon each rule definition. For example, consider the following rule definition again

```
?[[ var x ta := n ]]
; rules( PropConst : [[ x ]] -> [[ n ]] )
```

If this fragment is applied to `[[var k : int := 3]]`, then the stored closure will look like the following, where the dynamic pattern for this application is `Var("k")`, `"1_0"` is the unique stamp and `Int("3")` is the only needed context information.

```
Var("k") => Closure("1_0", [Int("3")])
```

Actually, the value stored for a key in the hash table is not just a closure, but a list of closures instead. This allows for extending a rule set, i.e. having multiple applicable rule instances for the same dynamic pattern. The special-purpose application strategies *bagof-L* and *once-L* are available for rewriting terms with such an extended rule set.

Lifting To illustrate the lifting of dynamic to static rules, consider a fragment from the constant propagation in Figure 13:

```
DeclarePropConst =
  ?[[ var x ta := e ]]
  ; if <is-value> e
    then rules( PropConst+x : [[ x ]] -> [[ e ]] )
    // ...
```

The compiler transforms this into an instrumented strategy:

```
DeclarePropConst =
  ?[[ var x ta := e ]]
  ; if <is-value> e
    then dr-label-scope(| "PropConst", x)
      ; where(dr-set-rule(| "PropConst", x, Var(x), ("d_0",e)))
      // ...
```

and two co-operating static rules

```
PropConst :
  f_0 @ [[ x ]] -> <fetch-elem(aux-PropConst(| x, f_0))> closures
  where dr-lookup-rule(| "PropConst", f_0) => closures

aux-PropConst(| x, f_0) :
  Closure("d_0", [e]) -> e
  where <id> f_0
```

In the above, the scope is labeled at run-time using the context value for `x`. The dynamic rule definition was assigned a unique stamp `"d_0"`. The only context info that will be stored in the closure is the value of `e`. The lifted, static rewrite rule `PropConst` retrieves any closures for `"PropConst"` using the instantiated dynamic pattern as a key. Next, it applies the helper `aux-PropConst` to the list of closures, until the first closure that rewrites the current input term (matched in `f_0`) successfully. The

helper `aux-PropConst` is quite similar to the original dynamic rule definition. The only difference is that it receives any variables from the dynamic pattern (here: x) as extra term arguments, and its input term is now a closure, thus passing on any definition-time context bindings that are needed in either the right-hand side or the condition (here: e). The static pattern contains no variables in this example.

Generation of Alternative Application Strategies Besides the normal, lifted, static rewrite rule providing normal rewriting functionality, some alternative application strategies are generated by the compiler. These have been introduced in Section 6.2. For the example above, the generated `bagof-PropConst` will be almost identical to `PropConst`, except that `fetch-elem` will be replaced by a `filter`, thus filtering out only the succeeded applications in a list. The generated `once-PropConst` will perform rewriting just as `PropConst` does, but upon successful application it retracts the closure from the scope (i.e. undefines that specific rule instance).

8.3. Scoping Rule Sets

The previous sections all referred to a hash table in which closures are stored. This is a bit more subtle though, since dynamic rule scopes have to be handled as well. All this information is difficult to represent efficiently in a single hash table. Although closures are stored in a stack-like list for each dynamic pattern, it is impractical to represent the scopes in this list as well. The list would then contain all closures for all scopes and list operations would be needed for manipulation of outer scopes, which will become expensive soon. Furthermore, an end of scope might become expensive, since all entries might need to be updated.

Instead, we opt for a stack of scopes, similar to a control stack, which is used for organizing activation records (or stack frames) that contain local variables and other data related to a function call. In the same way, a stack of scopes is maintained for each dynamic rule name in the implementation of dynamic rules. Each scope has its own hash table for the dynamic rules that are defined for this scope.

Operations Scope entry now involves creation of a new scope (compare to stack frame) on top of the stack. The costs of this operation are $O(1)$. Leaving a scope neither involves any list traversing whatsoever: the top stack frame is simply dropped off and costs are again $O(1)$. The labeling of scopes is simply adding the new label to the existing list of labels attached to the current scope.

Since hash tables in Stratego have state, only a reference to the (list of) hash tables has to be maintained during transformation, so no further terms have to be dragged around. Upon defining a new dynamic rule the hash table is extended, which is basically a side-effect when transforming a term. This side-effect may be undesirable when forking context-sensitive information in program transformation, but the next section sketches our solution for this. When the dynamic rule definition is labeled, the scope stack is first traversed until the first scope that has the appropriate label.

Looking up rule closures upon application of a dynamic rule involves traversing the stack of scopes until the first scope in which that dynamic rule was defined (thus shadowing any outer scopes for application).

Example An example will illustrate the representation of dynamic rule scopes and the closures stored within. Figure 15 shows a small Tiger fragment. It consists of three nested `let` blocks, of which the

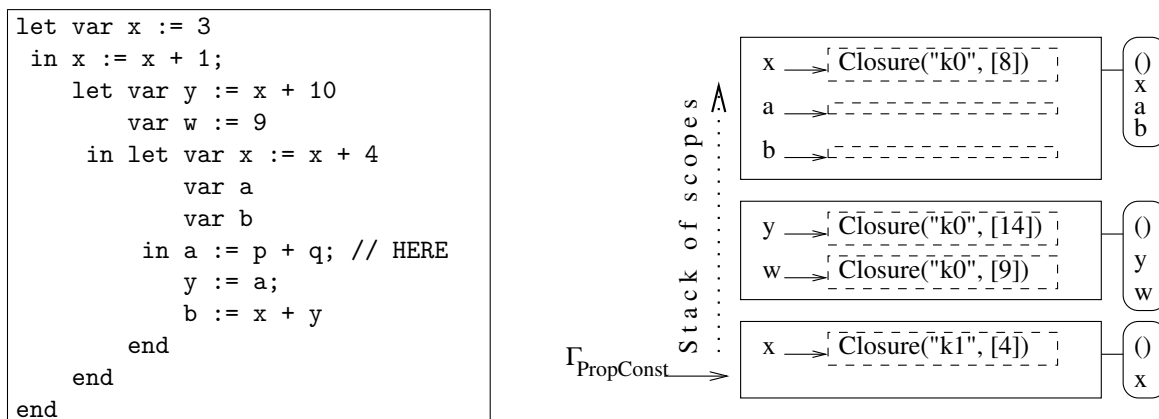


Figure 15. Scope representation during constant propagation. Left: input Tiger program. Right: contents of scope stack when constant propagation is at // HERE.

inner variable declaration for x shadows the outermost one. When constant propagation is applied to this fragment three dynamic rule scopes will eventually be entered, each with the appropriate labels attached, and the closures of any generated rules within.

The picture on the right in Figure 15 represents the state of the scope stack *right after* the constant propagation has treated the assignment $a := p + q$ (marked '// HERE'), so only two assignments are left. Note that in this picture the stack grows upward if a new scope is entered, and shrinks downward if a scope is left. The two rule generators that play a role here are from within `DeclarePropConst` and `AssignPropConst` as can be seen in Figure 13. The unique stamps for these rule generators are "k0" respectively "k1" in this case.

The first, outermost scope contains a rule $\llbracket x \rrbracket \rightarrow \llbracket 4 \rrbracket$ (where x is not a meta variable). This rule was defined at the assignment $x := x + 1$. The definition of this rule has overridden a previous rule for `var x := 3`, which is no longer visible in the scope stack. The dynamic pattern `Var("x")`, serves as the key in the hash table; for ease of reading just denoted 'x' in the right picture, just as numbers like 4 are actually `Int("4")`. The closure for the rule $\llbracket x \rrbracket \rightarrow \llbracket 4 \rrbracket$ refers to its static rewrite rule by "k1". The second scope contains closures for the two assignments, hence with stamp "k0".

The third, innermost scope is labeled x . Therefore the rule $\llbracket x \rrbracket \rightarrow \llbracket 8 \rrbracket$ is tied to that local scope, and not to the outermost one. The assignment $a := p + q$ can not be evaluated, so a previous rule for a will be undefined in this scope, as the empty list of closures for a in the top scope depicts. The same is the case for b since the variable declaration is uninitialized.

Now, this is the point of the transformation that is depicted in Figure 15. There are still two things left: The assignment $y := a$ cannot be evaluated since no rule for x exists, so the previous rule for y will be undefined. This will occur in the second scope, since that has the correct label, and the closures list will be empty just as it is already for a . A similar thing will happen for b , but now in the innermost scope.

8.4. Forking and Change Sets

Section 7 introduced the concept of forking a dynamic rule environment and intersecting or joining the two rule sets afterwards.

The unwanted side effect in one branch of control flow that would influence the other branch is prevented by ‘freezing’ the rule set at the fork point and push two fresh *change sets* for the two branches that will capture the changes to the existing scopes. A change set is basically the same key-value hash table for closures, but it additionally uses scope labels in its keys. This way, one change set can maintain changes for multiple scopes.

Intersection or union at the meet point is cheap now, since only the two change sets need to be intersected, all rule sets for the surrounding scopes need not be intersected: they have not been modified, since the change set has captured all changes. The final action to be performed is committing the intersected or joined change set to the frozen initial rule set. Only the appropriate rule set manipulations have ended up in the final rule set and normal transformation can continue.

8.5. Implementation Costs

In the preceding sections we have described the implementation of dynamic rules in Stratego, which has been carefully designed to comply with the formal semantics *and* be efficient in all operations. The use of hash tables saves time on rule definitions and calls. Maintaining a stack of scopes, each with its own hash table saves time on traversing scopes. Finally, the use of change sets makes the intersection and related operations relatively cheap. The costs for the various rules now should come down to:

Defining, calling and undefining a rule	$O(1)$ (or $O(s)$ if $s > 0$)
Entering and exiting of scope	$O(1)$
Forking of dynamic rule environment	$O(1)$
Intersection/union of rule sets	$O(n)$
Committing a change set	$O(n')$

Here, s is the number of enclosing scopes upon rule (un-)definition or calling, n is the total number of rules in the two change sets to be intersected and n' is the number of rules in the change set to be committed. In the next section we will validate the implementation against these performance requirements.

9. Performance of Dynamic Rules

As with all rewriting that occurs within iterative or recursive traversals, it is necessary to have a good idea of the costs of defining and applying dynamic rules. Also the costs of (nested) scoping, scope labeling and rule set intersection are important.

To benchmark the several aspects of dynamic rules use we use the constant propagation for Tiger (see Figure 13). The subject Tiger programs are automatically generated and typically contain $O(10^5)$ statements. The constant propagation implementation is profiled using a built-in equivalent of the Unix `times` command. User and system time are accumulated and child processes play no role here. All experiments are performed five times and running times are averaged.

<pre>v_1 := 1; // ... v_m := m; w_1 := v_1; // ... w_m := v_m</pre>	<pre>v_1 := 1; // ... v_m := m</pre>	<pre>v_1 := 1; w_1 := v_1; // ... w_m := v_1</pre>
---	--------------------------------------	--

Figure 16. Left: Tiger program with m integer assignments, followed by m variable assignments. Middle and right: programs which only contain the integer or variable assignments respectively.

9.1. Benchmark: Definition and Application

A first test will show how dynamic rules behave for a growing amount of rule definitions and applications. Figure 16 gives the typical test inputs. The general test consists of a sequence of m assignments of unique integers to distinct variables, followed by m assignments of these variables to other distinct variables. Hence, the latter sequence consists entirely of statements that can be optimized by replacing variable references with propagated integers. Scope labeling and definition of dynamic rules in specific labeled scopes has been turned off for this test, i.e. all dynamic rule definition and application occurs in the current scope. This has no effect on the resulting propagation transformation for this type of input, but it avoids any possible labeling related costs in this benchmark.

Since dynamic rule definition and lookup essentially comes down to constant time saves and lookups of context values, runtime is expected to be linear in the amount of statements in the subject Tiger program. Figure 17 shows that overall this is indeed the case, but that the runtime scales differently around certain program sizes. This is most likely due to some low-level memory management mechanisms, such as memory allocation and garbage collection. The size or fill percentage of hash tables is not of any noticeable influence here, as we have noticed in some experiments that varied the initial table size.

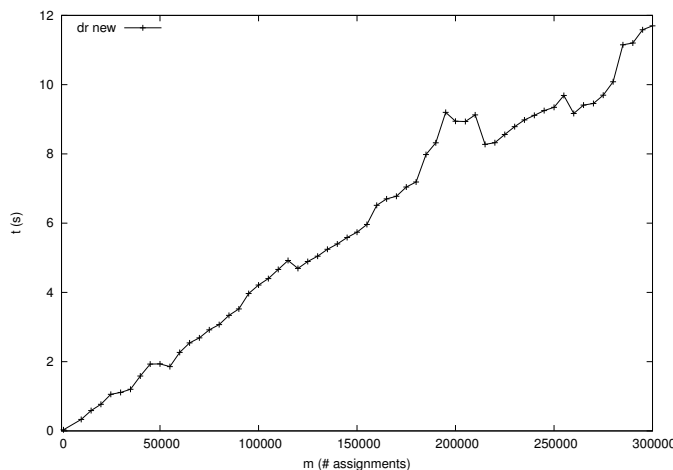


Figure 17. Performance of Tiger-PropConst on a sequence of assignments of growing length.

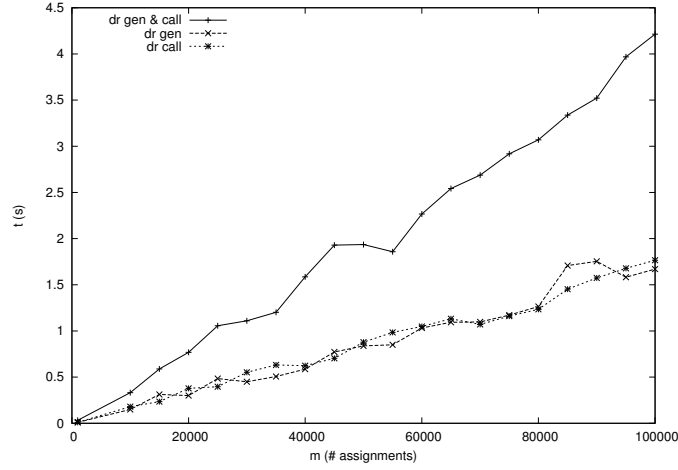


Figure 18. Comparison of dynamic rule definition to application costs.

Comparing dynamic rule definition costs to application costs is done by leaving out respectively the second or the first assignment sequence from programs like the first one in Figure 16. More precisely, for testing application costs only one initial rule is defined, followed by assignments $x_i := x_0$ ($i = 1 \dots m$). Figure 18 shows that definition and application dynamic rules are equally expensive, on the average.

9.2. Benchmark: Dynamic Rules Scope

The runtime representation of dynamic rewrite rules has been designed for very cheap entry and exit of dynamic rule scopes. Creation and destruction of hash tables (one per scope entering and leaving) is very efficient in the underlying ATerm library [7].

Figure 19 shows the typical input files for the tests executed for benchmarking the performance of dynamic rule scopes. `let`-blocks are nested up to depth n and at each level there is a sequence of $2m$ assignments, similar to the ones in the previous benchmark. Note that the depth n only amounts to n times longer runtimes, and has no special meaning of its own. The cost of scope nesting is supposed to be influenced by increasing the parameters p and q , which we will explain first.

First, three kinds of scopes are involved here. We say that a dynamic rewrite rule is *defined in* a scope s_1 if the dynamic rewrite rule is part of this scope s_1 . That is, if scope s_1 is left, then the dynamic rewrite rule is no longer available. We say that a dynamic rule is *generated in* a scope s_2 if the program is in scope s_2 when the dynamic rule is defined in a scope s_1 . The third scope that is involved, is the scope where a dynamic rule is applied.

To explain the parameter p , consider the second Tiger fragment in Figure 19. At top level, Tiger-PropConst will create a labeled scope for `a` and `b`. The body of this `let` first contains two additional nested `lets`, which results in extra scopes (labeled `e` and `f`). It is only then that a dynamic rule definition occurs, namely for the two integer assignments to `a` and `b`. This scope distance between the scope of definition and generation is denoted p , which is $p = 2$ in this example.

Similarly, the distance q between generation and application scope can be measured. In the same

<pre> let var c var e in (c := 1; e := 2; c; e; let var a var b in (a := 1; b := 2; a; b) end) end </pre>	<pre> let var a var b in (let var f in let var e in (a := 1; b := 2; let var c in (a; b) end) end end) end) end </pre>
---	--

Figure 19. Left: Tiger program with $n = 2$ nested `let`-blocks, each with $m = 2$ assignments. Scope labeling, rule definition and rule application are in the same scope ($p = 0$, $q = 0$). Right: Tiger program with $n = 1$ `let`-block, each with $m = 2$ assignments. Distance between scope labeling and rule generation is $p = 2$ (scopes for `f` and `e`), and distance between rule generation and rule application is $q = 1$ (scope for `c`).

example, notice that after the two integer assignments, an additional scope for the `let var c...` is introduced, which then contains the statements that will receive propagated values. The distance between the generation and application scope is $q = 1$. Notice that the distance between the *definition* and the application scope is $p + q$. Therefore, q can be negative as well.

The aim of this benchmark is to find out how the performance is affected if there is a certain distance between definition, generation, and application scope. The costs of nested scopes is expected to be linear in the distance between the scopes, since both rule generation and application have to traverse from the current scope up to the scope in which the rule definition is stored. This is simply a sequential walk through the stack of scopes.

Three tests were performed, all with $m = 1000$ and $n = 10$. Figure 20 shows the results of these tests. The first test varies p from 0 to 30 ('distant generations'). In this test, rule application only occurs in the generation scope, i.e. $q = 0$. The second test varies q from 0 to 30 and keeps generation of rules in the definition scope, i.e. $p = 0$ ('distant calls'). First and foremost, the results of these two tests shows linear behaviour for the scope traversing. However, 'distant generations' are far more costly than 'distant calls'. This is explained by the fact that rule generations have to inspect the list of scope labels at each scope, whereas rule calling only does a constant time lookup in the hash table of each scope. The costs of scope labeling and the inspection of the labels list is considered in more detail in the next section.

Figure 20 shows a third line as well, which also depicts 'distant generations', but in this case no rule calls are included. This is achieved by leaving out the PropConst-sensitive expressions (`a`; `b` in Figure 19, right program). The reason for this is that for $p > 0$ and $q = 0$, the application scope is actually also 'distant', namely p scopes away from the top-level definition scope. Hence, in the 'distant generations' experiment, the scopes list had to be traversed for both the generation and the application. By leaving out the calls, a more fair comparison can be made between generation and application. Of course, the cost increase per scope is similar. More interesting is the crossing at distance ≈ 8 . Apparently,

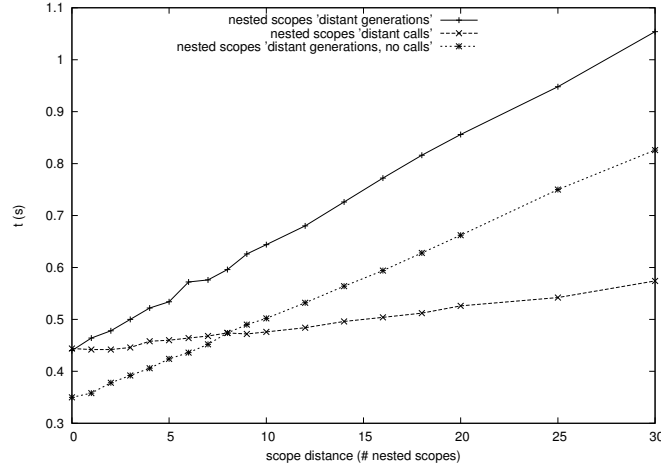


Figure 20. Performance of Tiger-PropConst on nested lets with increasing distance between definition and generation scope, and between generation and application scope.

the inspection of the scope labels, which are singleton lists in this case, is cheaper than the hash table lookup until that point.

Nevertheless, the main conclusion is that although costs are always linear, distant rule generation is relatively expensive because of the scope label inspection.

9.3. Benchmark: Scope Labeling

Labeling of scopes may play an even more influential role than the preceding benchmark suggests, since the current implementation of scope labels is straightforward and not specifically efficient. For each scope label, a new label value is pushed onto the head of the label list. For rule definition within some labeled scope a linear search is performed on the label list. Given that a scope is labeled with m labels, an average lookup of any label will cost $O(m)$. If within that scope m dynamic rules are defined with a label, total definition costs will be of $O(m^2)$.

The tests performed for this benchmark keep all definitions and applications in the same scope (i.e. $p = 0$ and $q = 0$), but since scope labeling (in the declaration section of a `let`) is separate from labeled rule definition (in the body of the `let`), the costs should be quadratic.

Figure 21 indeed shows the quadratic behaviour of the running time. This is a good argument for improving the labeling mechanism in the near future. A simple set data structure for maintaining the scope labels will turn the quadratic costs into linear without doubt. On the other hand, it is important to realize that applications seldomly will have the amount of scope labels tested in this benchmark. It will depend on this amount whether the use of sets is preferable to the use of a simple list.

9.4. Benchmark: Dynamic Rules Set Intersection

If transformations using dynamic rules have to deal with control-flow structures, then rule sets have to be combined. Section 7 already introduced the concept of computing with rule sets, which usually comes

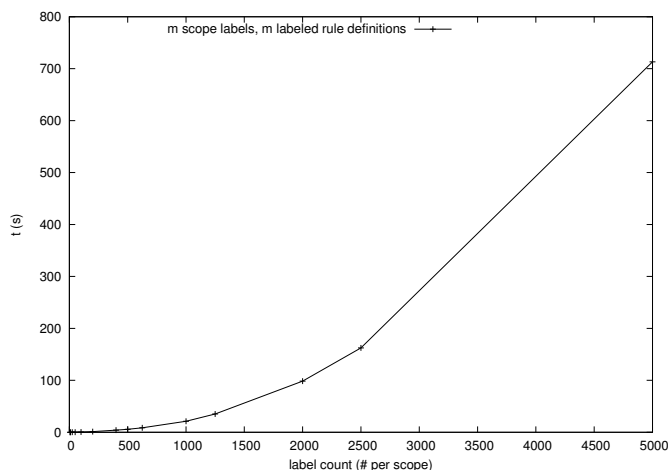


Figure 21. Performance of Tiger-PropConst on nested lets of with increasing amount of labels per scope.

down to intersection or union of two rule sets.

Intersection is potentially an expensive operation, having to intersect two entire rule sets that consist of several scopes, with several rule definitions inside each scope. Our implementation has a more efficient approach to this however, as section 8.4 described, by using change sets on top of rule sets when control-flow is involved.

The benchmark input now consists of n `if then else` blocks, where each branch contains $m/2$ assignment sequences as seen in the initial benchmark in section 9.1. Thus, the same amount of rule definitions and rule calls is involved, but additionally there are costs for the intersection operations. One variant will have unique variable names in both branches. In this case intersection will produce empty rule sets and no additional propagation rules remain. Another variant will have the same sequence of assignments in both branches. This does not change the local propagation in any manner, but the intersection afterwards will now produce a full set of rules. The third type of test input has m assignments that are all in the `then` branch, and a dummy statement in the `else` branch. Figure 22 lists three sample inputs. Note that the nesting does not play an important role here, since each intersection just operates on the local PropConst rules from that `if` block.

The two described branch-variants, one and two branches, were run for $n = 10$ and m varying from 100 up to 10000. The results have been compared to the measurements in section 9.1. Figure 23 shows that the intersection operations takes an additional 60% and 120% on top of the normal propagation costs.

At first sight, the difference between the two branch-variants seems strange. Intersecting two distinct sets of $m/2$ rules is about twice as expensive as intersecting a list of m rules with an empty list. The probable cause for this is that although the same amount of rules have to be compared in both cases, comparing a rule to nothing is cheaper than comparing a rule to another, different rule.

The costs of intersecting either two identical or two disjoint sets do not differ very much here. That is because each propagation rule has only one instance in the rule set. Intersecting thus comes down to walking over the set of rules and intersect two singleton lists for each element.

<pre> if cond then (f := 1; g := f; if cond then (a := 1; b := a) else (c := 1; e := c)) else (h := 1; i := h) </pre>	<pre> if cond then (c := 1; e := c; if cond then (a := 1; b := a) else (a := 1; b := a)) else (c := 1; e := c) </pre>	<pre> if cond then (f := 1; g := 2; h := f; i := g; if cond then (a := 1; b := 2; c := a; e := b) else 1) else 1 </pre>
---	---	---

Figure 22. Left: Tiger program with $n = 2$ if-blocks, each with $m = 1$ assignments in both branches, rule sets are disjoint. Middle: Tiger program with $n = 2$ if-blocks, each with $m = 1$ assignments in both branches, rule sets are identical. Right: Tiger program with $n = 2$ nested if-blocks, each with $m = 2$ assignments in just its (then) branch.

This benchmark shows that although the costs increase by a serious constant factor, behaviour is still linear. When the rule sets to be intersected contain multiple instances of rules the intersection of these instance lists results in quadratic costs (in the number of instances), but the number of instances is generally small.

Finally, the fix-point manipulation of rule sets, as discussed in section 7.2 basically comes down to a repeated application of rule set intersection. Hence, the costs will behave similar, multiplied by the number of fix-point iterations. For most applications this will be constant (2 or 3), but some delicate input programs can be created that cause the fix-point iteration to run just as long the actual loop in the

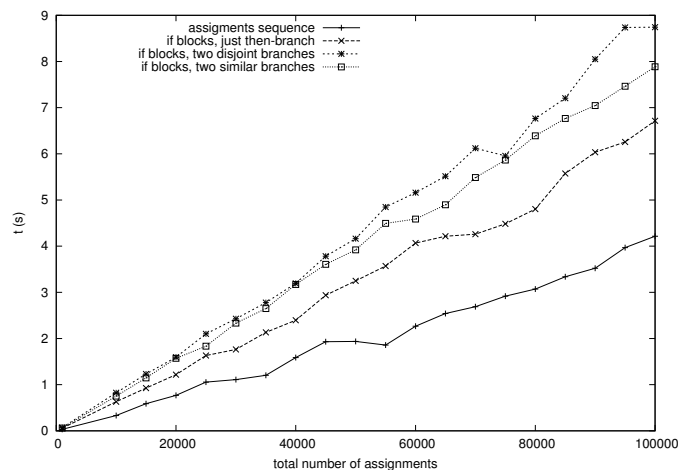


Figure 23. Performance of Tiger-PropConst on nested if blocks. Increasing length of assignment sequence tests the intersection costs.

Tiger program would!

9.5. Evaluation

The described benchmarks made several things clear. Costs are linear in the amount of rules and the nesting depth of scopes. This is due to the constant time lookups in the hash tables per scope. Costs of intersections are linear in the number of distinct rules in one scope. Besides, intersection costs are quadratic in the number of alternative, generated instances for one rule definition. This number is generally small though, so we do not consider this a problem.

What is more problematic is the representation of scope labels in a list attached to the scope. Upon labeled generation of a rule these lists have to be inspected, which takes linear time, whereas we required rule definition to cost $O(1)$. Replacing the list representation by a set data structure will improve efficiency here.

A final realization on the performed tests; the size of the Tiger programs that were transformed was not always too realistic. Programs with 600.000 assignments are not common. However, the tests were able to reveal certain qualitative behaviour of the performance, which is of fundamental importance when reasoning about the efficiency of a concept like dynamic rules. Quantative results depend on so much more factors of a transformation, and input programs are hardly ever of such regularity as the tests used here.

10. Applications

Scoped dynamic rewrite rules target a wide range of applications involving context-sensitive program transformations. The abstractions introduced by dynamic rewrite rules are very general, i.e. they are not specific to a certain kind of programming language or program transformation. This is shown by the successful application of dynamic rewrite rules in a great variety of non-trivial program transformations, especially program optimizations in compilers. In this section we discuss some of the applications of Stratego's dynamic rules in compilers for Tiger, Octave and Stratego itself.

10.1. Tiger Compiler and Tools

The Tiger compiler in Stratego [44] is used as a testbed for new approaches of implementing program transformations. The design space of dynamic rules has been discovered by implementing program transformations for Tiger in this compiler. Tiger [3] is an interesting language for this purpose, since it is a compact language, yet still includes some interesting features, such as nested functions with lexical scope.

Type checker In an abstract interpretation style type checker for Tiger, scoped dynamic rewrite rules are used to generate type checking rules for variables and functions. Thus, there is no need for threading type environments along traversals, and type rules can be expressed directly as rewrite rules.

Interpreter In the Tiger interpreter, dynamic rules are used to represent mappings from variables to values on the stack or heap. Variable bindings are dealt with using a scoped traversal similar to that of

the bound variable renamer. Globally visible heap objects are represented by an unscoped dynamic rule that maps reference values (pointers) to values. Evaluation for individual constructs is expressed using reusable constant folding rules.

Optimizers The Tiger compiler contains a variety of components that optimize Tiger programs. These components can be reused and composed at will. The implementation of constant propagation, copy propagation, and dead code elimination can be expressed elegantly using dynamic rules, as has been shown in this article. In forward transformation problems, dynamic rules rewrite variables to constants or to copy expressions. In backward problems, dynamic rules keep track of use/def and neededness information.

Instrumentation Students of our program transformation course have applied dynamic rules for the *instrumentations* of Tiger programs for tracing and profiling. They must apply dynamic rules to selectively extend functions with extra functionality. Also, the students apply dynamic rewrite rules in the implementation of a partial evaluator for a large, but pure, subset of Tiger.

Code Generation Dynamic rewrite rules have been applied in an implementation of *instruction selection* [9] using an algorithm based on dynamic programming [2], which is also used in the well-known BURG [14] code generator. In this implementation the mapping of an intermediate representation (IR) to machine instructions is represented by means of rewrite rules. Dynamic rewrite rules are used to keep track of the best tile for a given IR tree.

10.2. Stratego Compiler

Dynamic rules are used extensively in many parts of the Stratego compiler itself. Some examples are

- Bounds/unbounds analysis determines whether or not a variable is bound at some program point. This analysis is used in static checks and in optimizations.
- The Stratego inliner uses dynamic rules of course.
- Innermost fusion is a domain-specific optimization that generates efficient code for applications of the generic strategy `innermost` [17].
- The dynamic rule lifter uses dynamic rules to keep track of context variables.
- The C generation back-end uses dynamic rules for memoization of common terms.

10.3. Octave Compiler

Another ongoing project in which dynamic rules are used extensively is a compiler for Octave written in Stratego.

Parsing and Disambiguation For parsing and disambiguation of a complete Octave program, it is necessary to know all the functions that are invoked in a program. Octave defines a function per file and is a declaration free language. Identifiers in Octave can represent variables or function calls. Moreover, an identifier can represent a variable in a function and be a function call in another function. A mechanism to determine to which namespace an identifier belongs is needed. Octave assignments are closest to variable declarations. Thus, an occurrence of an identifier in the left-hand side of an assignment defines it as a variable. At this point, a dynamic rewrite rule is created to state that the identifier is a variable. This dynamically defined rule is used to disambiguate further occurrences of this identifier. Remaining identifiers can be user-defined functions or built-in functions. With information of built-in functions, complete Octave program disambiguation is achieved.

Online Function Specialization Octave function definitions are flexible. Octave function signatures define a number of incoming or returning arguments, however, a function call can provide a different number of arguments or specify a different number of returning values. To achieve a cleaner internal representation in the Octave compiler, functions are tailored to match the function call. Dynamic rules are used to bind the function definitions at the function call points, and to propagate available expressions. A specialized function is memoized.

Octave Type Inferencing Type inferencing is specified in an abstract interpretation style, comparable to the type checker for Tiger. Function calls in Octave can have context-dependent behaviour. A function call can yield different number of results as is defined in the left-hand side of an assignment. A dynamic rewrite rule provides the number of expected results to the function call. With this information proper types are inferred.

Optimizations A growing number of (data-flow) optimizations are included in the compiler, such as constant propagation, global constant propagations, dead code elimination.

11. 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 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.

11.1. 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 hashtable to have efficient lookup of information for a symbol. Since symbol tables are concerned with

names, they have to handle the scoping rules of the subject programming language. For example, the symbol table implementation used in Appel's Tiger compiler [3] remembers the state of a hashtable 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 is efficient and general enough to handle all kinds of scope 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 fix-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 definitely comes at a cost, since basically nothing can beat the compactness and performance of operations on bit vectors.

Value Numbering *Value numbering* is used in a wide range of program optimizations. Initially, it was a method used for common sub-expression 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 hashtable, 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 hashtable, of which the key is based on the dynamic part of the left-hand side the rewrite rule.

11.2. Language Independent Traversals

Language independent traversals [35] implement some program transformation of which the traversal is independent of a specific subject language. Language independent 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 [42] and are parameterized with strategies for handling language specific issues, such as the representation of variables and the constructs for binding of variables.

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 subject program and in both cases the traversals are more difficult to understand and maintain.

Scoped dynamic rewrite rules are a very useful language extension for making these language independent traversals more concise. First, the threading of context-sensitive information can now be dele-

gated 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 tiresome handling of possible name conflicts if a separate global table of information is constructed first.

The traversal functions in ASF+SDF [8] 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 tiresome 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. This is a powerful abstraction for implementing context-sensitive program transformations.

11.3. 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 make 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. Instead, 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 very 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 right-hand side of a rewrite rule. This means that clauses are 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 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. This 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. Unfortunately, in dynamic rules there is at this point a distinction between static and dynamic rewrite rules.

Static rules with the same name cannot be applied to produce all alternative solutions. Stratego has a *global-choice* operator (`++`) and there is a strategy combinator for exhaustive application to produce all possible results [9] (`bagof`). However, static rewrite rules with the same name are combined by the *local-choice* operator (`+`). This combinator commits the choice for a strategy argument if it succeeds.

In contrast, all possible results of applying a dynamic rule can be produced. However, this must be done with a special purpose strategy `bagof-L`, which immediately returns all results. Global backtracking can not be used to consider all alternative applications, although applying a continuation with a global choice over the actual results is possible. Moreover, Stratego's `bagof-L` only applies rewrite rules in a single scope. This was indeed the semantics of `bagof` that was needed for our use cases from the domain of program transformation, but it would be useful to allow some more flexibility in applying sets of rewrite rules.

A useful extension of the current implementation would be to allow a user-definable strategy combinator for static and dynamic rewrite rules. For static rules there does not seem to be an immediate need for this, but the alternative ways in which dynamic rules can be applied to produce a list of possible rewritings indicates that there is a need for user-definable combination there.

11.4. 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 then discuss 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 name concerns the visibility of it, 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 the end of it. In other words, names are evaluated in the environment of the definition. 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, Scheme), 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 all 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 my 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, Scheme provides 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 variable. Instead, it temporarily assigns a different value to variables that have already been defined in some outer scope. This bindings 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. Dynamically scoped 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 used 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. A 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 its own copy of the variable: it can set the value of the thread local variable, but this the value of the parent's variable will not be modified.

XSLT's Tunnel Variables Most recently, Schadow proposed dynamically scoped variables for XSLT [31]. This proposal has been accepted for XSLT 2.0 [19], where they are now called tunnel parameters. A parameter of a template can be defined as a tunnel parameter. Tunnel parameters are then 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.

11.5. Fresh O’Caml and FreshML

FreshML and Fresh O’Caml [32, 30] 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 can not be transformed with the variable binding facilities of FreshML.

12. Future Work

Generic Programming The current syntax for dynamic rewrite rules does not allow abstraction over dynamic rule names. In other words, it is not possible to implement generic strategies that are parameterized with dynamic rule names, or rather, this is not possible without falling back on the internals of the dynamic rules implementations. If abstraction of rule name is possible, then there might be more opportunities for implementing generic traversals, although scope is typically a language specific issue.

Furthermore, we would like to find out how the implementation can offer the atomic operations of dynamic rule application, without exposing too much of the internals of dynamic patterns and closures. Providing these atomic operations is useful to enable the implementation of alternative dynamic rule application strategies. It is currently not possible, without extending the Stratego compiler, to implement variants of dynamic rule application. The `bagof-L` operator shows that there is a clear need for this, since it only applies dynamic rewrite rules from a single rule scope, which is not always desired. This indicated that there are several variation points bound in the `bagof-L` operation. The dynamic rule facility should allow the implementation of variants of such dynamic rule applications, based on some more atomic operations.

Dynamic Scoping Dynamic Scoping is an important aspect of dynamic rewrite rules, however, it may also give rise to unexpected behaviour. The dynamic rule environment is not part of the closure of a dynamically defined rewrite rule. Therefore, an application a dynamic rule L will be applied in the current dynamic rule environment, and not the one of the definition-time of the applied rule L . However,

for some application it is required to have the dynamic rule environment as a part of the closure. This is already possible, but again this requires the use of internals of dynamic rules. Some abstraction might be useful at this point.

Object Variables In a dynamic rule definition only the static parts of the left-hand side can abstract over object values by either using wildcards or term variables. The dynamic parts of the left-hand side are closed terms, hence they cannot abstract over the object values and come down to a literal match. In some cases this is undesirable, since generation time values are not literally wanted as the part of the left-hand side.

This occurs for example in Wadler’s deforestation algorithm [43]. The algorithm eliminates intermediate terms in a program and does so, amongst others, by inlining functions. To prevent infinite regression during transformation, helper functions are generated when a function call is encountered that was inlined before already. The algorithm can be expressed using rewrite rules and a simple strategy. Dynamic rules can be used to implement the folding of recursive occurrences of the function composition being deforested. However, this requires abstracting over object variables. Whenever some function call with a unknown number of arguments `FunApp(f, a*)` is encountered, the dynamic rule to be generated should have as its left-hand side this same function call, but with any variable names in the argument terms replaced by wildcards, or fresh term variables. For example:

```
?FunApp(f, [BinOp("+", Var("x"), Var("y")), Int("3")])
; rules(RecursiveHelper: FunApp(f, [BinOp("+", u, v), Int("3")]) -> ...
```

would make sense, since at the call site a function call of the same structure, but with possibly different values for the `BinOp` arguments.

The problem is that only at runtime the actual shape of the input term is known, so no sensible left-hand side can be put in the rule definition. There are two approaches that still achieve the goal. At the points where wildcards are actually needed, dummy terms could be inserted. These serve as ‘closed term-wildcards’. An other option is to ‘pre-match’ the left-hand side term at the definition site, and insert static wildcards or fresh variables where appropriate. In the above example, this would come down to:

```
?FunApp(f, a*)
; rules(RecursiveHelper: FunApp(f, a'*) -> ...
      where <is-renaming> (a*, a'*)
```

Note that almost all matching will now be performed in the condition (`is-renaming` is a custom strategy here), and not in the left-hand side of the dynamic rule. The disadvantage of this is that the dynamic part of the left-hand side (i.e. the key in the hash table) is not very expressive, so more closures might be indexed by this same key. The advantage is that no effort has to be put in getting the dummy terms into the left-hand side, both at definition and call site. Besides, the ‘matching’ in the condition is much more powerful than when using dummies in the left-hand side, since the latter merely represent structural information, no value information.

The above has been successfully applied in a Stratego implementation of Wadler’s deforestation programs. The implementation also shows the use of rule set extension, bagof-rules and dynamic identity rules.

13. 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, online partial evaluation, and dead function elimination. The techniques are equally well applicable to many other program transformations.

Acknowledgments We would like to thank Patricia Johann, Oege de Moor, 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.

References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools*. Addison Wesley, Reading, Massachusetts, 1986.
- [2] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.
- [3] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- [4] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.
- [5] 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.
- [6] 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. Birkhäuser, 1997.
- [7] 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.
- [8] M. G. J. van den Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, April 2003.
- [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. Lämmel, 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] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
- [14] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.
- [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] P. Johann and E. Visser. Strategies for fusing logic and control via local, application-specific transformations. Technical Report UU-CS-2003-050, Institute of Information and Computing Sciences, Utrecht University, February 2003.
- [18] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [19] 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/>.
- [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 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.
- [26] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.
- [27] R. Paige. Future directions in program transformations. *Computing Surveys*, 28A(4), December 1996.

- [28] 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.
- [29] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.
- [30] 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, Heidelberg, 2000.
- [31] G. Schadow. Request for dynamically scoped variables in XSLT. <http://lists.w3.org/Archives/Public/xsl-editors/2002JanMar/0002.html>, 2002.
- [32] 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.
- [33] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] E. Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
- [38] 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.
- [39] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Spinger-Verlag, June 2004. (To appear).
- [40] E. Visser. A survey of strategies in rule-based program transformation systems, March 2004. (Draft).
- [41] E. Visser and Z.-e.-A. Benaïssa. 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.
- [42] E. Visser, Z.-e.-A. Benaïssa, 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.

- [43] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
- [44] <http://www.stratego-language.org/Tiger>.

Contents

1	Introduction	1
2	Program Transformation by Term Rewriting	6
2.1	The Tiger Language	6
2.2	Representing Programs as Terms	8
2.3	Term Rewriting	9
2.4	Concrete Syntax	9
3	Rewriting Strategies	10
3.1	Syntax and Semantics	11
3.2	Matching and Building Terms	13
3.3	Strategy Combinators	15
3.4	Strategy Definitions	18
3.5	Generic Term Traversal	19
4	Defining Rules Dynamically	23
4.1	Example: Constant Propagation in Basic Blocks	23
4.2	Semantics: Defining and Undefining Rules	26
5	Dynamic Rule Scope	26
5.1	Example: Bound Variable Renaming	27
5.2	Example: Function Inlining	29
5.3	Semantics: Scope	30
5.4	Example: Constant Propagation for Local Variables	32
5.5	Semantics: Labeled Scopes	33
6	Extending Dynamic Rules	34
6.1	Example: Common Subexpression Elimination in Basic Blocks	35
6.2	Semantics: Extend Rule	36
6.3	Example: Function Specialization	37
7	Intersection and Union of Rule Sets	41
7.1	Example: Constant Propagation with Control Flow	41
7.2	Semantics: Intersection and Union of Dynamic Rules	44
7.3	Example: Dead Code Elimination	45
8	Implementation	46
8.1	Static and Dynamic Patterns	46
8.2	Dynamic Rule Application	47
8.3	Scoping Rule Sets	49
8.4	Forking and Change Sets	51
8.5	Implementation Costs	51

9	Performance of Dynamic Rules	51
9.1	Benchmark: Definition and Application	52
9.2	Benchmark: Dynamic Rules Scope	53
9.3	Benchmark: Scope Labeling	55
9.4	Benchmark: Dynamic Rules Set Intersection	55
9.5	Evaluation	58
10	Applications	58
10.1	Tiger Compiler and Tools	58
10.2	Stratego Compiler	59
10.3	Octave Compiler	59
11	Related Work	60
11.1	Data-Structures in Program Optimization and Compilation	60
11.2	Language Independent Traversals	61
11.3	Runtime Extension of Logic Programs	62
11.4	Dynamic Binding	63
11.5	Fresh O’Caml and FreshML	66
12	Future Work	66
13	Conclusion	68