Extending Dynamic Rules

An Application Oriented Study Into Stratego's New Dynamic Rules

A. van Dam
adam@cs.uu.nl

MSc Thesis

INF/SCR-04-25

Center for Software Technology,
Institute of Information and Computing Sciences,
Utrecht University,
P.O. Box 80089, 3508 TB,
Utrecht, The Netherlands.

August 26, 2004

Preface

"And here we are...", presenting to you a master's thesis for the second time. Being a graduated master student in Computational Science (2002), who always wants to keep in touch with practical applications, I continued a second master studies in Computer Science. The course on Program Transformation by Eelco Visser was quite an eye opener. My main interest in computers and programming has always been to let it work for me, compute, generate, and that just happens to be all contained in program transformation.

To combine both applied mathematics and program transformation, it appeared like the perfect project to work in CodeBoost on optimizing transformation of Sophus programs. Sophus is a software library which aims to model the structure of systems of partial differential equations and their solution techniques, without fixing the coordinate system in an early stage. CodeBoost is a C++ transformation framework, currently mainly aimed at Sophus. Eelco Visser had some good contacts with the project members in Bergen, Norway and arranged a short stay.

In the following months, I created some example optimizations, but mainly exercised to become entirely familiar with many subparts of the Stratego project. It started with exploration of the SSL, making extensive use of the excellent xDoc documentation. Next was some new work on creating developer tools for Stratego users. Better control over debug output and inspection of the inner workings of Stratego's dynamic rules. In the end I even found myself debugging and contributing patches to the Stratego compiler.

At the end of 2003, it slowly became clear that the Sophus software wasn't quite ready for my research yet, and an extensive redesign was planned for CodeBoost as well. For these reasons, a new direction was found for my master's project. Having worked as an 'end user' with the old dynamic rules, I had already met with some of their limitations. Reasoning about this, and much inspiring brainstorming with Eelco, Karina and Martin led to the design and implementation of the new dynamic rules. Some early ideas on a redesign of dynamic rules were expressed during the Fifth Stratego User Days in March 2004, the resulting discussions even influenced some of the later design decisions. The new dynamic rules are an accomplished fact now and we have described them in a joint paper, a somewhat hectic but challenging task. My own research further focused on two example applications, built around two papers. Together with the ongoing research on the various optimizing Tiger transformations, they formed good case studies into the use and efficiency of the new dynamic rules.

Looking back, my project has lasted quite long. The reason is a good one: getting enthusiastic when making use of Stratego, slowly becoming a 'team member', contributing, helping new Stratego users, it is all very encouraging. Although still an academic project, it gives good experience in practice with operating within a large project where one can't know all the parts himself, communicating with colleague developers, the regular releases, introducing new tools to improve workflow, and more.

I must say, I have had a nice time at the ST-lab, interesting to see what each student is working on, all in different phases of a master's project. The lab-movies, four-o'-clock-soup, the countless litres of coffee, and the bizarre range of musical genres that came by, unforgettable. Thanking each one individually would yield another page of anecdotes, so in short: We all know what great fun it's been, guys, thanks!

Of the professional people I have to thank, Martin Bravenboer and Rob Vermaas are the first. After Eelco, Martin knows the most about Stratego, and especially in my initial learning phase, he's been very helpful. Rob contributed to my steep Stratego-learning-curve by having made xDoc, a generator of time-saving API documentation and visual code browser. We did some good extreme programming on the Stratego compiler as well.
From the more informal talks at the ST-lab, some good ideas arose as well. Jurriaan Hage hinted on the fixpoint counter example (page 57), and Alexey Rodriguez provided interesting insight into higher-order AGs and AG views.
Outside of Utrecht, I want to thank Otto Skrove Bagge for being my helpful host in Bergen, and Magne Haveraaen at Bergen University for providing such comfortable housing.

Last, but not least, my supervisor Eelco Visser, you have let me do my research quite independently, but gave the crucial hints where needed. You arranged a superb three week stay in Bergen, Norway, before my project had even really started. Also, letting me cooperate on compiler implementation, paper writing and more, was really encouraging. Thanks.

Arthur van Dam
August 2004

Contents

Chapter 1

Introduction

Program transformation comes in many variants. Input and output programs can be in the same or in different languages, the transformation can be an optimizing one or generalizing/specializing. A more complete and structured discussion can be found in the survey by Visser [Vis04b]. We focus here on source-to-source optimizing transformations.

Optimizing transformations receive the input program in a structured representation that still contains all syntactic information. By analyzing the shape and contents of the input tree, parts of the input can be transformed into more optimal parts, after which the structured representation can be pretty-printed again in the original object language, thus yielding a transformed program.

Whether a transformation yields optimized terms depends on the purpose of the transformation. Constant propagation tries to propagate assignments of constants to variables onto their usage sites, thus helping the often combined constant folding to evaluate simple (e.g. arithmetic or boolean) expressions. The more expressions that can be evaluated at compile-time, the better.

Maintaining call counts for functions and either inlining or eliminating their definitions yields a smaller program without some of the function calling overhead. Dead code elimination is also a good way of making programs smaller, while still preserving the semantics of the program.

Type inferencing is not an optimizing transformation in the sense of performance. Instead it 'enriches' programs by inferring types and annotating expressions and declarations with their type.

The research described in this thesis is part of ongoing work in the development of the Stratego system [VBT98, Vis00, Vis04a]. Focus lies on the concept of dynamic rules that allow the use of context information in rewrite rules, thus making more sophisticated rewriting possible. Dynamic rules were introduced in Stratego 0.6, and had some occasional improvements up to and including Stratego 0.9.3. We will sometimes refer to this 0.9.3 release as the 'old-style dynamic rules'. In Stratego 0.9.4 the `extend rules` feature was already silently introduced, but a major redesign including many new features was released in Stratego 0.10, followed by some small internal optimizations in 0.11. At the time of writing Stratego 0.11 reflects the current state of our research.

1.1 Static Rewriting in Stratego

Stratego as a program transformation language is part of the full fledged program transformation framework StrategoXT. The basis for XT is formed by the ATerm library. The Annotated Term format [vdBdJKO00] describes an abstract data type for tree-structured data and is memory efficient thanks to maximal sharing of subterms and automatic garbage collection. Besides, the binary exchange format makes it suitable for efficient program transformation systems. Next, the syntax definition formalism SDF2 [Vis97, dBSVV01] allows for a very concise definition of the programming language or data format under consideration (the object language). From an SDF2 syntax definition a parser can be generated that will parse an input file into an ATerm, which can directly be fed into the actual transformation executable. Finally the Generic Pretty Printer package [J00] generates pretty-printers for various output formats, thus forming the tail of the transformation pipeline.

Basic program transformation can be easily expressed using term rewriting. Stratego employs static rewrite rules for this. A rule `p1 -> p2 where s` denotes that a program fragment matching pattern *p1* can be replaced by its instantiated form of *p2*, with optional conditions or helper computations in `s`. For example, the following rule evaluates the sum of two constant values, as part of the *constant folding* transformation:

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

Note that terms between |[ ]| are intuitively expressed using *concrete syntax*, which will be discussed shortly in Section 2.1.2.

Rewrite rules are typically local operations, that only transform small parts of an input tree. Exhaustive application of a set of rules to transform an entire tree is not effective in most cases, since the system may be non-confluent or even non-terminating.

Instead, Stratego employs programmable rewrite strategies to significantly enhance the specification of a transformation. Strategies operate on terms as well, but may additionally distinguish the structure of an input term and define which rewrite rules should be applied to certain subtrees and in what order. Strategies thus take care of a certain traversal of the input tree, whereas rewrite rules maintain their local behavior. The separation of rules and strategies makes an intuitive specification of transformations possible.

An even more powerful way of traversing a tree is by means of *generic traversals*. Stratego offers some first-class constructs than can break up an arbitrary term and steer any parameter strategy into one or more of the child terms. Transformations in Stratego thus can be reused for programs in different object languages, by slightly adapting some strategies specifying only the relevant syntactic structure of the object language.

1.2 The Need for Context Information

Rewrite rules, when applied to some input term, only have knowledge of the local structure and contents of a term. Information from other parts of the tree being transformed are unknown. This limits the power of static rewriting to the basic laws of e.g. mathematics, logic or program analysis:

```
AndCommute : |[ p & q ]| -> |[ q & p ]|

ElimIf : |[ if true then e1 else e2 ]| -> |[ e1 ]|
```

The above rules are part of *expression simplification* and *unreachable code elimination* respectively.

More sophisticated rewriting needs context information. Consider for example the unfolding of function calls, i.e. the inlining of function bodies at their call sites. At the call site the body of the function is unknown, since the function definition is probably at some entirely different position in the tree. A normal rewrite rule thus does not have the necessary information available for inlining at the call site.

Several approaches have been made to get context information to the point where it is needed. The information could be passed on as parameters to the various strategies and rules up to the rule that needs it. This requires manual manipulation of all strategies on the execution paths that lead to a certain rule. If the transformation needs to be changed or extended, all strategy calls need to be modified as well. It is evident that this is very impractical.

Attribute grammar systems such as UUAG [SAS98] feature the concept of copy rules that automatically pass on inherited or synthesized attributes to the children or parent terms respectively. This is however strictly based on term structure (i.e. scope and data flow of the object language) and not on a traversal environment (i.e. the transformation itself).

ASF+SDF [dBKV03] allows the declaration of *accumulation parameters* that are automatically carried along during a transformation. This complicates the data structures being transformed.

The above approaches thus complicate strategy definitions, proper scoping or inheritance of context information, and term structure, i.e. the pure rewriting nature of terms is lost.

A more intuitive approach is the one of *contextual rules* that allow specification of patterns with their replacement values, e.g. in the case of function inlining:

```
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 ]|
```

Since function calls can be nested deeply inside the body of the `let` expression, a local traversal is needed to find them. A contextual rule `p1[q1] -> p2[q2]` rewrites a term t matching `p1` to the instantiated `p2`, and additionally performs a topdown traversal on the matched part of t, and rewrites occurrences of `q1` to `q2`. The local rewrite rules thus has a traversal of its own. Being itself part of a complete traversal, this rule leads to quadratic complexity. Note that the contextual pattern `p1[q1]` need not be the entire left hand side, in the inlining example above it is `e2[f(e3)]`.

Keeping context information available – as discussed for attribute grammars and accumulation parameters – is better than the expensive contextual rules, but it should not pollute the rewriting specification. The idea behind dynamic rules is to maintain a separate environment that contains context information, always available, but not showing up in the literal rewriting specification or terms under transformation.

1.3 Dynamic Rules

Dynamic rules are a way of representing a context environment by adding context information to it at runtime and using it for rewriting at any time in the transformation. This is realized by runtime definition of rewrite rules. Hence, besides the existing static rules from the original Stratego source program, new rewrite rules can be added to the rule set environment during execution. At the definition site, context info (i.e. variable bindings) may be used in the rule definitions, it is stored and kept available for use at the call site. Whereas the use of contextual rules states a static rewrite rule in the original source, now an inlining rule is defined at runtime for each function definition encountered:

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

The above strategy matches any function definition and defines a rewrite rule `UnfoldCall` for that specific function *f*. The variables *f*, *x* and *e1* are bound in the definition context of the rule (and stored in the dynamic rule environment as such), whereas *e2* is unbound at definition time and represents a call time variable.

The maintenance and use of this context environment is handled transparently by the dynamic rules implementation. The programmer just uses the `rules(..)` construct, and is able to use *dynamic rule scopes* such that rules are automatically retracted from the environment upon exit of that scope. Rules can also be *undefined* in a scope itself, thus hiding the context information to prevent further use of it. *Redefining* rules allows for renewing the context information (e.g. when the function body of an inlineable function has been transformed itself, thus requiring a new definition of `UnfoldCall`). *Overriding* rules in outer scopes is a way of redefining existing rules without losing them upon exit of the inner scope.

Dynamic rules are especially suitable for use in data flow optimizations, previous examples already showed some fragments of constant folding and function inlining, other optimizations include dead code elimination, loop fusion and common subexpression elimination. Since many of the (data flow driven) context information closely follows the scoped structure of the input, integration of the actual traversal and the definition and scoping of dynamic rules is almost seamless.

Other uses of dynamic rules are for example the Tiger interpreter (using dynamic rules for mapping variables to their global value on the heap), the Tiger type checker and TFOF type inferencer (mapping type variables to declared/inferred types) and the Stratego interpreter (mapping both pattern variables and strategy variables to their value or closure respectively).

1.4 Extending Dynamic Rules

The above describes dynamic rules in the 'old style'. Especially the *override rules* construct resulted from a very pragmatic approach to dynamic context. In the new dynamic rules all

existing concepts have been formalized, resulting in theoretical semantic rules.

New concepts were added, such as *extending* a rule set, allowing multiple rewrite rules with the same left hand side to exist alongside each other, with the possibility of retrieving all possible rewritings of a term in a list (*bagof* rewritings).

A more fine-grained scoping mechanism was designed, introducing the labeling of scopes. Rule scopes are identified by the rule name they are associated to *and* an optional list of labels. For nested `let` blocks the rule scopes may for example be labeled with the variable identifiers from the declaration section of each respective `let` block.

Previously, in data flow optimizations a certain technical knowledge of rule sets and available API strategies was needed to split the traversal into two or more paths and compute the intersection or union of the rule sets afterward. Now, forking the rule set and meeting the various paths afterward is done almost transparently using simple syntax. Moreover the performance of rule set joining is much better

Also for data flow optimizations, fixpoint meeting of rule sets is now possible. For example when constant propagating over a `while` loop, the propagation rule sets need to be intersected repeatedly until no changes occur anymore, thus guaranteeing that only valid propagations after any number of iterations remain after exiting the loop.

In general, the implementation of dynamic rules has been entirely revised and made more efficient, resulting in a more mature language feature for the Stratego language.

1.5 Outline

To describe the context in which this research was performed, Chapter 2 introduces the key parts of the Stratego language. Readers who are already familiar with the Stratego language constructs may skip this chapter, although the formal semantics described within will come back in the semantics of dynamic rules. Appendix A contains all formal semantics in one overview. Chapter 3 further explains the need for context-aware rewriting and describes the dynamic rule constructs that were available in the 'old-style dynamic rules', i.e. as in Stratego 0.9.3 and before.

Chapters 4, 5 and 6 describe three cases where dynamic rules are used in some optimizing transformation. Starting the reasoning from the old-style dynamic rules, undesirable or missing features of these dynamic rules are identified, and a solution described. Chapter 4 introduces a small lambda calculus and a *shrinking optimizer*, based on an article by Andrew Appel and Trevor Jim [AJ97]. Basic rule definition and rule scoping is of importance here. Chapter 5 describes constant propagation in the Tiger language. Previous work on this matter was done by Olmos and Visser [OV02], but new dynamic rules enable a cleaner implementation of constant propagation. Rule undefining, dynamic scoping and proper handling of control flow (i.e. forking and meeting of dynamic rule environments) are the important issues here. Chapter 6 is a study built around the deforestation algorithm for first-order functional languages described by Philip Wadler [Wad90]. Implementation of the deforestation algorithm shows the need for extending rule sets an higher-order matching. Within the same study, also a type inferencer was implemented, which allows for a comparison between type inferencing using attribute grammars (e.g. as implemented [DS01] in

the UUAG system) and type inferencing using dynamic rule sets. Both implementations are based on the $\mathcal{W}$ algorithm by Milner [Mil78, DM82]. To keep the original chapter focused on deforestation, the type inferencer is described separately in Appendix B.
Each case thus introduces one or more improvements or new features of dynamic rules in Stratego, and altogether they form the new set of dynamic rule constructs as they were initially released in Stratego 0.10.

Most new features of dynamic rules have been made possible by a smarter representation of rule sets and scopes. Chapter 7 describes how stacks of hash tables provide the appropriate functionality *and* efficiency. The supposed efficiency of the new rule set representation has been benchmarked. Chapter 8 describes these benchmarks, compares the results to the expected qualitative costs behavior and concludes which constructs are the most costly.

Chapters 9 and 10 describe related and future work and Chapter 11 concludes.

Chapter 2

Stratego: Syntax and Semantics

Stratego forms the basis for all research described in this thesis. This chapter will provide an (incomplete) overview of the language, highlight some relevant concepts and introduce some notation used elsewhere in this thesis.

The object language under consideration in this chapter is Tiger, unless stated otherwise. Tiger is the example language from the compiler construction textbook by Appel [App98], and is further introduced in Section 5.1.

2.1 Term Rewriting

Transformation of a syntax tree is based on rewriting of terms. Rewriting by means of rewrite rules tries to match a term against the left hand side of a rule, and then replaces it by the instantiated right hand side of the same rule.

2.1.1 Terms, Patterns and Variables

In Stratego, programs are represented using the ATerm format [vdBdJKO00]; each syntax tree (either abstract or concrete) can be represented as an ATerm. The basic form[1] of a term is one of the following:

- an integer, real or string constant,
- a constructor application $c(t_1, \ldots, t_n)$ to n terms,
  e.g. `BinOp("+", Int(3), Var("y"))`,
- a list of terms, e.g. `[A, B, C]` or tuple of terms, e.g. `(A, B, C)`.

Unless stated otherwise t or any t_i represents a term.

An additional feature of ATerms is the possibility of attaching a list of terms as annotations (The 'A' in 'ATerm' stands for 'Annotated'). This allows for moving some meta information around when transforming trees. An annotated term is notated as $t\{anno1, anno2\}$.

A *term pattern* describes a set of terms, by introducing zero or more pattern variables (or meta variables), such as x in a term. A pattern variable itself is also a term pattern.

[1] Actual ATerms are a bit different than sketched here, but the overview reflects the relevant forms here.

The term pattern `BinOp("+", Int(`x`), Var(`y`))` represents all expressions where some integer is added to some variable. Note that y here is a meta variable, whereas in the preceding item list `y` is an object variable. Object variables are literal variables from the underlying object language or program. The difference between meta variables and object variables is sometimes subtle, but crucial. In formal notation, p and q typically represent patterns, whereas x, y and z (including indexed variants, x_i, evidently) typically represent meta variables. Meta variables in both abstract and concrete syntax are typeset in typewriter italics (i.e. *x*, *y*, etc.) as some preceding terms already showed.

2.1.2 Concrete Syntax

A parsed input program is an abstract syntax tree. Writing abstract syntax can become quite verbose as the previous section already showed. Concrete syntax is the syntax of the object language itself. A few examples of concrete syntax translated into abstract syntax are:

```
|[ 1 + x ]|                          ≡  BinOp("+", Int(1), Var(x))
|[ if e1 then e2 else e3 end ]|      ≡  IfThenElse(e1, e2, e3)
|[ fadd a b = a + b ]|  ≡  FunDef(
                             "fadd"
                           , [Var("a"), Var("b")]
                           , FunApp("(+)", [Var("a"), Var("b")])
                           )
```

The first two examples are in the Tiger language, the third is in the TFOF language, which is introduced in Section 6.1. Note that abstract syntax is not always the most verbose as the second example shows, but in general, concrete syntax allows for a much more clear and intuitive specification of terms and patterns.

Specifying a rewrite system is greatly facilitated by using concrete syntax of the object language at the rewriting level [Vis02]. To use concrete syntax in Stratego quotation markers are required to specify the transition from meta level syntax (Stratego) to object level syntax (language of input program). In this case `|[ ]|` is used to delimit the concrete parts. When concrete syntax is used, the Stratego compiler translates all concrete term patterns to equivalent abstract term patterns and then continues with the normal compilation process.

As said, fragments in concrete syntax need not represent closed terms, they may also contain meta variables. The first two examples above contain meta variables, whereas the third example only contains object variables. The specification of the language embedding determines which characters in concrete syntax are parsed as meta variables, the rest is parsed as the underlying language.

2.1.3 Substitution, Matching and Rewriting

Term patterns can be instantiated by substituting their variables with terms. A substitution σ is a mapping from meta variables to terms, which can be applied to a term pattern to yield the instantiated term. A finite mapping for n variables is denoted as $[x_1 := t_1, \ldots, x_n :=$

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

Figure 2.1: Some rewrite rules for Tiger arithmetic expressions

$t_n]$. The application of a substitution σ to a term pattern follows the rules below:

$$
\begin{aligned}
[\ldots, x_i := t_i, \ldots](x_i) &= t_i, \\
\sigma(x_j) &= x_j && \text{if } x_j \notin [x_1, \ldots, x_n] \\
\sigma(x_j) &= [x_1 := t_1, \ldots, x_n := t_n](x_j) && \text{if } x_j \in [x_1, \ldots, x_n] \\
\sigma(c(p_1, \ldots, p_m)) &= c(\sigma(p_1), \ldots, \sigma(p_m))
\end{aligned}
$$

Note that to fit in this definition, the list and tuple constructs are in fact just special constructor applications. Numeric and string literals are of course not affected by the substitution. The result of such a substitution applied to a pattern is itself a pattern as well if not all of the variables have been substituted. A term pattern is sometimes referred to as a term. Whenever a pure term without any variables is meant, we speak of a *closed term*.

Matching a term t against a term pattern p amounts to finding a substitution σ such that $\sigma(p) = t$. A pattern p matches a term t if such a σ exists.

Basic rewriting can be performed by means of rewrite rules. A rewrite rule is a pair of term patterns, named by an identifier, with an optional condition and is denoted as $f : p_1$ -> p_2 `where` s. Application of this rule to a term t succeeds if p_1 matches t and the condition s succeeds as well. The resulting term t' is obtained by instantiating p_2 with the same substitution, i.e.: $\sigma(p_1) = t$, and $\sigma(p_2) = t'$. Note that σ also may contain variable mappings that originate from the condition s. If p_1 does not match t, application of the rule fails. The rules in Figure 2.1 evaluate the addition and multiplication operator for constant values: Variable `e` matches any Tiger expression, which can be lifted out of the zero addition expression. Variables `i` and `j` match integer constants, after which Stratego's built-in `add` and `mul` strategy compute the result and associate it to a new variable `k`. The notation <s> p indicates application of s to p, and s => v binds the result from s to v, that is if v was not already bound. The right hand side of the rule can now use this new variable to yield the computed value as result.

2.2 Strategies and Rewrite Environments

Rewrite rules are quite simple and not too powerful rewrite constructs for large transformations. Instead, Stratego implements *programmable rewrite strategies*, which add much functionality and rewriting power to the language. The next section discusses how strategies control a rewriting process, but first we will introduce some of the most basic constructs in Stratego along with a way of describing operational semantics formally.

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

Figure 2.2: Syntax of core Stratego.

2.2.1 Syntax and Semantics

The Stratego language is split into a core language with all the first-class constructs and syntactic abstractions defined in those constructs. Not all constructs will be discussed in this text, but Figure 2.2 provides an overview of core Stratego, whereas Figure 2.3 lists most relevant extensions (or sugar) in the language.

The execution of a Stratego program is controlled by strategies. Even rewrite rules are desugared to a combination of simpler strategies. Each strategy can be considered a small program that transforms its input term to some output term, or fails. A recent article by Visser et al. [BvDOV04] defines the operational semantics of Stratego's language constructs. This prescribes the behavior of a strategy applied to an input term, hereby taking the variable environment $\mathcal{E}$ and system state Γ into account.

An application of strategy s to subject term t within the context of state Γ and environment $\mathcal{E}$, either yields a term t' with a possibly updated state Γ' and environment

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

Figure 2.3: Extensions (sugar) of the syntax of core Stratego.

$\mathcal{E}'$:

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

or fails:

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

with the special value $\uparrow$ denoting failure. The notation $\bar{t}$ indicates either a normal term, or failure. In all operational semantics, $\equiv$ denotes semantic equivalence.

State Γ models the system of dynamic rules, as discussed in Chapter 3 and beyond. Variable environment $\mathcal{E}$ will be introduced now.

2.2.2 First-class Pattern Matching and Building

Rewrite rules as described in Section 2.1.3 are not the most basic operations in Stratego. Three first-class constructs in Stratego are the *match*, *build* and *scope* constructs.

Environments The variable mappings σ as described in Section 2.1.3 are maintained as a variable environment $\mathcal{E}$ in Stratego. An environment is not a *set* of variable mappings, but a stack of variable mappings instead, i.e. multiple mappings for variables may exist, and the order determines the result of substitutions. An environment is typically represented as

$\mathcal{E} \equiv [x_1 \mapsto \overline{t_1}, \ldots, x_n \mapsto \overline{t_n}]$ and an application of it to a variable x behaves as follows:

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

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

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

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

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

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

As example of the match operation consider the following: applying `?|[(e1 | e2) & e3]|` to the term `|[(a < b | 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:

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

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

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

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

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.

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

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 | e2) & e3]|; !|[(e1 & e3 | 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 `|` is only valid if *e3* is a *pure* Tiger expression (i.e. without side-effects), since the *e3* computation is duplicated.

2.3 Controlling Rewriting with Strategies

Section 2.1.3 discussed the concept of basic rewriting with rewrite rules. Now, given a set of rewrite rules and a term t, which rule is first applied to t? There may be more than one rule whose left hand side matches t. Besides, one may want repeated application of rules, or application of rules to specific child elements of t (if any). Exhaustive application of rules to a tree is generally no good solution, since the set of rules is often non-confluent or even non-terminating. Stratego controls the process of rewriting with strategies.

2.3.1 Strategy Basics

Two primitive strategies in core Stratego are `id` and `fail`. Applying `id` to a term succeeds always and leaves the term unchanged. The opposite 'zero strategy' `fail` always fails.

Strategy definitions are like function definitions in imperative languages, except we distinguish two types of arguments here: *strategy arguments* and *term arguments*. Strategy arguments allow passing any strategy upon calling. Term arguments allow passing values (closed terms) to a strategy. The latter is useful for passing values without touching the current subject term in the rewriting process. Appendix A lists all operational semantics of the strategies and strategy combinators discussed here.

2.3.2 Traversing Trees

Rewrite rules generally describe transformation on the level of small, local terms. The transformation of more complex terms, including treatment of child terms and combining several rules on one term, is controlled by strategies. Often, strategies 'steer' the transformation through an input tree, thereby defining a traversal.

A transformation in Stratego should be thought of as a walk over a syntax tree, where all strategies part of the transformation receive some input term (depending on the current position in tree) and produce a (possibly unchanged) output term, or they fail after which the runtime system backtracks to the strategy that originally called the failing strategy.

Congruences are a way of matching a certain term and applying strategies to its children. $c(s_1, \ldots, s_n)$ matches any term that has c as its constructor and has n children, and applies s_i to its respective children. The congruence `Assign(id, PropConst)` for example, matches an assignment term, applies `PropConst` to its right hand side expression and leaves the left hand side variable untouched by applying the identity strategy `id`. Assuming for example that `PropConst` rewrites `|[ a ]| -> |[ 3 ]|`, this congruences behaves as follows:

```
<Assign(id, PropConst)> |[ b := a ]| => |[ b := 3 ]|
```

Recursive strategy definitions make repeated or even exhaustive application of strategies possible. For example `repeat(`s`)` applies s repeatedly to its input term, until it fails, and yields the last successful result. Normalizing a term with respect to a set of rules $\{\mathtt{R}_1, \ldots, \mathtt{R}_n\}$ is done by `innermost(`$\mathtt{R}_1$ `+ ... +` $\mathtt{R}_n$`)`, which performs a bottomup traversal, and recursively applies itself at each term, possibly traversing downward again.

2.3.3 Strategy Combinators

Strategies determine *where* rewriting is applied, but it is also useful to determine *which* strategies are used for rewriting.

Sequential composition of two strategies s_1 `;` s_2 applies s_1 to the input term and if it succeeds, applies s_2 to the output of s_1.

The *left deterministic choice* $s_1 \mathtt{<+} s_2$ first applies s_1 and only if that fails, applies s_2 to the original input. It is used in the implementation of `try`:

$$\mathtt{try}(s) \equiv s \mathtt{<+\ id},$$

which tries to applies s, but is the identity strategy if s fails. The *non-deterministic choice* s_1 `+` s_2 applies either of the two and applies the other if it fails, without any preference for either one. It is often used to combine several mutually exclusive rewrite rules, e.g. `ElimDead = ElimDeadFun + ElimDeadRec`. The *guarded choice* s_1 `<` s_2 `+` s_3 is the generalized choice; it applies s_1, followed by s_2 if it succeeds, or s_3 if it fails.

The *test strategies* `where(`s`)` and `not(`s`)` test whether application of s succeeds or fails, respectively, and restore the original input term afterward, while *maintaining* any changes to state and – only for `where` – environment.

Although all combinators have a well-defined meaning, it is often convenient to have some sugar strategies available. A good example is the *if-then-else-end* construct, which

performs a test strategy and applies either the then-strategy or the else-strategy after that to the original input term:

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

Two strategy combinators were already mentioned in the beginning of this chapter, for applying strategies and binding the result. Instead of applying s automatically to the current subject term, it is applied to p using <s>p. Matching the result of some strategy against a pattern is done by s => p. These two constructs can be easily desugared to core Stratego:

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

Often these two constructs are combined, as we already saw in the example
<add>(*i*, *j*) => *k*

When matching or, especially, building terms the subterms are often the result of subcomputations. It is convenient when these strategy calls can be done in place, e.g. !FunDefs(<map(rename)> *fds*). This is implemented by lifting out the subcomputations and binding the result, which can be used in the actual match or build:

$$\texttt{!}\,p_1[\texttt{<}s\texttt{>}p_2] \;\equiv\; \{x\texttt{: where(<}s\texttt{>}p_2 \texttt{ => } x); \texttt{!}\,p_1[x]\} \qquad \texttt{?}\,p_1[\texttt{<}s\texttt{>}p_2] \;\equiv\; \{x\texttt{: ?}\,p_1[x]; \texttt{!}\,x; \texttt{<}s\texttt{>}p_2\}$$

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 <s> application in a term, defined as

$$\texttt{!}\,p[\texttt{<}s\texttt{>}] \;\equiv\; \{x\texttt{: where(}s \texttt{ => } x); \texttt{!}\,p[x]\} \qquad \texttt{?}\,p[\texttt{<}s\texttt{>}] \;\equiv\; \{x\texttt{: ?}\,p[x]; \texttt{<}s\texttt{>}x\}$$

For example, when the current term is a list of Tiger expressions, that need to be placed in a let block for some known variable declarations *d**, the following term wrap applies: !Let(*d**, <id>). This can be done the other way around, i.e. lifting out one child of some term, using term projection: ?Let(_,<id>).

2.3.4 Generic Traversal Strategies

Section 2.3.2 already introduced congruence strategies, which match a specific constructor and apply strategies to its children. The larger the object language gets however (i.e. the more non-terminals it has), the more congruences are needed to fully define a traversal. A more generic approach is desirable here. Stratego offers three primitive generic strategies: all, some and one.

all(s) matches any term and applies s to all of its children (maybe zero), and failing if any of these child applications fails. The resulting term is the same constructor with all rewritten child terms. Several types of generic traversal are implemented using all:

```
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)))
```

The `topdown` strategy applies `s` to its input term and recursively applies itself to all children of the resulting term. `bottomup` first descends into all subterms after which `s` is applied at the root level. `alltd` is a topdown traversal that stops a traversal once the application of `s` succeeds. `downup` is a combined topdown and bottomup traversal. `innermost` is a fixpoint traversal that applies a strategy exhaustively starting with innermost terms.

`one`(s) is like `all`, except that it rewrites exactly one subterm or fails if none of the subterms can be rewritten by s. Example strategies using `one` are:

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

`oncetd` and `oncebu` find the first subterm to which `s` applies and leave the rest intact, in a topdown and bottomup manner respectively. `spinetd` traverses a tree along one single path (a 'spine') applying s at each term, until no subterms can be rewritten anymore. An example use of the above is when detecting whether a some term x is subterm of another term e:

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

Finally, there is the `some`(s) strategy, which is similar to `all` and `one`, but tries to applies s to as many subterms as possible, and *at least once*. An example is

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

which finds as many applications of `s` as possible, and descends into subterms if application at root level fails. The use of `rec x` is explained in the next section.

2.4 Strategy Definitions and Contexts

Stratego is modular in the sense that it allows modular setup of source code. All strategy definitions, and hence rule definitions, are global however. A strategy definition $f(f_1,\ldots,f_n\,|\,x_1,\ldots,x_m)$ = s introduces a new strategy combinator f with body s parameterized with strategy variables f_1, ..., f_n and term variables x_1, ..., x_m. An application $f(s_1,\ldots,s_n\,|\,p_1,\ldots,p_m)$ 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.

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:

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

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:

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

2.4.1 Local Strategy Definitions

Global strategy definitions start with an initial environment containing only the values of all term arguments, since Stratego does not know global variables. Local strategy definitions, inheriting the current runtime environment, are possible by the use of the `let-in-end` construct. For `let` $d_1...d_n$ `in` s `end` the definition environment is extended with the local definitions $d_1...d_n$, and the body s is executed within that definition environment. Any previous definitions with the same name from outer definitions scopes are hidden and become visible again upon exit of the `let`.

2.4.2 Recursive Closure

The *recursive closure* `rec`$f(s)$ is sugar for a local recursive definition:

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

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

2.4.3 Rewrite Rules

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

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

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

$$dsig\ \texttt{:}\quad p_1\ \texttt{->}\ p_2 \quad\equiv\quad dsig\ \texttt{:}\quad p_1\ \texttt{->}\ p_2\ \texttt{where id}$$

Example: the rewrite rule

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

corresponds to the strategy definition

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

2.4.4 Multiple Definitions

It is sometimes useful to give a set of rules the same name. For instance the EvalBinOp rules in Figure 2.1 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.

$$dsig = s_1 \ \ldots \ dsig = s_n \quad \equiv \quad dsig = (s_1 + \ldots + s_n)$$

For example, the EvalBinOp rules from Figure 2.1 reduce to a single definition

```
EvalBinOp = {e : ?|[ e + 0 ]|; !|[ e ]|}
          + {i,j,k: ?|[ i + j ]|; where(<add>(i, j) => k); !|[ k ]|}
          + {i,j,k: ?|[ i * j ]|; where(<mul>(i, j) => k); !|[ k ]|}
```

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.

Chapter 3

Scoped Dynamic Rewrite Rules

Stratego transformations are based on traversals of syntax trees. Rewrite rules, or in general strategies, have no knowledge of the traversal they are part of. They only operate on the current term, and possibly have some bound variables from the execution environment $\mathcal{E}$ they occur in. Techniques such as function inlining, value propagation and other partial evaluators require rules to be context-sensitive, i.e. they should be able to use information gained somewhere previously in the transformation.

The introduction in Chapter 1 already signaled the problem of this lack of context, and discussed some ready available solutions and why a better solution was needed. Stratego employs the concept of dynamic rules to add context awareness to the transformation system. This chapter mainly describes the basic dynamic rules as they were added in Stratego 0.6. Figure 3.1 lists the syntax of the then available dynamic rule constructs and can be used as a reference in the first two sections. These sections consider some updated versions of the Tiger examples from the original dynamic rules paper [Vis01] and add operational semantics to that, as it was formalized in the new dynamic rules paper [BvDOV04]. In the final section, a brief but complete overview of all new dynamic rule constructs is given. The practical use and operational semantics are treated in the next three chapters. Readers who are unfamiliar with the Tiger language should refer to Section 5.1 for a quick introduction.

3.1 Runtime Definition of Rewrite Rules

Without dynamic rules, the *state* of the transformation system is *static*, which means that all rewrite rules and strategies are already available at compile time and do not change at run time. Adding rule definitions to this state at runtime allows to specify rewritings specifically aimed at the input currently under consideration.

3.1.1 Example: Bound Variable Renaming

The majority of programming languages, including Tiger, allows semantically distinct variables to have the same name. The scoping rules of the language determine to which variable declaration an occurrence of a variable corresponds. When transforming programs with non-

s	::=	`rules` $(drd_1 \ldots drd_n)$	dynamic rule generation
		`override rules` $(drd_1 \ldots drd_n)$	dynamic rule overriding
		{| $f_1, \ldots, f_n$: s |}	dynamic rule scope
drd	::=	sig : p_1 -> p_2 (`where` s)?	dynamic rule definition
		sig : p_1 -> `Undefined` (`where` s)?	dynamic rule undefining

Figure 3.1: Extension of syntax of Stratego with *old* dynamic rules.

unique variable names, these scoping rules should always be taken into account.
Bound variable renaming replaces each non-unique variable name in variable declarations with a new name, and renames the corresponding variable occurrences anywhere in the program accordingly. Transformations are made a lot easier when it can be assumed that two identical variable names refer to semantically the same variable. Besides, a transformation such as function inlining can cause variable capture, making bound variable renaming indispensable. A renamed program is suitable as input to other transformations that do not consider local scopes of variables and need unambiguous variable names for that reason.

An example of bound variable renaming is shown below, only local variable declarations (`let`-blocks) are considered, but extension to function arguments and loop index variables is trivial.

```
let var a := x
    var x := x + a
    var y := let var a := a + x
             in x * a
             end + a
 in print(a+x+y)
end
```

⇒

```
let var a_0 := x
    var x_0 := x + a_0
    var y_0 := let var a_1 := a_0 + x_0
               in x_0 * a_1
               end + a_0
 in print(a_0 + x_0 + y_0)
end
```

Declared variables become directly visible after their declaration. In the example: the initial declaration of a is directly used in the right hand sides of the following declarations. Any not yet declared variables refer to the outer scope in which the `let` expression is itself. The example shows that all left hand sides are renamed. Not yet declared (i.e. unbound) variables are not renamed here, but since any new occurrence of a same variable will be renamed (a new `RenameVar` rule defined by each `RenameVarDec`), there is no problem in that. Also note how the defined rule `a -> a_1` in the inner `let` does not outlive that scope to the `+ a` expression nor the `print` function, where the old mapping `a -> a_0` has become visible again.

The implementation of a bound variable renamer, as shown in Figure 3.2, traverses the input tree, renaming variables and variable declarations and introducing dynamic rule scopes that follow Tiger's underlying variable scopes. The renaming rules are defined for each declaration at run time, and can directly be applied to variable terms. First, `RenameVar` is tried, renaming variables when needed. If instead the input is a `let` expression, a new scope is introduced for `RenameVar`, ensuring that all rules potentially defined in its children, will

```
exprename =
rec rn(
  RenameVar
  <+ |[ let <*id> in <*id> end ]|; {| RenameVar : all(rn) |}
  <+ |[ var <id> <id> := <rn> ]|; RenameVarDec
  <+ all(rn)
)

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

Figure 3.2: Simple variable renaming with local variable scope.

be removed afterward. When the current term is a variable declaration, first the right hand side is renamed, since any occurrences of the newly declared variable name still refer to a previous declaration of it. Directly afterward, `RenameVarDec` renames the declared variable using `newname`, which generates unique names. Any other input is traversed recursively by `all(rn)`.

Defining Rules Definition of dynamic rules occurs by placing one or more 'normal' rule definitions inside a `rules(..)` construct. The rule definition can use variables that are bound in its definition context. In the condition of `RenameVarDec` both x and y are already bound, so the defined rule `RenameVar` is specifically for those two variables. Variables that are not bound in the context are considered static variables, just as they can be used in normal (static) rewrite rules. These variables will be bound (matched against) when the dynamic rule is being applied.

Dynamic Rule Scopes The whole point of variable renaming was the disambiguation of variables names across any variable scope in the input. This implies that for any local scope, the generated new variable names should not outlive that scope, i.e. the defined dynamic rules for that scope should be retracted upon exit of that scope.

The dynamic rules scoping construct {| f : s |} applies strategy s and any rules named f defined during that application are retracted afterward. The strategy for renaming a `let` expression shows how all its children are recursively renamed within one local scope, thus ensuring that any newly defined rules from the declaration section of that `let` apply only to its body.

Rules from some inner scope shadow any rules with the same name and left hand side from outer scopes. Upon exit of the inner scope, the rules from outer scopes become 'visible' again.

Applying Rules Application of dynamic rules is nothing different from application of static rules or strategies. In `exprename`, `RenameVar` is applied as any other rule would have been.

If no instances of a dynamic rule have been defined yet, application simply fails.

3.1.2 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 behavior 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 Chapter 7 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.

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

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

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

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

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

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

The new strategy is a prioritized choice that first tries to apply the new rule, 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.

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

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

3.1.3 Semantics: Scoping Rule Sets

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

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

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

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

Since no rules have been defined yet, the strategy for the new scope corresponds to `fail`. After application of the strategy s, the new scope is removed.

The definition and undefinition of a dynamic rule modifies the strategy expression in the current scope

$$\frac{s'_1 \equiv \mathsf{define}(drd, \mathcal{E}, s_1)}{\Gamma_{L(s_1|s_2|...|s_n)}, \mathcal{E} \vdash \langle \texttt{rules}(drd) \rangle\, t \implies t\ (\Gamma_{L(s'_1|s_2|...|s_n)}, \mathcal{E})}$$

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

$$\mathsf{define}(L : p_1 \to p_2 \texttt{ where } s_1, \mathcal{E}, s_2) \equiv \{\mathcal{E}(\texttt{?}\, p_1; s_1; \texttt{!}\, p_2)\} \leftarrow s_2$$

$$\mathsf{define}(L \texttt{ :- } p_1, \mathcal{E}, s) \equiv \{\texttt{?}\mathcal{E}(p)\texttt{;!}\bot\} \leftarrow s$$

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

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

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow ... \leftarrow s_n \rangle\, t \implies t'\ (\Gamma', \mathcal{E}')\ t' \not\equiv \bot}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle L \rangle\, t \implies t'\ (\Gamma', \mathcal{E})}$$

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

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

3.2 Undefining and Overriding Rewrite Rules

We saw how dynamic rules can be defined and automatically removed upon exit of their definition scope. Sometimes however, it is necessary to manually undefine certain rule instances, in the middle of some scope.

Another typical need is to redefine previously defined rules, where the original definition is unknown, the current state of the transformation may be several scopes deeper (but still inside the definition scope though). The old dynamic rule system facilitated the so called `override rules` construct for these situations.

Both situations are explained in one familiar Tiger example: elimination of dead functions. Also the replacement new dynamic rule constructs are mentioned.

3.2.1 Example: Dead Function Elimination

Dead code elimination aims at removing all unnecessary code from a program, code of which it can be safely assumed that not any program execution will ever reach those code parts. A special instance of this is dead function elimination. By monitoring any function calls, the functions that are never called can be declared 'dead', hence their definition can be removed. The following example shows how the definition of `sqr` is removed from the fragment:

```
let function sqr(x : int) = (x * x)
 in let var a_0 : int := (3 + y)
     in (a_0 * a_0)
    end
end
```

$\Rightarrow$

```
let var a_0 : int := (3 + y)
 in (a_0 * a_0)
end
```

An implementation of dead function elimination is shown in Figure 3.3. The main strategy treats three cases: either the current term is a `let` expression which needs proper scoping, or it is a new function declaration which needs to be initialized, or it is a function application which should be registered. The three cases recursively transform all subterms (`all(dead-fun-elim)`), and if none of the cases apply, the transformation defaults into this recursion anyway.

When the input is a `let` expression `dfe-let` enters a new scope for `FunctionIsDead`, the dynamic rules which will maintain for each function whether it is dead. After transformation of both the declarations and the body, any dead declarations are removed by `filter-dead-declarations`, if none remain the entire `let` expression is simplified to just its body.

When the input is a new function definition, it is initially declared 'dead'. Once a function call is encountered, this is made undone so that only dead functions maintain their 'dead' declaration. `DeclareFunDead` defines a new instance of the dynamic rule `FunctionIsDead`. This instance rewrites any function definition with the same identifier to itself. Note that both the function arguments, type annotation and body are static variables and can be seen as wildcards here, only f matters.

```
dead-fun-elim =
  dfe-let(dead-fun-elim)
  <+ DeclareFunDead
     ; all(dead-fun-elim)
  <+ try(DetectFunUsed)
     ; all(dead-fun-elim)

dfe-let(s) =
  |[ let <*id> in <id> end ]|
  ; {| FunctionIsDead :
      all(dead-fun-elim)
    ; filter-dead-declarations; try(RmEmptyLet)
    |}

filter-dead-declarations =
  |[ let <*filter(
            not( |[ <fundecs:id> ]| )
          <+ |[ <fundecs:filter(not(FunctionIsDead)); not([])> ]|)>
      in <*id>
     end ]|

DeclareFunDead =
  ? |[ function f(x1*) ta1 = e1 ]|
  ; rules(
      FunctionIsDead :
        |[ function f(x2*) ta2 = e2 ]| -> |[ function f(x2*) ta2 = e2 ]| )

DetectFunUsed =
  ? |[ f(a*) ]|
  ; override rules(
      FunctionIsDead : |[ function f(x2*) ta2 = e2 ]| -> Undefined )

RmEmptyLet :
  |[ let in e* end ]| -> |[ (e*) ]|

RmEmptyLet :
  Let([FunDecs([])], e*) -> |[ (e*) ]|
```

Figure 3.3: Strategies for eliminating dead functions.

Undefining Rules Finally, when a function call is encountered, the function should be declared 'undead'. `DetectFunUsed` does so by undefining the previously defined rule, using the syntax `rules(Rule : t -> Undefined)`. In the new dynamic rules the same functionality has been maintained, with new syntax though: rules(Rule :- *t*), so the reserved constructor `Undefined` has disappeared.

Overriding Rules In dead function elimination, simply undefining the rule is not the desired behavior. Dynamic rules are always defined or undefined in the current inner scope. In this case however, the rule `FunctionIsDead` needs to be undefined in the scope that the rule was originally defined in. Otherwise the effect of undefining becomes undone upon exit of the scope, whereas only upon return at the top level scope where the function was encountered the rule is used for checking the 'deadness' of the function.

Besides the normal rule definition blocks `rules(..)`, the old dynamic rule system facilitated the `override rules(..)` blocks. All dynamic rules defined or undefined inside such a block will be placed in the scope of the most recent previous rule definition. If no prior rule definition for that rule with the same left hand side existed, the rule definition failed. The new dynamic rule system has no native support for this form of overriding anymore, although its behavior can be emulated by the compiler, by consequently using the left hand side terms as extra scope labels. Scopes can be labeled with any closed term, and rules can be defined inside any labeled scope instead of the inner scope by default. Section 3.3.3 shortly introduces this concept, and the constant propagation implementation in Chapter 5 uses it in practice.

3.3 New Dynamic Rules

When more and more program transformations were built making use of Stratego's dynamic rules, many ideas for new functionality arose. With the release of StrategoXT 0.10 not only the representation and compilation of dynamic rules was entirely redesigned 'under the hood', but also new dynamic rule constructs were introduced, delivering the new functionality that had gradually been figured out. This section provides a brief, but complete overview of the new dynamic rules constructs, whose use will be further demonstrated in the next three chapters. Figure 3.4 lists all the current dynamic rule syntax and should be seen as a whole with the syntax shown in Figures 2.2 and 2.3. It replaces and extends the constructs as listed in Figure 3.1.

3.3.1 Extend Rules

When a dynamic rule is defined, it shadows any previously defined rules with the same name and left hand side. In certain situations this is too restrictive. For example in Wadler's deforestation algorithm, as presented in Chapter 6. The typical dynamic rules that are defined there look like:

```
HelperFun :+ FunApp(f, _) -> newfun
              where ?x
                  ; <is-renaming> (t, x)
```

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

Figure 3.4: Extension of syntax of Stratego with new dynamic rules.

Notice the wildcard _ in the left hand side, and the t in the condition: this rule is typically defined with the same f in its left hand side, but for various values of t. The left hand side is the same for all defined instances, but they are intended to rewrite different terms, due to the additional check `is-renaming` in the condition. Hence, the instances should not shadow each other, even though they have identical left hand sides. Instead of *defining* a single dynamic rule, it is possible to *extend the rule set* that is already present for a name and left hand side. Rule definitions are still placed in a `rules` block, but now with '`:+`' instead of '`:`' in the definition, as the above fragment already showed.

When a rule is applied, the available instances are tried, the most recently defined first, until the first successful rewrite. It is also possible to get all successful rewritings in a list. For each dynamic rule L, a strategy `bagof-`L becomes automatically available, and it is applied just as the normal rules would. The `bagof-` feature is also useful to collect information during a transformation and retrieve it in one list afterward.

Another automatically generated strategy is the `once-`L. When applied, it performs just as the normal rule application, but on the first successful rewrite, that instance is removed from the rule set. This ensures that any defined rule is used for rewriting only once, for example to ensure that a function application is unfolded just once.

3.3.2 Dynamic Identity Rules

A small but comfortable addition to dynamic rule syntax is the dynamic identity rule. By using `rules(`L `:` t`)` a normal dynamic rule is generated that matches t as its left hand side, and rewrites it to itself, without any condition. In the previous section on dead function elimination, the rule `FunctionIsDead` could typically benefit from this syntax:

```
rules( FunctionIsDead : |[ function f(x2*) ta2 = e2 ]| )
```

3.3.3 Scope Labeling

When new rules are defined, they are always associated to the inner scope. This is not always desirable, since the retraction of rules might then occur too early. The dead function elimination was a first example where a rule should be associated with some higher scope. The `override rules` construct was a pragmatic solution there. In the more general case, it would be a good thing to be able to literally associate a rule to a specific scope by some means. Labeling of scopes is an intuitive and clean way to do so, which opens up many new possibilities for dynamic rules.

A scope may be assigned labels upon entering: {| $L.p$: s |}, where L is again the rule name, s the strategy executed withing this scope, and p a fully instantiable pattern that acts as the label. Hence, any term or pattern can be used as a label. In most applications, the labels will be identifiers of e.g. functions and variables. Assigning labels to a scope can also be done in the middle of s.

Dynamic rule definitions now can specify a label attached to the rule name, as to specify to which scope the defined rule should be associated. In the dead function elimination implementation, the `override rules` functionality can now be replaced by:

```
DeclareFunDead =
  ? |[ function f(x1*) ta1 = e1 ]|
  ; rules( FunctionIsDead+f : |[ function f(x2*) ta2 = e2 ]| )

DetectFunUsed =
  ? |[ f(a*) ]|
  ; rules( FunctionIsDead.f : |[ function f(x2*) ta2 = e2 ]| )
```

In both rules again a dynamic identity rule is defined, but the name has the function identifier attached as a label now. The scopes themselves are still entered in the same way by `dfe-let`. In `DeclareFunDead`, the notation `FunctionIsDead+`*f* is used, meaning that the current scope will be labeled with *f*, *and* that the rule definition will also be associated to that scope. `DetectFunUsed` defines a rule which is associated to the scope that has the current function identifier as a label, using `FunctionIsDead.`*f*.

Scopes may be labeled with any number of labels, and when a labeled rule definition can not find a scope with the specified label the rule definition fails.

3.3.4 Rule Set Forking, Union and Intersection

Program transformations often follow the normal control flow of the input program, for example data flow optimizations such as constant propagation, common sub-expression elimination and dead code elimination. The most important requirement to these transformations is of course that the output program is semantically the same as the input, for any program execution. For example in dead function elimination, when an `if-then-else` expression is encountered, and in one branch a function is called, but in the other it is not. Upon exit of the expression, it is only safe to mark the function as being 'not dead'.

In all transformations where control flow plays a role, one would like to split (or fork) the current system state with its rule sets when the control flow of the input program splits itself. In this way, the various paths of control flow are analyzed with a split off system state, thus recording only their own new rule definitions. When the forked paths meet again, the various rule sets should be merged again. Typical ways of doing so are intersection and union (per rule name and left hand side)

The new dynamic rules offer built-in syntax and a flexible underlying API for forking rule sets and automatically intersecting or unifying them afterward. An even more extreme variant of this is the fixpoint operations on rule sets. These are typically used for loop expressions and repeatedly apply a strategy to current term, intersecting or unifying the forked rule sets each time until no changes occur in the resulting rule sets. Hence, for any execution of the input program, the produced analysis in the dynamic rules is safe.

Chapter 5 makes extensive use of rule set forking and meeting and will further discuss the details, syntax and semantics.

Chapter 4

Shrinking Inlining in a Small Lambda Calculus

Optimizing program transformations always bear the risk of becoming from program *transformation* into program *evaluation*, for example when value propagating and folding over loops leads to undesirably long iterations, or when unfolding function calls leads to a major blowup of the code. These reductions are often part of functional language compilers, and Appel and Jim have described a shrinking algorithm [App92], which is part of the Standard ML of New Jersey compiler. The *Contract* algorithm only performs optimizations that make the code smaller, e.g. dead-variable elimination, δ-reduction (constant folding), and a *shrinking* β-reduction (function inlining based on call counts).

The algorithm is effective, since it was shown to deliver a speedup factor of 2.5, but the compilation itself is expensive. Appel and Jim describe a smarter, hence faster algorithm, which is even proved to be confluent [AJ97].

The case study considered in this chapter is built around this latter article. A small variant of the CPS lambda calculus was realized in SDF2, and Stratego implementations for both the naive and smarter algorithm were made. Focus does not lie on verifying the experimental results in the article, but rather on showing how the algorithms can be elegantly and intuitively implemented in Stratego, partially by the use of dynamic rules.

4.1 The CPS Language

The *Contract* algorithm is part of the SML/NJ compiler, which uses a *continuation passing style* (CPS) lambda calculus as intermediate representation [AJ89]. A representative subset of this language is shown in Figure 4.1. An input program is a term M, which is a definition or use of a function or a record. Below is a small example fragment:

```
let v = <q,r>
 in let w = #1(v)
     in w(v,r)
```

Since we treat such a small subset of the language, and we are not working in the context of the actual SML/NJ compiler, it is hard to use real-world test programs like the

M, N	::=	`let` $f(x_1, \ldots, x_n) = M$ `in` N	recursive function definition	`FunDefR`
		$f(a_1, \ldots, a_n)$	function application	`FunApp`
		`let` $r = \langle a_1, \ldots, a_n \rangle$ `in` M	record creation	`RecBuild`
		`let` $x = \#i(a)$ `in` M	record field selection	`RecFieldSel`
		Uppercase identifier	meta-term (see text)	`MetaTerm`
i	::=	integer	record field selector	
a	::=	identifier	atom (here: just variables)	
f, r, x	::=	identifier	variables	

Figure 4.1: Syntax of the CPS based lambda calculus.

ones in Appel's original benchmark set. We're doing a qualitative comparison of the original specification and our Stratego implementation here, rather than a quantitative one, so the lack of large test input is no problem.

The actual syntax definition contains one additional production that allows the use of uppercase identifiers as terms, e.g. the following fragment is syntactically valid:

```
let f() = M
 in N
```

Here, `M` and `N` are parsed as `MetaTerms`. These are just included to facilitate creation of small tests. We make the assumption that none of the variables used in the rest of the program play a role inside `MetaTerms`, hence these terms never prevent any reductions by the *Contract* algorithm. They are usually just used for ending a number of nested `lets`, without any change to the semantics, while still being valid syntax.

4.2 A Naive Contract Algorithm

The *Contract* algorithm, both the naive and less-naive variant, performs a couple of optimizations. It eliminates dead variables, in this case dead function definitions and dead record builds, by analyzing any uses of the function or record identifier. Besides, it inlines records in a record field selection, if the record fields are known (projections). Similarly, it inlines function definitions at a call site, but *only* if that is the one and only site where the function is applied. Record inlining always makes the code smaller, but function inlining doesn't, hence the limit of 1. More advanced versions of *Contract* could do constant folding as well, but it is not incorporated here. The algorithm thus makes the code smaller and saves some overhead on function calls and record field selections when they have been inlined.

4.2.1 Algorithm

The major part of the algorithm is its $\mathrm{contract}$ phase, which performs all shrinking reductions: dead variable elimination, record-field selection and inlining of functions called only

repeat Initialize σ, Bind, $\mathrm{Count}_{\mathrm{app}}$ and $\mathrm{Count}_{\mathrm{esc}}$ to empty,
Gather usage counts (census),
Perform contractions based on usage counts ($\mathrm{contract}$).
until no redexes left

Figure 4.2: Iterative contractions. (Source: [AJ97])

$$\begin{array}{ll}
\mathrm{census} & (\Delta, \texttt{let } f(x_1, \ldots, x_n) = M \texttt{ in } N) = \\
& \mathrm{census}(\Delta, M);\ \mathrm{census}(\Delta, N) \\
\\
\mathrm{census} & (\Delta, f(a_1, \ldots, a_n)) = \\
& \mathrm{Count}_{\mathrm{app}}[\sigma(f)] \leftarrow \mathrm{Count}_{\mathrm{app}}[\sigma(f)] + \Delta \\
& \mathrm{Count}_{\mathrm{esc}}[\sigma(a_i)] \leftarrow \mathrm{Count}_{\mathrm{esc}}[\sigma(a_i)] + \Delta, \quad 1 \leq i \leq n \\
\\
\mathrm{census} & (\Delta, \texttt{let } r = \langle a_1, \ldots, a_n \rangle \texttt{ in } N) = \\
& \mathrm{Count}_{\mathrm{esc}}[\sigma(a_i)] \leftarrow \mathrm{Count}_{\mathrm{esc}}[\sigma(a_i)] + \Delta, \quad 1 \leq i \leq n \\
& \mathrm{census}(\Delta, M) \\
\\
\mathrm{census} & (\Delta, \texttt{let } x = \#i(a) \texttt{ in } N) = \\
& \mathrm{Count}_{\mathrm{app}}[\sigma(a)] \leftarrow \mathrm{Count}_{\mathrm{app}}[\sigma(a)] + \Delta \\
& \mathrm{census}(\Delta, M)
\end{array}$$

Figure 4.3: Gathering usage counts for the contract algorithms. Use $\Delta = +1$ to increment. (Source: [AJ97])

once. It needs information on usage counts and values of passed arguments, which are maintained in global mapping tables:

Bind A table mapping function variables to (argument, body) pairs and record variables to tuples of atoms.

σ A substitution mapping variables to atoms.

$\mathrm{Count}_{\mathrm{app}}$ A table mapping variables to their number of occurrences in function-call position, and record variables to their number of occurrences in selected-from position.

$\mathrm{Count}_{\mathrm{esc}}$ A table mapping variables to their number of occurrences as record fields or function arguments.

The algorithm iterates a number of times, until no more reducible expressions ('redexes') are left and a shrink-normal form has been reached, see Figure 4.2. Appel proposes that in practice, the iteration is ended when only a few contractions are done, thus preventing too many iterations. In our context however, input programs are small enough to safely

$$
\begin{array}{ll}
\text{contract} & (\texttt{let } f(x_1,\ldots,x_n) = M \texttt{ in } N) = \\
& \text{Bind}[f] \leftarrow ((x_1,\ldots,x_n), M) \\
& \text{if } \text{Count}_{\text{app}}[f] \leq 1 \text{ and } \text{Count}_{\text{esc}}[f] = 0 \\
& \text{then } \text{contract}(N) \\
& \text{else } \texttt{let } f(x_1,\ldots,x_n) = \text{contract}(M) \texttt{ in } \text{contract}(N) \\
\\
\text{contract} & (f(a_1,\ldots,a_n)) = \\
& \text{if } \text{Count}_{\text{app}}[\sigma(f)] = 1 \text{ and } \text{Count}_{\text{esc}}[\sigma(f)] = 0 \\
& \text{ and } \text{Bind}[\sigma(f)] = ((x_1,\ldots,x_n), M) \\
& \text{then } \sigma \leftarrow \sigma + \{x_1 \mapsto \sigma(a_1),\ldots,x_n \mapsto \sigma(a_n)\};\ \text{contract}(M) \\
& \text{else } \sigma(f)(\sigma(a_1),\ldots,\sigma(a_n)) \\
\\
\text{contract} & (\texttt{let } r = \langle a_1,\ldots,a_n\rangle \texttt{ in } N) = \\
& \text{Bind}[r] \leftarrow \langle a_1,\ldots,a_n\rangle \\
& \text{if } \text{Count}_{\text{esc}}[r] = 0 \\
& \text{then } \text{contract}(N) \\
& \text{else } \texttt{let } r = \langle \sigma(a_1),\ldots,\sigma(a_n)\rangle \texttt{ in } \text{contract}(N) \\
\\
\text{contract} & (\texttt{let } x = \#i(a) \texttt{ in } N) = \\
& \text{if } \text{Count}_{\text{app}}[x] = 0 \text{ and } \text{Count}_{\text{esc}}[x] = 0 \\
& \text{then } \text{contract}(N) \\
& \text{else if } \text{Bind}[\sigma(a)] = \langle b_1,\ldots,b_n\rangle \\
& \text{then } \sigma \leftarrow \sigma + \{x \mapsto \sigma(b_i)\};\ \text{contract}(N) \\
& \text{else } \texttt{let } x = \#i(\sigma(a)) \texttt{ in } \text{contract}(N)
\end{array}
$$

Figure 4.4: Reduction phase of the naive Contract algorithm. (Source: [AJ97])

iterate towards a shrink-normal form. In each iteration the contract call is preceded by a call to census, which updates the usage counts. Figure 4.3 lists the original algorithm for the census phase, which is fairly straightforward, and Figure 4.4 lists the original algorithm for the contract phase. Although the contract function is structured on each type of input term, the actual reductions and recursive rewriting of subterms are completely intertwined. Our implementation aims at separating these two.

4.2.2 Implementation in Stratego

We will mainly focus on the contract phase of the algorithm, but for completeness the implementation of the iteration and census part are included as well. Figure 4.5 lists how the iterative execution of census and contract takes place. The main strategy `shrink` calls `exhaust` with `shrink-iteration` as parameter.

```
shrink = exhaust(shrink-iteration)

exhaust(s) =
  rec x({prev: ?prev; s; (?prev <+ x) })

shrink-iteration =
{| CountApp, CountEsc, VarSubst, InlineFunDef, EvalRecField :
  census(|1)
; try(contract)
|}
```

Figure 4.5: Strategies for iterative execution of the algorithm.

```
census(|inc) =
  ?|[ let f(x*) = M in N ]|
; where(<census(|inc)> M
        ; <census(|inc)> N)

census(|inc) =
  ?|[ f(a*) ]|
; where(<inc-app(|inc)> f
        ; <map(inc-esc(|inc))> a*)

census(|inc) =
  ?|[ let r = <a*> in M ]|
; where(<map(inc-esc(|inc))> a*
        ; <census(|inc)> M)

census(|inc) =
  ?|[ let x = #i(a) in M ]|
; where(<inc-app(|inc)> a
        ; <census(|inc)> M)

census(|inc) =
  MetaTerm(id)
```

Figure 4.6: Strategies for performing the census phase (gathering usage counts).

```
var-subst      = try(VarSubst)
add-var-subst  = ?(x, a); rules(VarSubst: x -> a)
add-var-substs = zip(add-var-subst)
```

Figure 4.7: Strategies for maintaining variable substitutions.

```
contract =
  try(Bind)
; try((ElimDead <+ prep-inline(var-subst); ContractApp); contract
      <+ contract-td(contract, var-subst))

contract-td(con, subst) =
  FunDefR(id, id, con, con)
+ FunApp(subst, map(subst))
+ RecBuild(id, map(subst), con)
+ RecFieldSel(id, id, subst, con)

prep-inline(subst) =
  FunApp(subst, map(subst))
+ RecFieldSel(id, id, subst, id)

Bind        = BindFunDef + BindRecBuild
ContractApp = InlineFunDef + EvalRecField
ElimDead    = ElimDeadFun + ElimDeadRec + ElimDeadRecField

BindFunDef =
  ?|[ let f(x*) = M in N ]|
; rules(
    InlineFunDef :
      |[ f(a*) ]| -> |[ M ]|
      where <is-inlineable> f
            ; <add-var-substs> (x*, a*)
  )
ElimDeadFun :
  |[ let f(x*) = M in N ]| -> |[ N ]|
  where <is-dead + is-inlineable> f

BindRecBuild =
  ?|[ let r = <a*> in N1 ]|
; rules(
    EvalRecField :
      |[ let x = #i(r) in N2 ]| -> |[ N2 ]|
      where <add-var-subst> (x, <index> (<string-to-int> i, a*))
  )
ElimDeadRec :
  |[ let r = <a*> in N ]| -> |[ N ]|
  where <get-esc-count> r => 0

ElimDeadRecField :
  |[ let x = #i(a) in N ]| -> |[ N ]|
  where <is-dead> x
```

Figure 4.8: Strategies for the naive contract phase.

```
inc-app(|inc) =
  ?t
; where(get-app-count; <add> (<id>, inc) => n)
; rules(CountApp: t -> n)

get-app-count =
  CountApp <+ !0

set-app(|n) =
  ?t
; rules(CountApp: t -> n)

is-inlineable =
  where(get-app-count => 1)
; where(get-esc-count => 0)

is-dead =
  where(get-app-count => 0)
; where(get-esc-count => 0)
```

Figure 4.9: Strategies for maintaining and checking call counts.

The generic `exhaust` strategy binds its input term to a variable for later reference, applies parameter strategy `s` to it and compares the result to its input `prev`. If the result is different, `exhaust` applies itself recursively again, or returns otherwise, thus achieving an exhaustive transformation.

Figure 4.6 lists the almost literal implementation of census. The usage counts are determined by a recursive strategy based on the structure of the input term. The parameter Δ allows for both incrementing and decrementing the counters. Figure 4.7 lists some helper strategies for extending and applying the variable substitution relation σ. Finally, Figure 4.9 lists three representative strategies that maintain the call counts $\mathrm{Count}_{\mathrm{app}}$. The dynamic rule `CountApp` acts as the table mapping terms (variables) to their application counts. Similar strategies are available for $\mathrm{Count}_{\mathrm{esc}}$. Two additional shorthand strategies are also available: `is-dead` checks whether both counters are equal to zero, and `is-inlineable` checks whether the application count is equal to one, and $\mathrm{Count}_{\mathrm{esc}}$ is equal to zero, hence whether a function may be inlined or not.

The strategies that implement the contract phase are shown in Figure 4.8. The `contract` strategy locally tries to prepare future reductions (`Bind`) and either reduce terms (`ElimDead`, `ContractApp`) or do a topdown traversal into child terms (`contract-td`).

Function definitions and record creation are potential candidates for inlining and `Bind` prepares the inlining by defining dynamic rules for them. The decision whether or not to inline is left to the defined rule itself (note the checks on usage counts, through `is-inlineable` in its condition), so the checks are performed at the very moment of inlining, thus using

the most up to date usage counts. The rule definition replaces the $\mathrm{Bind}[f] \leftarrow \ldots$ and $\mathrm{Bind}[r] \leftarrow \ldots$ fragments in the original algorithm, and a call to these rules replaces the various inlining statements in the second and fourth $\mathrm{contract}$ variant in Figure 4.4.

After this preparation phase, `contract` tries to perform the actual reductions `ElimDead` and `ContractApp`, which on their turn call the actual elimination and inlining strategies. If any reduction succeeds, the resulting term is contracted again ('`; contract`'), thus taking care of the recursive nature of the algorithm. Finally, if no reductions were possible, the transformation descends into the appropriate child terms by calling `contract-td`.

The static rewrite rules (`Elim*`) in Figure 4.8 represent the most basic reductions, merely checking the usage counts. The defined dynamic rules (`InlineFunDef` and `EvalRecField`) specify basic reductions as well, but additionally extend the variable mapping σ. The latter is performed by calls to `add-var-subst`. Applying the mapping, i.e. $\sigma(a)$, amounts to calling `var-subst`. Behind these variable substitutions are basic dynamic rules as well.

4.2.3 Use of Dynamic Rules

This code only contains basic use of dynamic rules. The helper strategies for maintaining usage counts and variable-to-atom mapping are by means of dynamic rules, but these are just a way of using tables in Stratego. The two inlining rules however show the power of extending the rewrite system at runtime by adding new rules. The defined inlining rules contain runtime information on function bodies or record fields and take care of counter checking and variable renaming, thus simplifying the code at the actual inlining site significantly.

The only place where dynamic rule scoping is used, is at top level, to automatically reset the counters and variable mappings between two iterations. Further scoping is unnecessary, since the original algorithm was designed for unambiguous input programs with unique variable names. This requirement is here taken care of by an initial call to a bound variable renamer, similar to the one in Section 3.1.

4.3 A Less Naive Algorithm

The original *Contract* algorithm is effective, but expensive. This is mainly due to the sequential nature of the census and $\mathrm{contract}$ phase. The usage counters are updated, but during the entire $\mathrm{contract}$ phase no counters are intermediately updated. Elimination of function calls or record selections will only have its effect on usage counters in the next iteration, hence the potential elimination or inlining of the corresponding function definition or record build will also need another iteration.

Figure 4.10 shows the four rewriting steps on a sample term. A quick glance on the input term makes clear that the resulting term will eventually have to be `M`, but the usage counters prevent a one step elimination of all record creation and selections. The first step is fairly effective by inlining `a_0` and `c_0`. It also eliminates the definition of `g_0` since that is the only definition with both usage counters equal to zero. In the second step, only `f_0` has its usage counters equal to zero, and `d_0` can not be eliminated yet. This happens in the third and last step. The nested structure of this kind of input term, together with the

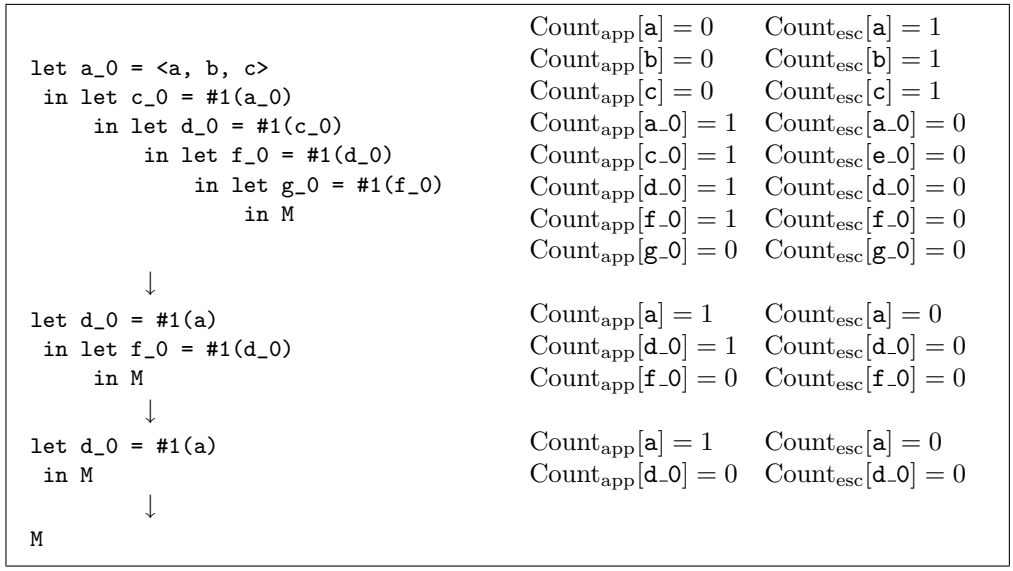


Figure 4.10: Iterative contract steps for the naive algorithm.

'upward dependency' of each variable use generally leads to $O(n)$ needed iterations where n is the number of nested `lets`.

The example shows that new reductions are not possible until the census pass of the next iteration step will update the usage counters. If we could maintain the counters more accurately and more directly, more reductions could potentially be performed in one iteration step.

4.3.1 Updated Algorithm

The new *Contract* algorithm is a quasi-one-pass algorithm, by directly recording the effect of each optimization on the usage counts, and by changing the order of some optimizations. The same census algorithm from Figure 4.3 is used, and the new *Contract* algorithm is shown in Figure 4.11.

Previously, during a contract phase, the usage counter would generally be overestimated, since the performed optimizations were not reflected in the usage counters yet. Some improvements in the new counter updating are:

- In dead-variable elimination: if `let` $f(\vec{x})$ = M is deleted, because f is dead, the usage counts of all free variables in M are decremented, by calling $\mathrm{census}(-1, M)$.

- In δ-reduction: when rewriting[1]

$$\begin{array}{l}\texttt{let } r = \langle \vec{a} \rangle \\ \texttt{in } C[\texttt{let } x = \#i(r) \texttt{ in } M]\end{array} \quad \rightarrow \quad \begin{array}{l}\texttt{let } r = \langle \vec{a} \rangle \\ \texttt{in } C[M\{x \mapsto a_i\}]\end{array}$$

  the usage count of r is decremented and the usage count of a_i is adjusted, depending on how many times x appears in M.

- In shrinking inlining: if a function definition `let` $f(\vec{x})$ `=` M is removed and its body inlined as $M\{\vec{x} \mapsto \vec{a}\}$ at one single call site $f(\vec{a})$, the usage counters of the a_i are updated depending on the original counts of the x_i. The counters of all free variables in M are now *not* decremented, as with dead function elimination, since M is not eliminated but placed inline.

Besides the more accurate counter maintenance, a second improvement is made to the original algorithm. Previously, elimination of definitions in expressions such as `let` r=$\langle\vec{a}\rangle$ `in` N was done prior to examination of the body N. Optimization of N however, might now lead to decreasing counters, including the ones for r, so possibly its definition can still be eliminated. If so, the counters for the a_i will be decremented as well, possibly leading to more optimizations on the way back up. So in the new algorithm, if normal elimination fails, an additional 'look-ahead' approach is taken by first contracting N, followed by an additional elimination attempt.

To correctly update the usage counts after each optimization, more care needs to be taken when handling recursive-dead-function and shrinking-inlining redexes. Consider the expression `let` $f(\vec{x})$=M `in` N, where $\mathrm{Count}_{\mathrm{app}}(f) = 1$ and $\mathrm{Count}_{\mathrm{esc}}(f) = 0$. This is either a recursive-dead-function or shrinking-inlining redex. Previously, $\mathrm{contract}$ would just eliminate the function definition and recurse on N. The new $\mathrm{contract}$ needs to distinguish the two cases: only if f is a genuine dead function, the usage counters of free variables in M should be decremented. This is detected by recurring on N and setting $\mathrm{Bind}[f]$ to the special value **inlined** when f is inlined at the call site. Upon return the value of $\mathrm{Bind}[f]$ is examined and if it was **inlined** the counters for M are decremented.

Finally, there is the case of $\mathrm{Count}_{\mathrm{app}}(f) > 1$ or $\mathrm{Count}_{\mathrm{esc}}(f) > 0$. The above does not apply here, but possibly during optimization of N the counters decrease because of other optimizations. Upon return, three cases are considered:

- $\mathrm{Bind}[f] = \mathbf{inlined}$, meaning that during optimization of N the counters for f decreased sufficiently to inline its body at a call site $f(\vec{a})$. Now, the function definition `let` $f(\vec{x})$=M can be eliminated, but no counters need to be decremented.

- $\mathrm{Bind}[f] \neq \mathbf{inlined}$, but during optimization of N, both counters for f are now zero. Again `let` $f(\vec{x})$=M is eliminated, but the counters for all free variables in M need to be decremented as well.

[1] Note the use of *contextual rewriting notation* `p1[q1] -> p2[q2]` here, as mentioned on page 3.

```
contract   (let f(x_1, ..., x_n) = M in N) =
           Bind[f] ← ((x_1, ..., x_n), M)
           if Count_app[f] = 0 and Count_esc[f] = 0
           then census(−1, M); contract(N)
           else if Count_app[f] = 1 and Count_esc[f] = 0
           then N' ← contract(N)
                 if Bind[f] ≠ inlined then census(−1, M)
                 N'
           else N' ← contract(N)
                if Bind[f] = inlined then N'
                else if Count_app[f] = 0 and Count_esc[f] = 0
                     then census(−1, M); N'
                     else Bind[f] ← ()
                          let f(x_1, ..., x_n) = contract(M) in N'
contract   (f(a_1, ..., a_n)) =
           if Count_app[σ(f)] = 1 and Count_esc[σ(f)] = 0
            and Bind[σ(f)] = ((x_1, ..., x_n), M)
           then σ ← σ + {x_1 ↦ σ(a_1), ..., x_n ↦ σ(a_n)}
                Count_app[σ(a_i)] ← Count_app[σ(a_i)] + Count_app[x_i] − 1,   1 ≤ i ≤ n
                Count_app[σ(f)] ← 0
                Bind[σ(f)] ← inlined
                contract(M)
           else σ(f)(σ(a_1), ..., σ(a_n))
contract   (let r = ⟨a_1, ..., a_n⟩ in N) =
           Bind[r] ← ⟨a_1, ..., a_n⟩
           if Count_app[r] = 0 and Count_esc[r] = 0
           then Count_esc[σ(a_i)] = Count_esc[σ(a_i)] − 1,   1 ≤ i ≤ n
                contract(N)
           else N' ← contract(N)
                if Count_app[r] = 0 and Count_esc[r] = 0
                then Count_esc[σ(a_i)] = Count_esc[σ(a_i)] − 1,   1 ≤ i ≤ n
                     N'
                else let r = ⟨σ(a_1), ..., σ(a_n)⟩ in N'
contract   (let x = #i(a) in N) =
           if Count_app[x] = 0 and Count_esc[x] = 0
           then Count_app[σ(a)] ← Count_app[σ(a)] − 1; contract(N)
           else if Bind[σ(a)] = ⟨b_1, ..., b_n⟩
           then σ ← σ + {x ↦ σ(b_i)}
                Count_app[σ(b_i)] ← Count_app[σ(b_i)] + Count_app[x]
                Count_esc[σ(b_i)] ← Count_esc[σ(b_i)] + Count_esc[x]
                Count_app[σ(a)] ← Count_app[σ(a)] − 1
                contract(N)
           else N' ← contract(N)
                if Count_app[x] = 0 and Count_esc[x] = 0
                then Count_app[σ(a)] ← Count_app[σ(a)] − 1
                     N'
                else let x = #i(σ(a)) in N'
```

Figure 4.11: Reduction phase of the less naive Contract algorithm. (Source: [AJ97])

```
let a_0 = <a, b, c>
 in let c_0 = #1(a_0)
     in let d_0 = #1(c_0)
         in let f_0 = #1(d_0)
             in let g_0 = #1(f_0)
                 in M
         ↓
M
```

Figure 4.12: Single contract step for the less-naive algorithm. (Compare with Figure 4.10)

- $\mathrm{Bind}[f] \neq \textbf{inlined}$, and the counters for f are still nonzero. No elimination is possible, and to finish the optimization, M needs to be contracted as well. Before doing so, inlining of recursive function calls to f should be prevented, which is done by setting $\mathrm{Bind}[f] \mapsto ()$.

Looking back at our original example in Figure 4.10, the effect of the counter updating becomes immediately clear. In the original transformation, `let g_0 = #1(f_0)` was eliminated, but the definitions of `d_0` and `f_0` remained in the first iteration step, since their application count was still bigger than zero. In the new algorithm, when trying to eliminate `d_0`, first the body of the `let` is contracted, and within that the body of the `let` for `f_0`. It is there that `g_0` is eliminated, and now additionally the application counter for `f_0` is immediately decremented, thus becoming equal to zero. This allows for elimination of `f_0`, leading to a decrement of the application counter for `d_0`, which can then be eliminated as well. The definitions for `a_0` and `c_0` had already been eliminated in the first place (since they had been inlined), so in the first and only iteration step the entire term is rewritten to `M` at once, as Figure 4.12 shows as well.

4.3.2 Updated Implementation

Again, the implementation looks quite different from the algorithm, due to the separation of traversal and actual contracting by rewriting. As said, the new *Contract* algorithm differs from the naive algorithm on two key points: changed order of recursion and elimination, and better maintenance of the usage counts. The first point shows up in the traversal part of the code for `contract`, the second point in the rewriting part of the code.

The two differences are discussed separately, starting with the new traversal, the code is listed in Figure 4.13. Comparing the new `contract` strategy with the old one, shows that the first part is unchanged. Future inlining is still prepared by `Bind`, `ElimDead` tries to directly eliminate some definitions and `ContractApp` tries to inline record fields or function calls. The changed and new parts of the traversal are discussed now:

- The final choice alternative of the traversal, which used to be a `contract-td` is more complex now. `LookAhead` recursively applies `contract` only to the bodies of `let` expressions. This implements the various $N' \leftarrow \mathrm{contract}(N)$ fragments in

```
contract =
  try(bind)
; try((ElimDead <+ prep-inline(var-subst); ContractApp)
      < contract
      + LookAhead(contract, var-subst)
        ; (LookedAheadElim <+ CompleteTD(contract, var-subst))
     )

LookAhead(con, subst) =
  FunDefR(id, id, id, con)
+ FunApp(subst, map(subst))
+ RecBuild(id, map(subst), con)
+ RecFieldSel(id, id, subst, con)

CompleteTD(con, subst) =
  try(FunDefR(?f,id,rules(InlineFunDef :- |[ f(a*) ]| ); contract, id))

prep-inline(subst) =
  FunApp(subst, map(subst))
+ RecFieldSel(id, id, subst, id)

bind        = bind-FunDef + bind-RecBuild
ContractApp = InlineFunDef + EvalRecField
ElimDead    = ElimDeadFun + ElimDeadRec + ElimDeadRecField

LookedAheadElim =
  (ElimInlined <+ ElimRecDeadFun <+ ElimDeadFun)
+ ElimDeadRec
+ ElimDeadRecField
```

Figure 4.13: Traversal strategies for the new contract phase.

Figure 4.11. Additionally it applies all necessary variable substitutions $\sigma(\cdot)$ scattered around in the original algorithm.

- Once the input term has been sufficiently prepared, a second try is given to eliminate function definitions, record builds or record field selections, by `LookedAheadElim`. For most terms, this is just done by the original `elimDead..` rule, except for function definitions. The discussion on page 40 showed that a function definition can be eliminated for several reasons. It may have been inlined during the look-ahead. If not, but the application count was equal to one before the look-ahead already with no other uses, the function is apparently a recursive dead function. The final possibility is that during the look-ahead both counters have been decremented to 0, meaning that it is a normal dead function after all. The first two cases depend on contextual knowledge (Has the function been inlined during look ahead? Was the application counter already 1 before the look-ahead?), hence are handled by dynamic rules `ElimInlined`

and `ElimRecDeadFun`. These will be discussed hereafter. The final possibility is just the normal `elimDeadFun` that inspects the counters after the look-ahead.

- If elimination still fails, the topdown traversal is completed by `CompleteTD`. This topdown traversal is different from the old `contract-td`, since part of the topdown has already been performed in the `LookAhead`. This ensures that no unnecessary duplicate traversals occur. Note how it matches a function definition for f, and before recurring on the function body first undefines the dynamic inlining rule `InlineFunDef`. This implements the $\mathrm{Bind}[f] \leftarrow ()$ in Figure 4.11. The four other term sorts need no further treatment. Instead of using identity congruences for them, the function definition congruence is placed inside a `try(..)`.

In short: if normal elimination fails, the new algorithm first 'looks ahead' by recurring on `let` bodies and then tries to eliminate again. If that still fails, normal topdown behavior is completed.

The second major change in the algorithm was the better maintenance of all usage counts during the optimizations already. This is reflected in the various rewrite rules. All rewritings are shown in Figure 4.14, again using the counter manipulations from Figure 4.9. The core rewriting behavior for most rewritings is still the same, but as a side-effect, counters are updated. These and other differences include:

- The two dynamic rules defined by `BindFunDef` and `BindRecBuild` now contain counter updates. Additionally, new dynamic rules are defined, as will be discussed in the third point.

- The dead-variable elimination rules `ElimDeadFun`, `ElimDeadRecBuild` and `ElimDeadRecField` are even more similar. Only some additional calls to `census(|-1)`, `inc-esc(|-1)` and `inc-app(|-1)` are made.

- The main difference in the traversal was the look-ahead, finalized by `LookedAheadElim`. For elimination of function definitions, two new dynamic rules were used, which both can be seen in the definition of `bind-FunDef`. The existing dynamic rule definition for `InlineFunDef` now contains another `rules(..)` block, so when the dynamic rule `InlineFunDef` is applied, it defines a new dynamic rule `ElimInlined` itself. This eliminates the corresponding function definition, leaving only the body of the `let`. Hence, the inlining takes care of later elimination of the function definition itself. Note that this elimination rule does not include any counter decrementing, since that is not needed when the function has been inlined.
  Besides the inlining rule, `bind-FunDef` also generates a rule `ElimRecDeadFun` that detects and eliminates recursive dead functions. If the application count is 1, and further usage count is 0 (`is-inlineable`), the function definition is potentially a recursive dead function. If no elimination or contraction takes place in `contract` (Figure 4.13), the `LookAhead` immediately follows `bind`, hence this is a valid point to check the counters. When applied, the elimination rule decrements the counters for `M` (taken care of by `<census(|-1)> M`), since now `M` appears nowhere anymore

```
bind-FunDef =
  ?|[ let f(x*) = M in N ]|
; rules(
    InlineFunDef :
      |[ f(a*) ]| -> |[ M' ]|
      where <is-inlineable> f
            ; <zip(add-var-subst
               ; {?(x,a); <inc-app(|<get-app-count; dec> x)> a })> (x*, a*)
            ; <set-app(|0)> f
            ; <cps-rename> M => M'
            ; rules( ElimInlined : |[ let f(x*) = M in N' ]| -> |[ N' ]| )
  )
; try(
    where(<is-inlineable> f)
  ; rules( ElimRecDeadFun : |[ let f(x*) = M in N' ]| -> |[ N' ]|
                            where <census(|-1)> M )
  )

ElimDeadFun :
  |[ let f(x*) = M in N ]| -> |[ N ]|
  where <is-dead> f
        ; <census(|-1)> M

bind-RecBuild =
  ?|[ let r = <a*> in N1 ]|
; rules(
    EvalRecField :
      |[ let x = #i(r) in N2 ]| -> |[ N2 ]|
      where <index; var-subst> (<string-to-int> i, a*) => b'
          ; <add-var-subst> (x, b')
          ; <inc-app(|<get-app-count> x)> b'
          ; <inc-esc(|<get-esc-count> x)> b'
          ; <inc-app(|-1)> r
  )

ElimDeadRec :
  |[ let r = <a*> in N ]| -> |[ N ]|
  where <is-dead> r
      ; <map(var-subst; inc-esc(|-1))> a*

ElimDeadRecField :
  |[ let x = #i(a) in N ]| -> |[ N ]|
  where <is-dead> x
        ; <var-subst; inc-app(|-1)> a
```

Figure 4.14: Inlining and elimination rules for the new contract phase.

in the code.
The described two elimination rules implement the two 'if $\mathrm{Bind}[f] = \ldots$' cases in the first variant in Figure 4.11.

The change from an 'imperative approach' as used in the algorithm to a 'strategic approach' as used in the implementation is non-trivial and might seem unnecessary difficult. However, the eventual implementation shows a clean separation of term rewriting and term traversal. The actual traversal or rewriting strategy is formulated in a mere three lines of code (definition of `contract`), and the implementation of the actual inlinings and eliminations are intuitive by themselves. Much less checks are needed at elimination sites, since elimination rules have only been generated when necessary.

The elimination rules apply variable substitution where appropriate themselves, whereas the inlining rules `InlineFunDef` and `EvalRecField` don't do this for f, r and $x*$. `prep-inlining` has already taken care of this. The renaming could of course also have been put in the inlining rules themselves, using:

```
bind-FunDef =
  ?|[ let f(x*) = M in N ]|
; rules(InlineFunDef :
          |[ f'(a*) ]| -> |[ M' ]|
          where <var-subst> f' => f
                ; <is-inlineable> f
                // ...
```

and similar for record field selection. The disadvantage of this is that none of the left hand side variables of the dynamic rule `InlineFunDef` are bound in the context. Generated rules might start conflicting with each other, since their left hand sides overlap. Partially this can be solved by using *extend rules* (see also Chapter 6), but we chose the `prep-inline` solution here.

In short: the handling of function inlining works, because of the way it is placed inside the traversal. The other rules, for record field selection, and especially the elimination rules work perfectly by themselves and can be reused in different versions of the traversal.

4.3.3 Use of Dynamic Rules

The use of dynamic rules for counters and function inlining has not changed. One additional dynamic rule feature has been used, though.

We have seen how dynamic rules can be defined, but sometimes it is necessary to manually remove certain rules from the active rule set, even before the automatic removal upon exit of the dynamic rule scope. The manual removal is based on rule name and left hand side. The general syntax is `rules(RuleName :- t)`, in this case inlining rules were removed in `CompleteTD`, the function identifier is the left hand side term here.

This new implementation has also shown the nested use of dynamic rules: the definition of one dynamic rule contains another `rules(..)` block within its `where` condition. All context bound variables for the first rule are also context bound for the second (inner) rule definition. Take for example the definition of `ElimInlined` in `bind-FunDef`. Variables f,

*x** and *M* are context bound, whereas *N'* is not and is a static variable. Also note that *N'* has no relation whatsoever to *N* (at least not to the compiler). Here it is just used to reflect that a function definition *f*(*x**)=*M* may be eliminated, even though the body of the `let` it appears in has changed in the meantime.

Chapter 5

Constant Propagation in Tiger

Constant propagation is a forward data flow analysis that maintains information on which variables are known to be constant and replaces uses of these variables by their constant value [AK01]. Often, the constant propagation analysis delivers valuable information to and is combined with other optimizing transformations such as constant folding and unreachable code elimination [NNH99]. The basic principle is very simple, but when the transformation has to support more diverse language constructs such as local `let` bindings and control flow statements such as conditionals and loop statements, more care has to be taken when maintaining constant values and replacing variable occurrences with them. As such, this is a good case study for new dynamic rule constructs such as scope labeling, rule set forking and meeting, and fixpoint meeting.

The Tiger language was mentioned before and will again be the language under consideration in this chapter. As part of the Tiger-in-Stratego project [TIG], many optimizing transformations have already been implemented, of which constant propagation was one of the earliest. This chapter describes work by Olmos and Visser, initially described in [OV02], which has recently gained full benefit of the new dynamic rules.

5.1 The Tiger Language

Tiger is the example language from the compiler construction textbook by Appel [App98] and has served as the object language for many program transformations in Stratego. Although being an example language, Tiger is a versatile imperative language with support for typing, modular setup and the familiar statements, expressions and other language constructs. Figure 5.1 shows the syntax of the language constructs that are relevant in this chapter.

Note how there is no separation between statements and pure (side-effect-free) expressions. Simple expression values are either variables or string and integer constants. Boolean values are represented by integers, where 0 denotes `false` and any other integer `true`. Furthermore, all the familiar binary expression operators are available. The control structures are also expressions, but when used at a 'value position' they should yield a value of course. For example, the following expression is semantically invalid: `if x:=3 then ()`, since the

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

Figure 5.1: Abstract syntax of Tiger (incomplete).

assignment `x:=3` *only* has a side-effect and produces no further result at all.

Declarations in the declarations list of a `let` expression are directly visible to the subsequent declarations and to the `let` body, evidently. Function definitions may be nested by the use of nested `let` expressions, but we don't consider them here at all. The discussed constant propagation is *intra-procedural*, hence only applies to function bodies without nested function definitions inside. Local variable declarations as well as all other expressions are valid input.

5.2 Constant Folding and Propagation in Basic Blocks

Constant folding is a transformation that evaluates simple, static expressions. Static means in this context that no runtime information is needed, neither any data flow analysis. Often, constant folding is 'helped' by a combined constant propagation transformation. The latter maintains which variables are known to have a constant value and replaces occurrences of them with it. This usually gives rise to additional applications of constant folding rules.

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

EvalRelOp : |[ i = j ]| -> |[ k ]| where fail-succ-to-int(<eq>(i,j)) => k
EvalRelOp : |[ i < j ]| -> |[ k ]| where fail-succ-to-int(<lt>(i,j)) => k

fail-succ-to-int(s) =
  s < !1 + !0
```

Figure 5.2: Some constant folding rewrite rules for Tiger expressions.

5.2.1 Constant Folding

Constant folding is entirely based on basic algebraic properties. For example, an expression e+0 can be immediately simplified to e. Besides, binary arithmetic operators can be evaluated statically if both arguments are known constants. The same holds for relational operators. Some examples of constant folding rules are shown in Figure 5.2. In the tiger optimizations package about 40 rules of this sort exist, when primitive string operations are included.

The input of a transformation is of course seldomly a pure arithmetic or relational expression. When only constant folding is applied, all other expressions, declarations and further Tiger terms should be kept unchanged. Furthermore, the transformation should first descend to the leaves of the input tree, applying the folding rules to constant values, and then descending back up, i.e. a basic bottomup traversal:

```
fold = bottomup(EvalBinOp + EvalRelOp)
```

This bottomup approach ensures that constant folding result may give rise to new foldings higher up in the tree, for example:

```
|[ (10*3) + 1 ]|  ⇒  |[ 30 + 1 ]|  ⇒  |[ 31 ]|
```

As said however, constant folding is practically always combined with a data flow analysis that creates more opportunities for folding. The running example here is the constant propagation analysis.

5.2.2 Constant Propagation in Basic Blocks

We have seen the versatility of the Tiger language, but we will restrict ourselves to *basic blocks* for now. A basic block is a sequence of simple statements without control flow, i.e. a sequence of assignments, for example:

```
(x := 1; y := (p & q) + a; x := x + y)
```

The constant propagation analysis should inspect each variable assignment x := e, and if the right hand side expression e is a constant value, this should be registered for the

```
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

is-value = Int(is-int)
```

Figure 5.3: Strategy for constant propagation in basic blocks.

assigned variable x for later propagation. A basic dynamic rule `PropConst` does this, and is directly suitable for application at the site of a variable use:

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

Now, if an assignment is encountered, whose right hand side is not constant, no `PropConst` rule has to be defined, but instead any previously gained knowledge on that same variable should be discarded. This is to prevent unwanted propagation. Undefining `PropConst` for the current variable achieves this. The following example shows how a sequence of assignments (left box) is traversed in a forward manner, and which propagation information is known after each statement (middle box), resulting in an optimized program (right box).

```
(x := 1 + 2;
 y := 2 * x;
 x := x + 1;
 x := x + a;
 z := 2 * x)
```

⇒

```
x ↦ 3
x ↦ 3, y ↦ 6
x ↦ 4, y ↦ 6
x-,    y ↦ 6
x-,    y ↦ 6, z-
```

⇒

```
(x := 3;
 y := 6;
 x := 4;
 x := 4 + a;
 z := 2 * x)
```

The third assignment shows how an existing mapping can be redefined, just by defining a new rule. The notation `x-` means that `x` has been undefined and any previous mappings have been discarded.

Finally, how should the various parts be fitted together? An input term could be replaced by its constant value if it is a variable, or a bottomup traversal of its children should be made. If it is an assignment, only the right hand side child expression should be evaluated, followed by either a rule definition or undefinition. For any term other than an assignment, the normal bottomup traversal is applied, followed by an attempt to apply constant folding. Figure 5.3 lists the strategies for this approach. Notice how `AssignPropConst` examines the right hand side expression (which has already been transformed) and either defines or

undefines `PropConst` for the matched variable x. The check whether an expression is constant only succeeds for integer constants now, as the definition for `is-value` shows.

5.3 Constant Propagation for Local Variables

Although constant propagation is intra-procedural, local declarations for variables are valid input. Extending the transformation to handle not only basic blocks, but also `let` expressions, requires some additional measures in the implementation. Consider the following example of constant propagation:

```
(x := 1 + 2;
 y := 2 * x;
 let var x := x + 1;
     var y := a
  in x := x + a;
     let z:= 3
      in printint(z);
         y := 7
     end;
     x := y + z
 end;
 z := x*y)
```

⇒

```
x ↦ 3
x ↦ 3, y ↦ 6
x ↦ 4, y ↦ 6
x ↦ 4, y-
x-,    y-
x-,    y-,    z ↦ 3
x-,    y-,    z ↦ 3
x-,    y ↦ 7, z ↦ 3
x-,    y ↦ 7, z-
x-,    y ↦ 7, z-
x ↦ 3, y ↦ 6, z-
x ↦ 3, y ↦ 6, z ↦ 18
```

⇒

```
(x := 3;
 y := 6;
 let var x := 4;
     var y := a
  in x := 4 + a;
     let z:= 3
      in printint(3);
         y := 7
     end;
     x := 7 + z
 end;
 z := 18)
```

The middle box again displays the current state of the `PropConst` mapping after each statement. First of all, it will be clear that dynamic rule scopes are needed here for each `let` expression. The rule definitions (or undefinitions!) generated while traversing a `let` expression should not outlive the scope of that same `let`, instead any previous mappings should become visible again. A bit more subtle issue turns up when considering the generation of the mapping `y` $\mapsto$ 7. It is generated in the innermost scope of the `z-let`, but it actually applies to the scope for `y`. Normal dynamic rules are always associated to the inner scope they are defined in though, so upon exit of the inner scope, the rule for `y` would be discarded unwantedly. We need a more fine-grained control over association of rules to scopes. The bound variable renaming in Section 3.2.1 already sketched the use of defining rules within `override rules(..)` blocks. We will present the new and more expressive scope labeling mechanism now, which is just the right solution for this issue.

5.3.1 Implementation with Local Scopes

The new implementation resembles the basic blocks variant, with some scoping related additions. Figure 5.4 lists the new code. Before defaulting to the general bottomup traversal, `prop-const` checks whether the current term is a `let` expression and if so, it enters a new scope for the dynamic rule. A new strategy `DeclarePropConst` handles new variable declarations. It is similar to the original dynamic rule definition, but additionally it *labels* the current scope with the declared variable using:

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

```
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 5.4: Constant propagation for basic blocks with local variables.

A rule definition `rules(`L`+`p `:  `$p1$ `->` $p2$ `where` s`)` is shorthand notation for `rules(`L`+`p`); rules(`L `:  `$p1$ `->` $p2$ `where` s`)`. It assigns a new label to the current scope, and defines a new rule in the current scope. Hence, upon each new variable declaration its corresponding scope (which has just been entered) is labeled with the variable identifier. The rule definitions for normal assignments are now labeled using:

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

The rule definition looks up the most recent scope that has the specified label and associates this rule instance to that scope, so not the innermost by default. The example already showed that the mapping `y` $\mapsto$ `7` is indeed preserved upon exit of the scope labeled `z`, since it is associated to the scope one level higher, which is labeled `x` and `y`. Scope labels may be any closed term, any number of labels may be associated to a scope, and labels can already be specified upon entering the scope using {| L`.`p `:  `s |}. Scope labels can not be removed intermediately. The next section discusses the operational semantics of all scope labeling related constructs.

5.4 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:

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

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

scope entails adding a label to this list:

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

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:

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

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

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

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:

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

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:

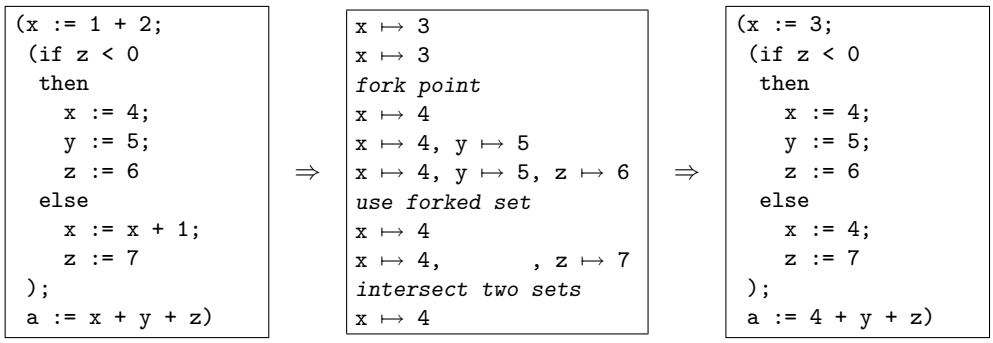
$$\{|\ L.p\ :\ s\ |\} \equiv \{|\ L\ :\ \texttt{rules}(L\texttt{+}p); s\ |\}$$

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.

5.5 Constant Propagation for Control Flow Statements

The final phase in our implementation of constant propagation is support for control flow statements. The first requirement for data flow optimizations is that they are safe: for any program execution the semantics of the transformed program should be identical to the original input program. When the input contains constructs that affect the control flow, such as conditionals and loop statements, the transformation should only use information that is consistent along all execution paths.

Conditionals: Forking and Meeting The following example shows constant propagation over an `if then else` statement:



In basic blocks, the constant propagation information was gained in a sequential manner, existing mappings could either be redefined or undefined. When control flow is split however, each branch should be evaluated starting with the information from directly before the split. In this example: constant propagation over the `else` branch should not use information gained in the `then` branch. The rule set is in fact *forked* into two sets, where each branch maintains its own rule set.

When both branches have been transformed, they meet again at the end of the `if` statement. For safe propagation over any following statements, only consistent rules from both branches should be maintained. This is achieved by computing the *intersection* of both rule sets and using that as the new rule set. In this example, only $\mathtt{x} \mapsto 4$ remains, since there is no mapping for `y` in the `else` branch, and the mappings for `z` are in conflict.

Loops: Reaching a Steady State Forking and meeting (intersecting) rule sets is also necessary for loop structures. The split is now not in two parallel branches, but one is over the loop body and the other is an unchanged branch, since it is not known whether the loop body will be executed at all. Furthermore, multiple iterations of the loop may change the rule set, so one single intersection is not enough in general. The initial approach [OV02] was to traverse the body twice, intersect with the original set, and traverse and intersect once again. Figure 5.5 shows a Tiger fragment in which each of the five variables are changed from 0 into 1, one per iteration. That is, if the loop would be traversed that often, which is unpredictable because of the unknown value for `p`. The only safe propagation results are produced when repeated propagation over the body followed by intersection with the rule set from the previous iteration yields no more changes in the resulting rule set: a fixpoint state should be reached. In this case, only $\mathtt{z} \mapsto 1$ is safe to propagate, and the other four variables can be either 0 or 1, hence should be considered unknown. The scheme at the right in Figure 5.5 shows the two intersection approaches. The old one first traverses twice, and then intersects with the initial rule set ('i'). A final traversal and intersection produces the resulting rule set in which `u` is still incorrectly considered to be constant 0. The new approach shows how each traversal ($\#1 \ldots \#5$) is followed by intersection with the previous

```
(v := 0;
 w := 0;
 x := 0;
 y := 0;
 z := 1;
 while (v = 0 & p) do (
   if (w = 1) then v := 1 else v := 0;
   if (x = 1) then w := 1 else w := 0;
   if (y = 1) then x := 1 else x := 0;
   if (z = 1) then y := 1 else y := 0
 ) ;
 printint(v + w + x + y + z))
```

#	v	w	x	y	z	Old intersection
i	0	0	0	0	1	
1	0	0	0	1	1	
2	0	0	1	1	1	→ 0 0 - - 1
3	0	-	-	1	1	→ 0 - - - 1
						Fixpoint intersection
i	0	0	0	0	1	
1	0	0	0	1	1	→ 0 0 0 - 1
2	0	0	-	1	1	→ 0 0 - - 1
3	0	-	-	1	1	→ 0 - - - 1
4	-	-	-	1	1	→ - - - - 1
5	-	-	-	1	1	→ - - - - 1

Figure 5.5: Forking and meeting over loop structures: reaching a fixpoint.

rule set, until the resulting set yields no changes anymore. Now, only z↦1 remains in the eventual rule set.

Unreachable Code Elimination Constant propagation delivers analysis information to a constant folding transformation to result in more folding than a static transformation would yield. In the same way, this analysis information can be used in an *unreachable code elimination* transformation. This transformation tries to eliminate all code parts that can be determined to never be reached by any execution path beforehand. This is simply done here by inspecting the conditions of `if then else` and loop expressions, which have already been evaluated as much as possible by the combined constant propagation. If the `if then else` condition evaluates to either true or false, one of the branches can be eliminated. If a loop condition eliminates to false, the entire expression can be eliminated. Figure 5.6 lists the rules that eliminate unreachable code. They will be used in the combined constant propagation implementation, which will now follow.

5.5.1 Implementation with Rule Set Forking and Meeting

The implementation that supports control flow statements is shown in Figure 5.7. It is identical to the previous implementation in Figure 5.4, except for the call to `prop-control`

```
EvalIf : ⟦ if i then e1 else e2 ⟧ -> e2 where <eq>(i,0)
EvalIf : ⟦ if i then e1 else e2 ⟧ -> e1 where <not(eq)> (i, 0)

EvalFor: ⟦ for x := i to j do e ⟧ -> ⟦ () ⟧ where <lt> (j,i)

EvalWhile: ⟦ while 0 do e ⟧ -> ⟦ () ⟧
```

Figure 5.6: Unreachable code elimination rewrite rules for Tiger expressions.

```
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 5.7: Intra-procedural constant propagation for expressions with local variables and structured control constructs.

in `prop-const`. This replaces the default bottomup behavior that otherwise would be defaulted to. Four variants of `prop-control` treat the two conditional and two loop statements.

Before doing any forking and intersection, first any unreachable statements are eliminated. Note how all four strategies apply constant propagation `s` to the condition or loop range, followed by a call to one of the unreachable code elimination rules. If this fails, normal constant propagation over the body of the statement is continued.

The fork-and-intersect behavior is implemented by the s_1 `/`L`\` s_2 strategy combinator. It applies two transformations s_1 and s_2 that both start with the original rule set for rules named L. The strategies s_1 and s_2 need no special modification whatsoever to fit in this forking approach. The resulting rule sets for L from the two paths are intersected afterward and stored as the new active rule set. For the so-called *may analyses* (as opposed to *must analyses*), the intersection operator should be replaced with the union operator. The combinator for this is s_1 `\`L`/` s_2. The treatment of an `if then else` statement shows how intuitive the combinators are:

```
⟦ if <id> then <s> else <id> ⟧ /PropConst\ ⟦ if <id> then <id> else <s> ⟧
```

The left strategy only applies constant propagation `s` to the `then` branch, and the right strategy does the same to the `else` branch. The forking and automatic intersection is in this case only performed for `PropConst`. A comma separated list of rule names may appear here though.

Finally, the fixpoint meeting is implemented by the `/`L`\*` s_1 strategy combinator. It remembers the current rule set for L, applies s_1 and intersects the resulting rule set with the originally remembered rule set, and repeats this until no more changes occur. s_1 is the only parameter strategy and in the case of loop statements, it should take care of descending into the loop body:

```
/PropConst\* ⟦ for <id> := <id> to <id> do <s> ⟧
```

Again, a variant for fixpoint *union* is also available: `\`L`/*` s_1.

5.6 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 afterward the intersection of the resulting rule sets is taken.

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

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

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

$$\vec{s} \cap \vec{s}' \equiv (s_1 \cap s_1')|...|(s_n \cap s_n') \qquad \vec{s} \cup \vec{s}' \equiv (s_1 \cup s_1')|...|(s_n \cup s_n')$$

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

$$s_1 \cap s_2 \equiv \texttt{<isect>(<}s_1\texttt{>,<}s_2\texttt{>)} \qquad s_1 \cup s_2 \equiv \texttt{<union>(<}s_1\texttt{>,<}s_2\texttt{>)}$$

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

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.

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

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

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

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.

Chapter 6

Deforestation of Functional Programs

Programs in functional languages often need intermediate data structures during execution. Intermediate lists, or trees in general, are the glue that keep composed functions together. For example, in the following expression:

$$\mathrm{sum}(\mathrm{map\ square\ (upto\ 1\ } n))$$

the list $[1, 2, \ldots, n]$ connects upto with map, and $[1, 4, \ldots, n^2]$ connects map with sum.

Intermediate trees come at a certain cost at runtime, taking up space when strict evaluation is used, or at least needing allocation and de-allocation when lazy evaluation is used. It would be best to avoid intermediate trees wherever possible, by allowing them only if they end up in the eventual result of the computation. For example, the above composition can be transformed into the following definition:

$$\begin{array}{l} h\ 0\ 1\ n \\ \text{where} \\ h\ a\ m\ n = \quad \text{if } m > n \\ \qquad\qquad\qquad\quad \text{then } a \\ \qquad\qquad\qquad\quad \text{else } h\ (a + \mathrm{square}\ m)\ (m + 1)\ n \end{array}$$

The resulting function h only performs the simple arithmetic computations, and does not need any intermediate lists anymore to traverse over the range $1 \ldots n$.

After some early work on listlessness [Wad84, Wad85], Philip Wadler presented an algorithm [Wad90] that transforms a term into a term without intermediate trees, a *treeless term*, that is when some necessary preconditions are satisfied. This chapter is a case study around Wadler's latter article. A small toy language, TFOF, was designed to create a working environment, and the strict deforestation algorithm was implemented in Stratego. Besides the convenient use of dynamic rules, the problem of the need for higher-order matching turns up here.

A less strict variant of the deforestation algorithm does allow intermediate values as long as they are simple values like integer and boolean constants, producing terms in so-called *blazed treeless form*. The above 'treeless' term is in fact in blazed treeless form.

M	::=	`module` M $d*$	module	`Module`
d	::=	`imports` $M*$	import declarations	`Imports`
		`functions` $fd*$	function declarations	`FunDefs`
		`terms` $td*$	term declarations	`TermDefs`
fd	::=	$f\ v_1 \ldots v_k = t$	function definition	`FunDef`
td	::=	$q := t$	term definition	`TermDef`
t	::=	v	variable	`Var`
		$c\ t_1 \ldots t_k$	constructor application	`Con`
		$f\ t_1 \ldots t_k$	function application	`FunApp`
		`case` t_0 `of` $p_1 : t_1 \mid \cdots \mid p_n : t_n$	case term	`Case`
		n	integer constant	`Int`
		`true` | `false`	boolean literal	`True` | `False`
p	::=	$c\ v_1 \ldots v_k$	pattern	`Pat`
M	::=	identifier	module identifier	
q	::=	identifier	term identifier	
f	::=	identifier	function identifier	
v	::=	identifier	variable identifier	
c	::=	identifier	constructor identifier	

Figure 6.1: Syntax of the TFOF language.

The algorithm is very similar to the original algorithm, hence its implementation is not discussed here. An additional requirement of this algorithm however, is that it receives full typing information for its input term. To facilitate this, a type inferencer was implemented, based on Milner's $\mathcal{W}$ algorithm. This forms a nice case study in itself and besides, it allows for a comparison with a similar type inferencer [DS01] in the UUAG attribute grammar system [SAS98]. Appendix B introduces some new syntax for a typed TFOF language, and discusses the implementation of the type inferencer, illustrated by some examples.

The resulting package, consisting of a parser and pretty-printer, the deforestation tools and the type inferencer, and several example files, is available on-line [TFO].

6.1 The TFOF Language

To create an accessible test environment, a toy language has been designed. The Tiny First Order Functional language (TFOF) is based on the grammar in Wadler's article, with some practical extensions around it. Figure 6.1 lists the syntax of the TFOF language, the following sections will explain the several parts of the syntax. Section B.1 will extend the syntax definition with syntax for types and type annotations.

6.1.1 Basic Terms

Basic terms t are the building blocks of TFOF, they are either variables, function or constructor applications or `case` terms. Patterns p are constructor applications with mere variables as children, so no nested patterns are allowed. Note that in this context terms and patterns should not be confused with terms and patterns in the more general context of program representation and rewriting, as seen in Chapter 2. The variables in the pattern of a `case` branch introduce a local scope for those variables in the body of the branch. Notice in `fappend` in Figure 6.2 how the second branch introduces a local scope for `x` and `xs`, the `xs` in the body refers to the matched part in the pattern, not the original function argument.

Function definitions allow for function composition in terms by names. Term definitions are just a convenient way of placing several terms in one test file. Terms can not be re-used in other terms as if they were variables, although this would be a very simple extension.

Linearity To support the upcoming definition of the 'treeless' concept, the linearity property should be introduced first. A term is said to be linear, if no variable appears in it more than just once. For example, `fappend xs ys` is linear, whereas `fappend xs xs` is not. This should hold for all term sorts in Figure 6.1, except for the `case`-term. A `case`-term is linear if no variable appears in both the selector and any branch, and if all branches are linear by themselves. Variables may appear in more than one branch though, as the linear definition of `fappend` shows (`ys` is in both branches but in both just once).

6.1.2 Modular Setup

On top of the core term syntax, a syntax for modular setup of TFOF programs is available. Each TFOF file should start with a `module` header that specifies its unique name, followed by an optional number of `import` statements, or function and term definitions in `functions` and `terms` blocks. Finally, for line and block comments the `//` and `/* .. */` syntax, familiar from several other languages, is used. The package comes with several tools that parse input files, resolve import statements, pretty-print output and such more. No special care is taken in handling multiple definitions with the same name, or verifying all function dependencies. Below is a small example program that relies on other modules:

```
module leaves
imports datatypes list-arith tree-basic

functions
  fsumleaves t = fsum (fleaves t)

terms
  t3 := fsumleaves (Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)))
  t6 := fleaves x
```

TFOF syntax does not explicitly require a definition for any user-defined data type, for example for the `Leaf` constructor. For later use as input to the type inferencer though, im-

```
module datatypes
types
  data List a = Cons a (List a)
              | Nil

  data Tree a = Branch (Tree a) (Tree a)
              | Leaf a

module list-basic
imports datatypes
functions
  fappend : (List a) -> (List a) -> (List a)
  fappend xs ys =
    case xs of
      Nil       : ys
      Cons x xs : Cons x (fappend xs ys)

module tree-basic
imports datatypes list-basic
functions
  flip : (Tree a) -> (Tree a)
  flip zt = case zt of
              Leaf z       : Leaf z
              Branch xt yt : Branch (flip yt) (flip xt)
```

Figure 6.2: Three TFOF modules with basic functionality on lists and trees.

porting the definitions (in module `datatypes`) is necessary. Furthermore, the `fsumleaves` function definition is in the module itself, whereas the definition for `fleaves` should be in one of the imported modules. Finally note that terms can contain undefined variables (`x` in `t6`), since we are not building an interpreter anyway, but instead deal with transformations that produce semantically equivalent programs.

Figure 6.2 lists some excerpts from three modules (so three files as well) that are useful as a reference for the examples in the text. Type annotations are intuitive here, but will be formally introduced in Section B.1.

6.2 Alpha Equality and Renamed Terms

Terms can of course always be compared with each other based on strict equality. The deforestation algorithm needs a less strict form of equality checking: *alpha equality*. This is a form of equality under α-conversion, denoted with $=_\alpha$. α-conversion is the process of renaming bound variables in a term, not necessarily each of them. Alpha equality originates from the general theory of lambda calculus [Wik], the first lambda abstraction below is a very simple example of it. The second and third equations below specify the two constructs

in TFOF that bind variables: function definitions and case branches.

$$\begin{array}{lll} \lambda x.x + 1 & =_\alpha & \lambda y.y + 1 \\ f\ x_1 \ldots x_k = t & =_\alpha & f\ y_1 \ldots y_k = t\{y_1/x_1, \ldots, y_k/x_k\} \\ \texttt{case}\ t_0\ \texttt{of}\ [c\ v_1 \ldots v_n : t] & =_\alpha & \texttt{case}\ t_0\ \texttt{of}\ [c\ w_1 \ldots w_n : t\{w_1/v_1 \ldots, w_n/v_n\}] \end{array}$$

In the above equations, all substituted variables y_i and w_i should be distinct, and not appear as free variables in the original expressions t.

6.2.1 Alpha Conversions and Renamings

Equality under α-conversion is one of the standard ways to compare two terms. Others involve β-reduction and η-conversion, but we don't consider them here. Instead, the concept of α-equality needs some generalization. To compare two terms on their similarity, renaming of *unbound* variables could also be allowed. If this leads to equality, a term is said to be a *renaming* of the other term, here denoted with $\sim$. The examples below summarize the idea and differences between the two:

α-equality			is renaming		
$f\ x = x + 1$	$=_\alpha$	$f\ y = y + 1$	$f\ x = x + 1$	$\sim$	$f\ y = y + 1$
$y + (f\ y)$	$\neq_\alpha$	$z + (f\ z)$	$y + (f\ y)$	$\sim$	$z + (f\ z)$
$y + (g\ x)$	$\neq_\alpha$	$y + (f\ x)$	$y + (g\ x)$	$\nsim$	$y + (f\ x)$
$y + (g\ x)$	$\neq_\alpha$	$y + (f\ x)$	$y + (g\ x)$	$\sim_f$	$y + (f\ x)$

Only the first example concerns α-equal terms. The second example is a renaming, since the unbound variable y can be renamed consequently to z. The third example is neither α-equal, nor a renaming, since the function identifier is different (f vs. g) and usually it is not desirable to allow function renamings (the functions may have entirely different semantics). In some cases though, function renaming may be desirable to allow. For example when the function definitions are checked upon similarity as well. If the function definition of g is a renaming of the definition of f, then the function application $g\ x$ can be considered a renaming of $f\ x$ without a problem. This kind of similarity is denoted with $\sim_f$.Given the above definitions, the following proposition holds:

$$t_1 = t_2 \quad \Rightarrow \quad t_1 =_\alpha t2 \quad \Rightarrow \quad t_1 \sim t2 \quad \Rightarrow \quad t_1 \sim_f t2$$

Term renamings with function renamings allowed are a useful method of comparing the sets of helper functions generated during the deforestation algorithm in Section 6.4. The 'normal' renamings are used to detect when infinite recursion is about to set in.

6.2.2 A Generic Equality Check

When comparing two terms to determine whether they are α-equal to each other, several approaches can be followed. One is to rename all bound variables in both terms in a standard way, e.g. by maintaining a fresh-variable counter. If the renamed terms are identical, the original terms were α-equal. Although simple and easy to implement, this approach has to

```
eq-gen(
  get-vars, get-nonvar-part, eq-bound, eq-unbound,
  get-bound-vars, get-bound-part, get-unbound-part,
  eq-over-set) =
{| AE-VarMap, AE-VarMapRev, AE-IsBound, AE-IsBoundRev:
  let compare =
        eq-gen-unscoped(get-vars, get-nonvar-part, eq-bound, eq-unbound,
                        get-bound-vars, get-bound-part, get-unbound-part)
   in if eq-over-set
      then set-eq(compare)
      else compare
      end
  end
|}

eq-gen-unscoped(
  get-vars, get-nonvar-part, eq-bound, eq-unbound,
  get-bound-vars, get-bound-part, get-unbound-part) =
rec x({| DontDescend:
  { ?(C#(as), C#(bs))
  ; ( where((get-vars, get-vars) => vs)
      < where(<zip((AE-IsBound, AE-IsBoundRev) < eq-bound + eq-unbound)> vs)
        ; where((get-nonvar-part, get-nonvar-part) < x + id)
        ; rules(DontDescend: ())
      + id)
  ; {| AE-VarMap, AE-VarMapRev, AE-IsBound, AE-IsBoundRev :
      where((get-bound-vars, get-bound-vars) => bvs)
      < where(<zip((mk-bound, mk-bound-rev); add-var-map)> bvs)
        ; where((get-bound-part, get-bound-part) < x + id)
        ; rules(DontDescend: ())
      + id
    |}
  ; where((get-unbound-part, get-unbound-part) < x + id)
  ; (where(<DontDescend> ()) <+ where(<zip(x)> (as, bs)))
  }
|}
)

eq-by-rename =
  ?(v1, v2)
; where(
  ( <AE-VarMap> v1
    <+ rules(AE-VarMap: v1 -> v2); !v2
  ) => v2)

add-var-map  = ?(u, v); rules(AE-VarMap: u -> v)
mk-bound     = ?v; rules(AE-IsBound: v)
mk-bound-rev = ?v; rules(AE-IsBoundRev: v)
```

Figure 6.3: Implementation of a generic equality checker.

traverse two entire terms, and compare them afterward, which can become unnecessarily expensive.

Instead, we designed and implemented a generic equality checker, parameterizable with strategies that specify the nature of equality checking. Figure 6.3 lists the generic code and shows that 8 parameter strategies need to be specified. The following assumptions and choices were made during the design:

- Variables can not always be identified by shape, it might depend on the direct context. For example within TFOF, function identifiers (which can be renamed in the $\sim_f$-check) are just string literals. However, they should not be confused with normal string literals, hence they need to be identified by their context (in the TFOF case either a `FunApp`, or a `FunDefR` in abstract syntax). The first parameter strategy `get-vars` should produce a list of variables contained in the term or its direct children.

- The second parameter strategy should produce the remaining non-variable parts of the term, which will *not* be treated by the (un)bound parts traversal strategies (see next points). Those will then be compared as well, by the recursive call to `x`. In this case only the arguments of a function call.

- For bound and unbound variables, different equality checks may be used. For α-equality only bound variables are compared using renaming where needed. Unbound variables are compared using strict equality. To check whether a term is a renaming of another, both bound and unbound variables are compared with optional renaming. When function renaming is allowed, even more liberal strategies should be passed as the third and fourth argument.

- Three strategies specify which are the newly bound variables by a language construct, which are the parts that fall in the scope of those bound variables, and which are the unbound parts that fall outside of that same scope.

- As we are allowing terms to contain variables that need to be checked upon equality *and* to act as a binding construct, both ways are tried in `eq-gen`. Besides, when either of these applies, it takes care of the further equality check of its children itself. The application of `x` to the child parts achieves this. Only if neither applied, `eq-gen` automatically compares each pair of child terms. To avoid code duplication and nested `if then else` statements, a dummy dynamic rule `DontDescend` is defined as to indicate that no automatic traversal into the children is necessary anymore. Note that the very first part of the implementation, checks whether the form (i.e. constructor name `C`) is identical at all, resulting in early detection of inequality.

- When allowing all kinds of renamings during equality checking, it makes sense to keep these renamings consistent across two sets of terms. As in the example with function renaming: that is only sensible for function applications if the function definitions themselves are also equal under renaming. Equality over sets is facilitated by `eq-gen`, notice how it enters the relevant dynamic rule scopes. The actual strategy

`eq-gen-unscoped` does not, such that renaming rules are maintained for upcoming comparisons, if needed. The comparison of two sets is performed by the `set-eq` in the SSL[1].

- The final strategy `eq-by-rename` is a helper strategy that can be used by instances of this `eq-gen` as the argument for the `eq-(un)bound` parameter strategies. It checks whether a renaming already exists for its first input term `v1`, if so it applies it and compares the result to the other term `v2`. If not, a new renaming `v1 -> v2` is generated and the strategy always succeeds.

6.2.3 Equality Checking in TFOF

Now that a generic equality checker is available, several equality checks can be implemented with very little effort, by defining some instances of `eq-gen`. Figure 6.4 lists the instances for the three types of equality introduced in Section 6.2.1.

- α-equality just selects real variables (`Var`) for renaming, and has no non-variable parts to return. Only bound variables are checked under renaming by the `eq-by-rename` from Figure 6.3, unbound variables are checked strictly (`eq`). The binding constructs are either patterns from `case` branches or function definitions.
- Checking of renamings is almost identical to checking of α-equality, except for the unbound variables, which are now also compared using `eq-by-rename`.
- Finally, equality with function renamings allowed also selects the function identifiers for renaming, and has a non-variable part to return. This is the argument list of a function application, which used to be treated by the automatic descend into child terms, but now falls under the check after `get-vars`.

Notice how the variable binding strategies are unchanged for the three variants. The real difference is just in which variable terms should be selected for renaming. The implementation with dynamic rules is also completely hidden from the user, all generations and proper scopings are handled by the generic strategy.

6.3 Treeless Form

To formally describe the elimination of intermediate values, *treeless form* is now introduced. Given a set of function names F, a term t is said to be treeless with respect to F if it is linear, it only contains functions in F, and every argument in a function call and every selector of a `case` statement is a variable. In our familiar BNF-notation, where tt now denotes a treeless term:

$$
\begin{array}{rcl}
tt & ::= & v \\
 & & c\ tt_1 \ldots tt_k \\
 & & f\ v_1 \ldots v_k \\
 & & \texttt{case}\ v_0\ \texttt{of}\ p_1 : tt_1 \mid \cdots \mid p_n : tt_n
\end{array}
$$

[1] Stratego Standard Library, http://www.stratego-language.org/Stratego/StrategoStandardLibrary

```
is-alpha-eq =
  eq-gen(get-vars, fail, eq-by-rename, eq,
         get-bound-vars, get-bound-part, get-nonbound-part)

is-renaming =
  eq-gen(get-vars, fail, eq-by-rename, eq-by-rename,
         get-bound-vars, get-bound-part, get-nonbound-part)

is-renaming-w-funs =
  eq-gen(get-vars-w-funs, get-nonvar-part-w-funs,
          eq-by-rename, eq-by-rename,
          get-bound-vars, get-bound-part, get-nonbound-part)

get-vars =
  ?Var(<![<id>]>)

get-vars-w-funs =
  ( ?Var(<id>)
  + ?FunDef(<id>, _, _)
  + ?FunApp(<id>, _))
; ![<id>]

get-nonvar-part-w-funs =
  ?FunApp(_, <id>)

get-bound-vars =
  ?(Pat(_, <map(?Var(<id>))>), _)
+ ?FunDef(_, <map(?Var(<id>))>, _)

get-bound-part =
  ?(Pat(_, _), <id>)
+ ?FunDef(_,_,<id>)

get-nonbound-part =
  fail
```

Figure 6.4: Instances of the generic equality checker for TFOF.

where, in addition, tt is linear and $f \in F$.

A set of function definitions F is treeless if each function body is treeless itself with respect to F. In Figure 6.2, both `fappend` and `flip` are treeless, whereas `fleaves` is not, the latter because it contains a call to `fappend` with arguments that are no variables. Note that in `flip` non-variable arguments *are* allowed, since it involves a constructor application there.

The reason for requiring function call arguments and case selectors to be variables is evident: otherwise intermediate trees would be allowed, which is the very thing we want to avoid. In constructor applications however, terms *are* allowed as arguments, since all constructor applications will end up in the eventual result anyway. The linearity requirement ensures that no expensive computations are duplicated when unfolding is used. Unfolding, or function inlining, replaces a function call with the function body where the formal parameters are replaced with the actual arguments. If the formal parameters appear more than once in the function body (i.e. the body is non-linear), and the actual arguments are intermediate computations themselves, these computations are duplicated or worse. In certain cases it is a good option to allow non-linear terms and function bodies and store the actual arguments in variables by using local `let` blocks. This does introduce intermediate terms, but that is acceptable when those are mere constant values. This flexibility is employed by the blazed deforestation algorithm, which is not further considered here.

Deforestation Theorem Now that the background and reasoning behind deforestation is clear, the leading theorem can be presented:

> Every composition of functions with treeless definitions can be effectively transformed to a single function with a treeless definition, without loss of efficiency.

The transformation is performed by the deforestation algorithm, which is treated in the next section. Input to the algorithm is a linear term and a collection of treeless function definitions, output is a treeless term and a collection of the same functions, possibly extended with new helper functions that achieve the treeless property. In our implementation the transformation can handle a *collection of* terms as input, and it shares the generated helper functions among them by memoization, as to avoid duplicate generation of identical helper functions. Finally, Figure 6.5 shows both a non-treeless and a treeless definition for list flattening. The next section will use these in another example.

6.4 The Deforestation Algorithm

The Deforestation Algorithm will now be treated. The first section introduces the original algorithm by Wadler and the reasoning behind it. The second section will explain the transition from the original algorithm to a strategic formulation and Stratego implementation, just as was done in Chapter 4.

6.4.1 Algorithm

The original algorithm by Wadler is specified by seven recursive relations, which are shown in Figure 6.6. The transformation of a valid input term t into its treeless form is denoted

```
flatten0 : (List(List a)) -> (List a)
flatten0 xss =
  case xss of
    Nil         : Nil
    Cons xs xss : fappend xs (flatten0 xss)

flatten1 : (List(List a)) -> (List a)
flatten1 xss =
  case xss of
    Nil         : Nil
    Cons xs xss : flatten1' xs xss

flatten1' : (List a) -> (List(List a)) -> (List a)
flatten1' xs xss =
  case xs of
    Nil         : flatten1 xss
    Cons x xs   : Cons x (flatten1' xs xss)
```

Figure 6.5: A non-treeless and a treeless definition for list flattening.

$$
\begin{array}{lll}
(1) & T[\![v]\!] & = v \\
(2) & T[\![c\ t_1 \ldots t_k]\!] & = c\ T[\![t_1]\!] \ldots T[\![t_k]\!] \\
(3) & T[\![f\ t_1 \ldots t_k]\!] & = T[\![t[t_1/v_1, \ldots, t_k/v_k]]\!] \\
 & & \quad \text{where } f \text{ is defined by } f\ v_1 \ldots v_k = t \\
(4) & T[\![\texttt{case}\ v\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!] & \\
 & = \texttt{case}\ v\ \texttt{of}\ p'_1 : T[\![t'_1]\!] \mid \cdots \mid p'_n : T[\![t'_n]\!] & \\
(5) & T[\![\texttt{case}\ c\ t_1 \ldots t_k\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!] & \\
 & = T[\![t_i[t_1/v_1, \ldots, t_k/v_k]]\!] & \\
 & \quad \text{where } p'_i = c\ v_1 \ldots v_k & \\
(6) & T[\![\texttt{case}\ f\ t_1 \ldots t_k\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!] & \\
 & = T[\![\texttt{case}\ t[t_1/v_1, \ldots, t_k/v_k]\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!] & \\
 & \quad \text{where } f \text{ is defined by } f\ v_1 \ldots v_k = t & \\
(7) & T[\![\texttt{case}\ (\texttt{case}\ t_0\ \texttt{of}\ p_1 : t_1 \mid \cdots \mid p_m : t_m)\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n]\!] & \\
 & = T[\![\texttt{case}\ t_0\ \texttt{of} & \\
 & \qquad p_1\ :\ (\texttt{case}\ t_1\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n) & \\
 & \qquad \cdots & \\
 & \qquad p_m\ :\ (\texttt{case}\ t_m\ \texttt{of}\ p'_1 : t'_1 \mid \cdots \mid p'_n : t'_n)]\!] &
\end{array}
$$

Figure 6.6: Transformation rules for the Deforestation Algorithm. (Source: [Wad90])

as $T[\![t]\!]$. It was already stated that valid input is linear, and the transformed term will be semantically equal.

Rules $(1)-(3)$ treat the first three kinds of term, whereas four rules $(4)-(7)$ treat a case term, for each possible selector separately. Variables remain the same (1), and (2) and (4) just apply the transformation recursively to each of the child terms. Function applications are unfolded in rules (3) and (6), after which the inlined function body is transformed recursively. Both in the algorithm and in the eventual implementation it is assumed that all called functions are known and available for unfolding. When the case-selector is a constructor application, it is immediately possible to find the matching pattern for that constructor. The selected branch is lifted out and transformed recursively (5). Finally, rule (7) lifts the inner selector t_0 to the top level and rearranges the branches to maintain an equivalent term. The result is again transformed recursively. In this latter case none of the pattern variables in p_i should occur free in any of the branches $p'_i : t'_i$, since otherwise variable capture would occur in the right hand side term. This requirement is easily satisfied by uniquely renaming all bound variables in the input term.

The word count for 'recursively' in the previous paragraph already suggests the risky possibility of infinite recursion. Indeed, the algorithm as given does not always terminate. Unfolding of self-recursive functions is guaranteed to run forever, and the recursion may be hidden even more deeply and through some intermediate function calls. In Wadler's article the step-by-step transformation of $\mathrm{flip}(\mathrm{flip}(zt))$ is shown, up to the point that $T[\![\mathrm{flip}(\mathrm{flip}(xt))]\!]$ is encountered (and for yt as well). This is the point where infinite recursion is about to set in. Now the trick is to introduce appropriate helper functions whose definition (i.e. body) needs no further transformation: $h\ zt = T[\![\mathrm{flip}(\mathrm{flip}(zt))]\!]$. We will further explain this approach using our own example.

Figure 6.7 shows all steps by the deforestation algorithm when applied to $\mathrm{flatten}_0\ bss$, with $\mathrm{flatten}_0$ defined as in Figure 6.5. The arrows with numbers denote which rules from the algorithm were used in each step, sometimes with additional explanation. The first four steps are basically following the rules, with $\mathrm{flatten}_0$ and append unfolded in the first and third step respectively. Now in the fourth step, $T[\![\mathrm{flatten}_0\ yss]\!]$ turns up and following the rules, another unfold would be done in the next step. However, since a function application of $\mathrm{flatten}_0$ was encountered and unfolded before (first step) and the current term is very similar to that one (it is a renaming), a helper function should be generated.

Infinite recursion is only possible when unfolding takes place, i.e. in rules (3) and (6). To detect possible recursion later on, for each application of these two rules, the input term is remembered. Before applying either of these rules, it is first checked whether the current term is a renaming of any term previously encountered (and unfolded). In our example, in the first step $\mathrm{flatten}_0\ bss$ was 'remembered', just before unfolding. Now, in the fifth step, before applying (3) to $\mathrm{flatten}_0\ yss$, it is detected that this is actually a renaming of $\mathrm{flatten}_0\ bss$. Instead of unfolding again, the term is replaced by a call to a new helper function h_0, with all free variables in the current term as arguments, in this case only yss. Directly in the next step, a similar situation occurs: $\mathrm{append}\ xs\ (\mathrm{flatten}_0\ yss)$ is a renaming of the term that was unfolded in the third step. Again, a new helper function h_1 is introduced, now with two arguments xs and yss.

$T[\![\text{flatten}_0\ bss]\!]$

$\downarrow$ (3) Unfold flatten0, remember as h_0.

$= T[\![$`case` bss `of`
Nil : Nil
Cons $ys\ yss$: append ys (flatten$_0$ yss)$]\!]$

$\downarrow$ (4)

$=$ `case` bss `of`
Nil : $T[\![\text{Nil}]\!]$
Cons $ys\ yss$: $T[\![\text{append}\ ys\ (\text{flatten}_0\ yss)]\!]$

$\downarrow$ (2), (3) Unfold append, remember as h_1.

$=$ `case` bss `of`
Nil : Nil
Cons $ys\ yss$: $T[\![$`case` ys `of`
Nil : flatten$_0$ yss
Cons $x\ xs$: Cons x (append xs (flatten$_0$ yss))$]\!]$

$\downarrow$ (4)

$=$ `case` bss `of`
Nil : Nil
Cons $ys\ yss$: `case` ys `of`
Nil : $T[\![\text{flatten}_0\ yss]\!]$
Cons $x\ xs$: $T[\![\text{Cons}\ x\ (\text{append}\ xs\ (\text{flatten}_0\ yss))]\!]$

$\downarrow$ $(h_0), (2)$

$=$ `case` bss `of`
Nil : Nil
Cons $ys\ yss$: `case` ys `of`
Nil : $h_0\ yss$
Cons $x\ xs$: Cons $(T[\![x]\!])$ $(T[\![\text{append}\ xs\ (\text{flatten}_0\ yss)]\!])$

$\downarrow$ $(1), (h_1)$

$=$ `case` bss `of`
Nil : Nil
Cons $ys\ yss$: `case` ys `of`
Nil : $h_0\ yss$
Cons $x\ xs$: Cons x $(h_1\ xs\ yss)$

$\downarrow$ Fold definition of h_1.

$=$ `case` bss `of`
Nil : Nil
Cons $ys\ yss$: $h_1\ ys\ yss$

$\downarrow$ Fold definition of h_0.

$= h_0\ bss$

Figure 6.7: Stepwise deforestation of flatten$_0$ bss.

It is still unclear how the exact definition of helper functions is determined. When rule (3) or (6) is about to unfold, it remembers its input term, as we just saw. The unfolded term is then transformed recursively, possibly using a new helper function. When the transformation of the unfolded term has finished, it should be checked whether the helper function based on the input term was indeed used. If this is the case, the new helper function should have exactly the transformed term as its body. In our example in the two last steps, this check is performed and leads to two new function definitions:

$$
\begin{array}{lcll}
h_0\ yss & = & \texttt{case}\ yss\ \texttt{of} & \\
 & & \mathrm{Nil} & : \quad \mathrm{Nil} \\
 & & \mathrm{Cons}\ ys\ yss & : \quad h_1\ ys\ yss \\
 & & & \\
h_1\ ys\ yss & = & \texttt{case}\ ys\ \texttt{of} & \\
 & & \mathrm{Nil} & : \quad h_0\ yss \\
 & & \mathrm{Cons}\ x\ xs & : \quad \mathrm{Cons}\ x\ (h_1\ xs\ yss)
\end{array}
$$

Together with these two generated helper functions, $h_0\ bss$ is the result of $T[\![\mathrm{flatten}_0\ bss]\!]$.

The attentive reader may have noticed already that the input to the algorithm was actually not valid at all: the input term is linear, but the set of functions, $F = \{\mathrm{flatten}_0, \mathrm{append}\}$ contains $\mathrm{flatten}_0$, which has no treeless definition. What is even more surprising is that the two generated helper functions h_0 and h_1 are in fact renamings of the treeless variants $\mathrm{flatten}_1$ and $\mathrm{flatten}'_1$ in Figure 6.5.

The described trick for preventing infinite recursion turns out to work fine in our example and Wadler proves in his article that termination is guaranteed for this approach.

6.4.2 Implementation in Stratego

The core of the Stratego implementation of the deforestation algorithm is of course formed by the seven rewrite rules from the original algorithm. To allow for easier testing and experimenting though, some additional language features were put into TFOF. Due to the modular setup, code (e.g. function definitions) need to be retrieved from other files before any deforestation can take place (Figure 6.8). Next, multiple terms may have to be transformed (Figure 6.9), with sharing of generated helper functions (Figure 6.10). After deforestation of all input terms, a valid TFOF-module should be produced as output.

Preparation of Input, Finalization of Output One of the tools in the TFOF package is `pack-tfof`, which given one input file, parses it and resolves all imported modules and packs them into one big self-contained module. This abstract syntax tree can be fed into `tfof-deforest` which is the wrapper around the actual deforestation algorithm. The relevant code is shown in Figure 6.8. Some self-explanatory strategies for manipulating TFOF modules and declaration blocks, and for (bound) variable renaming are not included. `tfof-deforest` receives the packed `Module` as input and splits the various declaration blocks into a tuple of type, function and term definitions. Strict deforestation is applied to the term definitions, with the function definitions as term argument, the type definitions play no role here. The eventual output of `tfof-deforest-strict` will be a tuple of newly

```
deforest =
  ?Module(m,_)
; where(collect-defs => (tpd*, fd*, td*))
; <tfof-deforest-strict(|fd*)> td*
; (MkFunDefs, MkTermDefs); TupleToList
; ![FunDefs(fd*)|<id>]
; <MkModule> (m, <id>)

tfof-deforest-strict(|fd*) =
{| InlineFun, VarSubst, HelperFun, HelperIsUsed, RecursiveHelper:
  where(<map(prepare-inlining)> fd*)
; map(tfof-deforest-strict-unscoped)
; !(<bagof-RecursiveHelper> (),<id>)
|}

tfof-deforest-strict-unscoped =
  TermDef(id, tfof-rename-bound-vars; transform-to-treeless)

prepare-inlining =
  ?|[ f v* = t ]|
; where(<length> v* => nargs)
; rules(
    InlineFun :
      |[ f t* ]| -> |[ t' ]|
      where <length> t* => nargs
          ; <tfof-rename-bound-vars; tfof-var-subst(|(v*, t*))> t => t'
  )
```

Figure 6.8: Strategies for input preparation and output finalization.

generated definitions for helper functions, and the transformed term definitions. These are put into appropriate declaration sections and packed together with the original function definitions into an output module.

`tfof-deforest-strict` enters the appropriate dynamic rule scopes, which are global across all functions and terms that are to be transformed. For each function definition an unfolding rule `InlineFun` is defined by `prepare-inlining`. Before inlining the argument count is verified, and the bound variables in the function body are renamed to avoid variable capture,followed by instantiation of the body by replacing all formal parameters with the actual calling arguments. Note that all of this will be performed at the call site of `InlineFun`, i.e. at the very moment of inlining.

Next, for each term definition bound variables are renamed in the body, followed by the actual deforestation transformation, which will be discussed hereafter. Once finished, all generated helper functions are retrieved by `bagof-RecursiveHelper`, which returns all successful applications of the dynamic rule `RecursiveHelper`. The used left hand side for

```
transform-to-treeless =
  alltd(
    RecFunApp(transform-to-treeless)          // (3), (6)
  + BranchSelect; transform-to-treeless       // (5)
  + TreelessCaseCase; transform-to-treeless   // (7)
  )

BranchSelect :
  |[ case C t* of b* ]| -> ti'
  where <length> t* => nargs
      ; <fetch-elem(
          {vi*, ti:
            ?(Pat(C, vi*), ti)
          ; <length> vi* => nargs
          ; <tfof-var-subst(|(vi*, t*))> ti
          })
        > b* => ti'

TreelessCaseCase:
  |[ case (case t0 of b*) of b'* ]| -> |[ case t0 of b''* ]|
  where <map(\ (pi, ti) -> (pi, |[ case ti of b'* ]|) \)> b* => b''*
```

Figure 6.9: Some of the strategies for the deforestation algorithm.

rewriting is a dummy and, as will be seen later, the rule itself has no `where` condition at all. This dynamic rule is just used to maintain a global collection of newly generated function definitions. This collection is wrapped into a `Module` again, together with the transformed term definitions, thus producing the final output term.

Strategic Formulation of Deforestation The algorithm rules $(3), (5) - (7)$ specify local transformations. They rewrite their input term (by either function unfolding or `case` branch simplification), and do not traverse into child terms afterward, only start a recursive transformation of the entire term. The other rules $(1), (2)$ and (4) only traverses into child terms. For this, our strategic formulation uses the generic topdown traversal strategy `alltd`, which first tries to apply its parameter strategy `s` and only if that fails, traverses into all child terms.

```
alltd(s) = rec x(s <+ all(x))
```

Figure 6.9 lists the code for the treeless transformation. `transform-to-treeless` parameterizes `alltd` with the appropriate strategy: the local rewrite rules $(3), (5) - (7)$. Note how they are followed by a recursive call '`; transform-to-treeless`' (the function inliner `RecFunApp` takes care of this itself).
`BranchSelect` selects the first `case` branch whose pattern matches the constructor *and* its arity in the selector. It instantiates the branch body with the child terms of the selector

by replacing all pattern variables with them (`tfof-var-subst(|(vi*,t*))`).
`TreelessCaseCase` distributes the outer `case` over the branches of the inner cases, as specified in (7). Variable capture can not occur here, since all bound variables have been renamed already.

Preparing and Using Helper Functions Two rules were still missing in the previous paragraph: the unfolding of function calls, rules (3) and (6). Along with that, infinite recursion should be detected and helper functions should be introduced and called. All of this is coordinated by `RecFunApp`, all code is shown in Figure 6.10.

The main strategy is sketched by the following pseudocode:

if helper function is available for current term
then use it
else remember current input term
inline function body
transform body recursively
if body contained a renaming of original input (hence used the helper function)
then create the actual helper function and call it here as well.

First, it is verified that the input is indeed a function application, possibly as the selector of a `case` term, and it is stored in variable *`funapp`* for later reference. Next, `UseRecursiveHelper` transforms a term t into a call to a helper function, if it exists. The helper function is determined by `HelperFun`, which is a dynamic rule, as will be shown hereafter. The free variables in t are exactly the arguments needed for the function call. The dynamic identity rule `HelperIsUsed` registers that the proposed helper function f has indeed been used.

If no term similar to the current has been encountered before, no helper function exists, resulting in a failing `UseRecursiveHelper`. The called function will now be inlined instead, using the `InlineFun` dynamic rules defined by `prepare-inlining`. If inlining succeeds, which should be the case since all called functions are assumed to be defined, the result can be transformed recursively.

First however, a fresh helper function is prepared for possible future use, by `PrepareRecursiveHelper`. The actual use of the helper function is contained in the dynamic rule `HelperFun`. Since a helper function is suitable for any term that is a *renaming* of the current term, the left hand side of the dynamic rule is largely unclear, most checking will have to be done at the call site by `is-renaming` in the condition of the dynamic rule. The only parts that are known at definition time are the structure of the function call (either standalone, or as a `case`-selector), and the function identifier itself. Note how two quite similar rules exist for `PrepareRecursiveHelper`. Since no higher-order matching [MS01] is possible in the left hand side of a dynamic rule, two different variants are needed to define rules with a left hand side as concise as possible. Higher-order matching and abstraction over object variables is further discussed in Section 10.2.
Finally to prevent argument count mismatches in a call to a helper function, the number of free variables in the term to be replaced is checked as well. For example, when preparing a

```
RecFunApp(trans) =
  (FunApp(id,id) + Case(FunApp(id,id),id)) => funapp
; (UseRecursiveHelper
   <+ where((InlineFun + Case(InlineFun,id)) => inlined)
    ; where(PrepareRecursiveHelper => fhelp)
    ; <trans> inlined
    ; (BuildRecursiveHelper(|fhelp)
       <+ where(<forget-recursive-helper(|fhelp)> funapp))
  )

UseRecursiveHelper :
  t -> |[ f t* ]|
    where <HelperFun> t => f
        ; <tfof-free-vars> t => t*
        ; rules(HelperIsUsed : f)

PrepareRecursiveHelper :
  t@Case(FunApp(f, _), _) -> newfun
    where <newname> "help" => newfun
        ; <tfof-free-vars; length> t => nargs
        ; rules(HelperFun :+ x@Case(FunApp(f, _), _) -> newfun
                  where <is-renaming> (t, x)
                      ; <tfof-free-vars; length> x => nargs)

PrepareRecursiveHelper :
  t@FunApp(f, _) -> newfun
    where <newname> "help" => newfun
        ; <tfof-free-vars; length> t => nargs
        ; rules(HelperFun :+ x@FunApp(f, _) -> newfun
                  where <is-renaming> (t, x)
                      ; <tfof-free-vars; length> x => nargs)

BuildRecursiveHelper(|f) :
  t -> |[ f t* ]|
    where <HelperIsUsed> f
        ; <tfof-free-vars> t => v*
        ; <length> v* => nargs
        ; rules(RecursiveHelper :+ _ -> |[ f v* = t ]| )
        ; !v* => t*

forget-recursive-helper(|fhelp) =
where(
  (Case(FunApp(id, ![]), ![]) + FunApp(id, ![])) => lhs
; dr-lookup-rule-pointer(|"HelperFun", lhs) => (vals, _, tbl)
; <filter(not(?(_,fhelp, _, _)))> vals => vals'
; <hashtable-put(|lhs, vals')> tbl)
```

Figure 6.10: Strategies for recursion detection and helper generation.

helper $h_0\ xs$ for a term $\mathrm{append}\ xs\ xs$, a later encounter of a term $\mathrm{append}\ ys\ zs$ should not lead to a call $h_0\ ys\ zs$, even though the term is a renaming of the original one.

Once the potential helper function has been prepared, the already inlined function body can now be transformed recursively, by applying the parameter strategy `trans`, which is in fact just the original `transform-to-treeless`. Finally, it should be checked whether the helper function that has been prepared, was used during the transformation of the inlined body. Instead of dragging this information around, again dynamic rules are used. Any successful call to `UseRecursiveHelper` has set a dynamic identity rule `HelperIsUsed` for the helper function under consideration. `BuildRecursiveHelper` uses this check and if successful, generates the function definition for the new helper. The inlined and transformed function body now becomes the body of the helper function, and the resulting output term is just another call to that same helper function. These are the two folds performed in the last two steps in Figure 6.7. The generated helper function is not part of the output, since we may still be deep inside transformations of parent terms and properly handling tuples of newly generated function definitions is unnecessarily complex. Instead the new function definition is registered in a dynamic rule `RecursiveHelper` with a wildcard left hand side and no condition. The preceding discussion of the finalization strategies already showed that once the entire term has been transformed, a call to `bagof-RecursiveHelper` is made, which yields the entire collection of newly generated helper functions.

More Refined Rule Undefining If the prepared helper function was not used, the current term can be left untouched, but some additional measures are to be taken to entirely discard the prepared helper function. Since helper functions are memoized between transformations of terms, the only dynamic rule scope for `HelperFun` is at the top-level strategy. Although the proposed helper function was not built for this term, the dynamic rule for it is still present in the rule set and could potentially cause undeserved future applications of `UseRecursiveHelper`, thus producing function calls to non-existing helper functions.

`forget-recursive-helper` removes the appropriate rule instance from the rule set. Since the left hand side may rewrite to multiple rule instances, the normal dynamic rule undefining can not be used. Only the instance with the appropriate *`fhelp`* in its right hand side should be removed. This is only possible by knowing the technical representation of dynamic rules, and using some API calls.

We are aware that this is not a clean solution, but we are working near the limits here. Another solution would be to define the `HelperFun` rules in a local scope, i.e.:

```
RecFunApp(trans) =
  ...
    {| HelperFun :
      where(PrepareRecursiveHelper => fhelp)
    ; <trans> inlined
    |}
    ; try(BuildRecursiveHelper(|fhelp))
  )
```

The proposed helper functions are always removed after the local recursion has completed.

To achieve memoization of the helper functions if they *have* been used and built, the rule `HelperFun` should be defined once again in the top-level scope. This would be possible by labeling the scope for `HelperFun` with a literal `"top"` in strategy `tfof-deforest-strict` (in Figure 6.8). Upon first successful use of `HelperGen` in its local scope – which ensures that it will be built afterwards – it could regenerate itself, but now in the top level scope:

```
PrepareRecursiveHelper :
  t@FunApp(f, _) -> newfun
    where <newname> "help" => newfun
        ; <tfof-free-vars; length> t => nargs
        ; rules(HelperFun :+ x@FunApp(f, _) -> newfun
                  where <is-renaming> (t, x)
                      ; <tfof-free-vars; length> x => nargs
                      ; rules(HelperFun."top" :+ y@FunApp(f, _) -> newfun
                                where <is-renaming> (t, y)
                                    ; <tfof-free-vars; length> y => nargs))
```

So, upon application of `HelperFun` it creates a new instance of itself that will *not* be removed at the end of the current local scope. This is possible, since `HelperFun` knows all the context bound variables that are needed to define a new instance. This regeneration of `HelperFun` could also be handled by `BuildRecursiveHelper`, to prevent generation of duplicates if a helper is used more than once. Although this approach yields a cleaner (more pure) implementation, the memoization results are worse. Deforestation results are still entirely correct, but less helper functions will be shared by terms. This is entirely caused by the introduced scoping. Since there are only two possible left hand sides per function identifier: `Case(FunApp(f, _), _)` and `FunApp(f,_)`, rules from inner scopes will quickly (but unwantedly) shadow rules rom the top level scope. Consider the example for `fappend` in the next section (Figure 6.12). Deforestation of the first term `t1a` has yielded a helper `help_1`, which is equivalent to `fappend`. In the just sketched approach this helper would be stored as a `HelperFun` in the "top" scope. Deforestation of the second term `t1b` will first prepare a helper for `fappend(fappend(.. ..) ..)` in a local scope. When the recursive deforestation of the inlined function reaches `fappend .. zs`, the (unsuitable) inner helper shadows the suitable helper from top level. Even though the helpers are entirely different, their left hand sides are both `FunApp("fappend",_)`.

So once again we encounter the limits that are caused by the lack of abstraction over object values. If the left hand sides could represent the structure of a term in more detail, shadowing would be much less of a problem. And in that case, the just sketched solution is a much cleaner one, than the one that is now shown in Figure 6.10.

6.4.3 Examples and Reflections

Wadler's deforestation paper already showed some nice examples of deforestations of small fragments. Figure 6.11 shows the transformation result for `flip(flip xs)` on binary trees (`flip` is defined in Figure 6.2). Notice how the generated helper `help_0` is in fact the identity function, hence `flip(flip(..))` is idempotent.

```
terms
  t1 := (flip (flip zt))
```

⇓

```
functions
  help_0 zt =
    (case zt of
      Leaf z_6 : (Leaf z_6)
      Branch xt_2 yt_2 : (Branch (help_0 xt_2) (help_0 yt_2))
    )

terms
  t1 := (help_0 zt)
```

Figure 6.11: Deforestation of double flip of a binary tree.

```
terms
  t1a := (fappend xs (fappend ys zs))
  t1b := (fappend (fappend xs ys) zs)
```

⇓

```
functions
  help_2 xs ys zs =
    (case xs of
      Nil : (case ys of
              Nil : zs
              Cons z_8 zs_2 : (Cons z_8 (help_1 zs_2 zs))
            )
      Cons z_9 zs_3 : (Cons z_9 (help_2 zs_3 ys zs))
    )

  help_0 xs ys zs =
    (case xs of
      Nil : (help_1 ys zs)
      Cons z_0 zs_0 : (Cons z_0 (help_0 zs_0 ys zs))
    )

  help_1 ys zs =
    (case ys of
      Nil : zs
      Cons z_2 zs_1 : (Cons z_2 (help_1 zs_1 zs))
    )

terms
  t1a := (help_0 xs ys zs)
  t1b := (help_2 xs ys zs)
```

Figure 6.12: Deforestation of terms concatenating three lists.

```
functions
  fnot b =
    case b of
      True : False
      False : True

  fmapnot xs =
    case xs of
      Nil : Nil
      Cons b bs :
        Cons (fnot b) (fmapnot bs)

terms
t1 := fmapnot (fmapnot bs)
```

⇒

```
functions
  help_0 bs =
    case bs of
      Nil : Nil
      Cons b_0 bs_0 :
        Cons (case bs_0 of
                True : True
                False : False)
             (help_0 bs_0)
    )

terms
  t1 := (help_0 bs)
```

Figure 6.13: Deforestation of double negated list of booleans.

Figure 6.12 shows the transformation of two terms that both append the same three lists: `fappend xs (fappend ys zs)` and `fappend (fappend xs ys) zs`. The resulting terms contain a single call to helpers `help_0` and `help_2`. Both helpers use another `help_1`, which is equivalent to the original `fappend`. Although `help_0` and `help_2` seem different, they are actually equivalent: the function call `help_1 ys zs` in `help_0` has been inlined in the first branch of `help_2`. The semantic equivalence of the two helpers tells us that apparently the two terms are equivalent, hence list concatenation by `fappend` is associative.

Figure 6.13 shows an example that applies the trivial, idempotent, double negation of a list of booleans (`fmapnot (fmapnot bs)`). The input term is transformed into a single treeless call to a new helper function `help_0`, which is basically the identity map over a list of booleans.

Limitations of TFOF and Strict Treelessness Besides the small but illustrative examples in Wadler's paper, it is difficult to think of many new example fragments. A more flexible extension is of course the beforementioned *blazed treeless form*, which allows intermediate use of simple computations on constant values.

The fact that TFOF is first order is not a true limitation here, but having to write specialized functions for normally higher-order functions (e.g. `fmapnot` instead of `map not`) becomes cumbersome in the end.

A final, but often occurring limitation is the unallowed use of non-variable terms (e.g. function calls) as function arguments. Even if one can see that the passed function arguments will appear literally in the result, treeless form forbids it. For example the definition of `fappend` in Figure 6.2 shows that its second argument `ys` will appear literally in the result. In such cases there is no problem in treating the argument just as the arguments of a constructor application, i.e. allowing non-variable but treeless terms. This extension would make functions like `unzip-list`, `flatten-tree` and `transpose-matrix` definable in a treeless form, whereas now we found it impossible. For example for `unzip-list`:

```
funzip xs =
  case xs of
    Nil : Tpl Nil Nil
    Cons x xs : fsplit x (funzip xs)

fsplit x us =
  case x of
    Tpl y z : case us of
                Tpl ys zs : Tpl (fappend y ys) (fappend z zs)
```

This example is even more subtle than e.g. tree-flattening, as it contains a split concatenation of the two already unzipped tails with two new head elements. Still it can be seen that the passed argument `(funzip xs)` to parameter `us` entirely ends up in the result of `fsplit`, hence can safely be allowed as argument in the `fsplit` call. More ideas on this matter are collected in an electronic note by Philip Wadler [Wad89].

6.4.4 Use of Dynamic Rules

The implementation of the deforestation algorithm suffers from the lack of higher order matching in dynamic rules. The defined rules that propose helper functions for a given term have very restricted left hand sides (`FunApp(`f`,_)`, or `Case(FunApp(`f`,_),_)`, where most of the actual matching is done in the condition by `is-renaming`. This means that many different rule instances may exist with the same left hand side. It is not desirable that these instances completely shadow each other, hence the use of *extend rules*. As mentioned in Section 3.3.1, this adds new rule instances to the existing rule set. When a rule is applied, the available instances are tried for rewriting and the first successful rewrite is used as the result.

Besides taking the first successful rewrite, an automatically generated `bagof-`L is also available, which will produce a list of all successful rewritings of the input term. This strategy always succeeds, possibly with an empty list if none of the rewritings were successful. Here, we used this functionality to collect all generated helper functions. Upon each build of a helper function, it is registered with an extend rule:

```
rules(RecursiveHelper :+ _ -> |[ f v* = t ]| )
```

Notice how the left hand side is a mere wildcard and there is no condition. A final application of `bagof-RecursiveHelper` to any term (here: `()` as a dummy) will produce a list of all generated helper functions.

During transformation of a tree, not only dynamic rewriting is important, but sometimes it is also desirable to remember certain properties for terms. In this case, we needed to remember whether a proposed helper function f had been used or not. Previously this was solved by defining rules with dummy right hand sides, like: `rules(HelperIsUsed :` f `-> ())`, and testing for this where needed: `?FunDefR(`g`,_,_,_); where(<HelperIsUsed>` g`)`. The *dynamic identity rule*, or *dynamic match* allows for using only the left hand side in the rule definition and when applied it matches on this term and leaves its input untouched. The rule definition is less verbose, and no `where` is need at the check site.

6.5 Semantics: Dynamic Identity Rule

All basic features of dynamic rules were already treated in Chapter 3, including their operational semantics. The dynamic identity rule is just syntactic sugar for a normal dynamic rule definition:

$$\texttt{rules}(L : p_1) \equiv \texttt{rules}(L : p_1 \rightarrow p_1 \texttt{ where id})$$

The semantics thus equal those of the equivalent rule definition.

6.6 Semantics: Extend Rules

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. Undefinition of a pattern (`:-`) is modeled by a strategy producing the empty list. Finally, extension (`:+`) is defined by a strategy that builds a list with the new right hand side as head element and any other applicable terms from applying the original strategy in the tail.

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

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

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

Thus, by using the empty list `[]` to model undefinedness, there is no more need for the $\bot$ value of Section 3.1.3.

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:

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

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

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow ... \leftarrow s_n \rangle\, t \implies [\,]\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle L \rangle\, t \implies \uparrow (\Gamma', \mathcal{E})} \qquad \frac{\Gamma, \emptyset \vdash \langle s_1 \leftarrow ... \leftarrow s_n \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle L \rangle\, t \implies \uparrow (\Gamma', \mathcal{E})}$$

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.

$$\frac{\Gamma, \emptyset \vdash \langle s_1 \mathbin{\Leftarrow} ... \mathbin{\Leftarrow} s_n \rangle\, t \implies [t_1, \ldots, t_m]\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle \texttt{bagof-}L \rangle\, t \implies [t_1, \ldots, t_m]\ (\Gamma', \mathcal{E})}$$

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

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.

$$\frac{\Gamma, \emptyset \vdash \langle \mathrm{once}_{L_1}(s_1) \mathbin{\Leftarrow} ... \mathbin{\Leftarrow} \mathrm{once}_{L_n}(s_n) \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}')}{\Gamma_{L(s_1|...|s_n)}, \mathcal{E} \vdash \langle \texttt{once-}L \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E})}$$

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

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

Chapter 7

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.

7.1 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.

7.2 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.

7.2.1 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 thesis 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 is more than one `rules(`L `: ...)` for a dynamic rule name L, then there is also more than one static pattern 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.

7.2.2 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 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")`, `"l_0"` is the unique stamp and `Int("3")` is the only needed context information.

```
Var("k") ⇒ Closure("l_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.

7.2.3 Lifting

To illustrate the lifting of dynamic to static rules, consider a fragment from the constant propagation in Figure 5.7:

```
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.

7.2.4 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 6.6. 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).

7.3 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.

7.3.1 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 set 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 rule definitions in outer scopes for application).

7.3.2 Example

An example will illustrate the representation of dynamic rule scopes and the closures stored within. Figure 7.1 shows a small Tiger fragment. It consists of three nested `let` blocks, of which the 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 7.1 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

```
let var x := 3
 in x := x + 1;
    let var y := x + 10
        var w := 9
     in let var x := x + 4
            var a
            var b
         in a := p + q; // HERE
            y := a;
            b := x + y
        end
    end
end
```

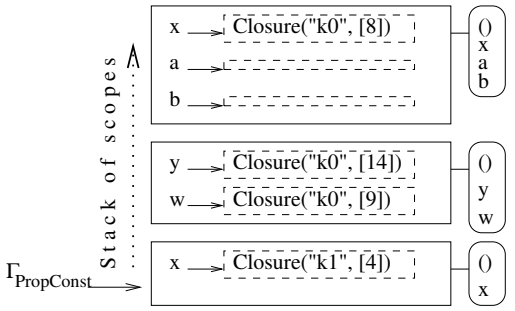


Figure 7.1: Scope representation during constant propagation. Left: input Tiger program. Right: contents of scope stack when constant propagation is at `// HERE`.

a role here are from within `DeclarePropConst` and `AssignPropConst` as can be seen in Figure 5.7. The unique stamps for these rule generators are `"k0"` respectively `"k1"` in this case.

The first, outermost scope contains a rule `⟦ x ⟧ -> ⟦ 4 ⟧` (where `x` is not a meta variable now). 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 `⟦ x ⟧ -> ⟦ 4 ⟧` 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 `⟦ x ⟧ -> ⟦ 8 ⟧` 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 7.1. 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.

7.4 Forking and Change Sets

Section 5.6 introduced the concept of forking a dynamic rule environment and intersecting or joining the two rule sets afterward.

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.

7.5 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. Storing scope labels in sets that have the same constant time operations as hash tables saves time on scope labeling and especially on labeled rule definition. 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, labeling 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.

Chapter 8

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 we use the constant propagation for Tiger (see Figure 5.7). 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. The first four sections each investigate separate dynamic rule constructs. Section 8.5 compares the performance of the old dynamic rules to that of the the new dynamic rules.

8.1 Benchmark: Definition and Application

A first test will show how dynamic rules behave for a growing amount of rule definitions and applications. Figure 8.1 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

```
v_1 := 1;
// ...
v_m := m;
w_1 := v_1;
// ...
w_m := v_m
```

```
v_1 := 1;
// ...
v_m := m
```

```
v_1 := 1;
w_1 := v_1;
// ...
w_m := v_1
```

Figure 8.1: 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.

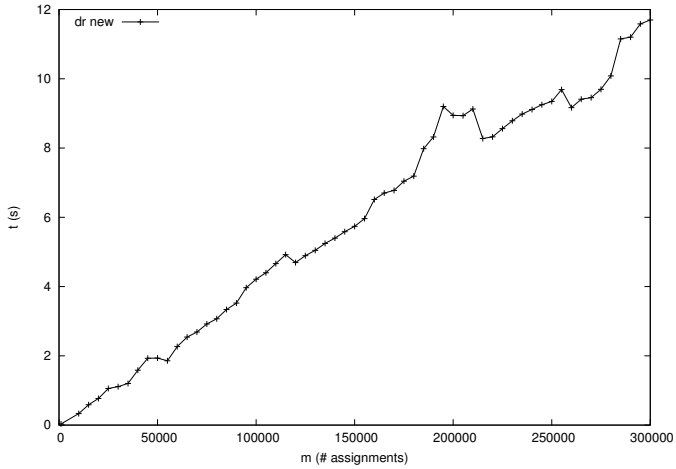


Figure 8.2: Performance of Tiger-PropConst on a sequence of assignments of growing length.

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 8.2 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. Only recently we found out that this is due to a minimal hashtable size which is automatically set by the ATerm library to 127, no matter what smaller table size was requested.

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 8.1. More precisely, for testing application costs only one initial rule is defined, followed by assignments $x_i := x_0 \quad (i = 1 \ldots m)$. Figure 8.3 shows that definition and application of dynamic rules are equally expensive, on the average.

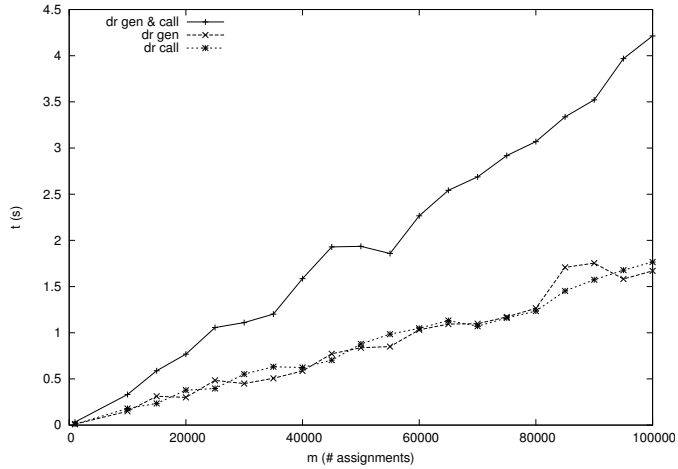


Figure 8.3: Comparison of dynamic rule definition to application costs.

8.2 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 [vdBdJKO00].

Figure 8.4 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 run times, 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 relations to 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 (possibly different, surrounding) 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 8.4. 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 definition and application scope can be measured.

```
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
```

```
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
```

```
let var a
    var c
in (let var f
    in let var e
       in (let var d
           in (a := 1;
               c := 2)
           end;
           (a;
            c))
       end
    end)
end
```

Figure 8.4: Left: Tiger program with $n = 2$ nested `let`-blocks, each with $m = 2$ assignments. Scope labeling, rule generation and rule application are in the same scope ($p = 0$, $q = 0$). Middle: 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 scope labeling and rule application is $q = 3$ (additional scope for `c`). Right: Tiger program with $n = 1$ `let`-block, each with $m = 2$ assignments. Distance between scope labeling and rule generation is $p = 3$ (scopes for `f`, `e` and `d`), and distance between scope labeling and rule application is $q = 2$ (one scope higher).

In the same 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 definition and application scope is $q = 3$. When $q \geq p$, the additional $q - p$ `lets` can be placed at the innermost (p^{th}) `let`. If $q < p$, the inner $p - q$ `lets` are closed, followed by the variable uses, and finally the remaining q `lets` are closed. The third fragment in Figure 8.4 shows this situation for $p = 3$ and $q = 2$.

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 8.5 shows the results of these tests. The first test varies p from 0 to 30 and keeps $q = p$. ('distant generations, distant calls'). In this test, rule application occurs in the same as generation scope, which is p levels nested inside the outermost definition scope. The second test varies q from 0 to 30 and keeps generation of rules directly in the definition scope, i.e. $p = 0$ ('direct generations, distant calls'). The third test does the opposite of the second: rule generations are deeply nested, whereas rule calls are in the top-level definition scope, i.e. $p = 0 \ldots 30$, $q = 0$ ('distant generations, direct calls'). First and foremost, the results of

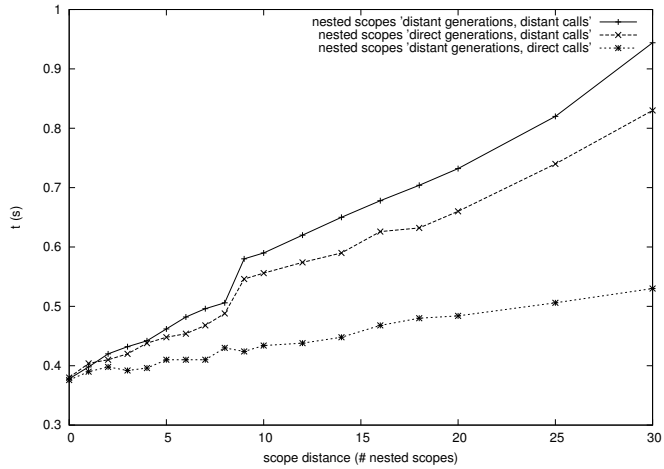


Figure 8.5: Performance of Tiger-PropConst on nested `lets` with increasing distance between definition and generation scope, and between definition and application scope.

these three tests shows linear behavior for the scope traversing. However, 'distant calls' are far more costly than 'distant generations'. Previous experiments with StrategoXT 0.10 showed the opposite and we attributed that to the inefficient scope labeling[1]. Now, rule definitions, which have to look up labels in each scope, are more efficient. This is probably because a dynamic rule call not only has to look up any closures, but also apply the actual rewrite rule to its input, whereas rule definitions only have to store the closure. The costs of scope labeling and the inspection of the labels list is considered in more detail in the next section.

8.3 Benchmark: Scope Labeling

The previous section already suggested that the usage of sets instead of lists for storage of scope labels seems to have improved the performance, as was the original goal. Now we will perform a final test that compares StrategoXT 0.10 labeling to that of StrategoXT 0.11 labeling.

The tests performed for this benchmark keep all definitions and applications in the same scope (i.e. $p = 0$ and $q = 0$). The number of scope labels m is varied. In the old, naive approach, for each scope label, a new label value was pushed onto the head of the label list. For rule definition within some labeled scope a linear search was performed on the label list, hence an average lookup of any label will cost $O(m)$. If within that scope m dynamic rules

[1] StrategoXT 0.10 offered full functionality in dynamic rules, but still stored scope labels in lists. StrategoXT 0.11 uses hash-indexed sets instead, which should change linear costs behavior to constant costs behavior.

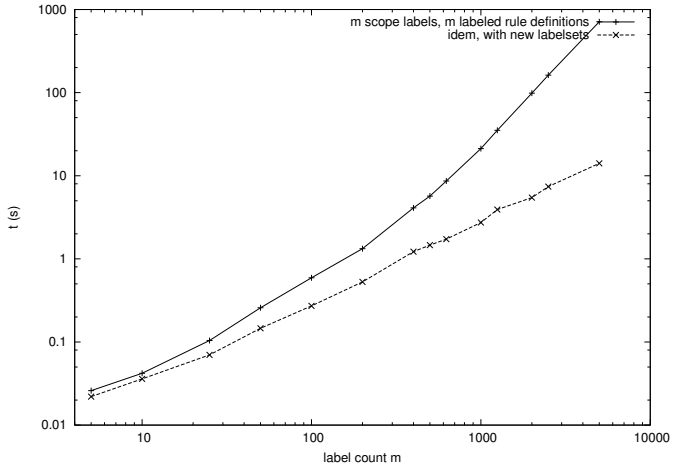


Figure 8.6: Performance of Tiger-PropConst on nested `lets` of with increasing amount of labels per scope.

are defined with a label, total definition costs will be of $O(m^2)$. The new approach should reduce the lookup costs to $O(1)$, yielding a total of $O(m)$.

Figure 8.6 indeed shows the quadratic behavior of the running time for the old approach and a linear behavior for the new approach. Notice the use of a double log scale, the bottom straight line has logarithmic slope of 1, hence is linear. The top line has an approximate slope of 2 (the second half of it), hence is more or less quadratic. The change in label storage definitely has its effect: for $m = 5000$, the average running time decreases from 713 seconds to a mere 14 seconds. Practical situations will never need this number of scope labels, but even for just a few labels, the new storage is faster: there is never an increase in costs.

8.4 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 5.5 already introduced the concept of computing with rule sets, which usually comes 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 7.4 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 8.1. Thus, the same amount of rule definitions and rule calls is involved, but additionally there are

```
if cond
then (f := 1;
      g := f;
      if cond
      then (a := 1;
            b := a)
      else (c := 1;
            e := c))
else (h := 1;
      i := h)
```

```
if cond
then (c := 1;
      e := c;
      if cond
      then (a := 1;
            b := a)
      else (a := 1;
            b := a))
else (c := 1;
      e := c)
```

```
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
```

Figure 8.7: 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.

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 afterward 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 8.7 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 8.1. Figure 8.8 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 an other, 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.

This benchmark shows that although the costs increase by a serious constant factor, behavior 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.

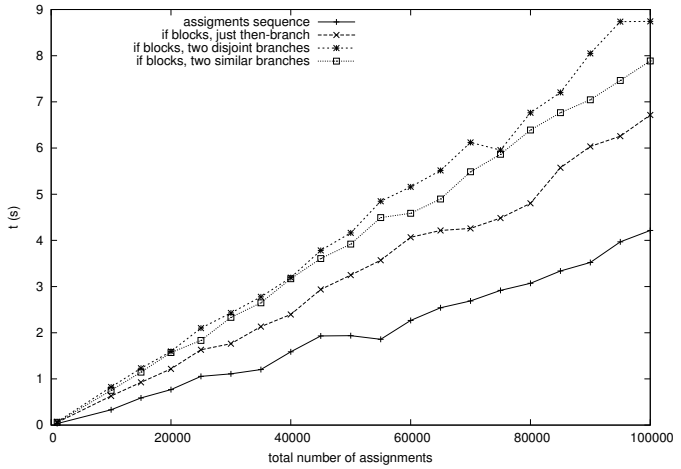


Figure 8.8: Performance of Tiger-PropConst on nested `if` blocks. Increasing length of assignment sequence tests the intersection costs.

Finally, the fix-point manipulation of rule sets, as discussed in section 5.6 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 Tiger program would!

8.5 Benchmark: Old versus New

The first and main reason for the redesign of the dynamic rule system was to obtain a more clean and more extensive setup of the dynamic rule system. Several new constructs were added and existing functionality was revised and its operational semantics was formally described. As part of the cleanup, the representation of dynamic rules was improved. The old system already used hash tables for storing the rule closures in, but the new system has split this in one hash table per scope, thus saving the costs of long list traversals over closures in inner scopes when a higher scope needs to be addressed. Another improvement is the forking and meeting of rule sets using change sets on top of the stack. In the old system, both entire rule sets had to be traversed when computing an intersection.

Using an old version of `Tiger-PropConst` that does not use any of the new dynamic rule constructs, and compiling it within the old system[2]yields an executable that allows for an honest comparison with the same file compiled in the new system. On a sequence of assignments, as used in Section 8.1, the new variant show a decrease in runtime of 20-30%.

[2]The old dynamic rule constructs can still be compiled in the 'old way' with a new Stratego compiler by passing the `--dr old` flag to it.

Comparing the same two variants, run on the nested conditionals, as used in Section 8.4 shows hardly any difference. The new variant uses the new intersection operation, but is identical to the old variant on all other parts. Since the conditionals do not introduce any scopes, the efficient new intersection using change sets has no advantage here: it has to intersect the one and entire big scope for the two branches just as well as the old variant has to. When the nested conditionals are given one scope per conditional, we get $n = 10$ nested scopes here, each with $m = 1000$ rule definitions. The new variant is now almost twice as fast as the old one. This is not as much as expected, but the intersection costs are of course not the only costs.

The above two variants used old dynamic rule constructs where possible. Comparing them with the new implementation as shown in Figure 5.7, shows the additional gain that can be obtained. The new `Tiger-PropConst` is approximately twice as efficient as the old one for a sequence of assignments. Exploiting the power of the new implementation, by applying it to the scoped nested conditionals as discussed before, shows that the new implementation is almost 8 times as fast as the old one.

Finally, the efficient storage of scope labels can not be compared with the old system, since it is a new feature. The only similar functionality in the old system was that of `override rules`. The costs for emulation of this construct probably have increased. In the old system, the new value could be put directly as a replacement at the head element of the list of closures, no matter how many scopes were in between. The emulated version however, needs to traverse some inner scopes until the first scope with the appropriate label. This is a price worth paying for all the extra functionality the generalized scope labeling mechanism delivers.

8.6 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 rule definitions for one rule and left hand side in one scope. This number is generally small though, so we do not consider this a problem. The use of change sets when forking and meeting afterward proves to be of great value when many surrounding scopes do not have to be intersected anymore.

The new, efficient scope label storage has delivered the $O(1)$ rule definition costs. Although using labeling for emulation of `override rules` is no doubt more expensive, the number of scopes between definition and generation scope will not be that big (< 10 probably). Besides, scope labeling is a powerful extension of the dynamic rule system.

A final, but important 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. This number will generally be much smaller. A program with this number of assignments would consist of many millions of lines of code. The measured running times that amount several seconds for the biggest tests (up to 12 seconds in Section 8.1)

hence should be considered with this realization in mind. More normally sized programs are transformed within fractions of seconds. These large test sizes were only necessary, to reveal certain qualitative behavior of the performance, which is of fundamental importance when reasoning about the efficiency of a concept like dynamic rules. Besides, quantitative results depend on so much more factors of a transformation, and input programs are hardly ever of such regularity as the tests used here.

Chapter 9

Related Work

Dynamic rules are a relatively new concept in strategic rewriting and have several uses. Firstly, they can be used as mere information carriers, propagating analysis information to other parts of the traversal that base the rewriting on this information. Rules such as `IsDead` are typical examples of this. Secondly, dynamic rules can be self-contained rewrite rules that enrich the rewriting system at runtime. Typical examples of this are the various function inlining rules discussed here.

The first use has been inspired by the various data structures, such as symbol tables, that other languages use to store program information, mainly during program compilation and optimization. Some languages also support dynamic specification of parameters or variables and their scope ('dynamic binding'). The dynamic rule constructs provide control over both definition time bindings and application time bindings, limiting the scope of generated rules, and the way available rules are applied. The second use, extending the runtime system, can also be seen in various other systems. These include some logical languages, such as Prolog, reflective rewriting systems, and some optimizing compilers.

This chapter elaborates on the above considerations, and just those. A good overview of program transformation in general is provided in a survey by Visser [Vis04b]. Dynamic rules as discussed in this thesis are also the subject of a recent publication on the new dynamic rules [BvDOV04]. Section 9.1 considers some approaches of combining program analysis and program transformation. Some uses of dynamic binding are considered in Section 9.2. Run time extension of rewriting systems is considered in Section 9.3 on reflective rewriting and Section 9.4 on user defined compiler optimizations.

9.1 Combining Analysis and Transformation

Program transformations, especially the optimizing source-to-source transformations, are often steered by information gained during a program analysis phase. Analysis of variable-liveness is used in dead variable eliminations, constant propagation analysis is used to support constant folding and unreachable code elimination. Instead of performing the analysis and transformation in two separate passes, it is often possible and beneficial to combine both traversals of the tree.

The problem that turns up then is how to pass the acquired analysis information to the point where the actual transformation needs it? Several systems have their own facilities for this.

Traversal Functions in ASF+SDF The ASF+SDF system [dBKV03] allows traversal strategies to be assigned accumulating parameters that are automatically carried along during the traversal. On each application of the strategy, the accumulated parameter is updated and the ongoing traversal will use this new value.

Higher Order Attribute Grammars Attribute grammars are a specification of attributes at the nodes of a tree structured input. Evaluation of such a grammar comes down to determining the order in which the various attributes need to be evaluated at the various nodes. The most common types are inherited (topdown propagated) and synthesized attributes (composed of child attributes). AG systems can either infer this order from a complete dependency analysis, or they can rely on the lazy evaluation mechanism of the language they are implemented in.

Mostly, the data contained in attributes concerns analysis information, such as usage counts or inferred types. The input tree itself remains unchanged: only at each node the attributes are evaluated. To transform an input tree means to modify the shape of it. In plain AG systems, this is only possibly by maintaining a synthesized tree attribute representing the transformed tree. Still, results of the one transformation computed in an attribute may still need to be reexamined in the other transformation, again supported by attributes. Higher order AGs (HAG) support the re-attribution of attributes, thus enabling combined transformations. The results of such an HAG evaluation become available at the original tree's root, and may be passed back to the child nodes as an inherited attribute. These nodes however receive the entire synthesized tree and can not identify which node in it originally corresponded to themselves. AG Views solve this by making the intermediate results at each node eventually available to that same node again.

There are still some hairy issues involved in these enhanced AG systems. Although originally the programmer didn't have to be concerned about proper propagation of attributes, now a clever specification of the (higher-order) attributes is needed to obtain a properly combined transformation. It is not yet clear either whether repeated transformations (e.g. to reach a normal form) are always easy to specify.

A general note for AG systems is: it saves one the effort of carefully specifying the traversal order, by automatic inferencing of attribute dependencies. On the other hand: no truly generic traversals can be specified and re-used, and in cases where one *does* want flexible control over the traversal, this has to be enforced by making delicate use of higher-order traversal-steering attributes. For a complete overview of these AG systems and a comparison with Stratego, consult the thesis by Alexey Rodriquez[Ro04].

Stratego Stratego is more or less the opposite of attribute grammar systems: it does not maintain inferred or synthesized attributes, let alone using an auto-inferred traversal. Instead it rewrites an input tree, and provides flexible control over both rewriting, and user-defined traversal of the tree. So now the analysis part is where the problem is: how

to propagate the analysis information to where it is needed? In the past, some generic traversal were defined (e.g. for bound variable renaming) that automatically carried along the renaming environment during the traversal. Although the use of such generic renaming strategies was intuitive, the original design of it was subtle enough, always handling the growing environment properly.

Dynamic rules are much more suitable for this. They don't need to be passed along anywhere, yet their contained information can be accessed anywhere (from within their dynamic rule scope, that is). Dynamic rules scopes restrict the availability of the information to the relevant parts of the traversal, and new information can be defined or added to existing information at any point.

9.2 Dynamic Binding

Dynamic binding is the binding of variables, parameters, function definitions and such in a *dynamic scope*. A dynamic scope for a variable can not be distinguished lexically, hence is not known at compile time. Static or lexical scopes and variables are used in most programming languages, using global scopes, local function scopes, and more variants. Variables are bound in their definition context. These scopes are known at compile time, since they can be lexically distinguished. Dynamically scoped variables are bound in their application context: any definition of a variable that is in the same execution path of that variable is visible. The advantage of dynamically scoped variables is that values can be assigned in some context that a program is currently in and any other fragments in the ongoing execution path can make use of them, no matter what the original program looked like.

Dynamic binding is not new. In Lisp 1.0 the dynamic scope of variables was considered an unintended feature, but gave rise to reasoning on the concept of closures. TEX allows local redefining of macros and variables, and shell scripting languages and XSLT 2.0 implement dynamic binding as well. Although most current languages rely on static variable binding, some recent papers have re-introduced the concept of dynamic binding in both functional [LLMS00] and imperative languages [HP01].

Implicit Parameters in Functional Languages Lewis et al. [LLMS00] describe an extension of functional languages with implicit parameters. It applies to languages that employ a static Hindley-Milner type system, such as Haskell. Implicit parameters in an expression e are defined using 'e `with` p=e_2' and are automatically passed on to their usage sites in e, through subexpressions and function calls. Implicit parameters are distinguished *lexically* from normal parameters (here they are preceded by a question mark, e.g. `?x`). The extended system infers at compile time where implicit parameters are used and which explicit parameter passing should be added. The inferencing is based on an extended type inferencing that now includes an additional environment for maintaining types of any implicit parameters.

The system is sophisticated enough to infer complex relations between uses and definitions of implicit variables, including proper handling of non-unique, local identifiers. A limiting requirement though, is that implicit parameters should be monomorphic, hence

can not be used to implement overloaded functions or operators. Besides, the definition of an implicit parameter is directly combined with its scope specification, which can be quite restrictive. Implicit parameters can not be redefined either: they are passed by value, hence are nothing more than automatically inserted parameters.

Dynamic Variables in Imperative Languages Hanson and Proebsting [HP01] describe a C++ implementation of *dynamic variables* used in the context of their imperative, object-oriented implementation of the Icon language. Dynamic variables are defined with an explicit '`set` id_1`:`tp_1`=`e_1 `in` S'. Hence, the definition immediately fixes the scope of the variable. Uses of dynamic variables are done with a similar construct '`use` id_1`:`tp_1 `in` S'. The definitions of dynamic variables are translated into a normal variable assignment 'id_1`:`tp_1`=`e_1' and a record that maintains scope and address information. The latter is maintained in a global table that stores for each definition its scope as two pointers to program counters, its type and name, and a local offset which is used to determine to absolute, runtime address of the variable id_1. Uses of dynamic variables just amount to a lookup in the global definition table, starting at the inner scope, following pointers to callers of the current function, until the first definition is found that matches name and type of the requested variable.

Compared to dynamic rules, the mandatory combination of variable definition and scope specification is very restrictive, at least for more fragmented programs. Compare this for example to the separation of scope specification for `RenameVar` in Figure 3.2.

Besides that, dynamic variables are mere variables, whereas dynamic rules are some sort of functions, of which the lexically scoped variables are stored in a closure at definition time, and the rest remains untouched and left for evaluation at call time.

Still, dynamic variables were designed for their own specific purpose. Icon is a very high-level language with extensive support for string and structure processing. The current object (string, structure, etc.) being examined and possibly a local position inside it is always available through keywords such as `&subject` and `&pos`. Any called functions can access these variables without them having been passed as arguments. The traversal of the caller-path using program counters and local offsets is an efficient approach to look up the most recent definition of a variable.

The minimalistic setup gives rise to further improvements though, as the authors mention themselves as well. Aliasing of dynamic variables to avoid name conflicts with local variables is one of them. And proper handling of exceptions when no matching dynamic variable definition is found, possibly by specifying a default value in a `use` statement. Stratego's dynamic rules could conflict with local strategy names, just as any normal rule or strategy could. Aliasing using a surrounding `let alias=rulename in s` easily solves this. When no rule definition is available at call time, application simply fails. One should be aware of this and backtrack to other alternatives, or revert to the original term. This is a basic property of program transformation anyway: try to rewrite an input term using some rewrite rule, or try another one instead, or backtrack to the calling strategy.

Of course the realization of dynamic variables is different from that of implicit parameters, but a more important difference between the two is that dynamic variables can be modified after their definition, whereas implicit parameters can not. Dynamic variables are

looked up and passed back by reference, whereas implicit parameters are passed by value. A dynamic rule set can be changed at run time by *defining* new rules or extended by *adding* new rules.

Dynamic Rules The two described variants of dynamic binding both lack separation between scope specification and variable definition. For program transformation, this is almost unbearable, since transformations are often specified by many separate pieces of code (in this case rules) that together form the entire transformation. For example in variable renaming, the strategies that define the various scoping rules of the object language can re-use the same renaming rules. If scope specification and variable definition need to fall together, this re-use becomes impossible.

Another important feature is the possibility to change dynamically bound variables. Especially during program transformation, new analysis information might be acquired during the transformation which should be reflected in the rewriting system as well. Dynamic rules can be redefined, or their definition can be added to any existing definitions.

When comparing dynamic rules to the concept of dynamic bindings, it should be mentioned that the names of dynamic rules have global scope. Any dynamic rule defined in a program can be applied anywhere, but application fails if no 'live' definition is available at that time. Dynamic rule scopes are actually the dynamically bound parts here: based on the execution path they are entered and exited. The scope to which a dynamic rule definition is associated depends on how the execution paths has entered scopes until the definition point. The labeling of scopes allows for context-sensitive scoping, besides the normal application-based scoping.

9.3 Reflective Rewriting

Reflective techniques originate from logic, where the logical rules and equations not only describe and operate on the logical domain, but also on the level of the logical axioms themselves, being able to modify the current logical system. In rewriting systems, reflection is the run time modification of the rewriting system by itself: it is able to perform rewritings at the meta level. Hence, using gathered information the rewriting system can be adapted to make it more suitable for the current rewriting being performed.

Maude Maude [MCM00, CM00] is a true reflective system, which supports 'normal' rewriting by rules and equations, but also provides a meta level. This meta level can be accessed using normal rewrite rules and equations to change the specification of the rewriting system itself based on run time information. In contrast to Stratego, in which rules are in fact composed of first class strategies, Maude uses the concept of 'inside strategies', which are created by applying rewrite rules to the rewriting system itself. As the designers summarize themselves: the system is not based on "Logic + Control", but on "Control $\subseteq$ Logic".

ELAN ELAN [BKK+96b] is a strategy language whose strategies and strategy combinators were an inspiration to the creators of the Stratego language. It has no built-in support

for reflection of its own, but a reflective extension of ELAN was made [KM96, BKK96a]. Although the extension needs some additional implementation in the C++ back-end (hence is not purely reflective), it provides flexible control over modifying the rewriting system.

Dynamic Rules in Stratego The rewriting engine, or Stratego Run Time System is not accessible to strategies, the language is not reflective. Dynamic rules provide a way of run time modification and extension of the available rewrite rules. Static rules and strategies can not be modified, so this is by far not as reflective as Maude is, but dynamic rules are a powerful mechanism still. Compared to purely reflective systems, dynamic rules are more pragmatic and can be used without too much formal knowledge of the higher level rewriting system.

9.4 User Defined Compiler Optimizations

Most compilers do not just translate their input source file into an executable program, they have intermediate optimizing phases. Although current compilers are more and more sophisticated in analyzing input files and optimizing their output, they usually have to rely on basic algebraic properties for their optimizations. The programmer has much better knowledge on the program, and especially on the user-defined data types and their properties. Several compilers allow the specification of such domain-specific information and use that to perform more advanced optimizations during compilation. Two such systems are discussed here.

These two application are only mentioned here to show the opposite of reflective systems, discussed in the previous section. Here, the meta-system (the compiler) is instrumented using object-level syntax instead of its own meta-level syntax. Moreover, the extension of the meta system is static, without use of context information.

Optimizing Rewriting in the Glasgow Haskell Compiler The Glasgow Haskell Compiler (GHC) is an optimizing compiler for Haskell. Peyton Jones et al. [PTH01] describe how GHC allows the developer to specify rewrite rules as pragmas in source files. These rules are in normal Haskell syntax, have no side condition and should have a function application as left hand side pattern. The compiler uses a simple repeated topdown traversal, applying the rules or eventually inlining function calls where possible. It also generates rules at compile time, rewriting more complex expressions to specialized functions that have just been inferred and generated.

The authors themselves stress that their principal point is simplicity: the rules are simple, and only very primitive control over the order of rule application is available to the user, rules are unscoped as well. Benchmark result show an average gain of a mere 5% in running times.

Domain Specific Optimizations of C++ programs CodeBoost is a transformation framework for C++ programs, currently aimed at large mathematical programs from the Sophus project. Besides some built-in domain specific optimizations, the user can specify additional rewriting relations in normal C++ syntax [BH03], all contained within the

reserved function `void rules(){..}`. The user can influence the rewriting strategy, by naming the rules with one of four predefined names, such as `topdown` and `bottomup_r`.

Benchmarks show a serious gain of several factors, but for this application domain, with its huge sets of multidimensional matrix data, seemingly small optimizations, may yield big results. The designers of CodeBoost identify the proper application order and strategy as the main points for further research.

Chapter 10

Future Work

Dynamic rules have evolved into a mature part of the Stratego language. Inspired by the current use of them, various improvements and new functionality can be thought of, still. The sections below highlight some of these ideas, sketching possible directions for further research in the future.

10.1 Generic Use of Dynamic Rules

In program transformation, there are many concepts that require a generic approach when traversing or performing a rewriting. Examples are bound variable renaming, splitting control flow and intersecting result sets afterward. Generic traversal strategies are sufficiently available in Stratego, but generic use of dynamic rules has not fully developed yet.

Abstracting over Rule Names When applying rules, or specifying rule scopes, the syntax prescribes the specification of the rule identifier L. For example, one can write {| `PropConst, IsDead :` s |}, but not {| `L* :` s |} where `L*` would then contain a list of rule identifiers. Only when making use of low level API strategies, this form of genericity can more or less be reached. And even then: application of strategies can not be achieved only by the textual strategy identifier. Strategies that generalize over dynamic rules, hence have to be parameterized with both the rule identifier and the rule itself. Dynamic rule definition is not trivial to generalize, since currently rule definitions themselves are lifted at compile time. Rule scoping can easily generalize over rule names, currently this is already possible for scope labels. Further generalization would allow specification of better reusable strategies involving dynamic rules.

Generic Selection of Applicable Rules Application of a dynamic rule L selects the most recently defined instance by default, or tries older ones if it fails. Now that rule definitions are available as rule sets, various other selection methods may be desirable. The `bagof-`L operator yields all successful rewrite results in a list, the `once-`L operator acts as L, but removes a rule definition directly after it was applied successfully. Users of dynamic rules may have their own wishes for the selection of applicable rules. This selection could

be made more accessible, without revealing technical details on the implementation and representation of dynamic rules.

10.2 Higher Order Matching in Dynamic Rules

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 [Wad90], as treated in Chapter 6. Dynamic rules are 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 an (at compile time) unknown number of arguments `FunApp(f,a*)` is encountered, the dynamic rule to be defined 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(HelperFun: 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(HelperFun: 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. Our implementation still puts the maximum possible amount of structure in the left hand side, as the two separate definitions for `PrepareRecursiveHelper` in Figure 6.10 show. The advantage of this approach 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.

Although the above approach is a pragmatic workaround, it requires more effort to implement and it is still lacking full abstraction over object values. In static rewriting,

higher order pattern matching has been investigated already. For example, De Moor and Sittampalam [MS01] have implemented higher order matching by representing the pattern variables as lambda abstractions. Proper substitutions for them are found by an algorithm that incrementally refines rules mapping patterns to (not necessarily closed) terms. Potentially, multiple substitutions exist, hence a choice should be made as to which one will eventually be used.

In dynamic rules, the problem is even more subtle, since at compile time, the left hand sides are already partially used (the dynamic part of it) to serve as the key when storing the context bindings in the rule state. It would be a very powerful extension, if dynamic rules *could* make use of abstraction over object values in their left hand side.

10.3 Dynamic Scoping

The dynamic scoping is not only an important feature of current dynamic rules, it can also give rise to unexpected behavior. When dynamic rules are defined, all variables that are bound at generation time are stored, representing some sort of closure, together with the lifted rule definition. The state of the system (i.e. the available dynamic rules) are not stored at generation time. So upon application of a dynamic rule, it is not executed in the *generation time variable context*, but in the *call time system state*. A dynamic rule `A` that uses `B` in its body might be intended to use the generation time state of `B`, but upon application of `A`, the definition(s) for `B` might have changed in the meantime. Sometimes, the application of `B` can be lifted to the generation site of `A`, but in general it requires storing true closures of both environment and state. With sufficient knowledge on the back-end of our system, this is already possible. Recent developments have shown this in the context of Stratego Interpreter [STR].

10.4 Dynamic Rules and Backtracking

The Stratego Runtime System performs automatic backtracking to alternative strategies, when strategies fail. Currently, the system state containing the available dynamic rules is not restored to its state at the original choice point. When aware of this fact, this does not form a limitation in most situations. In others it does however, and it would be desirable to offer the possibility of reverting any changes to rule sets upon backtracking.

In the type inference implementation in Appendix B, this facility is actually needed. When inferring a constructor application, multiple data types might be a candidate for the eventual type. When one fails, the other one is tried for unification. The mappings for type variables that have been generated as dynamic rules in any previous unification attempts should then be made undone.

Chapter 11

Conclusion

This thesis has fully described the work and research that formed the masters' project of the author. It provides both a broad and in-depth description of the new dynamic rule system in the Stratego language. The use of the various language constructs was illustrated by non-trivial case studies that were thoroughly worked out.

The Basics The context of the performed research was sketched by a description of strategies and combinators in Stratego, along with their formal operational semantics. The 'old-style' dynamic rules were described, as to see how the new dynamic rule system has improved on several points.

Case Studies Three case studies were used to show the use of dynamic rules in quite different, non-trivial contexts and with varying degrees of complexity. The shrinking inlining on a lambda calculus (Chapter 4) showed the basic use of dynamic rules, thereby forming a practical extension of the short overview in Chapter 3. Constant propagation on the imperative Tiger language (Chapter 5), for increasingly complex language elements, provided the motivation for scoping of rule sets and forking and intersecting of rule sets. Deforestation of functional programs (Chapter 6) showed the use of extending rule sets, instead of allowing just one rule instance per rule name and left hand side. The lack of abstraction over object values in a dynamic rule's left hand side was shown to be a limiting factor, which can be worked around, but would make a nice future improvement. The type inferencer for the TFOF language in this chapter formed a small sidetrack of the actual project, but produced some nice results of its own (Appendix B).

Redesign of Dynamic Rules The author has co-operated as a developer in the redesign and implementation of the new dynamic rule system, the redesign of dynamic rules itself has not been described in this thesis however. Instead, the old situation was shortly sketched, and the motivation for the redesign provided throughout the case studies. The implementation details of the new situation were described, thus revealing how efficiency is ensured. The efficiency requirements were successfully validated by a range of benchmarks that have been performed and analyzed. These also provide some better understanding of which parts lead to quantitative increase in costs.

Context and a Look Ahead The need for context information in rewriting system is not new. Other systems feature their own solutions for it, as discussed in 'Related Work' (Chapter 9). Purely reflective systems (e.g. Maude) allow true extension of the rewrite system at runtime. Dynamic rules could be seen as a light form of 'computational reflection'. Other systems purely focus on runtime propagation of contextual values to their places of need (e.g. implicit parameters and dynamic variables). Besides context-awareness, combining program analysis and traversal was also given some attention.

Future directions for research were sketched in Chapter 10. Abstraction over object values and the storage of true closures for later use would add serious new functionality and semantics to dynamic rules. Other pragmatic improvements include generalization of dynamic rule use, and reverting of rule operations upon backtracking.

Appendix A

Semantics of Stratego Constructs

This is an appendix to Chapter 2, which discussed rewriting in general and introduced the core Stratego syntax and several syntactic extensions. The extensions were discussed using equivalence relations with their core building blocks. This chapter lists the operational semantics of most of Stratego's language constructs. The operational semantics of all dynamic rule constructs have already been included in the preceding chapters (Sections 3.1.2, 3.1.3, 5.4, 5.6, 6.5 and 6.6).

The operational semantics are shown 'as is', in the form of assertions. For a detailed discussion, refer to the new dynamic rules paper [BvDOV04]. The semantics are expressed by their effect on variable environment $\mathcal{E}$ and rewrite system state Γ, as introduced in Section 2.2.1. Figures 2.2 and 2.3 can be used as a quick reference for all syntax.

A.1 Matching and Building

Match Match a term t against a pattern p by finding appropriate variable substitutions:

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

Build Build a closed term from a pattern p, by instantiating all pattern variables:

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

Scope Declare variables $x_1, \ldots, x_n$ local and initially unbound for a strategy s:

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

Convenience construct $\{s\}$ makes all free variables in s local:

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

A.2 Strategy Combinators

Unit and Zero Unit strategy `id` and zero strategy `fail`:

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

Sequential Composition Apply s_2 to result of s_1 or fail:

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

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

Guarded Choice Apply s_1, followed by s_2 if it succeeds, or s_3 if it fails:

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

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

Testing Test for success (or failure) of s but leave current term untouched:

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies t'\ (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \texttt{where}(s) \rangle\, t \implies t\ (\Gamma', \mathcal{E}')} \qquad \frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \texttt{where}(s) \rangle\, t \implies \uparrow (\Gamma', \mathcal{E})}$$

$$\frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies t'\ (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \texttt{not}(s) \rangle\, t \implies \uparrow (\Gamma', \mathcal{E})} \qquad \frac{\Gamma, \mathcal{E} \vdash \langle s \rangle\, t \implies \uparrow (\Gamma', \mathcal{E}')}{\Gamma, \mathcal{E} \vdash \langle \texttt{not}(s) \rangle\, t \implies t\ (\Gamma', \mathcal{E})}$$

A.3 Strategy Definitions

Global Strategy Definitions Add strategy definitions to the definition environment D:

$$\frac{\begin{array}{c} D_{\mathsf{fresh}}(f) = (f\texttt{(}f_1\texttt{,}\ldots\texttt{,}f_n\texttt{\,\vert\,}x_1\texttt{,}\ldots\texttt{,}x_m\texttt{)}\ \texttt{=}\ \texttt{s}) \\ \mathcal{E}' \equiv [x_1 \mapsto \mathcal{E}(p_1) \ldots x_m \mapsto \mathcal{E}(p_m)] \\ [f_1\texttt{=}s_1 \ldots f_n\texttt{=}s_n] D, \Gamma, \mathcal{E}'\mathcal{E} \vdash \langle s \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}'\mathcal{E}'') \end{array}}{D, \Gamma, \mathcal{E} \vdash \langle f\texttt{(}s_1\texttt{,}\ldots\texttt{,}s_n\texttt{\,\vert\,}p_1\texttt{,}\ldots\texttt{,}p_m\texttt{)} \rangle\, t \implies \overline{t'}\ (\Gamma', \mathcal{E}'')}$$

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

Local Strategy Definitions Add strategy definitions $d_1 \dots d_n$ to a local environment for execution of s, with surrounding global environment D, using a `let` construct:

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

A.4 Generic Term Traversal

Congruence Match a term on its constructor name and arity, and apply s_i to all its argument terms:

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

$$\frac{c \not\equiv c'}{\Gamma, \mathcal{E} \vdash \langle c(s_1, \dots, s_n) \rangle\, c'(t_1, \dots, t_n) \implies \uparrow (\Gamma_n, \mathcal{E}_n)}$$

$$\frac{\begin{array}{c}\Gamma, \mathcal{E} \vdash \langle s_1 \rangle\, t_1 \implies t'_1\ (\Gamma_1, \mathcal{E}_1) \\ \dots \\ \Gamma_{i-2}, \mathcal{E}_{i-2} \vdash \langle s_{i-1} \rangle\, t_{i-1} \implies t'_{i-1}\ (\Gamma_{i-1}, \mathcal{E}_{i-1}) \\ \Gamma_{i-1}, \mathcal{E}_{i-1} \vdash \langle s_i \rangle\, t_i \implies \uparrow (\Gamma_i, \mathcal{E}_i)\end{array}}{\Gamma, \mathcal{E} \vdash \langle c(s_1, \dots, s_n) \rangle\, c(t_1, \dots, t_n) \implies \uparrow (\Gamma_i, \mathcal{E})}$$

All Subterms Apply a single strategy s to all available child terms of the current term and require them all to succeed:

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

$$\frac{\begin{array}{c}\Gamma, \mathcal{E} \vdash \langle s \rangle\, t_1 \implies t'_1\ (\Gamma_1, \mathcal{E}_1) \\ \dots \\ \Gamma_{i-2}, \mathcal{E}_{i-2} \vdash \langle s \rangle\, t_{i-1} \implies t'_{i-1}\ (\Gamma_{i-1}, \mathcal{E}_{i-1}) \\ \Gamma_{i-1}, \mathcal{E}_{i-1} \vdash \langle s \rangle\, t_i \implies \uparrow (\Gamma_i, \mathcal{E}_i)\end{array}}{\Gamma, \mathcal{E} \vdash \langle \texttt{all}(s) \rangle\, c(t_1, \dots, t_n) \implies \uparrow (\Gamma_i, \mathcal{E})}$$

One Subterm Apply a single strategy s to the available child terms until the first successful application and fail if none succeeds:

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

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

Appendix B

A Type Inferencer for TFOF

Chapter 6 discussed the deforestation of functional programs to reach a treeless form, built around the deforestation algorithm by Philip Wadler [Wad90]. Wadler extends the treeless property to the *blazed treeless* property, and slightly extends the original algorithm such that it produces blazed treeless terms. This means that simple intermediate values (e.g. integer constants) *are* allowed, thus yielding a deforestation algorithm that is able to handle much more diverse terms. A requirement for the new algorithm is that the input term contains full type information including for all subterms. The blazed algorithm will not be further discussed here, as it does not add much to the already known uses of dynamic rules in Stratego, but we did implement a type inferencer for TFOF terms.

First, the TFOF syntax is extended with types and type annotations. Next, Milner's $\mathcal{W}$ algorithm for type inferencing is sketched, followed by the Stratego implementation. Finally, this approach is compared with a similar type inferencer in the UUAG attribute grammar system.

B.1 Extension of the TFOF syntax

To create a proper test environment for type inferencing, the TFOF language is extended with syntax for types, user defined data types and type annotations. Figure B.1 lists the new syntax, which should be seen as a whole with the existing syntax in Figure 6.1. Some existing entities, declarations d, function definitions fd and terms t, are extended. The rest of the syntax concerns new entities, of which types τ, type variables α and user defined data types tpd are the most important. A type comes in four sorts, other seemingly basic types (e.g. list-of types, type tuples) can easily be represented as data types. Note that the user defined data type is similar to data types in other functional languages, such as Haskell. A type specification like Haskell's `type` $\tau_1 = \tau_2$ is not included in the TFOF type system. This is usually just used for shorthand notation of more complex types anyway.

A second, minor extension are term expressions which can be summarized with:

$$t \ ::= \ t_1 \ op \ t_2 \quad \text{binary operator}$$

where op is one of the familiar arithmetic, relational or boolean binary operators. Before applying any transformation or type inferencing, these term expressions are desugared to

Types

τ	::=	α	type variable	`TpVar`
		ι	primitive type	`BasicTp`
		$\tau_1 \rightarrow \tau_2$	function type	`FunTp`
		$T\ \tau_1 \ldots \tau_m$	datatype	`TpCon`
α	::=	identifier	type variable	
ι	::=	`int` | `bool`	basic types	
T	::=	identifier	datatype identifier	

Declarations

d	::=	`types` $tpd*$	type declarations	`TypeDefs`
tpd	::=	`data` $T\ \alpha_1 \ldots \alpha_m = tpc_1 \mid \ldots \mid tpc_n$	user defined datatype	`DataType`
tpc	::=	$C\ \tau_1 \ldots \tau_k$	type constructor definition	`TypeConDef`
C	::=	identifier	type constructor	

Type annotations

fd	::=	$f : \tau$	function type specification	`FunTpSpec`
t	::=	$t :: \tau$	typed term	`Typed`

Figure B.1: Extended syntax for a typed TFOF language.

their functional notation, e.g.:

$$x + \text{sum}\ xs \rightarrow (+)\ x\ (\text{sum}\ xs)$$

Function identifiers may now also be (op), to fit this functional notation in existing syntax.

Functions can be separately typed, similar to Haskell style. Terms can be annotated with their types, including all subterms, similar to the *typing of prefix expressions* that Milner uses [Mil78]. Below is an example of a fully typed TFOF fragment:

```
fsum : (List int) -> int
fsum xs = fsum' (0 :: int) (xs :: (List int))

fsum' : int -> (List int) -> int
fsum' a xs =
  (case xs :: (List int) of
    Nil       : (a :: int)
    Cons x xs : (fsum' (((+) (a :: int) (x :: int)) :: int) (xs :: (List int))) :: int
  ) :: int
```

It is the definition of the sum operation over integer lists. Note how literally every subexpression that is of term sort is typed. Function arguments and case patterns are not typed,

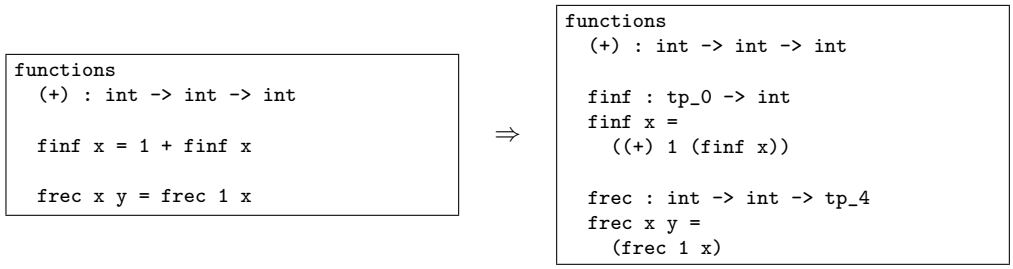


Figure B.2: Type inferencing for two simple recursive functions.

as their type can be inferred from the type of the function definition or `case` selector, respectively.

B.2 Examples of type inferencing

A type inferencer infers types for all subexpression of its input, unifying types for terms that should match, and refining previously inferred types when new typing information has become available. This section shows some peculiar examples of how the inferencing operates.

Functions and Argument Types Each argument in a function definition is initially typed with a fresh type variable. During examination of the body, more knowledge on the the argument types and result type becomes known. Figure B.2 shows two (infinitely) recursive function definitions. For `finf`, based on the type of its body that only contains an integer addition, the result type is inferred to be `int`. The type of argument `x` can not be further refined. For `frec`, the body contains a function call to itself. The first argument is inferred to be an `int`, since it is passed the integer constant 1. The second argument is also an `int`, since the first argument of the function definition is used as the second argument in the function call. The result type can not be further refined. Notice that the inferred types of functions are added to the code as separate function type specifications, and that the original function definition is left untouched. During inferencing the function body has been entirely type-annotated, but it is not needed for further processing anymore and it makes the output code look unnecessary complicated.

User-defined Data Types The extended TFOF format allows for new data type definitions. These are often used in combination with `case` terms in function bodies. The type inferencer has to unify the selector and all left hand side patterns of a `case` term with each other, and yield the unified type over all right hand side branches as result type. Figure B.3 lists some code that defines a list and tuple data type, and a function `findex` that adds indices to a list of arbitrary type (i.e. `["a","c","z"]` → `[(0,"a"),(1,"c"),(2,"z")]`).

Several elements play a role here: the helper strategy `findex'` is parameterized with some unknown `i`, which can later be determined to be an `int`, because of the type for `(+)`.

```
types
  data List a = Cons a (List a)
              | Nil

  data Tuple a b = Tuple a b

functions
  (+) : int -> int -> int

  findex' i es ies =
    case es of Nil : ies
               Cons e es : findex' (i+1) es (Cons (Tuple i e) ies)

  findex es = findex' 0 es Nil
```

```
types
  data List a = Cons a (List a)
              | Nil

  data Tuple a b = Tuple a b

functions
  (+) : int -> int -> int

  findex' : int -> (List tp_23) -> (List (Tuple int tp_23))
            -> (List (Tuple int tp_23))
  findex' i es ies =
    case es of Nil : ies
               Cons e es : findex' ((+) i 1) es (Cons (Tuple i e) ies)

  findex : (List tp_29) -> (List (Tuple int tp_29))
  findex es = findex' 0 es Nil
```

Figure B.3: Type inferencing with user-defined data types.

The accumulation parameter `ies` is used within a `Cons` application, together with a `Tuple` element. Determining that this is some instance of a `List` yields the type of `ies`, and thus the function's return type as well. From the inferred type of `findex'`, the type of `findex` is easy to infer.

Type Errors The type inferencer is intended to detect possible type errors in programs. As Section B.4 will show, the abstract syntax for types has an additional `TpError` constructor, that stores typing errors with an informational string. Real-world compilers will present type errors to the user during compilation, probably with the original line number. Such advanced error handling is not our goal here. Type errors are seen as a special type, and are used in normal type annotation. When inferencing has finished, type errors are pretty printed as comments in the final output. The type annotation itself is shortened to `t :: _`,

```
functions
  (*) : int -> int -> int
  (+) : int -> int -> int
types
  data List a = Nil | Cons a (List a)
terms
  t1 := (*) 4
  t2 := 1 * true
  t3 := case x of
          Cons x xs : 1+xs
          Nil : 0
```

```
functions
  (*) : int -> int -> int
  (+) : int -> int -> int
types
  data List a = Nil | Cons a (List a)
terms
  t1 := ((*) (4 :: int)) :: _ /* No matching function definition found for (*)/(1) */

  t2 := ((*) (1 :: int) (true :: _ /* Unification failed */))
        :: _ /* Type error in function argument(s). */

  t3 := (case x :: (List tp_1) of
          Cons x xs : ((+) (1 :: int) (xs :: _ /* Unification failed */))
                      :: _ /* Type error in function argument(s). */
          Nil : (0 :: int)
        ) :: _ /* Types in case-branches do not match. */
```

Figure B.4: Type inferencing with error handling.

indicating an uninferred type. Figure B.4 shows three terms that result in typing errors. Term `t1` lacks the second argument for the multiplication operator, and since TFOF is a first order language, this results in a 'function not found' error. $f/(n)$ refers to a function named f, with n arguments. The second term `t2` contains the correct amount of arguments in the function call, but their types do not match those of the formal function parameters. Finally, `t3` intends to compute the length of a list, but the recursive call to count `xs` has been forgotten. This results in an error in the call to `(+)`, similar to the one in `t2`. This error is propagated and results in a unification error over all case branches.

B.3 Milner's $\mathcal{W}$ Algorithm

In an initial publication, Robin Milner gives a formal description of type polymorphism, and introduces the well-type algorithm $\mathcal{W}$ [Mil78]. All of this is described of a simple functional language Exp, a later publication reconsiders the theory and algorithm and proves that $\mathcal{W}$ always yields the most general type for an expression [DM82]. The details of the algorithm

Type schemes			
σ	$::= \forall\alpha_1 \ldots \alpha_n\tau$	type scheme	`TpForall`
	τ	type	
	$\top$	any type	`TpAny`
	$\bot$ string	Type error (with message)	`TpError`
Type annotations			
t	$::= t :: \sigma$	partially typed term	`Typed`

Figure B.5: Term and type sorts for use of type schemes during inferencing.

are not discussed here, but some basic concepts need to be introduced.

Type Schemes When reasoning about polymorphic types, inferencing and unifying them, the type sort τ is too restrictive to use. Instead, *type schemes* are used during inferencing. A type scheme σ is a normal type τ, possibly generalized over a list of yet unbound type variables $\alpha_1 \ldots \alpha_n$. To support the inferencing implementation even more, two sorts are added. The top type $\top$, pretty-printed as _, denotes *any* type when nothing is known yet. The bottom type $\bot$ denotes a typing error and has one argument, a string containing a short informational message on that specific type error. Figure B.5 shows this formally. During inferencing, terms can be type annotated with type schemes as well.

Type Unification The algorithm aims to find the most generic type for each program fragment, but during this process various parts of the program should correspond with one another. The process of matching two types is called *type unification*. Given two types τ_1 and τ_2, the unification process finds a third type τ_3, which is a *subtype* or *instantiated type* of both τ_1 and τ_2. During inferencing and unification, a substitution S is maintained than maps type variables α to type schemes σ. Upon unification of types that still contain type variables, (some of) these variables are instantiated to a certain type or typescheme, which is recorded in the substitution relation S. Dijkstra and Swierstra [DS01] describe the process in more detail, below are the unification rules that are used here:

$$
\begin{array}{lcl}
U(\bot, \_) & = & \bot \\
U(\top, \tau) & = & \tau \\
U(\alpha, \tau) & = & \tau,\ S = S \cup [\alpha \mapsto \tau] \text{ iff } \alpha \notin \text{freevars}(\tau) \\
U(T\ \tau_{a,1} \ldots \tau_{a,m}, T\ \tau_{b,1} \ldots \tau_{b,m}) & = & T\ \tau_{c,1} \ldots \tau_{c,m}, \\
 & & \text{where } \tau_{c,i} = U(\tau_{a,i}, \tau_{b,i}) \text{ and } \forall i\ :\ \tau_{c,i} \neq \bot \\
U(\iota, \iota) & = & \iota \\
U(\_, \_) & = & \bot
\end{array}
$$

The rules should be tried in the specified order: only if none of the rules applies, the unification ends up in the default case, the final rule, which yields a new type error. The other rule handle the various types: a type error $\bot$ can never be unified with any other term, a yet unknown type $\top$ always unifies with any other type. Unifying a type variable

with some type leads to a new substitution relation, if the type variable did not occur free in the other type. Unification distributes over type constructors T, and finally primitive types only unify if they are identical. Note that the first three rules can also be applied with their two argument types switched.

The $\mathcal{W}$ Algorithm The $\mathcal{W}$ algorithm for type inferencing is in fact a bottomup traversal of the input. For each term a type is tried to be inferred using *type inference rules*. The rules used here can again be found in the course material by Dijkstra and Swierstra [DS01]. All inferred types for program variables and function identifiers are registered in an environment Γ, e.g $\Gamma = [x \mapsto \texttt{bool},\ \mathrm{square} \mapsto \texttt{int -> int}]$. This environment is consulted and extended intensively during the type inference process. In short: for each new program variable encountered, a fresh type variable is assigned. For each function application, a fresh instance of the function type is used (by instantiating the argument types τ_i). The type of a function definition is based on the inferred type of its body. For both function and data constructor application, the inferred types of the passed arguments are unified with the instantiated types of the function definition and data type definition respectively. Finally, for `case` expressions, the selector and all branch patterns are unified, and the branch bodies are unified to yield the result type.

B.4 Implementation in Stratego

The algorithm first descends into the smallest subterms, inferring their types and using that for further inferring and unifying upward. Functions and user defined data types need special registration. The following paragraphs discuss the various parts that together form the entire type inference implementation. A list of globally used helper strategies is shown in Figure B.6.

Data Types and Function Definitions A TFOF module consists at top level of type function and term definitions. The first two need to have their inferred type 'registered', so that later uses (i.e. function calls and constructor applications) are type-inferred using that information. Currently, our type inferencer does not resolve any yet undefined functions, resulting in type errors. The input thus should have every function and data type definition before its uses.

Figure B.7 lists the code that treats all definitions. For each data type constructor in a data type definition, a dynamic rule `KindOf` is defined, that maps a type constructor application to the correct data type, for example:

```
|[ Cons int tp_0 ]| -> |[ List int ]|
```

In this example, the call to `unifyTypes` has added a mapping $tp\_0 \mapsto$ `|[List int ]|` to the substitution relation S. Here, S is implemented by dynamic rules `SubstTpVar`.

Since a data type constructor may be used in multiple definitions (with different arities) an *extend rule* is defined. Upon application of `KindOf` any type errors in child terms are detected, using a `oncetd` traversal and reflected in the result type.

```
MkFunTp : (tpA , tpB) -> FunTp(tpA , tpB)

// Transforms (tpA -> (tpB -> (tpC -> tpD))) into ([tpA,tpB,tpC],tpD)
flatten-FunTp =
  !([], <?FunTp(_, _)>)
; rec r({
    if ?(a*, FunTp(tpA , tpR)) then <r> ([tpA | a*], tpR) else id end
  })
; (reverse, id)

extractType = ?Typed(_, <id>)
freshTpVar  = !TpVar(<new-tpvarid>)
new-tpvarid = <newname> "tp"

InitVarType:
  Var(v) -> tpV
  where freshTpVar => tpV
      ; rules(Id2Type: v -> tpV)

fresh-tp-vars(|tpV*) =
{| FreshTpVar :
  where(<map({tpV,tpN:
           ?tpV
         ; where(new-tpvarid => tpN)
         ; rules(FreshTpVar: TpVar(tpV) -> TpVar(tpN))
         })> tpV*)
; leaves(FreshTpVar, TpVar(id))
|}

generalize-type :
  tp -> tpgen
  where tfof-free-tpvarids => qv*
      ; if !qv* => []
        then !tp => tpgen
        else !TpForall(qv*, tp) => tpgen
        end

specialize-type =
  ?TpForall(tpV*, tp)
; <fresh-tp-vars(|tpV*)> tp

tfof-free-vars =
  free-vars(\Var(v) -> [Var(v)]\, \(Pat(_, v*), _) -> <map(Var(id))>v*\)
tfof-free-tpvarids =
  free-vars(?TpVar(<![<id>]>), ?TpForall(<id>,_))
```

Figure B.6: Various helper strategies used in type inferencing.

```
infer-types = {| Id2Type, SubstTpVar: Module(id, map(try(infer-from-decl))) |}

infer-from-decl =
  TypeDefs(map(HandleDataTpSpec))
+ FunDefs(map(HandleFunTpSpec + InferFunDef); flatten-list)
+ TermDefs(map(InferTermDef))

HandleDataTpSpec =
  ?DataType(D, tpv*, tpb*)
; where(<map(!TpVar(<id>))> tpv* => tpV*)
; where(!TpForall(tpv*, TpCon(D, tpV*)) => tpD)
; where(<map({
          ?tpb@TypeConDef(C, tp*)
        ; rules(
            KindOf :+
              TpCon(C, _) -> tpD''
              where <fresh-tp-vars(|tpv*)> (TpCon(C,tp*), TpCon(D, tpV*))
                    => (tpb', tpD')
                  ; <unifyTypes> (<id>, tpb')
                  ; (oncetd(TpError(!"Error in child types."); ?err); !err
                     <+ <subst-TpVars> tpD') => tpD'')
        })> tpb*)

HandleFunTpSpec =
  ?FunTpSpec(f, tpF)
; rules(Id2Type: f -> tpF)

InferFunDef :
  |[ f v* = t ]| -> [ts, |[ f v* = t ]| ]
  where {| Id2Type, SubstTpVar :
          <map(InitVarType)> v* => tpV'*
        // Introduce funtype here already (for recursive fundefs)
        ; freshTpVar => tpR
        ; <foldr(!tpR, MkFunTp)> tpV'* => tpF'
        ; rules(Id2Type: f -> tpF')

        ; <term-it> t => Typed(_, tt)
        ; <map(subst-TpVars)> tpV'* => tpV*
        |}
        ; <foldr(!tt, MkFunTp)> tpV* => tpF
        ; <generalize-type> tpF => tpFgen
        ; !FunTpSpec(f, tpF)  => ts
        ; rules(Id2Type: f -> tpFgen)

InferTermDef :
  |[ q := t ]| -> |[ q := t' ]|
  where <tfof-free-vars; map(InitVarType)> t // to allow unbound variables
      ; <term-it> t => t'
```

Figure B.7: Type-inferencing strategies for datatypes, function and term definitions.

```
term-it =
  bottomupS(term-it-basic, buIT)

buIT(s) =
  Int(id)
+ True
+ False
+ Var(id)
+ Con(id, map(s))
+ FunApp(id, map(s))
+ Case(id,id)
+ Pat(id,id)
+ Typed(id, id)

term-it-basic =
  Typed(id, id)
<+ TermIT
<+ !Typed(<id>, TpAny)

TermIT =
  IntegerIT
+ BoolIT
+ VarIT; try(SpecIT)
+ ConIT
+ FunAppIT
+ CaseIT
```

Figure B.8: Term traversal strategies for type inferencing.

For a function definition, its formal parameters are assigned a fresh type variable. Before recurring into the body, a simple function type based on these fresh type variables is registered in `Id2Type`, such that recursive function definitions can be handled as well. The dynamic rules `Id2Type` model the environment

Traversal Figure B.8 lists the code for the traversal part. The top-level strategy is a bottomup traversal that is steered by `buIT` to avoid descending into unnecessary subterms (e.g. function identifiers), or blocking further traversal to handle it in a specialized way later (`case` terms).

Once the appropriate subterms of a terms have been inferred, the current term itself should be considered. The top level strategy for this is `term-it-basic`, which usually calls `TermIT`. The various inference rules are discussed hereafter.

Type Inference Rules Type inference rules infer the type of their input term, given the already inferred types of their subterms. Figure B.9 lists the rules for the simplest terms: integer and boolean constants and variables. A variable is inferred from a mapping in the environment Γ (by `Id2Type`). In the meantime, new substitutions may have been found for

```
IntegerIT : |[ n ]| -> |[ n :: int ]|
BoolIT :    |[ true ]| -> |[ true :: bool ]|
BoolIT :    |[ false ]| -> |[ false :: bool ]|

VarIT : |[ v ]| -> |[ v :: tp ]|
  where ?Var(<id>) => vid
      ; (<Id2Type; subst-TpVars> vid
         <+ !TpError(<concat-strings> ["Variable '", vid, "' undeclared."])
        ) => tp
```

Figure B.9: Type-inferencing rules for basic term types.

any type variables in the mapped type. `subst-TpVars` applies them, if any, thus yielding an up-to-date type.

For the rest of the term sorts, some more effort is needed. Figure B.10 lists the strategies for them. Variables starting with *tt* represent typed terms, *t* is used for untyped terms, and *tp* for types.

A constructor application is handled by `KindOf`, which rewrites the type constructor to a user defined data type. Before doing so, the types of all argument terms are specialized, which means freshly instantiating any 'forall-types'. The application of `KindOf` might have led to new type mappings, which are applied to the original child terms *tt** to get them up-to-date.

Function applications look up the inferred type for the function definition, if it is known. The `FunTps` are flattened to the form `([`*tp_0*`,...,`*tp_k*`],`*tp_R*`)`. Now, all passed function arguments can be unified with the formal parameter's types in one list operation. Again, all freshly acquired type substitutions are afterward applied to the entire list of arguments, thus getting them up-to-date as well. Possible typing errors in a function call are incorrectly typed arguments, or the function definition itself may be unknown.

The rule for a `case` term has to unify the selector *t0* with all branch patterns, and all branch bodies with one another. These two unifications are handled in parallel by one `foldr`, where `CBWalker` performs the folding. The initial values for the fold are the selector type for the pattern part, and any type `TpAny` for the body part. When the entire list of branches has been traversed, the rule checks for any errors in them, and if all are correct, yields the body type as result type. Again, all branches receive updated type information by applying `subst-TpVars`.

Type Unification The rules for type unification were discussed in the previous section, at page 124. The implementation in Figure B.11 is straightforward and closely follows these rules. The correct order of application if maintained by a sequence of deterministic left-choices. The listing also show the implementation of `subst-TpVars` which updates a term with all the new type information, using `innermost` rewriting.

```
SpecIT : Typed(t, tp@TpForall(_, _)) -> Typed(t, tp')
  where <specialize-type> tp => tp'

ConIT : ct@Con(c, tt*) -> Typed(Con(c, tt'*), tpC)
  where <map(try(SpecIT); extractType)> tt* => tp*
      ; <KindOf> TpCon(c, tp*) => tpC
      ; <subst-TpVars> tt* => tt'*

FunAppIT : FunApp(f, t*) -> Typed(FunApp(f, t'*), tpR')
  where
    if <Id2Type; subst-TpVars; try(specialize-type)
        ; flatten-FunTp> f => (tpA*, tpR)
      ; <(length, length); eq> (t*, tpA*)
    then
      <zip(\ (tpA, Typed(t,tp)) -> Typed(t,<unifyTypes> (tpA, tp)) \)
      ; map(subst-TpVars)> (tpA*, t*) => t'*
    ; if one(?Typed(_, TpError(_)))
      then !TpError("Type error in function argument(s).")
      else <subst-TpVars> tpR
      end => tpR'
    else
      !t* => t'*
    ; !TpError(
        <concat-strings> ["No matching function definition found for ",
                          f, "/(", <length; int-to-string> t*, ")"]) => tpR'
    end

CaseIT : Case(t0, b*) -> Typed(Case(tt0, tb'*), tpc')
  where <term-it> t0 => Typed(t0',tp0)
      ; <foldr(!([], tp0, TpAny), CBWalker)> b* => (tb*, tps, tpc)
      ; <try(TpError(!"Types in case-branches do not match."))> tpc => tpc'
      ; !Typed(t0', <subst-TpVars> tp0) => tt0
      ; <map((id,subst-TpVars))> tb* => tb'*

CBWalker : ((pati@Pat(c,v*),ti), (tb*,tpj,tpj')) -> (tb'*,tpi_final,tpi')
  where {| Id2Type:
          <map(InitVarType)> v* => tpV'*
        ; <KindOf> TpCon(c, tpV'*) => tpiC
        ; <unifyTypes> (tpiC, tpj) => tpi
        ; <term-it> ti => Typed(ti',tpiB)
        ; <unifyTypes> (tpiB, tpj') => tpi'
        ; <subst-TpVars> tpi => tpi_final
        ; ![(pati, Typed(ti', tpi')) |tb*] => tb'*
        |}
```

Figure B.10: Type-inferencing rules for more complex term types.

```
unifyTypes =
let unify-basic =
      unifyTypes1
   <+ unifyTypes2
   <+ unifyTypes3
   <+ unifyTypes4
 in unify-basic
 <+ unifyError
end

unifyTypes1: (tp@TpError(_), _) -> tp
unifyTypes1: (_, tp@TpError(_)) -> tp
unifyTypes1: (TpAny, tp)        -> tp
unifyTypes1: (tp, TpAny)        -> tp
unifyTypes2: (TpVar(l), tp@TpVar(r)) -> tp
  where !l => r
        <+ rules(SubstTpVar: l -> tp)
unifyTypes3: (TpVar(l), tp) -> <TpVarBind> (l, tp)
unifyTypes3: (t, TpVar(r)) -> <TpVarBind> (r, tp)
unifyTypes4: (TpCon(c, tp1*), TpCon(c,tp2*)) -> tpC
  where <unifyLists> (tp1*, tp2*) => tp*
      ; if <one(?TpError(_))> tp*
        then !TpError("Error in data type argument(s).")
        else !TpCon(c, tp*)
        end => tpC
unifyTypes4: (BasicTp(b), BasicTp(b)) -> BasicTp(b)

unifyLists = zip(subst-TpVars; unifyTypes)

unifyError = !TpError("Unification failed")

/**
 * Unification is never called on polymorphic types (i.e. has always been
 * specialized), so...
 */
TpVarBind =
  ?(v, tp)
; if <not(oncetd(?TpVar(v)))> tp // ... no check on boundvars needed here.
  then
    rules(SubstTpVar: v -> tp)
  ; !tp
  else
    !TpError(<concat-strings>
             ["Type var '", v, "' also occurs freely in other type."])
  end

subst-TpVars =
  innermost(?TpVar(<SubstTpVar>))
```

Figure B.11: Strategies for type unification.

B.5 Comparison with an Attribute Grammar Approach

A basic type inferencer, implemented for the course on compiler construction in UUAG [DS01], formed a reference for the inferencing and unification rules used here. The implementations are hence similar in essence. Still, the observed differences between a strategic rewriting system and an attribute grammar system that were observed in Section 9.1, also apply here. The traversal of the input terms in AG follows purely from the attribute specification. In Stratego, the traversal is more present throughout the implementation. Partly, it is handled by `buIT`, partly it is inside the actual inference rules (`FunAppIT`, `CaseIT`). On the other hand, the manipulation of the assumption environment Γ, and the substitution relation S requires some more effort in the AG system, especially when abstracting over `let` terms (here: function definitions).

The Stratego implementation can make even more use of dynamic rules than is currently done. The inferencing of function applications is now almost statically specified, and just relies on the dynamic substitution `Id2Type`. An other approach would generate dynamic inference rules for function applications, for each just inferred function definition.

Bibliography

[AJ89] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–302. ACM Press, 1989.

[AJ97] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[AK01] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[BH03] Otto Skrove Bagge and Magne Haveraaen. Domain-specific optimisation with user-defined rules in CodeBoost. In Jean-Louis Giavitto and Pierre-Etienne Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE'03)*, volume 86/2 of *Electronic Notes in Theoretical Computer Science*, Valencia, Spain, 2003. Elsevier.

[BKK96a] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier Science Publishers.

[BKK+96b] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First Workshop on Rewriting Logic and Applications 1996 (WRLA'96).

[dBKV03] M. G. J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, April 2003.

[dBSVV01] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. Technical Report UU-CS-2001-39, Institute of Information and Computing Sciences, Utrecht University, October 2001.

[BvDOV04] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 2004. (Submitted).

[CM00] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier, 2000.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[DS01] Atze Dijkstra and S. Doaitse Swierstra. *Implementation of Programming Languages, Lecture Notes*. Utrecht University, Institute of Information and Computing Sciences, 2001.

[HP01] David R. Hanson and Todd A. Proebsting. Dynamic variables. In *Programming Language Design and Implementation (PLDI'01)*, Snowbird, UT, USA, June 2001. ACM.

[J00] Merijn de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, Limerick, Ireland, June 2000. Technical report, University of Wollongong, Australia.

[KM96] Hélène Kirchner and Pierre-Etienne Moreau. A reflective extension of ELAN. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First International Workshop on Rewriting Logic and its Applications.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 108–118. ACM, January 2000.

[MCM00] P. Lincoln M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MS01] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[OV02] Karina Olmos and Eelco 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.

[PTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001. ACM SIGPLAN.

[Ro04] Rodriguez Yakushev, Alexey. Attribute grammar extensions: Higher order attribute grammars and Views. Master's thesis, Utrecht University, Utrecht, The Netherlands, August 2004.

[SAS98] S. Doaitse Swierstra, Pablo R. Azero Alcocer, and Joao Sariava. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[STR] Stratego Interpreter website. http://www.stratego-language.org/Stratego/StrategoInterpreter.

[TFO] TFOFDeforest package website. http://www.stratego-language.org/Stratego/TFOFDeforest.

[TIG] Tiger package website. http://www.stratego-language.org/Tiger.

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew 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.

[vdBdJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[Vis00] Eelco Visser. Language independent traversals for program transformation. In Johan 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.

[Vis01] Eelco Visser. Scoped dynamic rewrite rules. In Mark van den Brand and Rakesh Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid 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.

[Vis04a] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[Vis04b] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 2004. (To appear).

[Wad84] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52. ACM Press, 1984.

[Wad85] Philip Wadler. Listlessness is better than laziness II: Composing listless functions. In *on Programs as data objects*, pages 282–305. Springer-Verlag New York, Inc., 1985.

[Wad89] Philip Wadler. The concatenate vanishes. Note originally published to FP electronic mailinglist, 1987 (Revised, November 1989).

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wik] Wikipedia. Lambda calculus. http://wikipedia.org/wiki/Lambda_calculus.