Strategies for Context Sensitive Program Transformation

Strategieën voor context afhankelijke transformatie van programma's

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. J.C. Stoof, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 27 mei 2009 des middags te 4.15 uur door

Karina Rosario Olmos Joffré

geboren op 4 oktober 1969 te Cochabamba, Bolivia

Promotor:	Prof. dr. S.D. Swierstra
Co-promotor:	Dr. E. Visser

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
This research was financially supported by Philips Research Laboratories.

Cover illustration: Patricia Garcia.
ISBN 978-90-393-5059-1

For my mother.

Contents

Part I

Preliminaries

1 Introduction

This chapter presents an introduction to program transformation, program transformation systems and discusses the different program representation levels at which program transformations are typically applied. The implementation of program transformations is not simple, and therefore we are interested in studying and finding abstractions with the aim to simplify their implementations, thus making the process simpler and improving the productivity of the transformation developer. For this purpose we investigate how to implement program transformations (compiler transformations) at a high level of abstraction. We use rewriting to implement such transformation systems.

The objective of *program transformation* is to change one program into another [79]. More broadly speaking, it changes the structure that represents a computer program or model. There are several reasons why we may want to generate a program from another one. For instance to obtain an optimised version or to improve the clarity of the code for maintenance purposes. An important issue to take into account is that the variation between the original and the obtained program must be *observable*. The observable behaviour of the execution of a program includes the computed result, speed of the computation, memory usage, energy consumption, failure, and the class of input programs for which the execution succeeds.

On the one hand, for certain generated programs the change will be observable when the program is being executed. On the other hand, for program transformations that aim at achieving modifications at source code level the change will be mainly

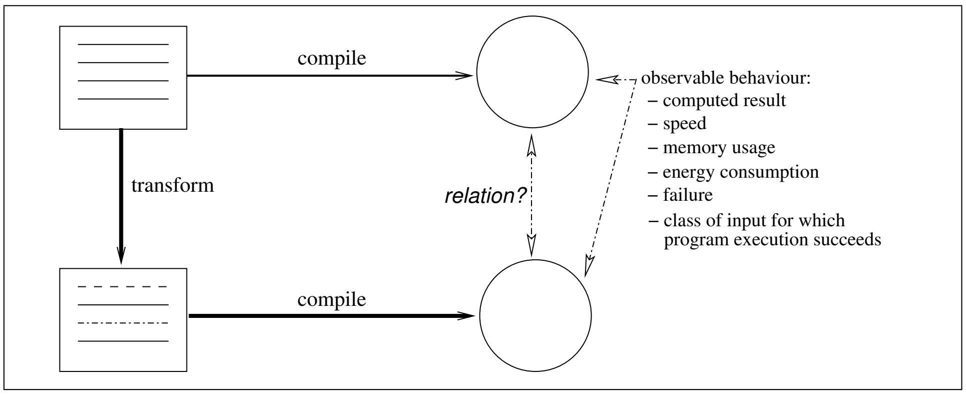

Figure 1.1: Relation of observable behaviour of a compiled program vs. transformed program.

observable in the source code [41]. Thus, the goal of program transformation is to obtain programs that preserve certain observable aspects while modifying others.

Figure 1.1 suggests differences in behaviour that might result from transforming a program. For the interpretation of the picture, we assume that both programs are compiled with the same compiler and executed in the same environment. The picture depicts two different flow paths for obtaining executable programs. The first path describes a program obtained by compilation. The second path (depicted with thick arrows) describes the process of first transforming a program and its subsequent compilation.

The manipulation of a program can be realised in a manual, semiautomatic (*interactive*) or fully automatic way. The last approach improves development productivity, reduces errors, and can be applied to multiple programs without any manual modification to the sources or to the process (i.e., it is reproducible; it has the same advantages as a computer program).

A *program transformation system* transforms and obtains a new version of a program from an old one. Programs are the input data of program transformation systems. Typically such systems apply meaning preserving modifications to the input. Such modifications do not alter the program's computed value(s). A typical example is a compiler. In order to compile a program, a series of modifications are applied to the input, and as a result an executable program in a machine language is obtained. Instances of transformation systems that preserve the computed result include program optimisers, program refinement tools, and compilers. Program transformation systems can also alter the semantics of a program in a controlled way. For instance, code maintenance tools can add a parameter to a function, change the type of a variable and update the program accordingly, adjust array boundaries or change data representations.

1.1 Applications of Program Transformation

The first automatic program transformation systems originated in the context of compilation [1]. Other instances of program transformation systems include program synthesis for the derivation of programs from a high level specification [112, 43, 87, 40], and intentional programming for software construction where the code reflects the design intention of the programmer [83, 33, 116]. These systems aim at increasing developer productivity.

Recently, new approaches for the support of the software engineering process that rely on program transformation have emerged such as generative programming [30, 86], refactoring [41, 55, 91], and code weaving [56, 114, 106]. Generative programming refers to a programming style that focuses on designing solutions for a family of software engineering problems, instead of solving each problem completely separately. Refactoring tools are used to improve code design by restructuring the source code of programs. Code weaving allows us to merge different aspects of a software model into a single implementation. In this way the software implementation can be specified without having to intertwine the different aspects by hand. Other examples of program transformation occur in the context of software renovation [96] which includes language migration and reverse engineering to support code maintenance [82, 92, 22].

Some applications of program transformation aim at changing structural aspects of the code such as code size, modularity, and clarity. Instances of such system are language migrators, code obfuscators, and code refactorers. A language migration system [109, 81] translates a program written in a programming language into a program written in a different programming language or to a newer or different version of the programming language which was originally used. Obfuscation [26, 108, 113] is a transformation that makes a program harder to understand by renaming functions and variables, inserting code that will not be executed, etc. Obfuscation is a preventive defence to hide engineering knowledge by making it harder to uncover any secret of trade hidden in the program text.

Although these new approaches all aim at different goals, they all involve program manipulation. Thus, we argue there is good reason for developing a common infrastructure as basis for program manipulations at source code level.

1.2 Realising Program Transformations

The realisation of program transformation systems has to consider different aspects of a programming language, such as its syntax and semantics and the computational equivalence between different program fragments. For building a program transformation system the following ingredients are required:

1. A meta-language to describe the transformation tool per se
2. The object language(s), the language(s) in which the input programs were developed

3. Intermediate program representations where applicable
4. Information facts required to trigger a transformation
5. Semantics of the object and target language

The meta-language is the language used in the implementation of the program transformation system. This can be a general purpose language such as C, Java, C#, or a domain specific language for program transformation such as Stratego [17, 88, 103]. In order to support the description of the program transformations, it is desirable to use a programming language with meta-programming capabilities. The language should provide idioms for writing program transformations in an easy manner, while preserving the intentions of the transformation developer.

Input programs were developed in some language, to which we will refer as the object language. A (input) program can be transformed into a program written in a different language. If the input and the target language are the same, we speak of source-to-source program transformation. A compiler is an example of a transformation where the input language and the target language differ.

Program transformation can be applied at different program representation levels. On the one hand, applications such as code refactoring or domain-specific transformations are usually applied at source code level. On the other hand, optimising compilers usually apply optimisations at intermediate representation levels. Optimising compilers are common program transformation systems and there are three major groups of transformations for procedural, imperative programming languages. For each group there are preferred representation levels at which transformations can be applied.

Peephole optimisations [72] are local transformations mostly based on computational equivalences.[1] They are machine-dependent optimisations which replace small sequences of instructions by equivalent ones, which usually execute faster or are smaller. These transformations are applied at low level intermediate representation languages.

Loop optimisations are transformations aiming at improving the execution time of iterative expressions [5, 72]. In order to transform loops such as `while` or `for`, the transformation uses information about the data dependencies between statements in the body of a loop. Common examples of loop optimisations are loop unrolling, loop-invariant code motion, strength reduction, and vectorisation. Such transformations are applied preferably when it is easy to determine the structure of a loop, which could be at source level or at an intermediate graph-based representation where nodes identify loop statements.

Data-flow optimisations use information about a program that is collected from many different locations. By considering all possible run-time execution paths, it is possible to perform evaluations or to establish relations at transformation-time.

[1] Computational equivalencies are different implementations of code fragments that compute the same value but usually one implementation improves certain behavioural aspects.

Classical examples of data-flow optimisations are constant propagation, dead code elimination and common subexpression elimination. Data-flow transformations can be applied at several representation levels, of which intermediate representation levels are the most common ones.

We are interested in the implementation of automatic program transformation systems to allow the developer (meta-programmer) to write executable program transformations at a *high level of abstraction*. We believe this way of constructing program transformations speeds up the development process.

1.3 Program Transformation with Strategic Rewriting

Term rewriting is a computational model based on rewrite rules. A rewrite rule represents a single stepwise modification to the input. Term rewriting is an elegant formalism to implement program transformation systems. When many rewriting rules are available there is a large chance that many can be applied at a specific point in time. The question which now arises is whether the order we select makes a difference, and if so what order to take. Chosen orders can also, despite giving the same result, exhibit a huge difference in the number of rewrite steps required to get to the result. The hypothesis this thesis is built upon is that a programmer has to specify the order in which to apply the rewrite steps explicitly: they have to give strategies for finding possible rewrite steps. Specifying such orders we will refer to as **strategic rewriting**.

1.3.1 Strategic Rewriting

Strategic rewriting extends the computational model of term rewriting. It approaches the problem of controlling the rewriting process by providing a programmable interface.

Stratego [17, 88] is a domain-specific language for the specification of program transformation systems based on the paradigm of strategic rewriting. Stratego provides a programming language to specify the control over the rewriting process. A strategy allows to define different aspects of the rewriting process, such as the definition of rule application entailing rule priorities and the definition of tailored/generic traversals. The description of a control strategy has the following advantages: (1) separation of rules from the reduction strategy, and (2) reusability of rules and strategies.

A problem associated with pure term rewriting systems is the intrinsic context-free nature of rewrite rules. Whether a rule can be applied is completely determined by its left-hand-side. Application of a rewrite rule has only access to the information in the left-hand-side, and it is limited to the term to which the rule is currently being applied. Program transformations systems often need to relate information that originates from several program points, which are most likely distributed all over the program. An example of this is inlining, which replaces a function call with the body of the function after having substituted the formal parameters with the actual ones. The substitution at the call site requires to have access to the definition of the function. Function definition and function calls are usually situated at different locations of a program. This is an example of the need for context-sensitive information.

1.3.2 Source-to-source Data-Flow Transformations

In contrast to compilers and program synthesis systems that aim at translating code into executable machine code, source-to-source transformation systems transform programs at source code level that needs to be compiled for its execution. An advantage of applying transformations at source code level is the presence of high-level programming constructs that can be exploited for domain-specific or aggressive transformations.

Program transformation systems often require the aid of so called program analysis techniques [73]. A subset of program analysis techniques is known as data-flow analysis. The objective of a data-flow analysis is to collect information which is valid for any possible execution path of a program. Inferred facts can be properties of variables, constant values, and in general any information that can be used for optimising a program. Thus, data-flow optimisations have to be able to collect and inspect context-sensitive information for their realisation.

1.4 Contribution and Outline

Program optimisations and transformations have been mainly concerned with performance improvement and memory usage of program transformation systems and the search for aggressive optimisations. However, the construction of software tools is mainly centred around support for parsing, code generation and pretty-printing [8, 31, 42, 66]. There have been far fewer attempts at building tools to support the construction of optimisations and transformations. [62, 69, 111] Despite of the support provided by these tools, the development of optimising compilers and transformation systems is still a laborious process.

The literature about optimisations describes program optimisations using succinct text explanations [1, 5, 72, 73]. There are few examples that attempt to provide program transformation descriptions that are executable [4, 62, 111]. In order to build optimising compilers that have good execution performance, compiler developers use low level languages for their implementation. Thus, the trade-off between compilation time versus developer productivity is reflected in the large amount of time and effort required to construct an optimising compiler or a specific program transformation system.

The contribution of this dissertation is the implementation of executable context-sensitive data-flow optimisations at a high level of abstraction. We have chosen data-flow transformations because many transformation systems have to deal with such concepts.

The high-level abstraction allows us to achieve the following results:

- Integration of analysis and transformation.
- Combination of multiple data-flow optimisations into a single program transformation.
- High-level definition of transformations without committing to a specific source language.
- The combination of different types of transformations.
- The correct treatment of variable binding constructs and lexical scope.

1.4.1 Thesis Outline

The thesis is organised in three parts. The first part provides a gentle introduction to program transformation systems and describes the object language used in the thesis. The second part describes the strategic approach for the realisation of program transformation systems and presents a concise introduction to Stratego, the language used for the implementation of the transformations. After the extensive introduction, the last part studies data-flow transformations and partial evaluation and how to implement them at source code level. We now provide a brief description of the content of each chapter.

Example Object Language (2) This chapter describes the syntax and semantics of Tiger, the programming language that will be used as the main object language in the rest of this thesis.

Realising Program Transformations (3) The goal of program transformation is to obtain a new program from an old one while improving some observable aspect. For its realisation a program transformation system may build an infrastructure specialised for the programming language at hand. This chapter discusses the tooling to support these activities. In particular it describes the Stratego/XT framework for the specification of the generic transformation components that comprise a program transformation system.

Strategic Rewriting (4) Program transformation systems are implemented as a sequence of consecutive program modifications. Such modifications can be represented by a collection of rewrite rules. Each rule defines a single transformation step. Term rewriting systems are required to guarantee that a set of rules induces a rewriting relation which is confluent and terminating. Extensions to term rewriting systems introduce several mechanisms to control the rewriting process. In this chapter, we focus on describing strategic rewriting by specifying strategies using Stratego. The Stratego language constructs are described and example program transformations are presented to illustrate the strategic rewriting model.

Context-Sensitive Transformation (5) Program transformation often needs information which is located at different sites in an abstract syntax tree. We present examples where context-sensitive information is required for the realisation of transformation systems. Simple rewrite systems only based on rewrite rules can only access the information available in the term that is being transformed. This kind of systems lacks high level abstractions to represent context-sensitive information. Dynamic rewrite rules are an extension of Stratego which allows us to inherit information and make it available at different program points, thus providing context-sensitive information. Implementations using dynamic rules are provided as motivating examples of context-sensitive transformations.

Data-flow Transformation (6) Data-flow transformations are not only used in compilers but also in other programming tools such as aspect weavers, code generators, domain specific optimisers and refactoring tools. Due to the broad applicability of data-flow transformations, there are several requirements that a framework for developing data-flow optimisations might fulfil. In this chapter we present the foundations for solving data-flow problems and resulting design issues for data-flow transformations. A review of related work and existing approaches to the challenges for realising data-flow transformations is presented. Finally, we introduce the Stratego approach for realising data-flow transformations.

Control-Flow and Dynamic Rule Sets (7) Data-flow optimisations are usually implemented on low-level intermediate representations. This is not appropriate for source-to-source optimisations, which reconstruct a source level program. In this chapter we show how constant propagation, a well known data-flow optimisation, can be implemented at source code level using Stratego by making use of dynamic rules and operations on dynamic rule sets. A particular feature of the implementation is the integration of analysis and transformation in a single traversal.

Dependent Dynamic Rewrite Rules (8) Dynamic rewrite rules were used to propagate data-flow facts concerning a single variable in the implementation of constant propagation. Data-flow optimisations such as copy propagation and common subexpression elimination involve more than a single variable for their realisation. To implement such optimisations, we introduce dependent dynamic rewrite rules which enables us to implement optimisations which involve more than a single variable.

Reusing and Combining Data-flow Transformations (9) Generic traversals for forward and backward data-flow propagation are designed in such a way that they can be reused to implement data-flow transformations. Different instantiations of the generic strategies for the implementation of constant propagation, copy propagation, common subexpression elimination, forward expression substitution and dead code elimination are provided. When two or more transformations share specific transformation features, it is possible to define a combination of transformations in a single traversal. Thus, a superoptimiser is presented which combines renaming, constant propagation, copy propagation and common subexpression elimination.

Partial Evaluation with Strategic Rewriting (10) This chapter presents a strategic rewriting implementation of partial evaluation. We extend constant propagation, constant folding and removal of dead code to be applicable at interprocedural scope. Furthermore, functions invocations are unfolded or specialised with respect to the statically known arguments of the call. An implementation of poly-variant online partial evaluation is presented. Strategies are defined in a natural way to control the process.

The Octave Compiler (11) Most array processing languages such as APL, Matlab and Octave rely on dynamic type-checking by the interpreter rather than static type-checking and are designed for user convenience with a syntax close to mathematical notation. Functions and operators are highly overloaded. The price to be paid for this flexibility is computational performance, since the run-time system is responsible for type checking, array shape determination, function call dispatching, and the handling of possible run-time errors. In order to produce efficient code an Octave compiler should address those issues at compile-time as much as possible. In particular, static type and shape inferencing can improve the quality of the generated code considerably. In this chapter we show how overloading in dynamically typed Octave programs can often be resolved by program specialisation. We discuss the typing issues in compiling Octave programs and give an overview of the implementation of an Octave specialiser.

Conclusion (12) This chapter concludes the thesis by discussing the contribution of this work, comparing with related work, and explores possible future research avenues.

1.5 Publications and Software

Parts of this thesis are adapted from earlier publications.

- E. Visser. *Scoped Dynamic Rewrite Rules* [102]. The paper introduces dynamic rules to Stratego. The initial implementation of dynamic rewrite rules had a reduced set of operations. This thesis extends the initial design of dynamic rules in several ways with main goal to enable implementation of data-flow optimisations.
- K. Olmos and E. Visser. *Strategies for Source-to-Source Constant Propagation* [75]. The paper introduces dynamic rules for the propagation of context-information and the need of operations on dynamic rules. Chapter 5 is based on the content of this paper.
- K. Olmos and E. Visser. *Turning Dynamic Typing into Static Typing by Program Specialization* [76]. The paper discusses how overloading in dynamically

typed programs can be resolved by program specialisation. The content of the paper is included and further extended in Chapter 11.

- K. Olmos and E. Visser. *Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite Rules* [77]. The paper introduces dependent dynamic rules for the realisation of data-flow optimisations. It also describes the generic propagation frameworks for the implementation of data-flow optimisations. The content of this paper is presented in Chapters 8 and 9.
- M. Bravenboer, A. van Dam, K. Olmos and E. Visser. *Program Transformation with Scoped Dynamic Rewrite Rules* [19]. This paper explores the design space of dynamic rules, their application to transformation problems, and their implementation. This work is not contemplated in this thesis. For a reader interested in the internal design and implementation of dynamic rules we refer to this publication.

The code fragments displayed in this thesis are part of the implementation of the following applications.

- *Tiger Compiler*
  The implementation of the transformations described in this thesis are included in the Tiger compiler developed with Stratego as an exercise of compilation by transformation.
- *Octave Compiler*
  Octavec is the Octave compiler developed with Stratego also using transformation as implementation method. The compiler reuses the Octave parser for its implementation. This compiler translates Octave programs into C++ programs. Executable programs are obtained reusing Octave scripts.

2
Example Object Language

This chapter describes the syntax and semantics of Tiger, the programming language that will be used as the main object language in the rest of this thesis.

2.1 Introduction

Tiger is a programming language designed by Andrew Appel [5]. In his book, Appel uses Tiger as an imperative procedural programming language at first, then adds small modifications to construct different versions of Tiger as an object oriented and as a functional programming language. Despite the fact that Tiger is a relatively small language, it addresses most of the standard features of procedural programming languages which makes it a good and amenable vehicle for research. Furthermore, Tiger is relatively well known in the academic community. In this dissertation, we refer to the initial procedural version of Tiger. For an easy understanding of the transformations presented in this dissertation, we start by introducing the Tiger syntax and semantics.

2.2 Tiger Syntax

Figures 2.1 and 2.2 combine a BNF notation for Tiger with the names of the constructors and the respective arity of each program element. The figures relate the concrete syntax of Tiger with the structure of the abstract syntax tree.[1] We will

[1] The Tiger grammar presented here is used to generate a parser with the Stratego/XT tools. Priorities over rules are used to disambiguate the generated abstract syntax tree.

d	$::=$	var x (: ta)? (:= e)?	VarDec/3	variable declaration
	$\mid$	$fd_1, ..., fd_n$	FunDecs/1	function definitions
	$\mid$	$td_1, ..., td_n$	TypeDecs/1	type definitions
td	$::=$	type t = ts	TypeDec/2	type definition
ts	$::=$	$\{ fe_1, ..., fe_n \}$	RecordTy/1	record declaration
	$\mid$	array of ta	ArrayTy/1	array declaration
ta	$::=$	$int \mid string \mid tid$	TypeId/1	type identifier
fe	$::=$	x : ta	Field/2	record elem. type
fd	$::=$	function f($arg_1, ..., arg_n$)(:ta)? = e	FunDec/4	function definition
arg	$::=$	x : ta	FArg/2	function argument
tid	$::=$	$identifier$	Tp/1	type declaration
ifd	$::=$	x= e	InitField/2	record elem.initialization
e	$::=$	lv	LValue/1	lvalue
	$\mid$	$str \mid i$	Str/1, Int/1	string, integer
	$\mid$	tid $\{ ifd_1, ..., ifd_n \}$	Record/2	record initialization
	$\mid$	tid [$e_1, ..., e_n$] of e_{n+1}	Array/3	array initialization
	$\mid$	nil	NilExp	nil expression
	$\mid$	lv := e	Assign/2	assignment
	$\mid$	f (e_1, . . . ,e_n)	Call/2	function call
	$\mid$	(e_1; . . . ;e_n)	Seq/1	sequence
	$\mid$	if e_1 then e_2 else e_3	If/3	conditional
	$\mid$	if e_1 then e_2	IfThen/2	conditional
	$\mid$	while e_1 do e_2	While/2	while loop
	$\mid$	for x := e_1 to e_2 do e_3	For/4	for loop
	$\mid$	let $d_1...d_n$ in e_1;...;e_n end	Let/2	let binding
	$\mid$	break	Break	unconditional jump
	$\mid$	$bexp$		Binary expression
	$\mid$	$rexp$		Relational expression
lv	$::=$	x	Var/1	variable
	$\mid$	$lv.f$	FieldVar/2	record element
	$\mid$	lv[e_1, . . . ,e_n]	Subscript/2	subscript

Figure 2.1: Abstract syntax of Tiger. It includes constructor and arity information for the defined elements, where f, x and t denote identifiers.

aexp	::=	e_1 `+` e_2	`Plus/2`	addition
	|	e_1 `-` e_2	`Minus/2`	substraction
	|	e_1 `*` e_2	`Times/2`	multiplication
	|	e_1 `/` e_2	`Division/2`	division
rexp	::=	e_1 `<` e_2	`GT/2`	greater than
	|	e_1 `<=` e_2	`GTE/2`	greater or equal
	|	e_1 `>` e_2	`LT/2`	less than
	|	e_1 `>=` e_2	`LTE/2`	less or equal
	|	e_1 `<>` e_2	`NE/2`	not equal
	|	e_1 `=` e_2	`LEQ/2`	equal
	|	e_1 `&` e_2	`And/2`	boolean and operator
	|	e_1 `|` e_2	`Or/2`	boolean or operator

Figure 2.2: Abstract syntax of Tiger arithmetic and relational expressions.

discus in Section 3.4.2 how an abstract tree can be obtained in a Stratego setting by means of parsing. For the better understanding of the relation between the concrete syntax and the constructors and arity presented in Figures 2.1 and 2.2 we provide an informal explanation. Consider for instance the construct `let` $d_1...d_n$ `in` $e_1;...;e_n$ `end` which is represented by the abstract syntax tree `Let(`ds`,`es`)`, where ds is a list of declarations corresponding with $d_1...d_n$. The subterm es is a list which has been constructed as a result of parsing the expressions $e_1;...;e_n$.

2.3 Informal Description of Tiger Constructs

2.3.1 Definitions and Declarations

Tiger programs have a (possible empty) declaration section which defines types, functions and declares variables.

Type Definitions

Complex data structures can be defined by combining primitive data types (*integer and string*) and user defined types into array or record data types. A type definition of the form `type` t `=` ts introduces a new type with name t.

Array An array type definition of is constructed by `array of` ta, where ta is the type of the array elements. The size of the array is not part of its definition. Examples of array type definitions are:

```
type intArray     =  array of int
type stringArray  =  array of string
```

Record A record type definition is constructed by `type` t `=` $\{fe_1, ..., fe_n\}$, where fe_i denote the element fields of the record type t. The elements of a record are defined with the syntax $x : tid$, where x stands for the identifier of the record element and tid is the type name of the element. An example of a record type definition is:

```
type personTy = {name: string, age : int}
```

The following record types `stringList` and `indexedStrList` are defined as recursive data types:

```
type stringList      =  {head : string, tail : stringList}
type indexedString   =  {index : int, value : string}
type indexedStrList  =  {head : indexedString, tail : indexedStrList}
```

Variable Declarations

A variable declaration `var` x (`:` ta)? (`:=` e)? introduces a variable x with optional explicit type ta and initial expression e. When the type of a variable is not given, the compiler will try to infer the type. If a variable is initialised with the value `nil` the type part is required.

Array variable declarations require the type, size and the initial value for the elements of the array to be explicitly specified. A record variable declaration requires the type of the record and the initialisation of each one of its elements. Uninitialised record variables can be declared using `nil`, although in this case it is required to provide the type of the record. A `nil` expression denotes a value *nil* and a run-time check can be carried out to prevent selecting fields from a record to which a *nil* value was assigned. We show some further example declarations:

```
var salary :  int       := 2300
var phone               := 2585
var numbers             := intArray[50] of 5
var client              := personTy{name:= "rob", age:= 23}
var owner :  personTy   := nil
```

Function Definitions

The definitions of functions and procedures share the syntactic construct: `function` f`(`$arg_1, ..., arg_n$`)` (`:` ta)? `=` e. Such definitions require an identifier f introducing the function name. The (possibly empty) list arg_1, ..., arg_n denotes the formal parameters of a function. A formal parameter is defined by the construct x : ta where x is the name of the parameter and ta its type. A function definition differs from a procedure definition by the presence of the result specification part. Examples of function and procedure definitions are:

```
function increment(x: int): int = x + 1
function printMessage(s: string) = print(s)
```

Functions and procedures can be (mutually) recursive. Mutually recursive functions must be defined in the same sequence of function definitions without intervening definitions of types or variable declarations.

2.3.2 Expressions

As in most imperative programming languages, an expression denotes a computation of a value and/or a side-effect. That is, there is no difference between expressions computing a value or statements having side-effects.

Integers and Strings The denotable primitive values are integers and strings. Integers are denoted by a sequence of decimal digits. Strings are denoted as sequences of characters enclosed by quotation marks, including the empty sequence.

Lvalues Before we can define *lvalues*, we need to define memory location. Memory is seen as a collection of locations which can hold values. An lvalue represents a reference to a memory location(s) that can be read from or written to. Instances of lvalues are variables, array elements, record fields and function arguments. An identifier refers to a variable or parameter, e.g., `x`. An expression enclosed in square brackets defines the selection of an element of an array, e.g., `x[i]`. Array indices enumerating the elements of an array start at zero and end at the size of the array minus one. Selection of a record field uses a dot to select the element of a record, e.g., `x.f`. Arrays and records are references to actual memory locations.

Arithmetic, Relational and Boolean operators Expressions can be combined using operators. Arithmetic operators are `+,-,*,/`, denoting addition, subtraction, multiplication and division respectively. A negation of an expression is represented with a prefix `-` sign. Relational operators are `=, <>, <=, <, >=, >`, denoting equality test, inequality, less or equal, less than, at least and greater. The result of a relational operation is a Boolean value. Boolean operators are `&` and `|`, defining the short-circuited Boolean conjunctions and disjunctions. Booleans are represented by integer numbers, where the number zero represents *false* and any other integer value represents *true*.

Assignment An assignment is an expression that modifies the value associated with an lvalue. The construct *lv* `:=` *e* denotes an assignment of the value of the expression *e* to the memory location represented by *lv*. Assignments of arrays or records are shallow, they do not involve a copy of the data structure. Therefore, aliases between two arrays or record references may arise as the result of an assignment.

Sequences A sequence consists of a list of expressions separated by semicolons and enclosed in parenthesis. The expressions are evaluated in the order of occurrence, that is from the beginning towards the end of the sequence. A sequence may be empty. If the last expression of a sequence determines a value, this is the value determined by the whole sequence. Consider for instance the code fragment:

```
x := (print ("value of x:  "); x := x + 1; x) * 82
```

The execution of this assignment updates the variable `x`. The execution of the sequence on the right-hand-side entails the following side-effects: it prints the message `"value of x:  "`, it increments the value of the variable `x` and this new value is multiplied by `82`. Finally, the variable `x` is updated with the result of this multiplication.

Let A `let` binding is an expression which defines types and functions and declares variables. This expression form introduces a lexical scope for variables, functions and types. A scope is an enclosing context which determines the contained entities (variables, functions, types, etc.) and how they can be used. The scope of the defined and/or declared constructs is restricted to the following declaration and to the expression in the body of the `let`. Consider the code fragment:

```
let
    type record := {element : int}
    var x : int := 4
    var r := record{element = x + 34}
    function dec(y : int) : int = y - 1
 in
    e
end
```

This code fragment defines a type, variables and a function. The `record` type definition can be used in the next declaration and is visible in the body of the `let`. The variable `x` can be used in the following declarations, i.e., `r` and `dec`. Thus, the variable declaration of `r` can use `x`. The function argument `y` is only visible in the body of the function `dec`.

Alternatives The construct `if` c `then` e_1 `else` e_2 denotes two alternative execution paths. After evaluating the condition expression c, either e_1 or e_2 will be executed. This construct can also yield a value, in which case both expressions of the branches must yield a value of the same type. Examples of `if` expressions are:

```
if c then 45 else 23
if c then print("hello world") else print_int(100)
```

The first expression yields an integer value, and the last expression acts as an statement. An expression `if` c `then` e is equivalent to the expression `if` c `then` e `else ()`, which has an empty sequence on the else branch.

Loops Iteration is expressed with `while` and `for` expressions. The construct `while` c `do` e requires two expressions: a conditional expression c that (amongst others) yields a Boolean value controlling the iteration, and an expression e which is repeatedly evaluated. When the evaluation of the condition is non-zero the repetitive expression is evaluated and subsequently the entire expression is re-evaluated.

```
while y > 0 do (print("value of y greater than 0 "); y := y - 1)
```

Expressions of the form `for` x `:=` e_1 `to` e_2 `do` e_3 are `for` expressions. The identifier x introduces the iteration variable. The expressions e_1 and e_2 specify the iteration bounds and are evaluated at the start of the evaluation of the `for` expression. The expression under repetition is e_3. Tiger defines a fresh variable for the iteration variable of a `for` expression. This variable is implicitly defined and valid only in the scope of the expression e_3 and it shadows outer definitions of the same identifier,

```
let
   var x :int := 100
in
   for x := 1 to 10 do print("value of x greater than 0 ");
   printint(x)
end
```

The first declaration of `x` is shadowed by the definition of the iteration variable of the `for` expression. The last expression prints the value 100 corresponding to the first declaration of `x`.

Break A `break` terminates the execution of the directly enclosing `while` or `for` loop. Breaks are not allowed outside the bodies of `for` and `while` statements.

Function Call A function invocation is specified with the construct: f`(`$e_1, .., e_n$`)`. The identifier f denotes the name of the invoked function and $e_1, .., e_n$ denote a (possible empty) list of expressions representing the actual parameters of the call. Formal parameters are bound to actual parameters and they follow static scoping rules. Function arguments are evaluated from left to right. Arguments of a function are passed by value.

2.4 Well-formedness of Tiger Programs

Verifying whether a program is well-formed involves more than syntactic correctness. Typically, semantic properties of programs should be correct as well.

Static semantic analysis collects information about the program, and is used to verify whether a program is well-formed with respect to the scope rules, type rules, etc.

Scoping Rules

Variable declarations, functions and types definitions for the same identifier shadow any previous definitions in the enclosing scope. Consider the example:

```
let
   var x : int := 10
   function print_value(x : int) :=
     let
         var y : string
      in (y := int_to_string(x);
          print(y))
     end
  in
     for x := 1 to 10 do
       print_value(x * x)
end
```

This code fragment shows three occurrences of `x`: as a variable, as a formal parameter of the function `print_value`, and as the iteration variable of the `for` loop. These three occurrences are part of different scopes and the last occurrence shadows the previous one. Thus the occurrence of `x` as a variable is shadowed by the occurrence of `x` in the function definition. The occurrence of `x` in the `for` expression shadows the variable `x`.

Name Spaces

Tiger distinguishes two different name spaces: one for variables and functions and one for type definitions.

Parameter Passing

Arguments of a function call have two different mechanisms for passing arguments: primitive types are passed by value and record and array types are passed by reference, i.e., their lvalue (reference) is passed by value.

Type Equivalence

Tiger uses *name equivalence*, meaning that two types are the same if they stem from the same type definition.

Memory Objects Lifetime

In Tiger lexical scope is related to the lifetime of a variable, at least for primitive data types. For complex data types (arrays and records) the lifetime of a variable may be determined by the execution of the whole program [5].

2.5 Examples of Tiger Transformations

In the following, we present some source-to-source transformations using Tiger. We introduce simple transformations that show the different aspects which are involved in a transformation: the syntax of the language, the intention of the program and transformations that change the structure of the program.

The transformation examples are code rephrasing, tail recursion elimination and strength reduction.

For-to-While Conversion

As a simple example of a source-to-source transformation, we look at a rephrasing transformation. The transformation replaces a `for` expression by an equivalent `while` expression. This transformation is commonly applied during compilation in order to reduce the number of different language constructs which has to be taken care of in further processing. An instance of this transformation is:

```
let
  var x : int := 100
 in
   for y := 1 to (x * 2) + 5 do
     print(x + y)
end
```

⇒

```
let
   var x : int := 100
 in
   let
     var y  : int := 1
     var ul : int := (x * 2) + 5
    in while y <= ul do
        ( print(x + y);
          y := y + 1 )
   end
end
```

This example shows two versions of a program which perform the same computation. The code in the left program uses a `for` expression (remember that in Tiger it is not necessary to declare the iteration variable and the scope of this variable is only the `for` expression). The code shown in the right performs the computation using a `while` expression. To accomplish that the second code fragment behaves in the same way as the code in the left, a new `let` expression was introduced to declare the variable `y` and the new fresh variable `ul` which is only valid in the `while` expression. The variable `y` is initialised with the initial iteration value. The variable `ul` contains the value of the upper limit of the iteration range. The iteration condition is constructed with the iteration variable `y` and the variable `ul`.

Tail Recursion Elimination

Tail recursion elimination [11] is a source-to-source optimisation that replaces a recursive implementation of a function by an equivalent iterative implementation. The advantage of this transformation is that it reduces the number of stack allocation operations and memory usage. As a result, the obtained implementation is likely to be more efficient. Consider the example:

```
let
 function fact(n:int, acc:int): int
     = if n < 1
       then acc
       else fact(n - 1, n * acc)
 in
   print(fact(10, 1))
end
```

⇒

```
let
 function fact(n:int, acc:int): int
   = let var a_0 := acc
      in (while not(n < 1) do
          (n := n - 1;
           a_0 := n * a_0);
         a_0)
     end
 in
    print(fact(10, 1))
end
```

This transformation rephrases the recursive implementation of the function `fac`. Tail recursion elimination requires three steps: (1) find out whether a function is tail recursive, i.e., check whether the last expression of its body is a call to itself (2) update some arguments of the call reflecting the state of the computation. A recursive implementation must advance towards a basic case or stopping state. The state of the computation is reflected by a change of at least one argument of the tail call. And (3) introduce a new fresh variable that contains the result of the computation and wrap these statements with a loop. The condition of the loop is the negation of the stopping condition of the recursive version.

Strength Reduction

Strength Reduction [25, 28] is a program transformation that replaces expressions by equivalent expressions containing computationally cheaper operators. This optimisation pays off when it is applied to a function or an expression which systematically uses previous computed values of such an expression.

The replacement takes three steps: finding candidate expressions, including temporaries that contain the values of the rephrased expressions, and inserting references to the variables in the proper places. Candidate expressions include induction variables, which are variables that are updated by a constant value in a loop.

As an illustrative example consider:

```
let
   var x := 0
   var s
 in
    for i := 1 to 10 do
      ( s := 3 * i;
         print(s) )
end
```

⇒

```
let
    var x := 0
    var s
    var a_0 := 0
 in
    for i := 1 to 10 do
     ( a_0 := a_0 + 3;
       s := a_0;
       print(s) )
end
```

This transformation modifies the computation of the values of the variable `s`. In the left code fragment the value is computed by performing a multiplication each time the loop body is executed. The result of applying strength reduction to this code fragment results in the code fragment on the right. A difference is that the values of `s` are computed using an addition and using a previously computed value. This transformation assumes that addition is cheaper than multiplication.

Part II

Strategies for Program Transformations

3 Realising Program Transformations

The goal of program transformation is to obtain a new program from an old one while improving some observable aspect. For its realisation a program transformation system may build an infrastructure specialised for the programming language at hand. This chapter discusses the tooling to support these activities. In particular it describes the Stratego/XT framework for the specification of the generic transformation components that comprise a program transformation system.

3.1 Introduction

The main objective of a program transformation system is the construction of a new program from an old one. In order to achieve this goal, we have to build the infrastructure that allows us to manipulate programs as data. A program transformation system has somehow to deal with most of the aspects of a programming language such as its syntax, semantics and has to discover facts[1] about the input program in order to safely transform it. Each one of these aspects has to be dealt with by the transformation system. Typically, a program transformation system consists of several software components such as a parser, (possibly) a type checker, a semantic analyser, transformation components and a pretty-printer. This chapter describes

[1] Program facts can be static information, invariants, constant values, etc

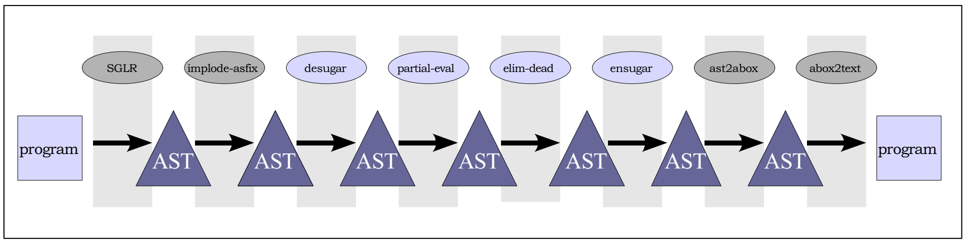

Figure 3.1: Stratego based program transformation pipeline.

these components and in particular the Stratego/XT framework for the specification of transformation components that comprise a program transformation system. The structure of the overall system is described in Section 3.2. Section 3.3 discusses the representation of programs by means of abstract syntax trees. In Section 3.4 we describe tools that are used to build the infrastructure of a program transformation system. The tools are automatically generated by the Stratego/XT framework.

3.2 Structure of a Stratego Program Transformation System

The Stratego/XT framework includes Stratego [88], a domain-specific language for the specification of program transformations. Moreover, the Stratego/XT framework includes a set of supporting tools that ease the development of program transformation systems. Transformation systems constructed with the Stratego/XT framework are organised as pipeline of transformation components.

Figure 3.1 shows a typical transformation pipeline corresponding to a partial evaluator. A partial evaluator may be composed of the following tools: a parser, desugarer, partial-evaluator, dead-code-eliminator, ensugarer and pretty-printer. A program is parsed resulting in an *abstract syntax tree*, i.e., a structural representation of the input program. The abstract syntax tree is simplified by removing the syntactic sugar of the language by a desugaring transformation pass. The obtained tree is the input for the partial-evaluator which generates specialised code for the arguments that are statically known. The result of this transformation pass may contain dead code, which is removed from the program by a separate transformation component. To map the resulting program back to a textual representation, the term is ensugared including layout information and syntactic sugar. Finally, the resulting term is pretty-printed. A pretty-printer generates a textual representation from a structural one.

The transformation system described above implements a complete source-to-source transformation, while individual transformation passes implement a single aspect. Each single transformation pass is implemented in a separate and independent manner.

For transformation systems that are not source-to-source, the data-flow pipeline model still applies, but they do not require a pretty-printer for the output target language, and it will be replaced by a tool that generates machine code, links the code, and thus generates executable programs.

The construction of complete transformation systems requires the composition of transformation tools. The transformation tool composition (XTC) [16, 47] supports the transparent construction of such systems. This tool composition framework is programmable and complete transformation systems can be composed with it.

3.3 Representing Programs as Terms

A program transformation system manipulates programs. Thus, an adequate program representation is crucial for transforming programs. The context-free nature of programming language descriptions induces tree structures for programs [1, 5, 44]. Terms are isomorphic to *trees* and they represent the object program to be manipulated. Terms are values which can be described by a signature.[2]

A *first-order term* is essentially a constructor c applied to a (possibly empty) list of first-order terms $t_1, ..., t_n$, as defined by the following grammar:

$$\begin{array}{rll} t ::= & c(t_1, ..., t_n) & n\text{-ary constructor application } n \geq 0 \\ c ::= & \text{identifier} \mid \mathit{str} \mid i & \text{constructors} \end{array}$$

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

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

To illustrate how programs correspond to terms, consider the constructors assigned to productions in the grammar for Tiger in Figure 2.1. Examples of terms over the Tiger grammar are `Var("x")` which represents the variable `x`; `Call(Var("f"), [Var("x")])`, which represents `f(x)`, the call of function `f` with the argument `x`; and `Let([VarDec("x", NoTp,Int("1"))],[Var("x")])` represents `let var x := 1 in x end` declaring the local variable `x` initialised with the integer constant `1`.

3.4 Tooling based on the Syntax Definition Formalism

The generation of a parser, pretty-printer and a signature is done by a family of tools based on the Syntax Definition Formalism (SDF from now on). To build the infrastructure, the tools can be replaced by similar tools that realise the same tasks. Since these tools are all part of the Stratego/XT framework the interaction with Stratego does not require additional work.

[2] Stratego signatures describe the term representation, arity of the terms, constructors, etc.

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

Figure 3.2: The Annotated Term (ATerm) Format.

```
module Tiger-Statements
signature

 constructors
    Assign : Var * Exp -> Exp
    If     : Exp * Exp * Exp -> Exp
    For    : Exp * Exp * Exp * Exp -> Exp
    While  : Exp * Exp -> Exp
    Call   : Var * List(Exp) -> Exp
    Plus   : Exp * Exp -> Exp
    Minus  : Exp * Exp -> Exp
    Var    : Id -> Exp
    Int    : IntConst -> Exp
```

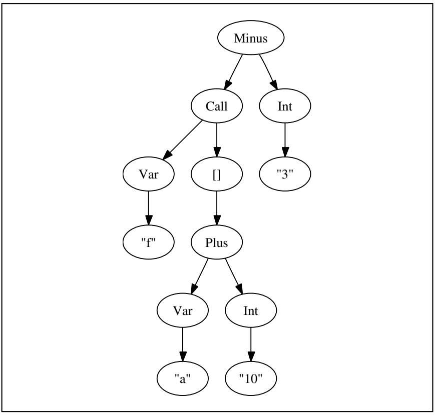

Figure 3.3: (Left) Tiger Signature and (Right) Graphical representation of an abstract syntax tree.

We have used this formalism due to its simplicity and elegance. The SDF formalism is based on a context-free grammar specified in EBNF notation. From a grammar description, term signatures, a parser and a pretty-printer are automatically generated.

3.4.1 Signatures

The abstract syntax of a programming language or a data format can be described by means of an *algebraic signature*. A signature declares for each constructor its arity m, the sorts[3] of its arguments S_1*...*S_m, and the sort of the resulting term S_0 by means of a constructor declaration $c : S_1$*...*$S_m \rightarrow S_0$. A term can be validated against a signature by a *format checker*.

Signatures are automatically derived from a given syntax definition. For each production rule of the form $A_1...A_n \rightarrow A_0\{$`cons`$(c)\}$ in a syntax definition, its corresponding constructor declaration is $c : S_1$*...*$S_m \rightarrow S_0$, where the S_i are the sorts corresponding to the symbols A_i after leaving out literals and layout sorts. Thus, the signature in Figure 3.3 describes the abstract syntax trees derived from

[3] Sorts are names for elements of similar syntactic category.

```
module Tiger-Statements
imports Tiger-Lexical
exports
  lexical syntax
    [a-zA-Z][a-zA-Z0-9]*                -> Id
    [0-9]+                              -> IntConst
  context-free syntax
    Var ":=" Exp                        -> Exp {cons("Assign")}
    "if" Exp "then" Exp "else" Exp      -> Exp {cons("If")}
    "for" Exp ":=" Exp "to" Exp "do" Exp -> Exp {cons("For")}
    "while" Exp "do" Exp                -> Exp {cons("While")}
    Var "(" {Exp ","}* ")"              -> Exp {cons("Call")}
    Exp "+"  Exp                        -> Exp {left,cons("Plus")}
    Exp "-"  Exp                        -> Exp {left,cons("Minus")}

    Id                                  -> Exp {cons("Var")}
    IntConst                            -> Exp {cons("Int")}
```

Figure 3.4: Syntax definition for the Tiger language (incomplete).

parse trees corresponding to the abstract syntax of Tiger in Figure 2.1. In the right side of Figure 3.3 we have given the graphical representation for the expression `f(a + 10) - 3`. Its corresponding textual representation is `Minus(Call(Var("f"), [Plus(Var("a"), Int("10"))]),Int("3"))`.

In Stratego, terms are described by a signature that defines the constructors used to represent the structure of the object language. In Figure 3.4 we give a subset of the syntax definition for Tiger. In the appendix B the complete syntax definition for Tiger is given.

The lexical and context-free syntax of a language are described using context-free productions of the form $S_1 ... S_n$ `->` S_0 stating that the concatenation of phrases of sort S_1 to S_n forms a phrase of sort S_0. Since SDF is modular it is easy to extend a language or to complete a definition to cover the whole language.

An example of a simple Tiger program and its corresponding abstract syntax tree is shown in Figure 3.5. The abstract syntax tree is constructed using the constructors introduced in Figure 3.4. Figure 3.5 shows two different formats for the resulting parsed term: in right top we have the textual representation and in the bottom right the graphical tree format.

3.4.2 Parsing

A programming language description starts with a context-free grammar. Compilers, code manipulator tools and program transformation systems use parsers to extract the structure of a program. A parser transforms a program written in a textual representation into a data structure that can be processed for code transformation, code generation, verification of program properties, etc. Although not

```
let
   var x := 0
  in x := x + 1
end
```

⇒

```
Let(
  [VarDec("x", NoTp, Int("0"))]
, [Assign(Var("x"), Plus(Var("x"), Int("1")))]
)
```

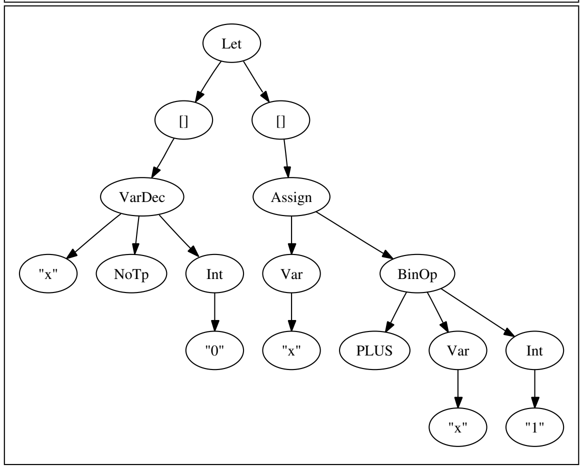

Figure 3.5: Example of a parsed Tiger program.

all programming environments use a textual interface, it is common that graphical environments store programs in a textual format too. Thus, transformation systems usually require a parser to obtain the structural representation of programs.

A *parse tree* is the result of applying a parser to the input program. This parse tree typically contains irrelevant information such as layout information or literal symbols that are no longer needed. An *abstract syntax tree* can be constructed from a parse tree by eliminating such redundant information. Furthermore, constructors can be used to represent the structural syntax of the input program.

3.4.3 Pretty-printing

After transformation, an abstract syntax tree may have to be converted back to a textual representation conforming to the syntax rules of the object language. Mapping a tree onto text is the inverse of parsing, sometimes called *unparsing* or *pretty-printing*. The outcome of a pretty-printer is a program that obeys the syntactic and semantic properties of the target programming language. GPP[31] is a generic pretty-printer tool included in the Stratego/XT framework. This generic pretty-printer uses the Box language to represent a generic intermediate formatted text output. There are different back-ends for Box that generate different output formats. Available back-ends for Box are plain text, HTML, and LaTeX.

A pretty-printer can be automatically generated from a syntax definition. Further customisation to the resulting pretty-printer is also possible.

3.5 Summary

In this chapter we have focused on the description of a complete set of tools that can be used to construct a program transformation system. Common tools for different transformation systems are parsers, desugarers, ensugarers and pretty-printers for the manipulation of object languages in Stratego. The Stratego/XT framework provides support for the easy construction of the required tools. It also provides the XTC composition tool to describe how a system is composed by a sequence of transformation components that comprise a transformation system. Now that we have described the main ingredients, we will focus in the next chapter on describing Stratego itself and how to describe actual program transformations. Some basic transformation examples will be presented.

4 Strategic Rewriting

Program transformation systems are implemented as a sequence of consecutive program modifications. Such modifications can be represented by a collection of rewrite rules. Each rule defines a single transformation step. Term rewriting systems are required to guarantee that a set of rules induces a rewriting relation which is confluent and terminating. Extensions to term rewriting systems introduce several mechanisms to control the rewriting process. In this chapter, we focus on describing strategic rewriting by specifying strategies using Stratego. The Stratego language constructs are described and example program transformations are presented to illustrate the strategic rewriting model.

4.1 Introduction

Program transformation systems are implemented as a sequence of consecutive program modifications. Modifications to a program are represented in a concise way by *rewrite rules*. Each rewrite rule defines a single transformation step.

Term rewriting [89] is an attractive formalism for expressing basic program transformations. It is based on terms and rewrite rules for rewriting these terms. The main objective of a term rewriting system is to reduce terms to a *normal form*. Extensions to term rewriting systems introduce several mechanisms to control the rewriting process. The range of additional mechanisms includes: goal driven engines, fixed application strategies such as outermost and innermost, a menu of strategies to be interactively applied, and programmable strategy languages for the specification of strategies [13, 12, 67].

We focus on describing programmable strategies included in Stratego to describe and control the rewriting process. A strategic rewriting system consists of a set of rewriting rules and a programmable way to specify and control the application of these rules, thus solving the problem of control over the application of rules while maintaining the separation of rules and strategies. With this separation, rules and strategies are kept clean and can be reused.

The rest of this chapter describes the foundations of strategic rewriting and introduces the Stratego language [88, 103]. Moreover, the main purpose of this chapter is to introduce the terminology and formalisms used in the rest of this dissertation. Section 4.2 discusses rewrite rules. Some basic concepts of term rewriting are covered in Section 4.3. Section 4.4 mainly presents the Stratego constructs and provides some examples. Section 4.5 describes traversals which are used to visit subterms of a tree in a particular fashion. Some examples of program transformation components are presented in Section 4.6.

4.2 Rewrite Rules

Before defining rewrite rules let us define a pattern. A *pattern* is a term in which every free variable x occurs in a subterm of the form $c(s_1, ..., s_n)$ where $s_1, ...s_n$ are different bound variables. A *rewrite rule* has the form R`:` p_1 `->` p_2 `(where` c`)?`. The application of a rewrite rule R replaces a term t with a term t'. The application matches the pattern p_1 with a term t if there is a substitution σ mapping all variables in p_1 to terms such that $\sigma(p_1) \equiv t$. Moreover, the condition c must succeed under σ producing an extension σ'. The result of applying the rule is $\sigma'(p_2) \equiv t'$.

An example of a conditional rewrite rule is:

```
EvalPlus :  Plus(Int(i), Int(j)) -> Int(k) where <addS> (i,j) => k
```

The `EvalPlus` rule matches a term expression which has two integer values as subterms and the condition evaluates and instantiates the value of `k`. The occurrences `i` and `j` are pattern variables and they will be instantiated in the application of the rule defining the substitution σ for a term t. The construct `<addS>` adds two integer values and the result is bound against `k`. This substitution is included in σ resulting in σ' which is used to construct the result.

Unconditional rewrite rules do not have a condition clause. An example of an unconditional rewrite rule is:

```
AddZero :  Plus(e, Int("0")) -> e
```

This rule matches a term that represents the addition of zero to some expression `e`. The expression `e` is a variable pattern that will be instantiated when the rule is applied. The rule is stated by the rule name `AddZero`. Note that pattern variables are typeset in italics.

4.2.1 Rewrite Rule Application

A rewrite rule is invoked using the name of the rule. As an example consider the following rule application:

$$\texttt{((y + 0) * (6 + 3)) + 0} \xrightarrow{\texttt{AddZero}} \texttt{((y + 0) * (6 + 3))}$$

where the rule `AddZero` is applied to the expression `((y + 0) * (6 + 3)) + 0` and as a result a new term is obtained. The substitution associates the pattern variable *e* with the subexpression `((y + 0) * (6 + 3))`. As a result of the rule application, the addition with zero is removed and `((y + 0) * (6 + 3))` is returned. The obtained term can be evaluated further by applying `EvalPlus` to subterms:

$$\texttt{(y + 0) * (6 + 3)} \xrightarrow{\texttt{EvalPlus}} \texttt{((y + 0) * (9))}$$

The application of the rule `EvalPlus` computes the two integer values and produces a new term with the result of their addition. The `EvalPlus` rule associates the pattern variables *i* and *j* with the values `6` and `3` respectively. The rule produces the a new pattern variable *k* that is bound by the condition part and the right-hand side of the rule.

In the examples the selection of the subterm to which the rewrite step is applied was left implicit. It is the task of our strategy program to make this explicit. In Stratego, rules are applied while visiting the term by a traversal. Traversals are discussed in Section 4.5.

4.2.2 Rewrite Rules with Concrete Syntax

Rewrite rules can also be written using *concrete syntax*, which allows us to use code fragments of the object language to denote Stratego terms [104]. Such code fragments can be used in rewrite rules and strategies. In order to make a clear distinction between the meta and the object languages, Tiger code fragments will be denoted between the delimiter symbols ⟦ and ⟧. Using concrete syntax for the rules previously defined with abstract syntax we get:

```
AddZero   :   ⟦ e + 0 ⟧ -> ⟦ e ⟧
EvalPlus  :   ⟦ i + j ⟧ -> ⟦ k ⟧ where <addS> (i,j) => k
```

These rules have exactly the same semantics as their counterparts expressed with abstract syntax. The only difference is their presentation and the guaranteed syntactic correctness of the code fragments. The use of concrete syntax allows us to

`[x,y,z][0-9\']*`	`->`	`Id`	tiger variable
`[f,g][0-9\']*`	`->`	`Id`	tiger function name
`[i,j,k][0-9\']*`	`->`	`IntConst`	tiger integer
`[e][0-9\']`	`->`	`Exp`	tiger expression
`[e][0-9\']"*"`	`->`	`{Exp";"}*`	list of tiger expressions
`[d][0-9\']`	`->`	`Dec`	tiger declaration
`[d][0-9\']"*"`	`->`	`Dec*`	list of tiger declarations
`[x][0-9\']"*"`	`->`	`{FArg","}*`	list of tiger formal parameters
`[a][0-9\']"*"`	`->`	`{Exp","}*`	list of tiger function arguments

Figure 4.1: Syntax definition of Tiger meta-variables.

write transformations that are clearer and concise.

The symbols |[and]| are referred to as *quotation* marks and delimit the fragments of the embedded object language. *Antiquotation* constitutes an escape to Stratego terms from within a quoted fragment. For Tiger the symbol ~ precedes such terms.[1]

A concise form of antiquotation is provided by *meta-variables*: variables that denote terms of some syntactic category. Such categories are defined by rules that relate the syntactic abstraction with their respective term sort. Figure 4.1 contains a list of the meta-variables most frequently used in this dissertation. The meta-variables are expressed as SDF rules for syntax definition of Tiger. In short they define parsing rules for the required syntactic categories, which are useful in defining concrete syntax rules. For a complete definition for Tiger meta-variable constructs available, we refer the reader to Appendix B.2.1.

The rule `EvalPlus` uses the meta-variables *i*, *j* and *k* to denote integer numbers. The occurrence *e* in the rule `EvalPlus` denotes an expression `Exp`. Other examples of uses of meta-variables are illustrated in the rule:

```
UnFold :  |[ let var x := e1 d* in e2 end ]| -> |[ let d* in e2 end ]|
  where <not-used> (x,[e2,d*])
```

The purpose of this rule is to eliminate variable declarations that are not used in a `let` expression. The meta-variable *d** denotes a list of declarations, the * is part of the name of the meta-variable and indicates that it refers to a list of declarations. The meta-variable *x* refers to a variable in the Tiger language, and *e1*, *e2* will match expressions. The condition `<not-used>` verifies whether the variable pattern *x* is not used in the rest of the `let` expression.

[1] Stratego quotation and antiquotation symbols are not fixed and can be defined according to suit the needs of the object language syntax.

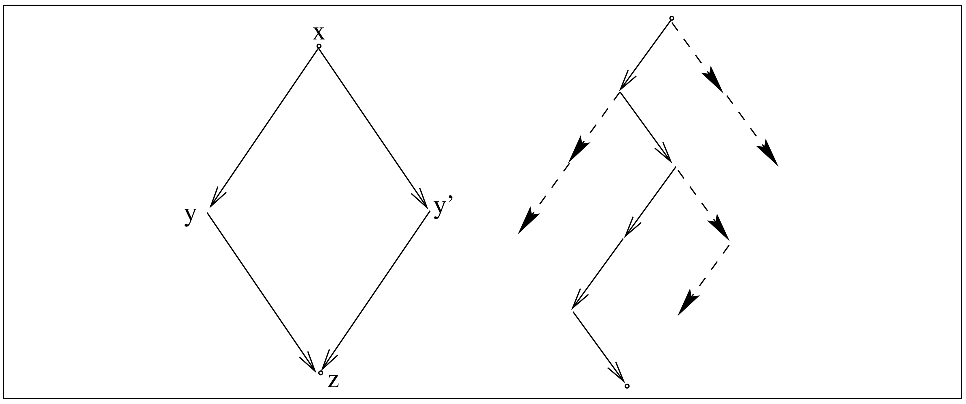

Figure 4.2: (Left) Diamond property of rewriting systems and (Right) Strategies for achieving termination and confluence.

4.3 Term Rewriting

A term is called a *reducible* expression (or *redex* for short) if it can be reduced by some rule. Rewriting entails normalisation, i.e., exhaustive application of the set of rules, until the term is in normal form. A term is in *normal form* if it contains no more redexes.

Basic properties of term rewriting system are termination and confluence [9]. A term rewriting system is **terminating** if any sequence of rule applications is finite. A term rewriting system is **confluent** if, for all x, y , and y' such that $x \to y$ and $x \to y'$, there exists a z such that $y \to z$ and $y' \to z$. Together these properties guarantee that non-deterministic selection of rules will result in a *unique* normal form of a term.

Figure 4.2 (*Left*) depicts the *diamond property*, which describes confluence in a rewriting system. A term rewriting system has the diamond property if and only if, whenever a term x reduces either to y or y', there is a term z, such that y and y' rewrite to z. The rewriting system depicted in Figure 4.2 (*Right*) suggests several reduction paths but only one path (solid lines) obtains a term in normal form. Thus, it diverges from a pure term rewriting system.

Programmable strategies give control to the developer to avoid non-termination and to specify which result is chosen in case of a non-confluent system. As long as we do not specify explicitly *when*, *where* and *which* rewrite rule should be applied, the outcome of the rewriting process is non-deterministically determined. In such situations it is common to require the rewriting system to be *confluent*, thus guaranteeing that the result is uniquely determined, and *terminating* so we can be sure that an answer will be produced, irrespective of the choices made. In strategic rewriting the situation is however completely different, since we specify the when, where and which by a strategy program, and hence no non-determinism is involved in our execution model, and the task to solve the aforementioned problems lies completely in the hands of the programmer.

4.4 Strategies

A **rewriting strategy**, or a strategy for short, is an algorithm describing how a sequence of transformations should be carried out. Strategies allow the developer to specify which computations are of interest and how to fine-tune them to accomplish a particular transformation goal. Stratego programs consist of both rules and strategy definitions which specify how selected rules are to be applied. A rewriting strategy transforms a term or fails in doing so. In the case of success, the result is a transformed term. In the case of failure, there is no resulting term.

Rewrite rules are the most common strategies. A rewrite rule applies a transformation to the root of a term. Strategies and rewrite rules can be combined into more complex strategies by means of strategy combinators.

The Stratego syntax is split into a core language, providing the fundamental constructs, and syntactic abstractions defined in terms of these constructs. The syntax of Stratego is presented in Figure 4.3, where s denotes a strategy and p a term pattern. Note that the syntax in this figure is not complete. For brevity, we are omitting generic term deconstruction [101], some traversal operators, and term annotations. We proceed by giving an overview and describing the semantics of the constructs of the Stratego language that define strategies. For clarity we provide a separate explanation of the strategy combinators according to their purpose.

Strategies without patterns are defined by the following constructs

$$s \ ::= \ \texttt{fail} \mid \texttt{id} \mid \texttt{not}(s) \mid \texttt{where}(s)$$

Stratego has two primitive strategies that do not depend on the current term: `fail` (which always *fails*) and `id` (the identity strategy that always *succeeds*). The negation strategy `not(`s`)` succeeds if the strategy s fails to the term that is applied. To test or verify conditional rewriting, the strategy `where(`s`)` verifies whether the strategy s succeeds but does not transform the term.

Basic strategies with patterns are defined by

$$s \ ::= \ \texttt{?}p \mid \texttt{!}p \mid \texttt{\{}x_1, ..., x_n\texttt{:} \ \ s\texttt{\}} \mid \texttt{<}s\texttt{>} \ p \mid s \ \texttt{=>} \ p$$

The basic strategies are *match* and *build*. A match strategy `?`p succeeds if the current term matches the specified pattern p. On the other hand, the strategy `!`p constructs a term t specified by p; all the variables in p must be bound for this strategy to succeed. In Section 4.2 we described rewrite rules as operations that first match their left-hand side pattern, then evaluate their condition, and finally

P	$::=$	$d_1 ... d_n$	program (list of definitions)
d	$::=$	$dsig$ `=` s	strategy definition
	$\mid$	$dsig$ `:` p_1 `->` p_2 (`where` s)?	rule definition (optional condition)
$dsig$	$::=$	f`(`$sd_1, ..., sd_n$ `|` $vd_1, ..., vd_m$`)`	definition signature
	$\mid$	f`(`$sd_1, ..., sd_n$`)`	definition without term arguments
	$\mid$	f	definition without arguments
sd	$::=$	f(`:`tp)?	strategy argument (optional type)
vd	$::=$	(x`:`tp)?	term argument (optional type)
p	$::=$	$str \mid i \mid r$	string, integer, real constant
	$\mid$	x	term variable
	$\mid$	c`(`$p_1, ..., p_n$`)`	constructor application
	$\mid$	`(`$p_1, ..., p_n$`)`	tuple
	$\mid$	`[`$p_1, ..., p_n$`|`p`]`	list of length $\geq$ n
	$\mid$	`[`$p_1, ..., p_n$`]`	fixed length list
s	$::=$	`fail`	failure
	$\mid$	`id`	identity
	$\mid$	`not(`s`)`	negative test
	$\mid$	`where(`s`)`	test
	$\mid$	`?`p	match
	$\mid$	`!`p	build
	$\mid$	`{`$x_1, ..., x_n$`:`s`}`	term variable scope
	$\mid$	`<`s`>`p	apply to pattern
	$\mid$	s `=>` p	apply and match against pattern
	$\mid$	s_1 `;` s_2	sequential composition
	$\mid$	f`(`$s_1, ..., s_n$`|` $p_1, ..., p_m$`)`	call
	$\mid$	f`(`$s_1, ..., s_n$`)`	call (only strategy arguments)
	$\mid$	f	call (no arguments)
	$\mid$	s_1 `<+` s_2	deterministic choice
	$\mid$	s_1 `<` s_2 `+` s_3	guarded deterministic choice
	$\mid$	`if` s_1 `then` s_2 `else` s_3 `end`	conditional choice
	$\mid$	`switch` s $cs_1 ... cs_n$ `otherwise` s_m `end`	n-ary branching construct
	$\mid$	`let` $d_1, ..., d_n$ `in` s `end`	local definitions
	$\mid$	c`(`$s_1, ..., s_n$`)`	congruence traversal
	$\mid$	tr`(`s`)`	traversal to subterms
	$\mid$	`rec` f`(`s`)`	recursive closure
	$\mid$	`{`s`}`	local scope for all free variables in s
cs	$::=$	`case` s_i `:` s_j	selective case operator
tr	$::=$	`all` $\mid$ `one` $\mid$ `some`	traversal operator
f	$::=$	identifier	strategy operator
x	$::=$	identifier	term variable
c	$::=$	identifier	constructor
tp	$::=$	...	type (omitted)

Figure 4.3: Syntax of (a subset of) Stratego.

instantiate the right-hand side pattern. Instead of taking rewrite rules as basic operations, in Stratego the operations that define rewrite rules are *first class*. That is, matching a term against a pattern and instantiating a pattern to build a new term are first class strategies. The strategy `?`p denotes matching against the pattern p, and `!`p denotes building a term specified by the pattern p. This decomposition allows many language constructs to be defined from first principles. For instance, a rewrite rule p_1 `->` p_2 corresponds to the sequential composition `?`p_1`;!`p_2. The sequential composition of strategies will be described below, but it boils down to first applying the first strategy and then the second in the context produced by the first one.

The *scope of pattern variables* can be controlled and restricted with the construction $\{x_1,...,x_n\text{: } s\}$ which introduces the *fresh* variables $x_1,...,x_n$ to be bound by the strategy s. Bound variables will result mainly from matching strategies. The binding to a variable x_i outside the scope $\{x_1,...,x_n\text{: } s\}$ is not visible inside it, nor is a binding to x_i visible outside its scope. Once a variable is bound, it cannot be rebound to a different term.

The *strategy application* to a term is described by the construction `<`s`>` p, which is equivalent to `?`p`;` s. It matches the term against the pattern p and then the strategy s is evaluated in the context produced by the matching.

A term can be constructed as a result of a successful strategy application. The strategy s `=>` p can be expressed as s`;` `?`p which applies the strategy s to the current term and matches and binds the resulting term with p.

Control strategies For the control of rewrites, Stratego has the following strategy combinators:

$$\begin{array}{lll} s & ::= & s_1 \texttt{ ; } s_2 \mid f\texttt{(}s_1,...,s_n\texttt{|}p_1,...,p_n\texttt{)} \mid f\texttt{(}s_1,...,s_n\texttt{)} \mid f \\ & \mid & s_1 \texttt{<+} s_2 \mid s_1 \texttt{ < } s_2 \texttt{ + } s_3 \mid \texttt{if } s_1 \texttt{ then } s_2 \texttt{ (else } s_3\texttt{)? end} \\ & \mid & \texttt{let } d_1 ... d_n \texttt{ in } s \texttt{ end} \end{array}$$

The *sequential composition* combinator `;` (*semicolon*) combines two strategies in a sequence. In order for $s_1;s_2$ to succeed, both strategies s_1 and s_2 have to succeed. This strategy combinator has precedence over the other combinators.

Strategy definitions such as f`(`$f_1,...,f_n$`|`$x_1,...,x_m$`)` `=` s define a new strategy f with body s and strategy parameters $f_1,..,f_n$ and term variables $x_1,...,x_m$. An application of the strategy f`(`$s_1,...,s_n$`|`$t_1,...,t_m$`)` entails applying the body s of f with the strategy parameters f_i bound to the strategy arguments s_i and the instantiated pattern parameters f_i to the term parameters x_i. The list of term arguments of a strategy combinator is optional and the symbol `|` can be left out if no term arguments are present. Similarly, if the list of strategy arguments and term arguments are empty the parentheses can be omitted. Thus we have the following equivalences for definitions and calls:

$$\begin{aligned} f(f_1,\ldots,f_n) = s &\equiv f(f_1,\ldots,f_n|) = s \\ f(s_1,\ldots,s_n) &\equiv f(s_1,\ldots,s_n|) \\ f = s &\equiv f(|) = s \\ f &\equiv f(|) \end{aligned}$$

Strategy examples are `succeed-always(`s`)= try(`s`)` and `elem-of(|`xs`) = elem(id,` xs`)`.

The semantics of the *guarded strategy* combinator s `<` s_1 `+` s_2 is defined as first applying the strategy s and then depending on its result, applying either s_1 or s_2. This strategy commits the result of the application of s and proceeds without local backtracking. The following equivalences illustrate the choice between the branches induced by success or failure of the guard strategy:

$$\texttt{id} < s_1 + s_2 \equiv s_1 \qquad \texttt{fail} < s_1 + s_2 \equiv s_2$$

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

The *deterministic* or *left choice* combinator `<+` is used to avoid non-determinism in the rewriting process. This strategy combinator enables us to define the preferred strategy application order. The strategy s `=` s_1 `<+` s_2 tries the strategy s_1 and only if it does not succeed attempts the strategy s_2. The strategy s succeeds if either strategy succeeds. As an example of use of this strategy, consider the following definition `try(`s`) =` s `<+ id` which attempts to apply the strategy s to a term. If the application of s succeeds, a new term is obtained, otherwise the original term is retained. The left choice combinator is a special case of the guarded choice combinator as expressed by the first of the following equations:

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

The equations assert the identity (id) is a left zero, but not a right unit or zero for left choice. A failure ($fail$) is a left and right unit for left choice. The inequality above indicates how left choice and identity can be used to turn a strategy that may fail into a strategy that always succeeds.

An example of the left choice combinator is the control over the application of rules that share the same name. Multiple rules with the same name may exist, possibly varying in their application conditions. The Stratego compiler applies a strategy to decide which rule is applied using the strategy:

```
apply-rule = R1 <+ R2 <+ R3 ....
```

All `R` rules are renamed by adding a suffix to be able to distinguish them. In this way it is possible to construct a strategy application that attempts to apply an `R` rule. The order of application of rules with the same name is decided by the compiler. If the developer wants to have control, he/she must provide different rule names and control the order of an application with an explict strategy.

The strategy `if` s `then` s_1 `else` s_2 `end` is syntactic sugar for the guarded strategy combinator s `<` s_1 `+` s_2.

The Stratego construction `let` $d_1 ... d_n$ `in` s `end` delimits the strategy definitions in $d_1 ... d_n$ to the strategy s.

The *recursive closure* `rec` f`(`s`)` is syntactic sugar for a local recursive definition, i.e., `rec` $f(s)$ is equivalent to `let` f `=` s `in` f `end`. The construct can be useful in strategy expressions and allow us to directly write rec x(try(s; x)) instead of `repeat(s) = try(s; repeat(s))`.

The `switch` is an n-ary branching construct. This construct is syntactic sugar for a nested `if-then-else` strategy combinator.

4.5 Traversals

So far the described strategy combinators rewrite a term at its root. To apply a rewrite to a subterm, it is necessary to traverse the term. Stratego defines several basic combinators which expose the direct subterms of a constructor application. These can be combined with the combinators described above to define a wide variety of complete term traversals. The basic strategy traversal combinators are:

$$s ::= \texttt{all}(s) \mid \texttt{one}(s) \mid \texttt{some}(s)$$

The combinator `all(`s`)` succeeds if for all the immediate subterms of the current term (i.e., children), s succeeds. The combinator `one(`s`)` succeeds if s succeeds for exactly one subterm of the current term. And finally, the combinator `some(`s`)` succeeds if at least for one subterm of the current term, s succeeds. Traversals are programmable by means of these basic strategy traversals and combinators. The following strategy definitions are examples of generic traversal strategies:

```
topdown(s)    =  s ; all(topdown(s))
bottomup(s)   =  all(bottomup(s)); s
oncebu(s)     =  one(oncebu(s)) <+ s
oncetd(s)     =  s <+ one(oncetd(s))
innermost(s)  =  bottomup(try(s; innermost(s)))
```

These strategy traversals are *generic* in two ways: (1) they can be applied to any tree, regardless of how a specific tree is represented and (2) they are parameterised with the transformation (s) to be applied to the subterms. The generic traversal strategy `topdown` first applies the strategy s to the root of the term and then recursively applies s to the subterms of the resulting term using the primitive strategy traversal `all`. Figure 4.4 depicts how a term is traversed in a topdown fashion. The solid lines represent the order in which the terms are traversed and the dotted lines represent how the term is reconstructed. The strategy s is applied during the descending traversal.

The generic strategy traversal `bottomup` traverses a tree by first applying itself recursively to all direct subterms of a node using `all` and then applying the parameter strategy s to the root of the term. The strategy s is applied during the ascending traversal. The strategy `oncebu` is a strategy that traverses a term in a bottomup fashion and succeeds as soon as it can apply a single rewrite s to a subterm. Figure 4.5 shows graphically how a term is visited with the strategies `bottomup` and `oncebu` respectively.

The strategy `innermost` is an example of a strategy traversal that applies the strategy parameter s exhaustively from inside out.

On the one hand, the strategies `topdown`, `bottomup` and `oncebu` are examples of simple traversals which visit a term only once while applying the strategy s. We will refer to simple traversals as *one pass traversals*. On the other hand, the `innermost` strategy defines an *exhaustive application* of the strategy s until every subterm of a term is in normal form.

Recall the rule application examples from Section 4.2.1, where the rules `AddZero` and `EvalPlus` were applied to the term `((y + 0) * (6 + 3)) + 0`. A possible application is to use the strategy `bottomup(try(AddZero <+ EvalPlus))` to obtain the expression `((y) * (9))`. A different result is obtained with the strategy `oncebu(try(AddZero <+ EvalPlus))`, which yields the expression `((y) * (6 + 3)) + 0`. With this strategy a single rewrite is performed. The outcome of the strategy `oncetd(try(AddZero <+ EvalPlus))` yields the expression `(y + 0) * (6 + 3)`.

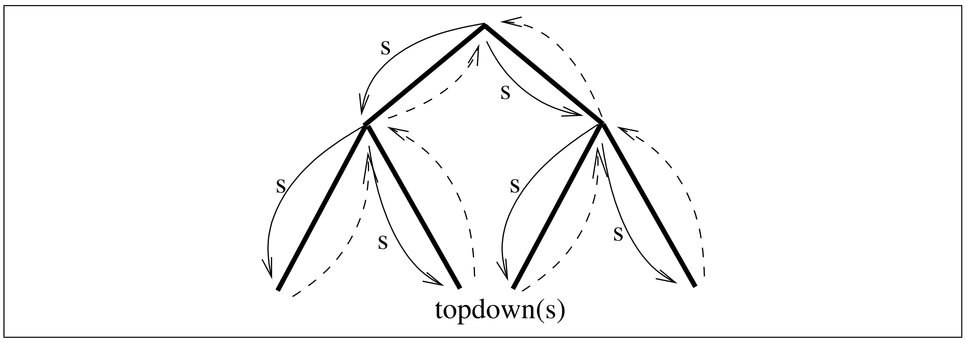

Figure 4.4: Generic strategy traversal topdown.

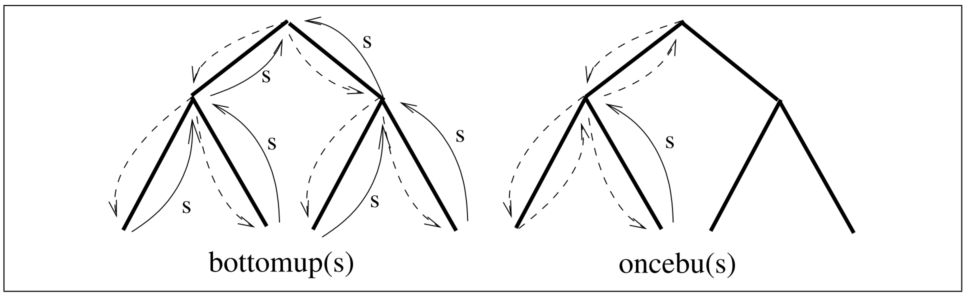

Figure 4.5: Generic strategy traversals bottomup and once-bottom-up.

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

```
c(s1,...,sn) : c(x1,...,xn) -> c(y1,...,yn) where <s1>x1 => y1;...;<sn>xn => yn
```

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

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

The `map` strategy applies a transformation to each element of a list, but fails when one of the applications fails. The `filter` strategy also applies the strategy parameter to every element of a list, but removes elements from the result for which the application fails.

4.5.1 Congruence Strategies with Concrete syntax

Stratego allows the definition of concrete object syntax for congruence strategies. We use concrete syntax for congruences over Tiger constructs using the notation `<`s`>` to embed a strategy within a Tiger expression. Thus, for example, the strategy expression $[\![$`if <`s_1`> then <`s_2`> else <`s_3`>`$]\!]$ corresponds to congruence `If(`s_1`,`s_2`,`s_3`)` over the `if-then-else` construct and applies strategy s_1 to the condition and the strategies s_2 and s_3 to the `then` and `else` branches, respectively. We use the notation `<*`s`>` to denote an strategy application to a list of terms. For example, Tiger's `let` construct has a list of declarations and a list of expressions as

`<` s `>`	`->`	`Exp`	congruence for expressions
`<*` s `>`	`->`	`Exp+`	congruence for list of expressions
`<typedecs:` s `>`	`->`	`TypeDec+`	congruence for type declarations
`<fd:` s `>`	`->`	`FunDec`	congruence for function definition
`<fd*:` s `>`	`->`	`FunDec+`	congruence for list of function definitions
`<bo:` s `>`	`->`	`BinOp`	congruence for a binary expression
`<ro:` s `>`	`->`	`RelOp`	congruence for a relational expression

Figure 4.6: Syntax definition for Tiger congruence strategies.

direct subterms. Thus, we use |[`let <*`s_1`> in <*`s_2`> end`]| to denote `Let(`s_1`,`s_2`)`, i.e., the application of the strategies s_1 and s_2 to the lists of declarations and expressions, respectively. Compare this to |[`let <`s_1`> in <`s_2`> end`]|, which is equivalent to `Let([`s_1`],[`s_2`])`. Figure 4.6 depicts the notation for congruence strategies which will be used in the examples. For a complete definition for Tiger congruences see Appendix B.2.1.

4.6 Example Transformations

This section presents some typical examples of program transformations for Tiger. The examples can be adapted with little effort for other imperative programming languages.

4.6.1 Desugaring

Desugaring is a simple program transformation which simplifies programs by defining constructs in terms of other, often simpler constructs or by simplifying their usage, e.g., splitting `let` bindings into several `let` expressions, each one containing exactly one binding. For program transformation developers, desugaring is a required transformation which reduces the amount of work in writing specifications.

We will desugar a small Tiger program by (1) simplifying multiple declarations of variables, (2) introducing a declaration scope for each variable declaration and (3) representing binary expressions in a prefix notation by using a single constructor for all binary arithmetic expressions as well as for Boolean expressions. The prefix representation of binary expressions makes it possible to match any binary expression by using only two constructors instead of using as many as the object language supports.

```
DefAnd  : |[ e1 & e2 ]| -> |[ if e1 then e2 else 0 ]|
DefOr   : |[ e1 | e2 ]| -> |[ if e1 then 1 else e2 ]|
DefUmin : |[ - e ]|     -> |[ -(0, e) ]|
DefTimes: |[ e1 * e2 ]| -> |[ *(e1, e2) ]|
DefDiv  : |[ e1 / e2 ]| -> |[ /(e1, e2) ]|
DefPlus : |[ e1 + e2 ]| -> |[ +(e1, e2) ]|
DefMinus: |[ e1 - e2 ]| -> |[ -(e1, e2) ]|
DefEq   : |[ e1 =  e2 ]| -> |[ =(e1, e2)  ]|
DefNeq  : |[ e1 <> e2 ]| -> |[ <>(e1, e2) ]|
DefGt   : |[ e1 >  e2 ]| -> |[ >(e1, e2)  ]|
DefLt   : |[ e1 <  e2 ]| -> |[ <(e1, e2)  ]|
DefGeq  : |[ e1 >= e2 ]| -> |[ >=(e1, e2) ]|
DefLeq  : |[ e1 <= e2 ]| -> |[ <=(e1, e2) ]|
DefSeq1 : |[ ( e ) ]|    -> |[ e ]|
EmptyLet: |[ let in e* end ]| -> |[ (e*) ]|
LetSplit: |[ let d d* in e* end ]| -> |[ let d in letd* in e* end end ]|
```

Figure 4.7: Desugaring operator expressions.

```
let
    var b:= 4
    var a:= 6
    var c:= 3
 in
    (c := b * a;
     a := b * c - a;
     b := b + c + a;
     print(b))
end
```

⇒

```
let
    var b := 4
 in let
        var a := 6
     in let
            var c := 3
         in (c := *(b,a);
             a := -(*(b,c),a);
             b := +(+(b,c),a);
             print(b))
        end
    end
end
```

Desugar Strategy

The desugar strategy performs a `topdown` traversal while repetitively applying a number of desugaring rules. The repetition is controlled with the strategy `repeat`.

```
Tiger-Desugar = topdown(repeat(Desugar))
```

The `Desugar` strategy is composed of rewrite rules defined in Figure 4.7.

```
Desugar =
  DefUmin  <+ DefTimes  <+ DefDiv   <+ DefPlus   <+ DefMinus  <+
  DefEq    <+ DefNeq    <+ DefGt    <+ DefLt     <+ DefGeq    <+ DefLeq <+
  DefAnd   <+ DefOr     <+ DefSeq1  <+ LetSplit  <+ EmptyLet
```

```
EvalBinOp : |[ +(i, j) ]| -> |[ k ]| where <addS>(i, j) => k
EvalBinOp : |[ -(i, j) ]| -> |[ k ]| where <subtS> (i, j) => k
EvalBinOp : |[ *(i, j) ]| -> |[ k ]| where <mulS>(i, j) => k
EvalBinOp : |[ /(i, j) ]| -> |[ k ]| where <not(?|[ 0 ]|)>j;<divS>(i, j) => k
EvalBinOp : |[ +(+(e, i), j) ]| -> |[ +(e, k) ]| where <addS>(i, j) => k
EvalBinOp : |[ -(+(e, i), j) ]| -> |[ +(e, k) ]| where <subtS> (i, j) => k

EvalRelOp : |[ =(i, j) ]|   -> |[ k ]| where eval-rop(<eq>(i,j)) => k
EvalRelOp : |[ <>(i, j) ]|  -> |[ k ]| where eval-rop(<not(eq)>(i,j)) => k
EvalRelOp : |[ <(i, j) ]|   -> |[ k ]| where eval-rop(<ltS>(i,j)) => k
EvalRelOp : |[ >(i, j) ]|   -> |[ k ]| where eval-rop(<gtS>(i,j)) => k
EvalRelOp : |[ <=(i, j) ]|  -> |[ k ]| where eval-rop(<leqS>(i,j)) => k
EvalRelOp : |[ >=(i, j) ]|  -> |[ k ]| where eval-rop(<geqS>(i,j)) => k
EvalRelOp : |[ =(i1, i2) ]| -> |[ k ]| where eval-rop(<eq> (i1, i2)) => k
```

Figure 4.8: Some rewrite rules for constant folding.

Thus, a fairly elaborate transformation changing many constructors is defined in only a few lines of code using separately defined and reusable rewrite rules. This strategy combines rewrite rules using the strategy left choice combinator.

4.6.2 Constant Folding

Constant folding is the simplification of operator expressions with known constant values as operands, i.e., reducing expressions of the form `c1` $\oplus$ `c2` for some operator $\oplus$ with constant arguments `c1` and `c2`. For instance the expression `3 + (6 * (5 - 2)) / 2` can be simplified at compile time to `12` by applying arithmetic laws. Constant propagation empowers constant folding; if a variable can be replaced by its constant value, run-time computations may be replaced by compile-time computations. Constant propagation is discussed in Chapter 7.

Recall that Tiger expressions consist of arithmetic and relational operations on integer values and Boolean values respectively. Figure 4.8 presents the rewrite rule set for constant folding evaluation. The strategies used to evaluate expressions such as `addS`, `subtS`, `mulS` are part of the Stratego library. For the evaluation of relational operations such as equality and greater-than the strategy `eval-rop(s)` is used. This strategy applies the parameter strategy to evaluate the relational operation. The result of the evaluation strategy produces an integer that represents the result of the evaluation. This strategy is defined as follows

```
eval-rop(s) = if s then !|[ 1 ]| else !|[ 0 ]| end
```

```
ElimIf    : |[ if 0 then e else e2 ]| -> e2
ElimIf    : |[ if i then e else e2 ]| -> e where <not(eq)> (i , |[ 0 ]|)
ElimIf    : |[ if 0 then e ]| -> |[ () ]|
ElimIf    : |[ if i then e ]| -> |[ e ]| where <not(eq)> (i , |[ 0 ]|)
ElimIf    : |[ if e then () ]| -> |[ (e) ]|
ElimIf    : |[ if e1 then e2 else () ]| -> |[ if e1 then e2  ]|
ElimIf    : |[ if e1 then () else e2 ]| -> |[ if not(e1) then e2 ]|
ElimFor   : |[ for x := e to e2 do () ]| -> |[ (e; e2) ]|
ElimWhile : |[ while e do () ]| -> |[ (e) ]|
```

Figure 4.9: Rewrite rules for unreachable code elimination.

Constant Folding Strategy

The specification of a strategy for constant folding determines which rules are applied and in which order. For the application of the constant folding rules a single bottom-up pass over the expression tree suffices to reduce all possible constant operator applications. Constant folding can be specified with the generic term traversal `bottomup`, as follows:

```
constant-folding = bottomup(try(fold))
```

The `fold` strategy used in the definition of `constant-folding` is defined as the choice between the `EvalBinOp` and `EvalRelOp` rules from Figure 4.8:

```
fold = EvalBinOp <+ EvalRelOp
```

The constant folding strategy defines a one pass traversal. Several rewritings are achieved with successful applications of the `EvalBinOp` and `EvalRelOp` rules.

4.6.3 Unreachable Code Elimination

Control-flow statements fork the execution paths of a program. At execution time some paths are not reachable. To avoid having to consider unreachable code, the constant folding rules are extended to eliminate branches from control-flow statements whenever the condition can be statically evaluated. Consider the `if` statement with three argument expressions, a `condition`, a `then`, and an `else` expression. If the evaluation of the condition results in a constant value, it determines which branch will be executed.

Unreachable code elimination can be expressed by means of rewrite rules shown in Figure 4.9. The application of an `ElimIf` rule selects the executable branch and the `If` expression is replaced by the surviving branch. If a loop will not be executed it can be removed from the object program as defined by the rules `ElimFor` and `ElimWhile`. The `constant-folding` strategy can be extended with rules defined in Figure 4.9 as follows:

```
constant-folding = bottomup(try(fold <+ unreachable-elim))
unreachable-elim = ElimIf <+ ElimWhile <+ ElimFor
```

4.6.4 Side-Effect Removal from Expressions

Tiger expressions can contain statements which may produce side-effects. This entails that statements can be used within expressions. Thus, the statement `x := a + (y := x + 1; y)` is a valid Tiger statement which produces side-effects. A side-effect is any change of state produced by the execution of an expression [45]. Side-effect removal is not a program optimisation per se, due to the fact that it does not improve any behavioural aspect of the resulting code. Instead, this program transformation generates code that is amenable for follow up program transformations, for its better comprehension and maintenance.

A side-effect expression removal transformation changes a program containing side-effect expressions into an equivalent program which contains only side-effect free expressions. Removal of side-effects requires proper ordering of statement evaluation. It also updates and cleans the structure of the code without changing its semantics. An example is:

```
let
    var x := 3
    var y := 3
    var a := 3
 in
    x := y + (a := x + 3; a * 2);
    print(x)
end
```

⇒

```
let
    var x
    var y
    var a
    var a_0
    var c_0
 in x := 3;
    y := 3;
    a := 3;
    a_0 := y;
    a := x + 3;
    c_0 := a * 2;
    x := a_0 + c_0;
    print(x)
end
```

The statement that defines the variable `x` as a side-effect also defines the variable `a`. The new value of `a` is used to define `x`. To avoid statements that have side-effects, we can lift the inner definitions and use the defined value in the right-hand side of the expression that defines `x`. This transformation requires the definition of temporary variables to hold intermediate values. The transformation is specified by using exhaustive rewriting by means of the strategy application `innermost`.

```
hoist-effects =
  innermost(
      HoistEffectFromBinOp
      <+ DetachInitFromDecl
      <+ MakeLetBodySeq
      <+ HoistLetFromSeq
      <+ AssignLet
      <+ AssignSeq
      <+ HoistDeclFromIf
      <+ HoistDeclFromWhile
      <+ HoistDeclFromFor
      <+ ...
  )
```

Figure 4.10: Removal of side-effects using the fixed strategy innermost.

Figure 4.10 shows the strategy `hoist-effects` which traverses a tree from inside out applying the transformations specified by its argument strategy. The rewriting involves many rules, here only a few are shown.[2] To give an impression of how to lift an expression out of a binary expression, consider the following rewrite rule:

```
HoistEffectFromBinOp :
  |[ bo(e1, e2) ]| -> |[ let var x := e1 in bo(x, e2) end ]|
  where <IsEffect> e1
      ; <not(IsEffect)> e2
      ; new => x
```

The rule checks whether the expression *e1* has side-effects and if so, it defines a new variable that will contain this expression, and it uses the newly introduced variable in place of the expression. Thus, we have cleaned the binary expression, but the side-effect expression is now at the declaration level. Other rules will take care of lifting side-effect expressions out of these constructs.

Figure 4.11 shows a subset of the rules that accomplish this transformation. The specification assumes no name clashes when restructuring of expressions takes place. The strategy `hoist-effects` presented in Figure 4.10 uses the `innermost` strategy which uses exhaustive rewriting. The set of rules to be applied with this strategy must be terminating. The criteria for termination of the applied rewrite rules is: (1) Lift expressions containing effects out of expressions and (2) Restructure the expressions in a way such that only sequences of expressions are present and declarations do not contain initialised expressions.

The strategy `IsEffect` checks whether an expression is a sequence or a `let` binding. Expressions containing sequence or lets are lifted. This is specified in the following

```
IsEffect = ?|[ (e*) ]| <+ ?|[ let d1* in e1* end ]|
```

[2] For a complete specification see Tiger in `http://www.stratego-language.org/Tiger`.

4.7 Further Reading

The contents of these chapter are based on the following publications. The reader may wish to refer to the publications that are of his/her particular interest.

- B. Luttik and E. Visser. *Specification of Rewriting Strategies* [68]. The paper introduces user definable rewriting programmable strategies. It describes the language to achieve programmable strategies.
- E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. *Building Program Optimizers with Rewriting Strategies* [105]. This paper introduces *rewriting strategies* to maintain the separation of rules from the control of the rewriting process. The separation allows reuse and concise specification of program transformation systems.
- E. Visser. *A Bootstrapped Compiler for Strategies* [99]. The article describes the Stratego compiler.
- E. Visser. *Strategic Pattern Matching* [100]. The paper describes System S as the base of the Stratego programming language for the specification of program transformations. In particular describes the mechanism of pattern matching and its use in contextual rules providing deep matching, recursive patterns and overlays to achieve code reuse.
- E. Visser. *Language Independent Traversals for Program Transformation* [101]. Generic traversals are described in this paper and its use to obtain language independent program transformations.
- E. Visser. *Meta-Programming with Concrete Object Syntax* [104]. The paper introduces the machinery to achieve programming with concrete syntax of embedded languages, thus improving readability of meta-programs.

```
HoistEffectFromRelOp :
  |[ ro(e1, e2) ]| -> |[ let var x := e1  var y := e2 in ro(x, y) end ]|
  where <IsEffect> e2; new => x; new => y

AssignLet :
  |[ x := let d* in e* end ]| -> |[ let d* in x := (e*) end ]|

AssignSeq :
  |[ x := (e1; e2; e*) ]| -> |[ (e1; x := (e2; e*)) ]|

MakeLetBodySeq :
  |[ let d* in e1; e2; e* end ]| ->  |[ let d* in (e1; e2; e*) end ]|

DetachInitFromDecl :
  |[ let var x ta := e in e* end ]| -> |[ let var x ta in x := e; e* end ]|

HoistDeclFromSeq :
  |[ (e; let var x ta in e* end) ]| -> |[ let var x ta in e; e* end ]|

HoistDeclFromSeq :
  |[ (let var x ta in e1* end; e2*) ]| ->
    |[ let var x ta in (e1*); (e2*) end ]|

HoistDeclFromWhile :
  |[ while e do let var x ta in e* end ]| ->
    |[ let var x ta in while e do (e*) end ]|

HoistDeclFromFor:
  |[ for x := e1 to e2 do let var x ta in e* end ]| ->
    |[ let var x ta in for x := e1 to e2 do (e*) end ]|

HoistDeclFromIf :
  |[ if e1 then e2 else let var x ta in e* end ]| ->
    |[ let var x ta in if e1 then e2 else (e*) end ]|
```

Figure 4.11: Code hoisting rules.

Part III

Context-Sensitive Program Transformation

5 Context-Sensitive Transformations

Program transformation often needs information which is located at different sites in an abstract syntax tree. We present examples where context-sensitive information is required for the realisation of transformation systems. Simple rewrite systems only based on rewrite rules can only access the information available in the term that is being transformed. This kind of systems lacks high level abstractions to represent context-sensitive information. Dynamic rewrite rules are an extension of Stratego which allows us to inherit information and make it available at different program points, thus providing context-sensitive information. Implementations using dynamic rules are provided as motivating examples of context-sensitive transformations.

5.1 Introduction

Program transformation uses context-sensitive information to accomplish many transformations. Classical examples are data-flow related optimisations such as constant propagation, value numbering and dead code elimination.

Rewriting rules are of a context-free nature, being only able to access the term being transformed. This is a problem for the implementation of program transformations that need context-sensitive information. For example, when inlining a function at a call site, the call is replaced by the function body in which the formal parameters have been substituted by the actual parameters. The substitution requires that the formal parameters and the function body are known at the call site, but these information is only available elsewhere in the parse tree.

There are many similar problems in program transformation, including bound variable renaming, type-checking, function specialisation, code hoisting, and data-flow transformations. Although basic transformations they all require access to contextual information.

There are many solutions available for making this context-sensitive information available. One solution to this problem is the use of *contextual rules* [6, 100, 105]. A contextual rule combines the context and the local transformation in a single rule by using a local traversal that applies the rule reusing information from the context. Thus, contextual rules solve the problem by applying the transformation at the context level instead of at the location where the actual transformation takes place. A context expression `e[e']` matches and replaces an expression `e'` occurring within `e`. For instance, the following contextual rule defines the inlining of a (unary) function definition at a function call site:

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

The rule is applied to an abstract syntax tree that contains both the function definition and its uses. Since function calls can be nested deeply in the body of the `let` expression, a local traversal is needed to find them. When such a rule is applied as part of a complete traversal over a program, e.g., to perform inlining for all function definitions, the extra local traversal leads to an unacceptable increase in execution time. To avoid this complexity, a more common solution to the problem is to extend the traversal over the tree (be it hand-written or generic) such that it carries the data needed by transformation rules. For example, traversal functions in ASF+SDF [15] can be declared to have an accumulation parameter in which data can be collected. Another solution is the *language independent definition* [101] of operations such as bound variable renaming in Stratego which capture a generic tree traversal schema that takes care of distributing an environment through a tree.

The disadvantage of such solutions is that the rewriting nature of the solution is lost. Instead of a rewrite rule performing a transformation, the traversal carries along a data structure that stores the context information. The traversal code maintains this data structure in order to add information at the appropriate places and to retrieve it at other places. For instance, an inlining algorithm needs to maintain a table, mapping function names to definitions. These data structures and operations are often complicated by the fact that the context information is governed by the scope and the data-flow of the object program. Further complications arise when multiple kinds of context information need to be carried along. Many variations of such data structures are used in transformation systems, e.g., symbol tables in type checking [5], and hash tables in value numbering [72]. Representing such data structures as terms within term rewriting systems has the disadvantage of the sub-optimal complexity of list manipulation and inspection.

This chapter describes the extension of rewriting strategies with *scoped dynamic rewrite rules* as a mean for providing context-sensitive information in transformation systems. It also presents examples of program transformations which use such context-sensitive information. The examples presented in Section 5.2 expose a clear need for an abstraction to represent context-sensitive information. Section 5.3 discusses other approaches for handling context-sensitive information. Section 5.4 describes the extension of Stratego with dynamic rewrite rules. Stratego implementations for the examples discussed in Section 5.2 using dynamic rewrite rules are presented in Section 5.5.

5.2 Examples of Context Sensitive Transformations

In this section we only motivate the need for such context information and in Section 5.4 we illustrate their implementation in Stratego using dynamic rewrite rules.

5.2.1 Bound-Variable Renaming

The scoping rules of the language determine which variable occurrences correspond to which variable declarations. Nested scoping enables us to use the same name identifier for different variables in a program, in which local declarations shadow declarations in enclosing scopes if they bind the same identifier.

Bound variable renaming is a transformation that gives a unique name to each variable, resulting in a program in which no declaration shadows any other declaration. This is a useful transformation that ensures that all occurrences of the same identifier refer to the same variable. Bound variable renaming is required in program transformations to avoid variable capture upon substitution; furthermore it can simplify subsequent program transformations.

The following example illustrates bound variable renaming for local variables (`var`), function arguments (`function`) and loop index variables (`for`) in Tiger. The program in the left box is transformed into the program in the right box:

```
let var a : int := x
    function foo(a : int) : int =
      let var a := a + 3
          var z := 6 + a
      in for a := a to a + 100 do
           z := z + a
      end
 in foo(a)
end
```

$\Rightarrow$

```
let var a : int := x
    function foo(b : int) : int =
      let var c := b + 3
          var z := 6 + c
      in for d := c to c + 100 do
           z := z + d
      end
 in foo(a)
end
```

The example illustrates different Tiger binding constructs and their scoping rules. Note that not all sub-terms are necessarily in the scope of a declaration. There are four different binding sites for the variable `a`. The first one is in a `let` construct, the second is as a formal argument of a `function` definition, shadowing other variables outside the body of the function. The third is a local definition of the variable `a` in the function `foo`. The last binding for `a` is in a `for` construct, defining it as an iteration variable. The iteration variable shadows variables outside the body of the `for`, but occurrences of `a` in the expressions defining the boundaries of the iteration are bound in the outer scope.

Occurrences of `a` are renamed to newly introduced identifiers in the program on the right. The variable `a` has been renamed into `b`, `c` and `d` which are fresh identifiers in the program.[1] The rewriting of `a -> b` is only valid till a new definition of `a` is found in a local declaration of `a` in the function `foo`. The rewriting of `a -> b` is only applicable in the function `foo`. In order to restrict applications of this rewriting we need to specify the context where it is valid. In Section 5.4 we show how this can be expressed using dynamic rewrite rules.

5.2.2 Function Inlining

In-line expansion, function inlining and procedure integration are different names that refer to the same compiler optimisation. They all replaces function calls with the body of the corresponding functions, instantiating the formal parameters with the actual parameters. The result of this optimisation is the avoidance of the overhead of transferring program control to the function. Another benefit is that it opens opportunities for further optimisations such as constant propagation and code hoisting. As a known drawback, it generally results in an increase of code size and therefore it may decrease performance, e.g., by introducing extra cache misses.

Figure 5.1 shows how the function `doSomething` is inlined at its call site. The depicted code fragment in the left box has three function definitions: `inc`, `doSomething` and `fact`. Binding the actual parameters to the formal parameters requires some care. Just substituting the actual for the formal parameters is not correct. Computations may be duplicated, or computations with a side-effect may be executed multiple times or out of order. Therefore actual parameters are bound to new variables in a local let declaration. Inlining a function call `f(`*`e1`*`,...,`*`en`*`)` is a substitution with `let` *`x1`* `:=` *`e1`* ... *`xn`* `:=` *`en`* `in` *`e`* `end`, where the formal parameters in *`e`* have been substituted with the newly introduced variables *`x1`*,..., *`xn`*. Here we assume bound variable renaming has been performed. Inlining the function `doSomething` causes the function `inc` to be inlined as well. Function definitions that have been inlined wherever applicable do not have to be present in the resulting code. The code fragment in the right box contains only one definition for `fact`. The code fragment resulting from function-inlining may have expressions that have side-effects: take for instance the condition of the `while` expression.

[1] To rename variables a generator of new identifiers is needed; at this point we assume that we have such a name generator available.

```
let
 var x := 1
 var z := 4
 var y := z
 var b := 2

 function fact(n : int) :int =
   if n < 1 then 1
   else (n * fact(n - 1))

 function inc(c: int) :int =
      c + 1

 function doSomething(a:int) :int =
   if a < 5 then (a := a - 1;
                  b := inc(a);
                  1 )
   else 0
 in
  while (doSomething(z)) do
    (z := z + y);
   z := z + x;
   print(fact(z - 4))
end
```

⇒

```
let
 var x := 1
 var z := 4
 var y := z
 var b := 2

 function fact(n : int) : int =
   if n< 1 then 1
   else n * fact(n - 1)

 in
   while (let var a :int := z
          in if a< 5 then
             (a := a - 1;
              b := let var c:int := a
                     in c + 1
                   end;
              1 )
             else 0
          end) do
      (z := z + y);
    z := z + x;
    print(fact(z - 4))
end
```

Figure 5.1: Example of function inlining.

Followup transformations can hoist statements from expressions and construct a side-effect free resulting program. This transformation was previously described in Subsection 4.6.4.

Unrestricted function inlining may cause non-termination when inlining of recursive functions. Thus, criteria to determine which functions are to be inlined is an important aspect of function inliners; code size of the function to inline and restricting inlining depth are the most common criteria. We refer to the reader to Chapter 10 where inlining strategies are studied.

5.3 Approaches to Context-Sensitive Transformation

In program transformation systems, different kinds of data structures, formalisms and mechanisms are used for dealing with context-sensitive information, in particular symbol tables, use-def chains, static single assignment format and attribute grammars. We describe the data structures and attribute grammars from the perspective of their use for handling context-sensitive information.

5.3.1 Symbol Tables

An abstract syntax tree contains relevant information regarding the syntactical structure of a program. Semantic information or contextual information of a program may be situated at different nodes of the program tree. A symbol table is a compile-time data structure which associates symbols in a program with contextual information, e.g., the type of the symbol. Symbol tables are often implemented by using a hash table for efficient lookup. Since symbol tables are concerned with names, they have to handle the scoping rules of the object programming language explicitly. For example, the symbol table implementation used in Appel's Tiger compiler [5] remembers the state of a hash table in a `beginScope` and restores this information in the `endScope`.

5.3.2 Use-Definition Chains

Use-Def and Def-Use chains are data structures frequently used in compilers [1]. A use-definition chain (*ud-chain*) contains for each use of a variable all possible definitions that may reach it. A definition-use chain (*du-chain*) contains for each definition pointers to its uses. Thus, it relates information from the definitions of a variable to its uses and vice versa. Different data structures can be used to implement `ud/du` chains. Graphs with directed pointer references are the most common. When transforming with such graphs the chains need to be updated accordingly.

5.3.3 Static Single Assignment

Static single assignment(SSA) is the property of a program representation where each variable is defined only once [5, 29]. This representation encodes information about definitions and uses of a variable by imposing a name convention. Thus, variable names reflect the control-flow of variables. Special ϕ functions are introduced at program points where the control-flow of a program joins, thus ensuring the SSA property. The SSA intermediate representation includes new assignments that contain distinct names for the same variable, possibly eliminating false dependencies. For its construction, it requires the insertion of ϕ functions and proper *variable renaming* that ensures that a variable is defined only once. The SSA property ensures the unique association between a variable and its uses. With this property, propagation of information can be easily achieved.

The advantage of the SSA representation is that it simplifies the construction of program transformations. For example, consider *sparse conditional constant propagation* [110]. As a drawback, this representation results in a code size increase and requires to be reconstructed into a program representation that is ϕ function free. Moreover, for source-to-source transformation a naive reconstruction process removing the ϕ functions and re-mapping the names back to the original may result in inconsistent code [20]. Furthermore, there are some problems regarding the reconstruction after aggressive program transformations, but this falls outside the scope of this thesis.

5.3.4 Attribute Grammars

Attribute grammars constitute a formalism for the construction of translation tools, especially compilers [60, 7, 85, 44]. An attribute grammar is a context-free grammar augmented with attributes, semantic rules, and conditions. An attribute grammar associates information with constructs by attaching attributes to grammar symbols or abstract syntax trees. The value of attributes is computed by the semantic rules and can be further restricted by semantic conditions. Attributes can be either synthesised or inherited. Inherited attributes are computed using synthesised attributes of their siblings and inherited attributes of their parents are suitable for passing context information. Inherited attributes can be propagated using a top-down traversal while synthesised attributes can be propagated with a bottom-up traversal. The advantage of an attribute grammar system is its ability to transport information from any point in the abstract syntax tree to anywhere else. The information is transported in a controlled way. An attributed tree is consistent with an attribute grammar if the values of all attribute instances in the tree satisfy the semantic rules.

A compiler translates a program written in a source language (SL) into the target language (TL). Frequently this translation requires many transformation passes involving intermediate languages (IL) and/or structural changes in the representation of a program. While attribute grammars are a powerful formalism for specifying transformations from a program from an SL into TL, the intermediate representations require the use of attributes to emulate such changes. The emulation enables the implementation of compilers however system properties such as modularity, encapsulation, reusability are lost in the way [80]. In order to alleviate the afore mentioned problems attribute grammars have been extended in several ways [107, 115, 37]. Combining the power of attribute grammars and program transformation systems is a topic of research [48, 80, 115, 54].

5.4 Dynamic Rewrite Rules

Pure term rewriting systems lack access to context-sensitive information. To satisfy this need, Stratego has been extended with dynamic rewrite rules [102, 95, 19]. Dynamic rules are rewrite rules defined at execution time, as opposed to static-time for (*normal*) rewrite rules. Dynamic rules are defined at run-time and can access variables available from their definition context. There are several operations on dynamic rules. The operations are: rule definition (creation), re-definition (update), undefinition (remove) and rule application itself.

s	::=	`rules` $(drd_1 \ldots drd_n)$	dynamic rule definition
	$\mid$	`{|` $f_1, \ldots, f_n$ `:` s `|}`	dynamic rule scope
drd	::=	$drsig$ `:` p_1 `->` p_2 (`where` s)?	dynamic rule definition
	$\mid$	$drsig$ `:+` p_1 `->` p_2 (`where` s)?	dynamic rule extension
	$\mid$	$drsig$ `:` p	dynamic identity rule definition
	$\mid$	$drsig$ `:-` p	dynamic rule undefinition
	$\mid$	f+p	label current scope
$drsig$	::=	sig	relative to current scope
	$\mid$	sig.p	relative to labelled scope
	$\mid$	sig+p	relative to current scope and label

Figure 5.2: Extension of Stratego syntax with dynamic rules.

5.4.1 Defining, Undefining and Application of Dynamic Rewrite Rules

In this section we describe basic operations on dynamic rules: definition, undefinition and rule application.

Definition and Re-definition of Dynamic Rules A dynamic rule is defined using the construct `rules(`$R : p_1$ `->` p_2 `where` c`)`, which introduces an instance of a dynamic rule with name R. The variable pattern p_1 is bound by the context where the dynamic rule is defined. As an illustrating example, consider the strategy expression:

```
?|[ x := i ]|; rules(ValueOf: |[ x ]| -> |[ i ]|)
```

This strategy first matches the current term against the assignment pattern `?|[ x := i ]|` binding the meta-variables *x* and *i*. Then a new rule with the name `ValueOf` is defined, with the bound values of *x* and *i*. The purpose of this dynamic rule is to record that the variable *x* contains the integer value represented by the pattern *i*. Application of the rule replaces an occurrence of *x* by this value. Such a rewrite rule is specific to a particular program segment, rather than being a universally valid transformation rule. Thus, dynamic rule instantiation at run-time requires a control mechanism to restrict its application to the program segments where the replacement is valid. Furthermore, there can be many instances of `ValueOf` dynamic rules, differing in the values bound to the pattern variables.

It is possible to redefine a rule that has the same variable bindings for *x* and as a consequence only a redefinition is applicable. To illustrate this behaviour consider the application of `?|[ x := i ]|; rules(ValueOf: |[ x ]| -> |[ i ]|)` to `a := 1; a := 2`. The first application defines the rule `ValueOf: a -> 1`, the second redefines the rule to `ValueOf: a -> 2`.

Undefining Dynamic Rewrite Rules Undefinition of a dynamic rule is specified with the construct `rules(`R` :- `p`)`. The application of an undefined dynamic rule always fails. As a concrete example let us consider the application of the strategy expression `?|[` x `:=` e `]|; where <not-constant>` e`; rules(ValueOf:- |[` x `]|)` to `a := foo(3)`. The application will undefine the rule `ValueOf`, because `a` does not longer bind a constant value.

Application of Dynamic Rewrite Rules Dynamic rule application is described as a normal rewrite rule application, i.e., by referring to the rule name. For instance the rule `ValueOf: a -> 2` from the previous example is applied with the syntax `<ValueOf>` t to the term t. If the term t matches with `a`, it will be replaced with `2`.

5.4.2 Dynamic Rule Scope

A feature of dynamic rules is that rule definitions are not constrained to a lexical scope, but are globally visible. This entails that rules are *implicitly* propagated. A transformation strategy does not need to pass around the current set of rules. Thus, a rule defined in one part of a strategy can be applied in another part, without parameter passing. In particular, this means that a transformation can be organised as a sequence of phases that implicitly pass on information in dynamic rules. For instance, one phase may define inlining rules for top-level functions, which are then used to inline function calls in subsequent phases. As a consequence of this design, the definition of a dynamic rule permanently redefines any previous definition for the same left-hand side. Likewise, the undefinition of a dynamic rule permanently erases that definition. Sometimes, however it is useful to redefine or undefine a rule only temporarily and restore the old definition after performing some local transformation. For instance, an inlining rule for a local function may redefine an inlining rule for an outer function with the same name. Thus, after traversing the subtree in which the local function is in scope, the inlining rule for the outer function should be restored. Achieving this with a single rule definition and undefinition requires maintaining information about dynamic rules in strategies, which is undesirable. Instead, Stratego provides a construct for limiting the lifetime of dynamic rules.

Delimiting dynamic rules The scope of a dynamic rule is specified with the syntax `{|` $f_1, ..., f_n$ `:` s `|}` . This construct delimits the visibility of dynamic rules specified by $f_1, ..., f_n$ to the scope of the strategy s. That is, all rules defined during the execution of s are visible and applicable, and are removed when s terminates, after which the original set of rules is restored. An example of rule scope use is:

```
{| ValueOf : |[ for <id> := <id> to <id> in <s> ]| |}
```

This strategy defines the scope of the dynamic rules `ValueOf` to be visible and applicable only in the body of the `for` expression. Here we assume the previous definition for the dynamic rule `ValueOf`.

5.4.3 Labelled Rule Scopes

To provide specific control over dynamic rule application, rule operations can be defined relative to a dynamic rule *label*. This allows the definition of an identifiable scope label for dynamic rules. Thus, dynamic rule definition and undefinition are relative to an scope label. The labelling scope mechanism of dynamic rewrite rules can model different scoping constructs of block structured programming languages.

The construct `rules(`R+l `:` p_1 `->` p_2`)` introduces and initialises the scope label l for the dynamic rule R. To re-define this rule, the rule operation is relative to its label as in the construction `rules(`R`.`l `:` p_1 `->` p_3`)`. Dynamic rule undefinition is also relative to the scope label as is shown in the construction `rules(`R`.`l `:-` p_1`)`. An example of rule scope use is:

```
?|[ x := i ]|
; rules(ValueOf.x : |[ x ]| -> |[ i ]|)
```

This strategy matches an assignment of an integer number *i* to *x*. To refer to the current scope, rule operations use the identifier as a *rule label.* The strategy requires that the label *x* has been introduced, as in:

```
?|[ var x ta:= e  ]|
; rules(ValueOf+x : |[ x ]| -> |[ e ]|)
```

5.4.4 Extended Dynamic Rules

The effect of re-defining a dynamic rule causes the old rule with the same left-hand side to be discarded. This behaviour can be altered with *extended dynamic rules*. The construct `rules(`R `:+` p_1 `->` p_2`)` makes it possible to keep multiple R dynamic rules instantiating the pattern p_1. Dynamic rule extension allow us to keep multiple rules with the same name and left-hand-side pattern. Additional operations for extended dynamic rules are `bagof-R` and `once-R`. The `bagof-R` operator gives as a result the list of all right-hand sides for which a rule with name `R` succeeds. The operation `once-R` describes a unique application of the dynamic rule `R`; after have been successfully applied once the rule is discarded.

5.5 Realisation of Context-Sensitive Transformations

We present examples of the use of dynamic rules for achieving context-sensitive rewriting without the added complexity of local traversals and abstracting over complex data structures. In addition to propagating information to different points in a tree, a rewrite rule can perform a transformation. These concepts are illustrated by bound variable renaming and function inlining.

5.5.1 Realisation of Bound Variable Renaming

A dynamic rule is defined for each variable to be renamed. We define a renaming `RenameVar` dynamic rule for each binding construct in Tiger: variable declarations,

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

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

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

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

strategies

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

Figure 5.3: Bound variable renaming specification.

`for` loops, and function parameters. For instance consider the renaming rule for variable declarations:

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

The rule matches a variable declaration and renames the variable declaration for *x* by replacing it with the newly generated identifier *y*. An application of the (static) rule `RenameVarDec` renames *x* and defines a dynamic rule `RenameVar` that rewrites *x* to *y*, overwriting a possibly already existing rule.

Renaming Strategy The renaming strategy `exprename` is a slight variation of a `topdown` (see Section 4.5) traversal over the program applying the renaming strategy on the way.

```
exprename = try(RenameVarDec <+ RenameVar); all(exprename)
```

However, the generic `topdown` traversal will not rename the variables correctly. A `topdown` traversal does not fit well the scoping rules of Tiger. Consider for instance the construct `|[ let var x := e1 in e2 end ]|` which defines the scope of the variable *x* to be the expression *e2*. A `topdown` traversal will rewrite occurrences of *x* in *e1* and *e2*, thus incorrectly renaming occurrences of *x* in *e1*.

An alternative traversal that visits the language constructs in the same way that the language bindings are defined for Tiger, has thus to be explicitly defined. The scope of a renaming rule is restricted to the traversal of that program segment in which the corresponding variable is in scope. Outside this scope, all previously defined renaming rules become active again.

The strategy expression `{| RenameVar : all(exprename) |}` defines the scope of the `RenameVar` rule. After leaving the scope for the strategy `all(exprename)`, all `RenameVar` rules are undefined: having control over the life range of dynamic rules allows us to rename variables according to the Tiger scoping rules.

Figure 5.3 presents the complete variable renaming transformation for Tiger. It uses the rules `RenameVarDec`, `RenameFor` and `RenameArgs` to rename identifiers in their respective binding constructs using the rule `NewVar`. The rule `NewVar` defines the `RenameVar` dynamic rule to rename variables. `RenameVar` rules are applied during the traversal performed by the `exprename` strategy. This traversal strategy uses the dynamic rule scope construct to restrict the scope of the `RenameVar` rule. For instance consider the strategy definition:

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

This strategy specifies that any `RenameVar` rule defined during the traversal of a `let` construct is restricted to that `let`. Similarly, the `exprename` traversal restricts the scope of function arguments to the body of the function definition and the scope of the `for` loop iteration variable to the *body* of the loop. Note that the loop bound expressions are visited before defining the renaming rule. In the same way, the initialiser of a variable declaration is visited before renaming the variable declaration itself, thus ensuring that any occurrences of the identifier within the initialiser are renamed first. Finally, note how the `NewVar` rule invokes the `RenameVar` rule to establish whether the identifier was already used in an enclosing scope; if `RenameVar` fails, then there are no occurrences of this identifier in the enclosing scopes, and there is no need for renaming.

5.5.2 Realisation of Function Inlining

The implementation of a simple function inliner for Tiger is shown in Figure 5.4. Although this is a simple inliner with simple criteria to determine whether to inline, the important aspects for a correct implementation are all present.

The `inline` strategy is recursively defined based on function definitions and function calls. Function definitions are only valid within the `let` binding in which they are defined.

```
inline = ( |[ let <*id> in <*id> end ]|; {| InlineFunction: all(inline) |}
           <+ Declare
              ; |[ <fd*: inline> ]|
              ; RemoveFunctions
           <+ InlineFunction; inline
           <+ all(try(inline))
         )

Declare =
  |[ let <fd*: map(try(DeclareFun(inlineable)))> in <*id> end ]|

RemoveFunctions =
  |[ <fd*: filter(not(DeclareFun(inlineable)))> ]|

DeclareFun(inlineable) =
 ?|[ function f(x*) ta = e ]|
 ; inlineable
 ; rules(
    InlineFunction+f : |[ f(a*) ]| -> |[ let d* in e end ]|
    where <zip(bind-arg)> (x*,a*) => d*
  )

inlineable =
  not(is-recursive)

bind-arg :
 (FArg|[ x ta ]|, e) -> |[ var x ta := e ]|
```

Figure 5.4: A simple function inliner.

The strategy `inlineable` prevents recursive functions to be inlined. It verifies whether a function is recursive by using a generic traversal strategy that looks for an occurrence of a function call to itself. The strategy `oncetd(`*s*`)` traverses a tree in a top-down fashion applying the strategy *s* in its visit. This strategy succeeds on the first successful application of *s*. The strategy `is-recursive` verifies if a function is self-recursive by matching the name of the function with the pattern *f* and searching for calls containing this identifier in its function body.

```
is-recursive = where(|[ function <?f>(<*id>) <id> = <oncetd(?|[f(a*)]|)> ]|)
```

The `inline` strategy when encountering a `let` defines and enters a new scope for the `InlineFunction` rule. When encountering a sequence of function definitions, inlining rules (`InlineFunction`) are declared for each inlineable function using the strategy `map` over the list of function definitions. After that, inlining is applied to the function definitions themselves. As result, inlineable function calls in the function definitions are inlined. Function inlining rules are refreshed, thus reflecting the changes of the optimised function bodies. Furthermore, the `RemoveFunctions` strategy removes inlined functions; after inlining all function calls, function definitions are not longer needed, and are removed from the code. The `DeclareFun(inlineable)` strategy defines a new `InlineFunction` rule for a function *f*, replacing a call to *f* with its body inside a `let` binding. The `let` construct binds the actual parameters *a** to the formal parameters *x** as local variables. The binding of arguments is achieved with the Stratego library strategy `zip(s)`. This strategy builds a list of variable declarations by applying the rule `bind-arg` to the list of arguments and the formal parameters of the function. If a function is not deemed inlineable, there will be no `InlineFunction` rule for such a function. Thus, the filtering strategy will keep the definition of non-inlineable functions.

Inlining of functions is specified based on the name of the function. The inlining strategy is not valid for languages that support function overloading, where we may have several function definitions binding the same name. Tiger does not support function overloading, but just for the following discussion let us assume that we have overloading based on the number of arguments. For that purpose, the dynamic rule for inlining can be modified into:

```
DeclareFun(inlineable) =
 ?|[ function f(x*) ta = e ]|
 ; inlineable
 ; rules(
     InlineFunction+f :+ |[ f(a*) ]| -> |[ let d* in e end ]|
     where <zip(bind-arg)> (x*,a*) => d*
  )
```

The difference in the implementation of this strategy is the use of *extended dynamic rules*[2] to keep multiple rule occurrences with the same left-hand-side pattern of the `InlineFunction` rule. Arity-based verification of the number of arguments of the function is achieved with the strategy `zip`, which succeeds only if the list of terms *x** and *a** have the same number of elements. This strategy is still very simplistic and it does not support type-based overloading. For the latter, type information of the arguments of the call is needed to select which function should be inlined. To achieve type-based overloaded function inlining, an extra clause in the condition of the `InlineFunction` rule must be specified.

5.6 Discussion

Scoped dynamic rewrite rules are a novel extension of Stratego. A defined rewriting strategy controls not only the application of static rewrite rules, but also controls the definition, scope, and application of dynamic rewrite rules. This language extension is inspired in previous work in program analysis. The main contribution of dynamic rules is the ability to supply information and to encapsulate a transformation.

[2] Extended dynamic rules are specified adding a + symbol following the semicolon after the rule name.

Program transformation has the main goal of altering the input program in an expected manner. Program analysis are the fundamentals for enabling the required changes to a program. In this chapter we referred to symbol tables, use-definition chains, static single assignment form and attribute grammars as approaches used to support the implementation of program transformation systems. From all them attribute grammars are the most powerful formalism to support the implementation of transformation systems. Attributes are used for collecting information and propagating it in an inherit and/or synthesised manner. An important aspect of attribute grammars is the reusability of the abstract syntax tree to allow several computation over the same structure to take place.

Several extensions are proposed to classical attribute grammars such as copy rules, collection of attributes and forwarding [98]. An attribute grammar requires an attribute evaluator which is responsible of scheduling a traversal over the abstract syntax tree to determine the values of the attributes. Some attributes have been extended in such a way that they can encode program transformations. For instance the UUAG system has SELF attributes which allow to keep local modifications to a tree [10]. However if the transformation system requires several modifications to take place, attribute grammar systems do not allow the specify attributes in the result of SELF attributes. In order to obtain the resulting tree a way to combine the pieces of the original tree and the nodes with modifications is required.

Although attribute grammars ease the implementation of language tools by allowing reusability of the data structure for multiple analysis it is still cumbersome to support the composition of multiple transformations, and yet alone to enable the combination of analysis and transformation while sharing a single traversal.

In this dissertation we look at the abstractions which enable us to simplify the implementation of program transformations and to bring transformation in front of the footlight and program analysis to the back stage. This is the main motivation for finding abstractions to allow the combination of transformation and data-flow analysis. The rest of the thesis will address such objectives.

5.7 Summary

In this chapter, we have focused on describing mechanisms for the propagation of context-sensitive information. We have motivated the need for an mechanism that allows us to implement program transformations requiring this sort of information. Furthermore, the propagation of context-sensitive information must be restricted to those places where the relation is valid, e.g., its enclosing scope.

Scoped dynamic rewrite rules are a novel extension of Stratego. A defined rewriting strategy not only controls the application of static rewrite rules, but also controls the definition, scope, and application of dynamic rewrite rules. This language extension is inspired by previous previous work in program analysis. Dynamic rewrite rules are used to relate context-information from the site where it is available to the site where is needed. Moreover, labelled scoped dynamic rules have the functionality required for handling scopes on the language object by adding special purpose language constructs to the meta-language.

6 Data-Flow Transformations

Data-flow transformations are not only used in compilers but also in other programming tools such as aspect weavers, code generators, domain specific optimisers and refactoring tools. Due to the broad applicability of data-flow transformations, there are several requirements that a framework for developing data-flow optimisations might fulfil. In this chapter we present the foundations for solving data-flow problems and resulting design issues for data-flow transformations. A review of related work and existing approaches to the challenges for realising data-flow transformations is presented. Finally, we introduce the Stratego approach for realising data-flow transformations.

6.1 Introduction

The specification of program transformation systems requires the specification of both **how** to transform and **when** to transform. For the specification of **how** to transform, term rewriting is a natural and elegant paradigm to represent changes or transformations to a program. The specification of **when** to transform refers to all those program points that satisfy the enabling conditions of a particular transformation. A program point is a location in the representation of the source program. If all specified conditions hold for a certain program point, the transformation can be performed.

The applicability conditions are the subject of study of program analysis. Data-flow analysis is a branch of program analysis concerned with gathering facts that are valid on any possible execution path of a program [1, 72]. How the data flows in a program in general depends to the sequence of computed values during the course of program execution. Compiler optimisations and program transformation systems use information collected by a data-flow analysis and apply transformations while preserving the computed values. Traditionally, a data-flow analysis framework constitutes the starting point for many compiler optimisations that depend on information collected from a program.

While data-flow optimisations in compilers usually work on fixed low-level intermediate representations, some applications require transformations at source code level. Furthermore, compiler optimisations are traditionally implemented in general purpose languages, optimising for speed while executing transformations rather than optimising the productivity of the transformation developer. Higher productivity can be achieved using a language and/or environment that provides adequate support for the program transformation domain. For a framework for source-to-source transformations to be widely applicable it should cover a wide range of transformation tasks. That is, it should not be specific to a single object language and should not be restricted to support only a single type of transformation. Rather, it should provide high-level abstractions for modelling control- and data-flow of the language under consideration, and it should support combining data-flow transformations with other types of program manipulation such as template based code generation. Also, the framework should provide abstraction from details of program representation and should for instance support handling issues such as variable scopes and help to avoid problems such as free variable capture.

In Section 6.2 some transformations that require data-flow information are introduced. Section 6.3 explains the importance of data-flow analysis for program transformation. Section 6.4 presents the preliminaries for data-flow analysis. An evaluation of the data-flow analysis from a different perspective is presented in Section 6.5. In Section 6.6 we discuss some features that are missing in traditional data-flow analysis frameworks. In Section 6.7 a review of tools and frameworks and the description of their approach of solving data flow problems is presented. Finally, in Section 6.8 we describe how data-flow transformations can be realised in Stratego.

6.2 Data-flow Transformation Examples

Data-flow optimisations such as constant propagation, copy propagation, and dead code elimination transform or eliminate statements or expressions based on data-flow information that is propagated along all possible execution paths. These optimisations depend on information collected at different program points. Furthermore, those program points may be distant from each other. Thus, a program optimiser needs to relate information spread over different program points and the transformation may involve rewritings at several program points. For an illustration of classical data-flow optimisations consider the example:

```
(a := 3;            (a := 3;
 b := a + 2;         b := 5;            (a := y;
 a := y;       ⇒     a := y;       ⇒     b := a + 5;
 b := a + b;         b := a + 5;         print(b))
 print(b))           print(b))
```

Here constant propagation in the first step is followed by dead code elimination. In the example constant propagation discovers that the variable `a` contains the value `3`, and the propagation of this information to the second assignment allows the evaluation of the expression `3 + 2`, after the replacement of the variable `a`. This process subsequently discovers that `b` contains the value `5`, and can be further propagated until the program point that redefines the value of `b` in the fourth assignment. As described, constant propagation progresses by using a *forward* directed traversal flow. A forward traversal visits the nodes of a tree in the same direction that they are going to be executed.

After constant propagation some of the expressions in the resulting program are no longer required to be executed. Dead code elimination is thus subsequently applied to the outcome of the first transformation. Dead code elimination visits the program fragment in the opposite direction, namely *backwards* to the flow of execution. In our example, the final expression depends on the value of `b`. In order to compute the last statement, we require the value of `b`, thus, a statement that defines `b` is *needed* on all paths to this program point. The fourth statement defines `b`, and for its definition needs the value of `a`. This process continues searching for a definition of `a` and then for `y`. Statements that compute variables that are not signalled as needed can be deleted. Thus, the first and second statements are dead code and are eliminated in the final code fragment. This example suggests that the following ingredients are needed for the implementation of data-flow based optimisers:

– Collect information for all program points. Constant propagation collects variables that contain constant values, and dead code elimination collects the *needed* property of variables.

– Relate collected information available from different program points (by means of propagation while traversing the abstract syntax tree). Thus, a relation is established between the program point where the information is available and where it may be used.

  For constant propagation the relation is between the definition of a variable and the uses of that variable, and more specifically the value assigned to that variable. For dead code elimination, the relation is the other way around, between the sites where a use of a variable is needed towards the definitions of that variable.

– The propagation of information requires a directed traversal flow on the abstract syntax tree. For constant propagation the direction of traversal is forward and for dead code elimination is backward.

– The specification of code transformations. The substitution of a variable by its associated constant value is an example of a code transformation. This code modification can be specified in a clear way by means of rewrite rules.

6.3 Data-flow Analysis for Program Transformation

Data-flow analysis is described as the process of abstractly interpreting a program thus collecting information about program properties [46]. Data-flow analysis is a tool for reasoning about different aspects (facts) of a program, such as modified variables, variables that preserve their value, and interdependencies between variables.

Criteria for code optimisations can be to minimise execution time, to minimise code size, or to minimise memory usage. This improvement can be achieved as a result of applying an optimisation, or by modifying the code so that optimisations in following transformations become enabled. Typically optimising transformations are constructed using the reasoning stated in a *data-flow analysis problem*. A data-flow analysis problem is a high-level, natural language description of some aspect of the run-time behaviour of a program. As a result of the separation of analysis and transformation, the transformation pass is often not the place where the really interesting work is done, and thus often left implicit. In contrast, solving a data-flow analysis problem is the main task for building data-flow program optimisers.

The compiler construction community has a collection of well known data-flow analyses, such as available expressions, reaching definitions, liveness analysis and very busy expressions.

Available expressions is a data-flow analysis problem which determines whether an expression e is already computed at a program point p on all execution paths that lead to p, i.e., it was computed on all execution paths passing through p. This analysis helps avoiding the re-evaluation of expressions.

Reaching definitions analysis looks for program segments where an assignment to a variable v reaches a point p without an intervening definition of v. This information is exploited by constant propagation and copy variable propagation.

Liveness analysis determines the program segments where a variable definition is alive. A variable is alive from the point where it is defined to the program point where its last use occurs. The results of this analysis can be used for dead code elimination and for register allocation.

Very busy expressions analysis identifies those expressions e which are computed along all paths from a point p without any intervening definitions of variables in e. A very busy expression e can be moved to a single, common location. The result of the analysis is used in code hoisting, which relocates the computation of a very busy expression to a program point where it will be always executed.

6.4 Preliminaries of Data-Flow Analysis

The goal of a data-flow analysis is to find and to gather information for a data-flow analysis problem. There are several aspects which characterise a data-flow analysis problems, such as the direction of the analysis, sensitivity to control-flow and sensitivity to context information.

Direction of the Analysis Collecting program facts is done while traversing a program. A program traversal that follows the order of possible execution is known as a *forward* traversal. The traversal of a program starting from the exit to the program towards its initial point is called a *backwards* traversal. An example of a forward data-flow analysis is the identification of available expressions, whereas liveness is a backwards analysis.

Some analyses are bidirectional, such as the analysis for identifying partially redundant expressions. Fortunately, such problems can be decomposed into a sequence of forward and backward analyses [59].

Flow Sensitivity Programs have special statements to control the order of execution. If the analysis considers the execution order of the expressions, the analysis is *flow sensitive*, otherwise the analysis is *flow insensitive*. Flow sensitive analyses are usually more precise at the price of more computational resources.

Control-Flow Sensitivity When the analysis takes into account control flow program constructs it has to collect information which is safe to propagate in all possible execution paths of a program. Control-flow constructs have the ability to fork and merge execution paths. A confluence program point is the location where two different execution paths join. An instance of a confluence point is where branches of an `if-then-else` expression join. At confluence program points information gathered has to be merged. If the state of the analysis is represented with sets a merge operation can be a intersection or a union. Typically the union operation is used in **may** type analyses, and the intersection operation in **must** type analyses. The first kind of analysis collects information that may hold at a program point, whereas the second one collects information which is guaranteed to hold. An example of a may analysis is the identification of reaching definitions, and an example of a must analysis is the identification of very busy expressions.

Context Sensitivity Imperative programming languages may modify the (state) context of a program by a function call execution. Semantics of function invocations have to be considered by a data-flow analysis.

Data-flow analysis restricts the boundaries of the analysis to inter-procedural or intra-procedural. If the analysis takes into consideration context sensitive information the analysis is known as **inter-procedural analysis**. In contrast, the analysis is **intra-procedural** when it is performed at a function level (not taking into consideration modifications of the context by a function call invocation). This type of analysis assumes that the complete context is changed after a function invocation.

6.5 Evaluation of Data-Flow Transformations

Data-flow transformations have been extensively studied. As a result, stable foundations have been developed for formulating and solving data-flow analysis problems. However, most of the conducted research was focused on developing algorithms attempting to improve performance, efficiency and scalability of data-flow analyses.

We believe that there are other factors that have to be considered when building program transformations in general, and data-flow optimisations in particular. The range of aspects to improve include the following:

Language Expressivity Implementation of data-flow transformations is not a trivial task, and any help from the meta-language[1] has a crucial impact on the way a data-flow problem is solved. Thus, if the optimiser developer has help from the meta-language for the specification of transformations, he/she can concentrate on specifying the conditions that control and specify the transformation.

Generic traversals, programmable rewriting, and separation of rules from strategies have a large impact on clarity and conciseness of the implementation. Abstractions and specific functionality for the task at hand simplify the development of program transformations.

Developer Productivity Algorithms and data representations lie at the base of performance improvement of the constructed transformation. To improve performance, the optimiser developer frequently struggles using a general purpose language suitable for achieving program performance but not aimed at increasing the productivity of the optimiser developer. Thus, development of optimisers has been unnecessarily complicated.

Complexity of the Object Language Differences between programming languages have impact on the complexity of the analysis and transformation of the object language. An incremental and modular specification of programs transformations is a desirable property. Common language constructs can be handled in a similar fashion and they can constitute a basis for building optimisers.

Effectiveness of the Transformation It is very difficult to predict how effective two different implementations of a particular transformation are. Since it requires the construction of two different transformations systems that operate on the same code.

[1] Meta-language is the language used to process the object language.

The variability conditions of the implementation of a program transformation can provide hints of its effectiveness. A flow-sensitive implementation will be more effective compared to a flow-insensitive one. The same holds for a context-sensitive implementation compared to a context-insensitive one. Another aspect to take into account is whether the combination of transformations at unison would increase the effectiveness of program transformations.

6.6 Challenges for Data-Flow Transformations

In the previous sections, we have reviewed preliminary notions for solving data-flow analysis problems. The focus of this section is on evaluating how a data-flow transformation is performed. For that purpose, we have defined requirements that the construction of a data-flow transformation ideally should satisfy. These requirements include integration of analysis and transformation in a single pass, combination of multiple transformations into a single pass, applicability to multiple languages, program representation and efficiency.

6.6.1 Integration of Analysis and Transformation

Traditionally data-flow optimisations consist of two phases: An initial phase collecting data-flow facts followed by a transformation phase. Although the purpose of the analysis is to provide information to be used in the transformation, the separation of these phases may confuse the whole task. Moreover, this separation requires at least two separate traversals over a program for every optimisation.

In order to decrease the amount of additional storage needed to communicate the result of the analysis, sparse data representations were introduced. These representations compacts data from each program point to those representative program points at which the information is needed [23], thus reducing the required storage for communicating the result of the analysis to the transformation pass.

The integration of analysis and transformation (all in a single program traversal) allows to directly specify the transformation to be performed. This approach is more expressive and avoids the use of auxiliary data storage for communicating the result of the analysis to the transformation phase. Instead, the transformation is done in one traversal. When the enabling conditions are satisfied, they directly trigger the transformation. Most importantly, better results can be obtained than with separate phases: (1) transformation changes a program and thus requires re-analysis (2) application of transformations may change the result of the analysis.

Data-flow analysis and transformation requires time and memory space. By combining and executing these two tasks at once, space and time can be saved. Another advantage is code clarity for the better understanding of the modifications in the program as well for further possible extensions and maintenance of the code.

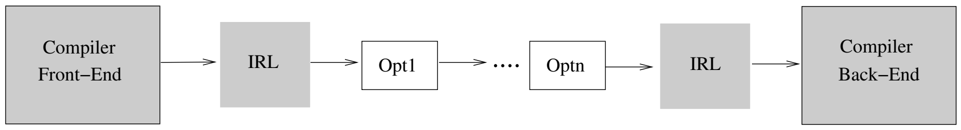

Figure 6.1: Classical compilation scheme.

6.6.2 Combination of Multiple Transformations

The interaction between program transformations can be two-fold. Given two transformations they can interact and cooperate with each other to accomplish some transformation criterion, or they may interfere with each other [111].

An approach to achieve more profitability from program optimisations is to combine transformations that interact with each other to achieve better results. Online partial evaluators combine different program optimisations to achieve good results and more importantly they exploit the result of the optimisations to feed into other optimisations. Online partial evaluators outperform offline partial evaluators [53]. The same principle is used for the combination of transformations that are applied in unison. For instance the combination of constant propagation with constant folding and unreachable code elimination increases the identification of known constants and the elimination of computations at run-time [110]. Other examples for the combination of different transformations were studied in [24, 63].

Although not all transformations can benefit from each other, the construction of superoptimisers can be a task of studying the interaction between transformations and the specification of their combination.

6.6.3 Applicability to Multiple Languages

Every programming language requires an (optimising) compiler or an interpreter to be useful. For the usability of a programming language, a standard compiler will be sufficient, but the end user will always expect the best translation possible for his/her code. Thus, a compiler is expected to apply program optimisations. Compiler technology addresses the problem of developing compilers for different languages by defining common intermediate representation languages (IRL). Figure 6.1 depicts the classical compilation scheme. With this translation scheme, many languages can reuse a compiler back-end and likely some program optimisations. In this way, we only need to build different front-ends for the different programming languages. Thus, the construction of program optimisers can be carried at an intermediate representation language. The compilation scheme provides reusability of compiler optimisations to some extent. Some data-flow analyser generator tools define a language that matches a possible standard low level intermediate representation language and generate data-flow analysis or data-flow (transformers) optimisers for it [69, 94].

On the other hand, source-to-source program transformation systems do not benefit from the compilation scheme. Thus, we consider as an evaluation criterion the applicability to different programming languages.

6.6.4 Data-Flow Representation

Most of the tools for data-flow optimisations define a specific data structure for the representation of programs. This data structure represents the direction of how the data flows in a program at execution time. The flow of a program is mostly represented by a directed Control Flow Graph (CFG). Nodes of a control-flow graph represent a sequence of statements and directed edges are used to represent jumps in the control flow. There are several representation variants based on a graph representation such as *single static assignment* [29], and program dependency graphs [38]. Construction of the control-flow graph requires semantic knowledge of the programming language. Unfortunately, the language representation level of the program is lowered when converted into a CFG representation.

A tree based representation of a program has a higher representation level when compared to control-flow graphs. However, a tree based representation language does not include per se possible execution paths or captures the direction of how the data may flow in a program, thus having to be captured by a tailored program traversal. In this approach part of the semantics of the language is incorporated into a traversal.

Transformations based on a control-flow graph representation are unfortunately not very suitable for source-to-source optimisations. They require an extra transformation to recover the original structure of the program. Another possible approach is to first convert the abstract syntax tree into a control-flow graph representation, transform the program and then reconstruct the representation back into a higher program representation level.

6.6.5 Efficiency

Efficiency of a program transformation is measured in an empirical manner. A program transformation may change different aspects of its behaviour such as execution speed, use of memory, for some embedded applications code size can be an issue to optimise, and even energy consumption is relevant for certain applications. Thus, determining the efficiency of a particular transformation is not an easy task. There are different factors involved that matter, such as the gain obtained with a particular transformation and the resources (time and space) used to compute such a result. There is a trade-off between these two factors. These are implementation decisions however, that have a direct influence while designing an optimisation such as the variability points of solving a data-flow problem. Whether the implementation is flow-sensitive and whether it is intra-procedural or inter-procedural influence the outcome of a transformation.

6.7 Review of Related Approaches

A discussion of techniques for data-flow transformations is beyond the scope of this thesis. Rather, we focus on languages, tools and frameworks that automate part of the effort of producing(generating) programs for data-flow analysers and optimisers.

6.7.1 PAG

PAG [4, 69] is a program analyser generator tool that uses a dedicated functional programming language FULA for the specification of program analyses and DATLA for the specification of data structures. FULA allows the definition of transfer functions, which compute the values corresponding to the analysis at hand for each node in the program. A transfer function relates the values of a program point before the node with the values after such a node. Transfer functions are a standard way to define data analysis problems [73]. This language also allows to define the analysis parameters such as direction flow, the combination (merge) function, the starting node of the abstract syntax tree. The tool does not support the combination of analyses, nor the specification of transformations; applications of analysis and transformations are separated.

6.7.2 Sharlit

Sharlit is a data-flow analyser generator tool [94, 93]. From data-flow generator descriptions it generates C++ classes which perform the actual analysis. The descriptors refer to the flow values and flow functions for the problem at hand, which are specified in C, are of a procedural nature. The data flow analyser generated is based on an iterative solver. The data flow analyser or solver can generate transformations which are specified by *action routines.* Action routines are analogous of a parser generator.

This framework incorporates *path simplification* operations to combine transfer functions and simplify flow paths. It supports a style of optimisation based on a model introduced by Killdal [57]. It combines two or more analyses, but it does not combine several analyses and transformations.

6.7.3 Gospel & Genesis

The Gospel language [111] allows the specification of program transformations by providing two ingredients: preconditions and actions. Preconditions describe the applicability of a transformation. Actions specify how to modify the code. In order to perform a transformation, all the preconditions have to be satisfied. The description of a program transformation combines analysis and transformation. Gospel uses a fixed input language (three operand code), which is a low level intermediate representation language and provides means for easily identifying loops.

The specification of program optimisations is of a flow-insensitive nature; it is not possible to specify transformations that take into account different execution paths. Data dependencies can be specified for loop transformations.

Genesis is an optimiser generator that generates optimisers as described by Gospel, on a one to one basis. Unfortunately, it is not possible to combine multiple transformations, since Gospel does not provide means for the specification of interaction between transformations.

6.7.4 Paleri's work

The optimiser generator of Paleri [78] is inspired by Gospel and Genesis. It aims at the generation of scalar optimisations using dependence relations. The specification language for the optimiser generator allows to define preconditions and actions. An improvement over Gospel is that flow-sensitive optimisers can be generated.

The object language is a based on three-address operand code and it is fixed. This optimiser generator generates code for single optimisers and there is no indication if multiple optimisations can be constructed.

6.7.5 Cobalt and Rhodium

Cobalt and Rhodium [63, 64, 65] are domain specific languages to write optimisations. They allow the development of program transformations that can integrate analysis and transformation. In Rhodium optimisations can be expressed by local rules and they are of context-insensitive nature.

These frameworks allow to specify analysis and transformations and the specifications can be automatically verified by this system. It operates on fixed control-flow graph representations, in a C-like intermediate language.

Furthermore, combination of program transformations is also possible if the transformations share the same flow direction.

6.7.6 David Lacey's work

This work provides a transformation-oriented approach, aiming at declarative specification of individual transformations. The tool developed by Lacey uses graph rewrite rules with temporal logic conditions to check properties of the control-flow graph; that is, enabling conditions are checked from the point of view of the node that is transformed, rather than being based on a global analysis.

The drawback of this approach is that pattern matching requires performing a global program analysis and a search for graph nodes that match a certain pattern. After applying a transformation, the analysis needs to be redone. Obtaining efficient optimisers requires incrementally updating the analysis information after applying transformations. There is some progress in this area [84] with a technique for compositional analysis based on path expressions. Furthermore, it is not possible to combine multiple transformations.

6.8 Stratego Approach

After a review of frameworks for realising data-flow optimisations, in this section we present our implementation of data-flow optimisations in Stratego.

Stratego has two main elements as a language for the construction of applications: rules and strategies. Dynamic rules are used to represent context-sensitive information. Scoped dynamic rules allow the handling of scoping rules in the object language.

Moreover, we introduce operations on dynamic rule sets to allow the propagation of context-sensitive information in the presence of control-flow statements. Strategies allow us to construct traversals for forward and backward propagation. Stratego is not tied to any object language. It also allows us to combine multiple transformations into a single one using a declarative approach.

6.8.1 Program Transformation with Stratego

Stratego provides an abstract interpretation style for data-flow transformations. A program is interpreted for the program facts of interest of a particular program transformation, which in the context of this work is through dynamic rules.

Thus, when the enabling conditions of a data-flow transformation hold, it is possible to exploit all these facts and directly transform the input program, in this way blending analysis and transformation. The mutable behaviour of dynamic rules provides means to reflect changes in the program.

The key idea of a data-flow transformation is to capture information facts and provide this information where it is needed but not present. For instance consider in the code fragment:

```
x := 1
 ...
 ...
z := ... x ..
```

The value of the variable `x` can be captured with a dynamic rule (`x -> 1`) and be applied at the point where the definition of `x` is used. Dynamic rules provide information where it is not available and/or apply the required rewriting. In this case a rewrite rule can rewrite the variable `x` to the value assigned to it. This principle is used in constant propagation. Stratego implementations combine analysis and transformation using a single traversal. Dynamic rules encapsulate variable binding constructs to control the application of rewritings.

The following chapters describe these extensions and illustrate their use in specifying classical data-flow optimisations such as constant propagation, common subexpression elimination, forward expression substitution and dead code elimination. Furthermore, generic frameworks for data-flow propagations are defined. These frameworks encapsulate the traversals and they can be parameterised with different transformations by using different dynamic rules and different operations over those dynamic rules.

6.9 Summary

In this chapter, we have described traditional variants for the implementation of data-flow analyses. The conducted research in this area is mainly focus on developing solutions which seek to improve in performance, efficiency and scalability of data-flow analysis. In contrast we choose to concentrate in other factors such language expressivity, and developer productivity. We also propose challenges for the implementation of data-flow optimisations. The following chapters present Stratego extensions which need to express source-to-source data-flow optimisations concisely. The specification examples are presented using Tiger as the object language, but the specifications are not restricted to this particular language, only requiring small modifications to handle other imperative programming languages.

7 Control-Flow and Dynamic Rule Sets

Data-flow optimisations are usually implemented on low-level intermediate representations. This is not appropriate for source-to-source optimisations, which reconstruct a source level program. In this chapter we show how constant propagation, a well known data-flow optimisation, can be implemented at source code level using Stratego by making use of dynamic rules and operations on dynamic rule sets. A particular feature of the implementation is the integration of analysis and transformation in a single traversal.

7.1 Introduction

Constant propagation is the process of discovering values that are known to be constant on all possible executions of a program and propagating these values through the program [1, 110]. This is a common data-flow optimisation and the techniques for implementing it can be generalised to other transformations that require data-flow analysis such as common sub-expression elimination, copy propagation and dead code elimination. We will introduce operations on dynamic rules sets to allow modelling optimisations that cope with the control-flow of programming languages. With dynamic rule operators, the implementation of optimisations can be described in a clear and concise way.

The constant propagation strategy defined in this chapter integrates analysis and

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

⇒

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

Figure 7.1: Flow-sensitive constant propagation integrating analysis and transformation.

transformation and it has more benefits than performing separate analysis and transformation phases. New obtained results can be immediately used by the transformation. Figure 7.1 shows an example from [63] which illustrates how the application of constant propagation (to determine that `x` is 10 at the condition), gives rise to elimination of code, reducing the `if-then-else` construct to the `then` branch. This elimination avoids considering the `else` branch which would interfere with the knowledge that `x` is 10. We achieve this with the generally applicable features of strategies and dynamic rules.

7.2 Flow-Sensitive Constant Propagation

Flow-sensitive propagation takes the order of execution (flow) of the expressions in the program into account. For illustration purposes let us consider a program consisting of a sequence of assignments. In this situation, constant propagation comes down to propagating constants through the sequence of assignments. It is important that variables are only substituted for constants at program points where the replacement is valid. Invalid points are variables in the left-hand side of assignments and replacing variables that are not known to contain a known constant value.

Consider the transformation shown in Figure 7.2 where it is possible to propagate the value assigned to the variable `a` from the first assignment. Thus the occurrence of `a` in the right-hand side of the second assignment is replaced by `3`. But the

```
(a := 3;
 b := a + 2;
 a := a * 4;
 a := y;
 b := a + b)
```

⇒

```
(a := 3;
 b := 5;
 a := 12;
 a := y;
 b := a + 5)
```

Figure 7.2: Flow-sensitive propagation of constant values.

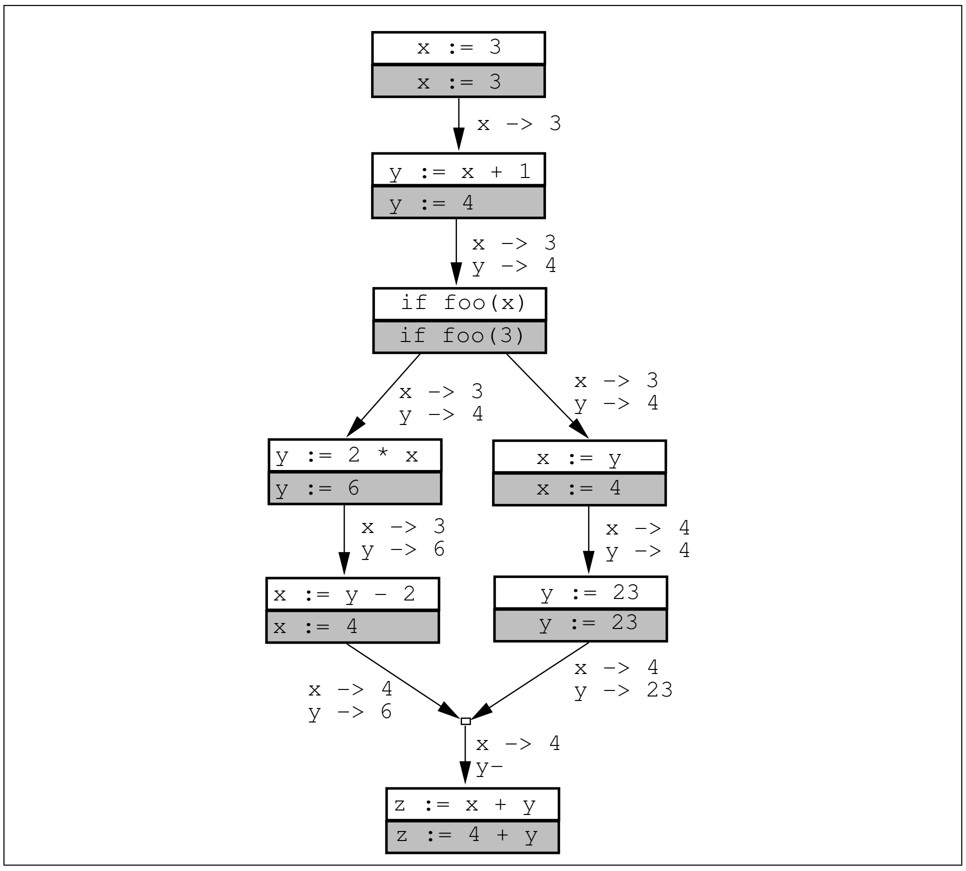

Figure 7.3: Control-flow graph annotated with propagation rules.

occurrence of `a` in the last assignment is not replaced because of the intervening definition of `a` in the previous assignment. Associations between variables and their constant values can be represented with dynamic rewrite rules.

7.2.1 Constant Propagation and Control-Flow

Real programs are more complex than a simple straight line sequence of assignments, since they include control-flow statements. For program transformations with control-flow constructs, we first consider the use of dynamic rewrite rules to propagate data-flow information in a control-flow graph and then argue that this approach can also be applied to abstract syntax trees.

The diagram in Figure 7.3 shows the control-flow graph of the example program in Figure 7.4. The nodes correspond to the assignments and conditions in the program before and after transformation. The traversal of the graph follows the

```
x := 3;
y := x + 1;
if foo(x)
then (y := 2 * x;
      x := y - 2)
else (x := y;
      y := 23);
z := x + y
```

⇒

```
x := 3;
y := 4;
if foo(3)
then (y := 6;
      x := 4)
else (x := 4;
      y := 23);
z := 4 + y
```

Figure 7.4: Example application of flow-sensitive constant propagation.

control-flow of the program, which corresponds to following the direction of the arrows from entry to exit. At nodes with more than a single outgoing edge, the traversal subsequently visits each branch and synchronises information at the merge point. Data-flow facts are represented by a set of dynamic rewrite rules (x -> i) that rewrite an occurrence of a variable to its constant value. Since the set of dynamics rules can be different at each point in the program, the edges of the graph are annotated with the rules that are valid at that point of the traversal.

At each assignment node of the graph, first the right-hand side of the assignment is transformed by rewriting variables to constant values, and if applicable attempting to apply constant folding rules such as the ones defined in Section 4.6.2. For example, y := x + 1 is transformed to y := 3 + 1 by application of the rule x -> 3 and then reduced to y := 4 by constant folding. Next, an assignment *x* := *e* (where *e* is not a constant value) causes the undefinition of any rule with *x* as left-hand side, since these are no longer valid. Finally, if the assignment has a constant value as right-hand side (*x* := *i*), a new rewrite rule *x* -> *i* is defined.

Propagation rules for *different variables* can be applied simultaneously. For example, after the y := x + 1 assignment both rules x -> 3 and y -> 4 are active. However, only one rule can be applied with the same left-hand side. Consider for instance, the assignment x := y which replaces the rule x -> 3 with the rule x -> 4. At a fork in the control-flow, that is at a node with more than one outgoing edge, each branch starts with the set of dynamic rules active at the branching node. Each edge is annotated with that rule-set. At the merge point only those rules that are consistent in all branches are maintained. In the example, the rules for y are inconsistent at the merge point and are not included in the rule-set of the outgoing edges. In the case of loops the search for a valid rule-set is repeated until an stable set of rules is obtained.

A control-flow traversal of a program can also be realised by traversing its abstract syntax tree. This requires visiting the nodes of the tree in the order that they would be visited in a traversal of the graph. The diagram in Figure 7.5 depicts the abstract syntax tree of the example program (Fig. 7.4). Simulation of the control-flow graph basically corresponds to a depth-first left-to-right traversal of the syntax tree [90]. Realisation of the constant propagation transformation on abstract syntax trees thus requires:

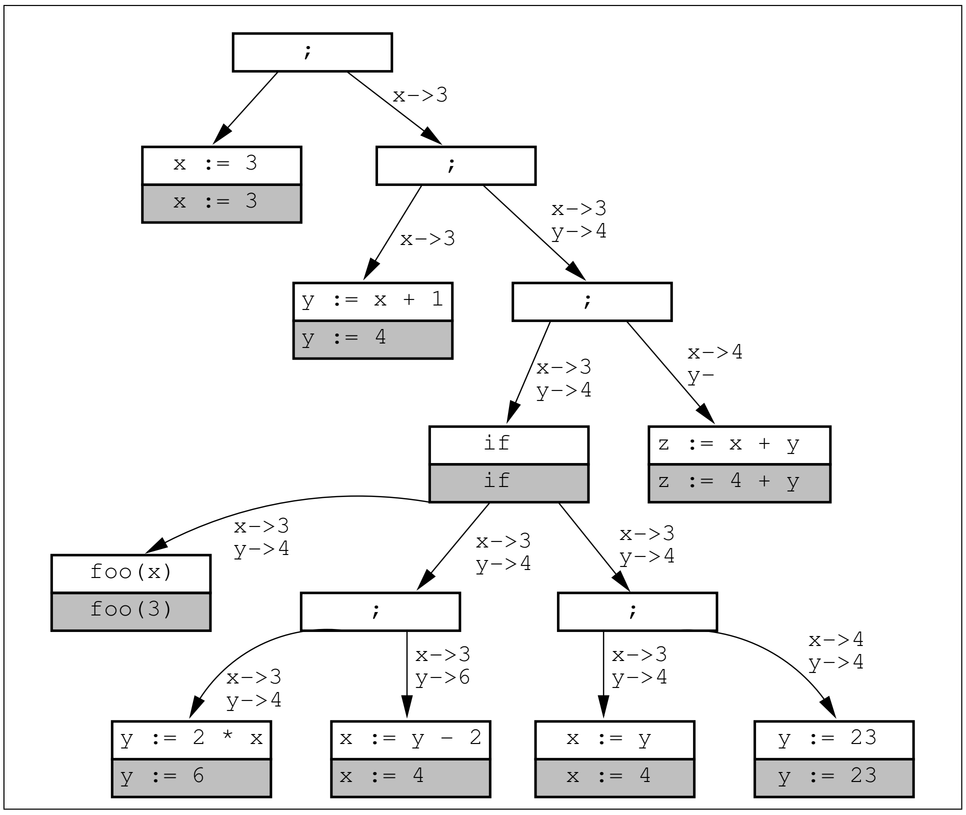

Figure 7.5: Abstract syntax tree annotated with propagation rules.

1. traversal of the abstract syntax tree to visit expressions in the right order
2. dynamic definition of rules to reflect the constant assignments
3. application of dynamic propagation rules and static constant folding rules
4. forking and combining rule-sets to model forks in data-flow

So far we have covered the first three required elements, next we explain how multiple path propagation is accomplished in Stratego. We first consider the use of dynamic rewrite rules to propagate data-flow information in straight line code.

7.3 Transforming Straight-Line Code

Basic transformations on abstract syntax trees can be defined using term rewriting. Term rewriting is declarative since rewrite rules can be independently defined and are automatically applied by a rewriting engine. The correctness of the integrated transformation can be established by the correctness of the individual rules. However, static rewrite rules are not sufficient for defining data-flow transformations.

To extend rewriting with propagation of context-sensitive information requires (1) the dynamic (run-time) definition of rewrite rules and (2) the adequate control of their application. For simplicity, we first consider a reduced version of Tiger which consists of expressions, assignments and sequential composition and then we will extend it to code containing control-flow constructs.

7.3.1 Dynamic Propagation Rules

As the above example illustrates, constant propagation works by replacing a variable occurrence in an expression by the constant value which it is inferred to hold. Since this information is not available at the place where a variable is used, the association between the variable and the value should be established at the place where the variable is defined, i.e., the assignments with constant expressions on their right hand sides. Dynamic rewrite rules [19, 102] were designed for exactly this purpose.

The `AssignPropConst` strategy defined in Figure 7.6 recognises an assignment and generates a dynamic rule `PropConst` which rewrites a variable *x* to the constant expression *e* assigned to it. The `?|[ x := e ]|` construct matches an assignment, and the `is-value` strategy verifies if the expression *e* contains a constant value. If this is the case, the `PropConst` rule is *defined*, otherwise the dynamic rule is *undefined*.

The `rules(PropConst : |[ x ]| -> |[ e ]|)` construct generates a new dynamic rewrite rule `PropConst`, which rewrites *x* to *e*, just like an ordinary rule R `:` p_1 `->` p_2. The difference is that for variables in p_1 and p_2 that are bound in the context of the rule, the bindings are injected from that context. Thus, in the case above, the (meta)variables *x* and *e* are bound when the rule is defined, and not when it is applied. Moreover, multiple `PropConst` rules for different variables may coexist.

The `AssignPropConst` strategy can be applied several times for the same variable while a program is being traversed. Whenever this strategy is applied a new rule is defined redefining a previous rule for the same variable, that is, for the same left-hand side pattern. However, rules for other variables remain untouched. In case the expression *e* assigned to the variable *x* is not a constant, it is necessary to undefine the `PropConst` rule for *x*. For instance, in the previous example the rule `PropConst : |[ a ]| -> |[ 3 ]|` is undefined by the assignment `a := y`. The application of an undefined rule always fails. Thus, attempting to apply this rule to the term `a` fails.

7.3.2 Propagation Strategy

The `prop-const` strategy in Figure 7.6 defines constant propagation for basic blocks using dynamic rules. The strategy has three cases. First, the application of the `PropConst` rule, which replaces a variable with a constant value. Second, when an

```
prop-const =
  PropConst
  <+ ⟦ <id> := <prop-const> ⟧; AssignPropConst
  <+ prop-all

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

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

Figure 7.6: Constant propagation for straight line code.

assignment is encountered, a congruence strategy is used to transform the right-hand side expression by a recursive invocation of the `prop-const` strategy, while leaving the left-hand-side of an assignment untouched (to prevent the replacement of the variable by a value in the left-hand-side). Finally, a generic traversal is performed and the subexpressions are transformed with the `prop-all` strategy. This strategy attempts to evaluate expressions with the discovered constant values by applying constant folding rules `EvalBinOp` and `EvalRelOp`. These rules were previously defined in Figure 4.8.

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

```
(a := 3;            a -> 3                 (a := 3;
 b := a + 2;        b -> 5, a -> 3          b := 5;
 a := a * 4;   ⇒    b -> 5, a -> 12    ⇒    a := 12;
 a := y;            b -> 5, a-              a := y;
 b := a + b)        b-,     a-              b := a + 5)
```

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

7.4 Block-structure (Scope)

Programming languages have scoping mechanisms to restrict the life time of defined entities such a variables. A `let` construct determines the variable scope in Tiger programs. Dynamic rules that propagate values contained in variables should only be valid in the program segment which comprises the scope of a variable. The

```
prop-const =
  PropConst
  <+ |[ <id> := <prop-const> ]|; AssignPropConst
  <+ prop-let
  <+ all(prop-const); try(DeclarePropConst <+ EvalBinOp <+ EvalRelOp)

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

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 7.7: Constant propagation for scoped code.

scoping construct for dynamic rules is defined with {| R : s |}, where R stands for the name of the dynamic rule that is defined and used during the application of the strategy s. Thus, we only have to match the scope of the variable with the scope of the dynamic rule. This is necessary in cases such as the following:[1]

```
let var x := 17
 in let var y := x + 1
     in let var x := y + 1
         in ()
        end;
        x := x - 7
    end;
    print(x + 10)
end
```

$\Rightarrow$

```
let var x := 17
 in let var y := 18
     in let var x := 19
         in ()
        end;
        x := 10
    end;
    print(20)
end
```

Without scoping the assignment of `x` in the inner scope would be used for the value of `x` in the next assignment.

The constant propagation strategy shown in Figure 7.7 extends the strategy of figure 7.6 with the `prop-let` strategy, which restricts the scope of `PropConst` rules. The strategy `|[ let <*id> in <*id> end ]|; {| PropConst: all(prop-const) |}` matches a `let` construct and defines a delimiting scope for rules. In fact, not all rules

[1] Note that we are considering source-to-source transformation and in particular we are avoiding renaming to maintain as much resemblance with the input program as possible.

s	::=	s_1 `/`$f_1, ..., f_n$`\` s_2	fork and intersect
	|	s_1 `\`$f_1, ..., f_n$`/` s_2	fork and union
	|	`/`$f_1, ..., f_n$`\*` s	fix and intersect
	|	`\`$f_1, ..., f_n$`/*` s	fix and union
	|	s_1 `/`$f_1, ..., f_n$`\`$f_{n+1}, ..., f_m$`/` s_2	fork and intersect/union
	|	`/`$f_1, ..., f_n$`\`$f_{n+1}, ..., f_m$`/*` s	fix and intersect/union

Figure 7.8: Extension of Stratego syntax with operations on dynamic rule sets.

defined within a strategy `prop-const` are applicable in the scope of the strategy `prop-const`.

Dynamic rules can be defined relative to a labelled scope. For this purpose the strategy `DeclarePropConst` *labels* the current scope using as label the name of the declared variable (notation: `PropConst+`x). The dynamic rule defined by `AssignPropConst` is relative to the scope of the variable being defined (notation: `PropConst.`x) to ensure that the dynamic rule is still applicable when inner scopes are exited. In the example the rule for `x` defined in the scope for `y` is not removed when leaving the scope of `y`. Note that shadowing of dynamic rules is accomplished in a transparent manner. An impression of labelled scoped rules can be seen in the following diagram, which shows dynamic rule definitions relative to the scope in which the rule operates.

```
let var x := 17
 in let var y := x + 1
     in let var x := y + 1
         in ()
        end;
        x := x - 7
    end;
    print(x + 10)
end
```

```
x -> 17
x -> 17  y -> 18
x -> 19  y -> 18 // shadows x
x -> 19  y -> 18
x -> 17  y -> 18 // leaving scope of inner x
x -> 10  y -> 18
x -> 10          // leaving scope of y
x -> 10
```

Scope labels provides a mechanism to control the life cycle of dynamic rules, that is, definition, undefinition, and rule application. The former two rule operations use a label to identify to which rule set the operation refers. Dynamic rule application does not require a label to identify which rules are applicable at a program point. Thus, a dynamic rule application **has no** different syntax.

7.5 Transforming Code with Control-flow Statements

In the previous sections, we have used dynamic rules to model data-flow facts (known constants). However, we have only considered straight line code in the transformations so far. That is, code without conditionals or loops. In straight line code there is a single execution path. A traversal strategy follows this path and maintains data-flow information along the way in the form of dynamic rewrite rules.

```
prop-const =
  PropConst <+ AssignPropConst <+ DeclarePropConst
  <+ prop-let <+ prop-if <+ prop-while <+ prop-all

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

prop-while =
  |[ while <id> do <id> ]|
  ; (|[ while <prop-const> do <id> ]|; ElimWhile
      <+ (/PropConst\* |[ while <prop-const> do <prop-const> ]|))
```

Figure 7.9: Constant propagation for control-flow statements.

For example, the `PropConst` rule set represents all known propagation facts at the current point in the program at any time during the constant propagation transformation. Real programs do not have a single execution path. Rather, execution forks at conditionals and iterates at loops. Thus, to model data-flow facts using dynamic rules, we need to account for these phenomena. To achieve this we need to fork dynamic rule sets for use in different branches, and join them again when branches merge. These operations are captured in several strategy combinators, which provide exactly the abstractions needed to define data-flow transformations for programs with structured control-flow in a concise manner. Figure 7.8 shows the operators that have added to Stratego for controlling the application of dynamic rules in the presence of control-flow statements. In this section we introduce these fork-and-merge operators, and illustrate their use with constant propagation.

Constant Propagation with Fork Statements. Thus far, we have considered constant propagation with straight line code, possibly with local variables. Figure 7.9 presents the Stratego program for intra-procedural constant propagation with structured control-flow constructs. Interesting about this program is that it combines analysis with transformation in a single traversal, similarly to the conditional constant propagation transformation of Wegman and Zadek [110] and the approach of Lerner et al. [63]. We achieve this with the generally applicable features of strategies and dynamic rules, which are not specific to constant propagation, or even data-flow transformations.

The issues that need to be solved are illustrated by the example in Figure 7.10. (1) Facts that are valid before a condition should be propagated into both branches of the condition. For example, `a` and `z` in the second branch get the value that they have *before* the condition. (2) Within a branch, local facts can be propagated as usual in a basic block. For example, the value of `x` can be propagated within the second branch. (3) Facts that are guaranteed to be the same after execution of each of the branches can be propagated after the condition; facts that are inconsis-

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

⇒

```
let var x := 1
    var y := z
    var z := 3
    var a := 4
 in x := 4;
    a := 5;
    if y then (y := y + 5;
               z := 8)
         else (x := 26;
               y := 27;
               z := 8);
    b := 13;
    z := 8 + x
end
```

Figure 7.10: Example application of flow-sensitive constant propagation.

tent should not be propagated. For example, the value of `a` is unchanged in both branches, so it is the same after the condition. While `z` is changed in both branches, its value is always the same, so it can be propagated. However, the value of `x` is changed in the second branch, and thus its value cannot be propagated afterwards.

From these requirements we can conclude that (1) transformation of the two branches of a conditional expression should start with the same set of dynamic rules. Hence, after propagation in one branch, the rule set should be restored to what it was *before* the conditional in order to correctly propagate in the other branch. (2) Within a branch, transformation proceeds as usual. (3) After the conditional, transformation proceeds with those rules from the rule sets for the branches that are consistent. These requirements are implemented by the s_1 `/`R`\` s_2 strategy operator. The `/`R`\` operator is language independent: it has no knowledge of what are the "branches" that should be treated separately. Instead this notion is expressed in the argument strategies. For example, the strategy expression

```
⟦ if <id> then <prop-const> else <id> ⟧
     /PropConst\ ⟦ if <id> then <id> else <prop-const> ⟧
```

applies the strategy `prop-const` first to the `then` branch of the `if` and then to the `else` branch. Next the resulting `PropConst` rule sets from the branches are intersected to maintain only those propagation rules that are the same in both branches.

The program in Figure 7.9 uses this confluence operator to express constant propagation over structured control-flow constructs. Furthermore, the transformation combines constant propagation with unreachable code elimination, which is a powerful combination [63, 110]. For example, the transformation in Figure 7.1 shows how the application of constant propagation gives rise to elimination of code, reducing the `if-then-else` construct to the `then` branch. This elimination avoids to consider the `else` branch which would interfere with the knowledge that `x` is `10`.

This is achieved by the following strategy expression

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

which first applies `prop-const` to the condition of the `if` expression. Then it tries to apply the `ElimIf` rule, which may discard one of the branches, after that the resulting branch can be transformed with an application of the `prop-const` strategy. If the conditional cannot be reduced, the intersection is invoked, instead.

Constant Propagation with Loops. The execution flow of a loop has an reentering path. As a motivating example for constant propagation on loops, consider the transformation:

```
(a := 5;                                  (a := 5;
 b := 4;                                   b := 4;
 c := 8;                                   c := 8;
 d := 5;                                   d := 5;
 while (c < 100) | x                       while (c < 100) | x
 do( b := a;                        ⇒      do( b := 5;
     e := a + b;                               e := 10;
     a := e * d + c;                           a := (50 + c);
     c := c + e;                               c := (c + 10);
     e := e + c;                               e := (10 + c);
     a := b );                                 a := 5);
 f := a + b + d + c + e)                   f := (15 + c + e))
```

In this example the variables `a, b, c` and `d` contain constant values before the while statement. This example illustrates the following situations:

1. `a` contains a constant value, regardless of the execution of the while statement.
2. `b` is defined in the body of the while, if the body of the while is executed at least once, b contains the value of `a`, i.e., `5`, and it can be propagated outside the while statement.
3. `c` does not contain a known constant value because it is iteratively defined in the while body.
4. `d` contains a constant value since it is not defined in the body of the while.
5. `e` contains a constant value from its first definition until its second definition locally in the while loop. After the while `e` does not contain a constant value.

When the assignment to `f` is reached, the variables `a`, `b` and `d` contain constant values. The implementation of constant propagation has to cover all these cases. To discover that the value of `a` contains a constant value, we repeatedly inspect the while body until an stable rule set is reached. This analysis corresponds to the reentering path of the loop.

An extension of the program to cover loop statements has to consider any number of iterations. Therefore, the application of the transformation to the loop is iterated until an stable rule set is obtained. This is expressed using the fixed-point operator `/`R`\*` s, which repeats the application of s until no more changes in the rule set R occur. For example, in the constant propagation transformation, the strategy

```
    /PropConst\* |[ while <prop-const> do <prop-const> ]|
```

expresses the fixed-point iteration over a while-loop. The program in Figure 7.9 uses this intersection operator to express constant propagation over structured control-flow iteration constructs.

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

Finally, the transformation is enhanced with unreachable code elimination, which gives the effect of conditional constant propagation as illustrated in Figure 7.9. This is achieved by the following strategy definition:

```
prop-while =
  |[ while <id> do <id> ]|
  ; (|[ while <prop-const> do <id> ]|; ElimWhile
     <+ (/PropConst\* |[ while <prop-const> do <prop-const> ]|))
```

which first applies the strategy `prop-const` to verify if it is possible to eliminate the while loop if it can be determined that it is unreachable code. The rule `ElimlWhile` attempts to remove the `while` loop if the condition evaluates to false.

7.5.1 Generalised control-flow operators

The strategy construction in its most general form s_1 `/`$f_1, ..., f_n$`\`$f_{n+1}, ..., f_m$`/` s_2 allows to fork the application of the different sets of active dynamic rules for the rule names $f_1, ..., f_m$. Intersection and/or union are operations applicable to the rule sets. As an instance consider the strategy s_1`/`R_1, R_2`\`R_3, R_4`/`s_2 which applies the strategies s_1 and s_2 in sequence while using the resulting rule sets of the operation applied to the rule sets. The operation on the rule sets R_1, R_2 is intersection and on the rule sets R_3, R_4 is union. The description of the semantics of these strategy operators and its efficient implementation is described and formalised in [19, 18].

A related operator is `/`$f_1, ..., f_n$`\`$f_{n+1}, ..., f_m$`/*` s which captures *fixed-point iteration* over rule sets. This dynamic rule operator also applies operations at confluence points, namely intersection and union until a fixed-point on the result of rule sets is achieved. The strategy `/`R_1`\`R_2`/*` s applies strategies s while using the obtained rule sets of active dynamic rules R_1 and R_2. It applies the operations intersection/union and it repeats the operation until there is no change in the rule sets.

Although in this thesis we have not used the full potential of the generalised dynamic rule combinator, we believe it can be used in the implementation of data-flow optimisations which could require to apply different operations on dynamic rule sets at confluence points.

7.6 Summary

In this chapter we have shown how a data-flow transformation can be implemented in Stratego by combining a control-flow traversal with dynamic rules for the collection and propagation of program facts.

We will discuss in following chapters how this data-flow transformation can be generalised to propagate facts that involve more variables as in common subexpression elimination.

8 Dependent Dynamic Rules

Dynamic rewrite rules were used to propagate data-flow facts concerning a single variable in the implementation of constant propagation. Data-flow optimisations such as copy propagation and common subexpression elimination involve more than a single variable for their realisation. To implement such optimisations, we introduce dependent dynamic rewrite rules which enables us to implement optimisations which involve more than a single variable.

8.1 Introduction

In the previous chapter we have seen how constant propagation can be specified with dynamic rewrite rules which propagate information about a single variable. We have described the correct treatment of lexically scoped variable bindings to restrict the application of transformation rules to the scope where the bindings are valid. While dynamic rules as presented in the previous chapter can be used to implement constant propagation, they are not sufficient to describe any kind of data-flow sensitive program transformations. In constant propagation, a propagation rule maps a variable to a constant value. Propagation rules are invalidated by encountering a new assignment in the control flow of the program. However, in optimisations such as common subexpression elimination there are multiple variables involved in a propagation rule.

This chapter describes several data-flow transformations that require control over variable interdependencies. Examples of such transformations are copy propagation, common subexpression elimination, and forward substitution. In particular, we introduce the concept of *dependent dynamic rewrite rules* to model the dependencies amongst variables. We illustrate the problems when more than a single variable is involved by using copy propagation as the running example. After that, we discuss the application of the newly introduced concept in common subexpression elimination, forward expression substitution and dead code elimination.

8.2 Copy Propagation

Statements of the form `x := y` indicate that `x` is a copy of `y`. Occurrences of `x` can be replaced by `y` at a program point, if there are no intervening definitions which invalidate that `x` and `y` refer to the same value. Consider the example of a copy propagation transformation:

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

The first statement makes the variable `a` a copy of `b` and this predicate holds until either the variable `a` or `b` is associated with a different value. When evaluating the right hand side of the assignment `c := a` we know that `a` and `b` refer to the same value, and can thus safely replace `a` by `b`. This information can be used to replace occurrences of `c` by `b`. The third assignment assigns a new value to `a`. This assignment invalidates the copy replacement of `a` by `b`. The last assignment has the effect that `c` and `b` are not longer copies.

Copy propagation is a variation of the constant propagation and it can be captured by the dynamic rule `CopyProp : ⟦ x ⟧ -> ⟦ y ⟧`. Here we present the previous example with an additional box showing the copy propagation rules:

```
( a := b;          a -> b             ( a := b;
  c := a;      ⇒   a -> b  c -> b  ⇒    c := b;
  a := y + x;      a-  c -> b           a := y + x;
  b := x + a )     a- c-                b := x + a )
```

The first and second assignments create the copy rules `a -> b` and `c -> b`. The third assignment changes the value of `a` and the rule `a -> b` is invalidated (represented as `a-`). The last assignment must invalidate the rule `c -> b`. The inspection of the assignment `b := x + a` does not give information to invalidate the rule `c -> b`. In Stratego rule invalidation needs the term in the left hand side of the rule `CopyProp.c :- ⟦ c ⟧`. For a copy propagation to be valid it demands no changes in the variables involved in the rule. Not being able to invalidate the applicability of the rule `c -> b` may certainly introduce erroneous replacements.

```
copyprop-assign =
  ? |[ x := y ]|
  ; if <not(eq)>(x,y)
    then rules( CopyProp.x : |[ x ]| -> |[ y ]| )
    else rules( CopyProp.x :- |[ x ]| ) end
```

Figure 8.1: Incorrect definition of copy propagation.

8.3 Dependencies Between Variables

For the specification of copy propagation, the dynamic rule definition in Figure 8.1 is a natural modification to the rule given for constant propagation. The strategy `copyprop-assign` matches with an assignment between two variables. If the variable occurrences are not the same, a dynamic rule `CopyProp` is defined. This rule replaces occurrences of variables in an expression by an occurrence of another variable. Here, we assume that the `copyprop-assign` strategy is embedded in a similar traversal strategy as that of constant propagation. However, it is incorrect in a number of ways:

(1) Insufficient Dependencies. The dynamic rule is not undefined when the variable in its right-hand side occurs in the left hand side of an assignment. For example, in the program sequence:

```
(a := b;           a -> b           (a := b;
 b := foo();   ⇒   b-   a -> b   ⇒   b := foo();
 c := d + a)       c-   a -> b       c := d + b)
```

The incorrect rule will replace the variable `a` in the last statement by `b` even though `b`'s value changed in the second statement. Thus, a `CopyProp` rule should be undefined when either variable in its pattern or its result is occurring in the left hand side of an assignment.

(2) Free Variable Capture. The rule is not undefined when a local variable shadows the variable in the right-hand side. The following example illustrates this issue:

```
let var a                 a-               let var a
    var b                 b-                   var b
 in a := b;               a -> b            in a := b;
    let var b := foo() ⇒  a -> b  b-  ⇒        let var b := foo()
     in print(a)          a -> b                in print(b)
    end                                        end
end                                        end
```

The occurrence of `a` in the call to `print` will be replaced with `b`, which refers to the variable in the inner `let`. Thus, a `CopyProp` rule should be undefined in a local scope when the local variable is used in the rule.

(3) Escaping Variables. The rule is not undefined (invalidated from the active set of rules) when its target goes out of scope. For an example consider the following program fragment:

```
let var a
 in let var b := foo()
     in a := b
    end;
    print(a)
end
```

⇒

```
a-
b-
a -> b
a -> b
a -> b
```

⇒

```
let var a
 in let var b := foo()
     in a := b
    end;
    print(b)
end
```

The variable `a` is replaced by `b`, which is then used outside the scope where it is defined. This suggests that a `CopyProp` rule should be defined in the local scope. However, consider the variant of the program:

```
let var a
    var c
 in let var b := foo()
     in a := b;
        a := c
    end;
    print(a)
end
```

⇒

```
a-
c-
b-
a -> b
a -> c
a -> c
a -> c
```

⇒

```
let var a
    var c
 in let var b := foo()
     in a := b;
        a := c
    end;
    print(c)
end
```

The last assignment leads to a copy propagation rule which *can* be applied in the outer scope, since neither `a` nor `c` are declared in the inner scope. Thus, a `CopyProp` rule should be restricted to the *innermost* scope in which one of its variables is in scope.

This sums up the problems with the naive use of dynamic rules. The first two problems are solved by means of *dependent dynamic rules*. The last problem is solved by defining rules in the innermost scope of all variables concerned.

8.3.1 Dependent Dynamic Rules

The construction `rules(` $R.l : p_1$ `->` p_2 `depends on` $d_1 ... d_n$`)` defines a dependent dynamic rule R, with scope label l and dependencies $d_1 ... d_n$. The `depends on` clause declares a list of dependencies consisting of pairs of the scope label and the dependency term. The dependencies provide extra means to control the applicability of rules. Figure 8.2 shows the syntax for operations of dependent dynamic rules. In the case of the Tiger transformations, variable names are used to label scopes and as well dependencies. The dependencies are used to undefine or *shadow* dynamic rules. That is, for each dependency a *mapping back to the rule is generated and maintained.* Thus, if the meaning of the dependency is changed, for instance through an assignment, the affected rule can be found and undefined.

s	::=	`new-dynamic-rules(`$\vert$ $rsig, l, dep$`)`	dynamic rule initialization
	$\vert$	`rules (`ddr_1 ... ddr_n`)`	dynamic rule definition
	$\vert$	`undefine-dynamic-rules(`$\vert$ $rsig, dep$`)`	dynamic rule undefinition
	$\vert$	`innermost-scope(`$s\vert$ $rsig$`)`	innermost scope label
ddr	::=	dri : p_1 `->` p_2 `depends on` d_1... d_n (`where` s)?	dynamic rule definition
	$\vert$	dri `:+` p_1 `->` p_2 `depends on` d_1... d_n (`where` s)?	dynamic rule extension
dri	::=	$rsig$`.`l	relative to labelled scope
d	::=	`(`l`,`dep`)`	dynamic rule dependency
l	::=	p	dynamic rule label
dep	::=	p	dependent term

Figure 8.2: Extension of Stratego syntax with dependent dynamic rules.

The `new-dynamic-rules(|`*`R`*` , `*`l`*` , `*`dep`*` )` strategy must be used for initialising a rule *`R`* when entering a new scope for a dependency *`dep`* with scope label *`l`* . This strategy labels the current scope with *`l`* and *locally* undefines any rule *`R`* that depends on *`dep`* , thus avoiding such rules from being applied with the risk of variable capture. To undefine all R rules depending on *`dep`* the strategy `undefine-dynamic-rules(|`*`R, dep`*` )` should be used. Using these strategies consistently guarantees that rules are undefined when any dependency is affected.

Finally, the problem of escaping variables is solved by defining a dynamic rule in the proper scope, which is the first scope label (starting with the innermost scope label) that is related to dependencies of the rule. This ensures that the rule is removed as soon as one of its dependencies goes out of scope. For the dynamic rule R the strategy `innermost-scope(`s`|`R`)` gets the first scope label for which s succeeds.

A correct definition of copy propagation using the newly introduced language constructs is presented in Figure 8.3. Note that the traversal part resembles the one of constant propagation. The main difference is the use of dependent dynamic rules and the strategy `innermost-scope` to refer to the proper scope where a rule operation is defined. The implementation uses the strategies:

```
repeat1(s)    =  s ; (repeat1(s) <+ id)
elem-of(|xs)  =  where(?x;<elem>(x, xs)); !x
```

The strategy `repeat1` succeeds if the parameter strategy s can be successfully applied at least once. The strategy `elem-of` takes as a term parameter a list of variables `xs`. This strategy succeeds yielding the current term (`x`) if it is contained in the list of terms `xs`.

8.4 Common Subexpression Elimination

Common subexpression elimination (CSE) is a transformation that avoids to evaluate expressions that were previously evaluated. The evaluation of an expression e is redundant if there is at least one other occurrence of the expression e from which the second occurrence is reached and none of its operands are assigned along the

```
copy-prop =
  repeat1(CopyProp) <+ AssignCopyProp <+ DeclareCopyProp
  <+ copy-prop-let <+ copy-prop-if <+ copy-prop-while <+ all(copy-prop)

DeclareCopyProp =
  |[ var x ta := <copy-prop => e> ]|
  ; where( <new-dynamic-rules(|["CopyProp"], x, x)> x )
  ; where( try(<AssignCopyProp-aux> |[ x := e ]|) )

AssignCopyProp =
  |[ x := <copy-prop> ]|
  ; undefine-dynamic-rules(|["CopyProp"], x)
  ; where( try(AssignCopyProp-aux) )

AssignCopyProp-aux =
  ? |[ x := y ]|
  ; where( <not(eq)>(x,y) )
  ; where( innermost-scope(elem-of(|[x,y])| ["CopyProp"]) => z )
  ; rules( CopyProp.z : |[ x ]| -> |[ y ]| depends on [(x,x), (y,y)] )

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

copy-prop-if =
  |[ if <copy-prop> then <id> else <id> ]|
  ; ( |[ if <id> then <copy-prop> else <id> ]|
      /CopyProp\ |[ if <id> then <id> else <copy-prop> ]|)

copy-prop-while =
  |[ while <id> do <id> ]|
  ; (/CopyProp\* |[ while <copy-prop> do <copy-prop> ]|)
```

Figure 8.3: Implementation of copy propagation with dependent dynamic rules.

path. An example of common subexpression elimination is:

```
let
    var x := foo()
    var y := bar()
    var z := foo()
    var a := 11
  in
    ( y := z + x;
      a := z + x;
      if q then x := 14;
      z:= z + x )
end
```

⇒

```
let
    var x := foo()
    var y := bar()
    var z := foo()
    var a := 11
  in
    ( y := z + x;
      a := y;
      if q then x := 14;
      z := z + x )
end
```

Common subexpression elimination identifies assignments such as `y := z + x` and replace subsequent occurrences of the expression `z + x` with the variable `y` provided the value of the expression can be guaranteed to be the same. The replacement is valid during a program segment not containing definitions for the variables `z`, `x` and `y`. In the example, the second occurrence of the expression `z + x` can thus safely be replaced by `y`. The third occurrence of this expression is not considered safe for a replacement because in the branch of the `if` statement the variable `x` gets a new value assigned. This transformation needs to keep track of the same dependencies between variables as in copy variable propagation.

Figure 8.4 presents the implementation of common subexpression elimination using dependent dynamic rewrite rules. The specification is restricted to replace expressions that are pure and not trivial, i.e., expressions that are guaranteed to be side-effect free and do not contain the variable being defined. The `pure-and-not-trivial` strategy is defined as:

```
pure-and-not-trivial(|x) =
  not(?|[ i ]| <+ ?|[ y ]|)
  ; pure
  ; not(oncetd(?|[ x ]|))

pure =
  ?|[ i ]| <+ ?|[ x ]|
  <+ |[ <bo:id>(<pure>, <pure>) ]|
  <+ |[ <ro:id>(<pure>, <pure>) ]|
```

The `pure-and-not-trivial` is a recursive strategy which succeeds for integer expressions, variables, and arithmetic (*bo*) or relational (*ro*) expressions of pure subexpressions. The strategy `pure` uses congruences defined for binary and relational operators in Figure 4.6.

The common sub-expressions are matched at assignments, the right hand side of which can contain simple or complex expressions. The propagation strategy replaces the expressions such they are found at assignment level and properties of

```
cse =
  repeat1(CSE) <+ AssignCSE  <+ DeclareCSE
  <+ cse-let <+ cse-if <+ cse-while
  <+ all(cse)

DeclareCSE =
  ? |[ var x ta := <cse => e> ]|
  ; where(new-dynamic-rules(|["CSE"], x, x))
  ; where(try(<AssignCSE-aux> |[ x := e ]|))

AssignCSE =
  |[ x := <cse> ]|
  ; where(undefine-dynamic-rules(|["CSE"], x))
  ; where(try(AssignCSE-aux))

AssignCSE-aux =
  ? |[ x := e ]|
  ; where(<pure-and-not-trivial(|x)> |[ e ]|)
  ; where(get-var-names => vars)
  ; where(innermost-scope(elem-of(|vars)|["CSE"]) => z)
  ; where(get-var-dependencies => xs)
  ; rules(CSE.z : |[ e ]| -> |[ x ]| depends on xs)

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

cse-if =
  |[ if <cse> then <id> else <id> ]|
  ; ( |[ if <id> then <cse> else <id> ]|
      /CSE\ |[ if <id> then <id> else <cse> ]|)

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

Figure 8.4: Implementation of common subexpression elimination.

operators such as commutativity are not handled. However the implementation can be extended in two ways: 1) a preliminary phase that converts expressions into a canonical form yielding assignments of simpler expressions with at most one operator, and 2) rewrite rules that take care of commutative operators.

The traversal strategy for common subexpression elimination is similar to the ones used in constant propagation and copy propagation. The only difference in the implementation of this strategy is the name of the dynamic rule. This code duplication will be addressed in the next chapter.

8.5 Forward Expression Substitution

Forward expression substitution is the inverse of common subexpression elimination. The transformation replaces variables by expressions that were used to compute their value. Thus, this is not an optimisation; it may introduce redundant computations. This transformation is part of the preliminary transformations that prepares the code for data dependency analysis used in vectorisation [2]. Consider the following example adapted from [2] and encoded in Tiger:

```
for i := 1 to 100 do
  ( k := i + 2;
    a[k] := a[k] + 5 )
```

⇒

```
for i := 1 to 10 do
  ( k := i + 2;
    a[i + 2] := a[i + 2] + 5)
```

In this code fragment, the programmer has factored and lifted the common subexpression `i + 2`. Thus, the variable `k` is an *auxiliary induction variable* and it could interfere with other analyses such as a data dependency analysis. Forward expression substitution finds expressions such as `i + 2` and introduces redundant computations as is shown in the code fragment on the right. Moreover, a follow-up transformation can eliminate the assignment `k := i + 2` in the body of the `for` from the code fragment, provided that there are no uses of the variable `k` outside the scope of the `for` expression.

The implementation of forward expression substitution is shown in Figure 8.5. The dependent dynamic rule `(FES.`*z*` : |[ `*x*` ]| -> |[ `*e*` ]| depends on `*xs*`)` is used to propagate the redundant expression and has the opposite direction of the common subexpression rule. Such as is the case with the `CSE` rule, the `FES` rule is valid while there are no interfering definitions of variables involved in the rule. The implementation shares considerable similarities with common subexpression elimination. The principal difference is in the specification of the propagation rule. The traversal strategy for forward expression substitution is the same as the previous data-flow transformations.

8.6 Dead Code Elimination

A program can be improved by removing dead code. Dead code is code that does not affect program behaviour and thus it can be removed. Dead code can arise in

```
fes =
  repeat1(FES) <+ AssignFES <+ DeclareFES
  <+ fes-let <+ fes-if <+ fes-while <+ all(fes)

DeclareFES =
  |[ var x ta := <fes => e> ]|
  ; where(<new-dynamic-rules(|["FES"], x, x)> x)
  ; where(try(<AssignFES-aux> |[ x := e ]|))

AssignFES =
  |[ x := <cse> ]|
  ; where(undefine-dynamic-rules(|["FES"], x))
  ; where(try(AssignFES-aux))

AssignFES-aux =
  |[ x := <fes => e> ]|
  ; where(<pure-and-not-trivial(|x)> |[ e ]|)
  ; where(get-var-names => vars)
  ; where(innermost-scope(elem-of(|vars)| ["FES"]) => z)
  ; where(get-var-dependencies => xs)
  ; rules(FES.z : |[ x ]| -> |[ e ]| depends on xs)

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

fes-if =
  |[ if <fes> then <id> else <id> ]|
  ; (|[ if <id> then <fes> else <id> ]|
      /FES\ |[ if <id> then <id> else <fes> ]|)

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

Figure 8.5: Implementation of forward expression substitution.

many properly written programs. This can occur after applying program transformations such as inline expansion, constant propagation and common subexpression elimination. Another source of dead code is code reuse which was written in a general way, where a particular instantiation may discover that only certain data-flow paths will be computed at execution time.

There are different criteria on how to determine whether certain statements are dead. Unreachable statements can safely be eliminated. It is also possible to remove computations of which results are never used. There is a gray area where memory stores are involved. Most compilers are conservative and assume stores to memory are critical. For the implementation presented in this section we only consider scalar dead code elimination. Thus, we consider as dead code all those statements that compute values that are not used. Consider the example:

```
let var b := read()
    var a := read()
    var x := a
    var y := b
 in x := y;
    let var x := a + y + y
        var t
     in t := x + y;
        y := a
    end;
    b := a;
    print(b)
end
```

⇒

```
let var b := read()
    var a := read()
 in b := a;
    print(b)
end
```

The example shows many statements computing values that are not used. The inner `let` locally declares the variables `x` and `t`. The only value that escapes the inner `let` is the value computed for `y`, which is not used by the rest of the program. All the computations in the inner `let` are dead and can be removed. The assignment `x := y` can also be removed since there is no use of `x` in the rest of the program.

For a simple implementation of dead code elimination we will separate two aspects of dead code elimination: dead assignment elimination and dead variable declaration elimination.

8.6.1 Dead Assignment Elimination

Dead assignment elimination searches for those statements which do not **need** to be computed. An assignment can be removed if the variable in the left-hand-side does not have the needed mark. The removal of assignments can only be done provided that there are no side-effects in the right-hand-side.

The traversal of dead assignment elimination visits the code from the end towards the beginning. During this traversal, it marks variables with the property *needed.*

A variable is needed if a computation uses its value. In the previous example, the statement `print(b)` needs the value of `b`. With this information we proceed marking statements that need to be computed. In a backward traversal flow, the statement `b := a` is reached. The left hand side of the assignment is inspected first followed by the right hand side. The assignment is *needed* because the left-hand-side of the assignment was marked as *needed.* Dead assignment elimination proceeds traversing the right-hand-side of the assignment after marking the variable `b` as not needed or removing the property *needed* of this variable. The variable `a` is marked as *needed* when the traversal reaches the right-hand-side of the assignment. After that, the statements in the inner `let` are visited. Note that none of the assigned variables in the `let` are marked as *needed*, and thus can be safely removed. Next the traversal visits the assignment `x := y` and the variable `x` is not marked as *needed* and the assignment is removed. An example of this transformation is:

```
let var b := read()
    var a := read()
    var x := a
    var y := b
 in x := y;
    let var x := a + y + y
        var t
     in t := x + y;
        y := a
    end;
    b := a;
    print(b)
end
```

⇒

```
let var b := read()
    var a := read()
    var x := a
    var y := b
 in let var x := a + y + y
        var t
     in
    end;
    b := a;
    print(b)
end
```

The resulting code contains declarations that are not longer needed. These declarations can be removed from the resulting code by dead declaration elimination. This transformation is explained in Section 8.6.2.

Realisation of Dead Assignment Elimination

Figure 8.6 shows the Stratego implementation of dead assignment elimination. This transformation requires proper scope handling of the declared variables. This can only be achieved by a forward traversal. Scoping of variables is handled similar to the already explained transformation that uses a forward-flow propagation. However, for a backwards transformation statements of a sequence are traversed in a reverse order by the strategy `reverse-filter`. This strategy traverses a list of statements in a reverse order while applying the strategy `dce-asg` and filters out empty sequences of statements.

For marking variables with the property *needed*, the dynamic rule `Needed` is used. This rule propagates the *needed* property of variables. If the rule is defined, it indicates that the variable is needed in the program. The absence of this property

```
dce-asg =
  VarNeeded  <+ ElimAssignment <+ dce-assign <+ dce-let(dce-asg)
  <+ dce-seq(dce-asg) <+ dce-if <+ dce-while <+ all(dce-asg)

dce-let(dce-asg) =
  |[ let <*id> in <*id> end ]|
  ; {| Needed
     :  |[ let <*map(declare-not-needed)> in <*dce-stats(dce-asg)> end ]|
    |}

declare-not-needed =
  (? |[ var x ta := e ]| <+ ? |[ var x ta ]|)
  ; new-dynamic-rules(|["Needed"], x, x)
  ; undefine-dynamic-rules(|["Needed"], x)

dce-stats(dce-asg) =
  reverse-filter(dce-asg; not(? |[ () ]| ))

dce-seq(dce-asg) =
   |[ (<* dce-stats(dce-asg)>) ]|

VarNeeded =
  ? |[ x ]|
   ; rules(Needed.x: |[ x ]| -> |[ x ]| depends on [(x, x)])

ElimAssignment :
   |[ x := x ]| -> |[ () ]|

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

dce-assign =
  ? |[ x := e ]|
  ; undefine-dynamic-rules(|["Needed"], x)
  ; |[ <id> := <dce-asg> ]|

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

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

Figure 8.6: Implementation of dead code assignment elimination.

or its unsuccessful application indicates that a following[1] definition of this variable can be removed.

When the traversal visits a `let` expression, it defines new dynamic rules for the variables declared by the `let` expression. The strategy `dce-let` initialises the scopes for the dynamic rules. The definitions of scopes is done in *forward* fashion. In contrast, the expression(s) in the `let` are traversed in a *backward* traversal flow.

The strategy `declare-not-needed` matches a variable declaration and it initialises the rule for that variable. As soon as the rule is initialised is undefined. The property *needed* of a variable can change when the variable is defined in the traversal. The strategy `VarNeeded` marks a variable as needed when it is successfully applied.

The elimination of assignments is accomplished with the rule `ElimAssignment`. On the other hand, the strategy `dce-assign` keeps needed assignments. This strategy removes the *needed* mark for the variable in its left-hand-side, and then it proceeds traversing the right-hand-side.

The Tiger control-flow constructs are handled in a similar fashion as the forward transformations but they use different operators for merging the execution paths of the constructs. The `dce-if` strategy applies the operation union `(\Needed/)` over the branches of the `if` statement. And the strategy `dce-while` applies the fixed-point union operator (`\Needed/*`) over the dynamic rules.

Applying a fixed-point union operator seems dangerous as it might not terminate in all situations. But for the transformation at hand this is not the case. The `while` constructs uses a limited number of variables which can be defined or undefined. The fixed-point operator will iterate a finite number which does not exceed the number of variables in the construct. Thus, for the implementation of dead assignment removal this operator is sound and it will always terminate.

8.6.2 Dead Variable Declaration Elimination

This transformation removes variable declarations for variables not longer used in a program. Consider the example:

```
let var b := read()
    var a := read()
    var x := a
    var y := b
 in let var x := a + y + y
        var t
     in
    end;
    b := a;
    print(b)
end
```

⇒

```
let var b := read()
    var a := read()
 in b := a;
    print(b)
end
```

[1] The following definition refers to the definition that is visited next in a backward traversal fashion.

The transformation visits the program in backward direction marking *uses* of variables in the program. First it marks the variables `b` and `a`. Next it traverses the body of the inner `let` without finding other uses of variables. Thus, all the declarations of this `let` can be removed. The resulting empty `let` can also be safely removed. A backward traversal reaches the declarations of the outer `let` and it keeps variable declarations for which the property *used* is set.

Realisation of Dead Declaration Elimination

The implementation shown in Figure 8.7 shares many similarities with dead assignment elimination. It uses the same traversal and a rule to propagate the property *used*. It defines scoped labelled dependent rules with a similar scope as the variables defined in a Tiger program. The declarations are traversed in a forward fashion, and the body of the `let` is traversed in backward fashion.

The rule `Used` propagates the property *used* of a variable to the declarations, which are visited twice. The first time to initialise the dynamic rule and the second time to decide whether the declaration can be removed from the program. The strategy `keep-declarations` does not remove the declaration if the declared variable is used in the program. This strategy is the parameter of `reverse-filter` which applies the `keep-declarations` strategy and filters out the declarations which are no longer required. The resulting code is cleaned up with the unreachable code elimination rules `ElimIf` and `ElimWhile` already shown in Figure 4.9. The rule `ElimLet` is defined as:

```
ElimLet : |[ let in end ]| -> |[ () ]|
```

The definitions for the strategies `dce-stats(s)` and `dce-seq(s)` are already presented in Figure 8.6. The control-flow constructs of Tiger are treated in the same way as for dead assignment elimination.

8.7 Summary

This chapter describes the need of dependent scoped dynamic rules to achieve clean implementations of context-sensitive data-flow transformations at source code level. With the introduction of dependent dynamic rules the dependencies between variables can be handled in a proper way. The rules alleviate the work of handling language scope and take aspects of a language such as variable shadowing and variable capture into account. The following chapter deals with the code duplication and defines generalised frameworks for data-flow transformations.

```
dce-decl =
  (VarUsed <+ dce-seq(dce-decl) <+ dce-let(dce-decl) <+ all(dce-decl))
  ; try(ElimIf <+ ElimWhile <+ ElimLet)

dce-let(dce-decl) =
  |[ let <*id> in <*id> end ]|
  ; {| Used
    : |[ let <*map(declare-not-used)> in <*dce-stats(dce-decl)> end ]|
    ; |[ let <*reverse-filter(keep-declarations(dce-decl))> in <*id> end ]|
    |}

declare-not-used =
  (?|[ var x ta := e ]| <+ ?|[ var x ta ]|)
  ; new-dynamic-rules(|["Used"], x, x)
  ; undefine-dynamic-rules(|["Used"], x)

keep-declarations(dce-decl) =
  ?|[ var x ta := e ]|
  ; where(<Used> |[ x ]|)
  ; |[ var x ta := <dce-decl>]|

keep-declarations(dce-decl) =
  ? |[ var x ta ]|
  ; where(<Used> |[ x ]|)

VarUsed =
  ? |[ x ]|
   ; rules(Used.x: |[ x ]| -> |[ x ]| depends on [(x, x)])

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

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

Figure 8.7: Implementation of dead variable declaration elimination.

9 Reusing and Combining Data-flow Transformations

Generic traversals for forward and backward data-flow propagation are designed in such a way that they can be reused to implement data-flow transformations. Different instantiations of the generic strategies for the implementation of constant propagation, copy propagation, common subexpression elimination, forward expression substitution and dead code elimination are provided. When two or more transformations share specific transformation features, it is possible to define a combination of transformations in a single traversal. Thus, a superoptimiser is presented which combines renaming, constant propagation, copy propagation and common subexpression elimination.

9.1 Introduction

So far we have observed considerable code duplication in the implementations of various data-flow transformations. In the implementation of constant propagation we have used dynamic rules for the propagation of values. Data-flow transformations that involve multiple variables are modelled with dependent dynamic rules.

In this chapter, we generalise the strategies for constant propagation, copy propagation, common subexpression elimination and forward substitution into a generic strategy for forward data-flow propagation for Tiger. The optimisations that require a backward traversal flow such as dead assignment and dead declaration elimination are generalised into the generic strategy for backward data-flow transformations. These strategies are implemented in such a way that they can be reused by other data-flow transformations. Instantiations of data-flow transformations based on these generic propagations frameworks are presented. Moreover, when two or more transformations share a flow direction and the dynamic rules do not interfere with each other, it is possible to implement a combination of transformations in a single traversal. Thus, a superoptimiser is presented which combines renaming, constant propagation, copy propagation and common subexpression elimination.

9.2 Generic Data-flow Transformation Strategies

The definition of constant propagation in Figure 7.6 is very similar to the definition of copy propagation in Figure 8.3. The difference between the two transformations is restricted to the implementation of the strategies for handling variable declarations and assignments. For the control-flow constructs such as fork and iteration, the only difference is the name of the dynamic rule used.

To reuse code and to focus on specifying enabling conditions at relevant program points such as definition of values, a generic forward propagation strategy (GFPS) for Tiger is introduced in Figure 9.1. This framework combines different strategies for the correct handling of control-flow constructs for the forward propagation flow. The strategy is parameterised with a list of rule names to be intersected (*`Rs1`*) and a list of rule names to be unified (*`Rs2`*) at join points, respectively. Moreover, it is also possible to supply rule names (*`Rs3`*) that are part of the transformation, but do not require a dynamic rule operation at confluence points, i.e., rules that do not depend on control-flow. Dependent dynamic rule names are not first class objects in Stratego, they are implemented as an extension to Stratego. That is the reason for implementing the strategies `new-dynamic-rules(Rs,l,dep)` and `undefine-dynamic-rules(Rs,dep)` to define and undefine rules dependent rules.

Generalised Strategy Operators The GFPS uses generalised versions of the dynamic rule strategy operators to deal with multiple dynamic rules. The strategies `new-dynamic-rules` and `undefine-dynamic-rules` define and undefine rules for all their parameter rules. Similarly, the `/`*`R1`*`\`*`R2`*`/` and `/`*`R1`*`\`*`R2`*`/*` operators generalise the intersection and union operators into a single combined operator, which performs intersection over the first set of rules and union over the second. Thus, it becomes straightforward to combine different analyses based on different rules, if they share a directed flow traversal over a tree, and if the rules make the system terminating and confluent. If the framework will combine the rules `CSE` and `FES` the transformation will be non-terminating. Here we say that this two transformations interfere with each other. The operators enable us to apply different dynamic rule operations at confluence points which allow different analyses or transformations to *coexist*.

```
forward-prop(transform, before, recur, after| Rs1 , Rs2 , Rs3) =
  let fp(transform, before, recur, after| Rs1 , Rs2 , RsSc , RsDF) =
        (  forward-prop-let    (transform,before,recur,after| RsSc)
        <+ forward-prop-vardec(transform,before,recur,after| RsSc)
        <+ forward-prop-assign(transform,before,recur,after| RsDF)
        <+ forward-prop-if    (transform,before,recur,after| Rs1 ,Rs2)
        <+ forward-prop-while (transform,before,recur,after| Rs1 ,Rs2)
        <+ (transform <+ before; all(recur); after))
   in fp(transform, try(before), recur, try(after)
         | Rs1 , Rs2 , <conc>(Rs1 , Rs2 , Rs3), <conc>(Rs1 , Rs2))
  end

forward-prop-let(transform, before, recur, after| Rs) =
  ? |[ let d* in e* end ]|
  ; (transform <+ {| ~Rs  : before; all(recur); after|})

forward-prop-vardec(transform, before, recur, after| Rs) =
  |[ var <id> <id> := <recur> ]|
  ; (transform
     <+ before
        ; ?|[ var x  ta  := e  ]|
        ; new-dynamic-rules(|Rs , x , x)
        ; after)

forward-prop-assign(transform, before, recur, after| Rs) =
  |[ <id> := <recur> ]|
  ; (transform
     <+ before
        ; ?|[ x  := e  ]|
        ; where(undefine-dynamic-rules(|Rs , x))
        ; after)

forward-prop-if(transform, before, recur, after| Rs1 , Rs2) =
  |[ if <recur> then <id> else <id> ]|
  ; (transform
     <+ before
        ; (|[ if <id> then <recur> else <id> ]|
              /~Rs1\~Rs2/ |[ if <id> then <id> else <recur> ]|)
        ; after)

forward-prop-while(transform, before, recur, after| Rs1 , Rs2) =
  |[ while <id> do <id> ]|
  ; (transform
     <+ before
        ; (/~Rs1\~Rs2/* |[ while <recur> do <recur> ]|)
        ; after)
```

Figure 9.1: A generic (transformation independent) strategy for forward propagation transformations.

Generic Forward Propagation Framework The framework shown in Figure 9.1 is a generic and reusable forward propagation framework for Tiger. The generic strategy is parameterised with strategies that are applied at certain stages of the transformation. The strategies `transform`, `before` and `after` are local rewrites of a construct and can be used to tune the transformation. The `recur` strategy is a recursive strategy that traverses the tree. The transformation process is controlled by the strategy parameters `before` and `after`, which define local modifications for the general treatment of the construct at hand. The strategy parameter `transform` provides an alternative to achieve desired or unconventional transformations not achievable by means of strategies `before` and `after`.

The GFPS is composed of the following strategies:

`forward-prop` defines the top level strategy. This strategy is parameterised with the parameter strategies `transform`, `before`, `recur`, and `after`. It has also three sets of dynamic rule names for rules that will be intersected *Rs1*, unified *Rs2* and rules insensitive to control-flow *Rs3*. Internally the strategy constructs four different set of rules and it instantiates every defined strategy with the required rule set(s). The strategy `conc`[1] is used to concatenate rule names and constructs the required rule sets. The third rule set (*RsSc*) contains all rule names to be initialised. The fourth rule set includes rules to handle control-flow and record definition and undefinition of rules.

`forward-prop-let` defines the application scope for each dependent dynamic rules (*RsSc*) used in a transformation. An alternative behaviour can be specified by supplying a definition strategy for the parameter strategy `transform`.

`forward-prop-vardec` matches a construct that declares a variable and defines dependent rule(s) for the variable. This strategy handles shadowing if a previous dynamic rule for the same variable was already defined.

`forward-prop-assign` matches an assignment and applies the strategy `before`, if defined. It also undefines all dynamic rules (*RsDF*) that depend on the variable defined with the assignment. This rewriting can be refined providing an implementation for the strategy `after`. If a different behaviour is not intended, the strategy `transform` can be specified to achieve the desired rewriting.

`forward-prop-if` and `forward-prop-while` match control-flow statements. These strategies apply operations over dynamic rules sets such as *intersection* or/and *union* at confluence points. All the rewriting is achieved with these strategies. Further adaptation can be done with the strategies `before`, `after` and `transform`.

Constant Propagation Figure 9.2 shows how the GFPS framework is instantiated to implement constant propagation. It uses dependent dynamic rules and it focuses on implementing the strategies that define the `PropConst` dynamic rule. Thus, the state of the variables is being reflected in the state of the dynamic rule.

[1] The `conc` strategy is defined in the Stratego library and listed in Appendix A.

```
prop-const = forward-prop(prop-const-transform(prop-const)
                         , id
                         , prop-const
                         , prop-const-after
                         | ["PropConst"], [], [])

prop-const-transform(recur) =
   ElimIf; recur <+ |[ while <recur> do <id> ]|; ElimWhile; recur

prop-const-after =
  try(prop-const-assign  <+ prop-const-declare <+
      PropConst <+ EvalBinOp <+ EvalRelOp)

prop-const-declare =
  ?|[ var x ta := e ]|
   ; where(<prop-const-assign> |[ x := e ]|)

prop-const-assign =
  ?|[ x := e ]|
  ; where(<is-value> e)
  ; rules(PropConst.x : |[ x ]| -> |[ e ]| depends on [(x,x)])
```

Figure 9.2: Instantiation of GFPS for constant propagation.

```
copy-prop =
    forward-prop(copy-prop-transform(copy-prop), id, copy-prop,
                 copy-prop-after | ["CopyProp"],[],[] )

copy-prop-transform(cp) =
   ?|[ x := <cp => x> ]|

copy-prop-after = copy-prop-assign <+ copy-prop-declare <+ repeat1(CopyProp)

copy-prop-declare =
    ? |[ var x ta := e ]|
    ; where(try(<copy-prop-assign> |[ x := e ]|))

copy-prop-assign =
    ? |[ x := y ]|
    ; where(<not(eq)>(x,y))
    ; where(get-var-dependencies => xs)
    ; where(innermost-scope(elem-of(|[x,y])|["CopyProp"]) => z)
    ; rules(CopyProp.z : |[ x ]| -> |[ y ]| depends on xs)
```

Figure 9.3: Instantiation of GFPS for copy propagation.

```
cse = forward-prop(fail, id, cse, cse-after | ["CSE"], [], [])

cse-after = cse-assign <+ cse-declare <+ repeat1(CSE)

cse-declare =
  ? |[ var x ta := e ]|
  ; where(<cse-assign> |[ x := e ]|)

cse-assign =
  ? |[ x := e ]|
  ; where(<pure-and-not-trivial(|x)> |[ e ]|)
  ; where(get-var-names => vars)
  ; where(innermost-scope(elem-of(|vars)|["CSE"]) => z)
  ; where(get-var-dependencies => xs)
  ; rules(CSE.z : |[ e ]| -> |[ x ]| depends on xs)
```

Figure 9.4: Instantiation of GFPS for common subexpression elimination.

The strategies to handle control-flow constructs are entirely reused from the framework and do not require further adaptation.

The implementation removes unreachable code which is achieved by providing an instantiation for the strategy parameter `transform`. The instantiation is defined by the strategy `prop-const-transform` which re-uses rules `ElimIf` and `ElimWhile` already presented in Figure 4.9.

Copy Propagation Figure 9.3 presents the instantiation of the GFPS for copy propagation. The instantiation is similar to the one for constant propagation. This transformation propagates variables instead of values, and it requires a proper control for propagating variables. Thus, it is only possible to propagate variables with different names. This transformation uses the strategy `innermost-scope` to control the scope to which the dependent dynamic rule operation refers to. Control-flow constructs are entirely handled by the generic propagation strategy.

Common Subexpression Elimination The instantiation of the GFPS for common subexpression elimination (CSE) is presented in Figure 9.4. CSE is a transformation that replaces common expressions with a variable that already contains the value of that expression. By instantiating the GFPS, we can focus on the definition of the conditions that control the propagation of non-trivial expressions by defining `CSE` rules. Scoping and undefining of dynamic rules are handled in the GFPS strategy. This is a major simplification of the implementation of CSE, since we do not have to handle all the control-flow constructs.

Forward Expression Substitution Figure 9.5 illustrates the implementation of forward expression substitution as an instantiation of the GFPS. This transformation has the opposite goal of common subexpression elimination. The only

```
forward-subs =
  forward-prop(fail, id, forward-subs, forward-sub-after|["FSubs"],[],[])

forward-sub-after =
  try(forward-sub-assign <+ forward-sub-declare <+ repeat1(FSubs))

forward-sub-declare =
  ? |[ var x ta := e ]|
  ; where(try(<forward-sub-assign> |[ x := e ]|))

forward-sub-assign =
  ? |[ x := e ]|
  ; where(<pure-and-not-trivial(|x)> |[ e ]|)
  ; where(get-var-dependencies => xs)
  ; where(innermost-scope-FSubs => z)
  ; rules(FSubs.z : |[ x ]| -> |[ e ]| depends on xs)
```

Figure 9.5: Instantiation of GFPS for forward expression substitution.

```
bvr =
  forward-prop(fail, bvr-before, bvr, bvr-after| [], [], ["RenameVar"])

bvr-before =
  bvr-declare <+ bvr-assign

bvr-after =
  bottomup(try(RenameVar))

bvr-declare :
  |[ var x ta := e ]| -> |[ var y ta := e ]|
   where <rename-variable> x => y

bvr-assign :
  |[ x := e ]| -> |[ y := e ]|
   where <bvr> |[ x ]| => |[ y ]|

rename-variable : x -> y
  where innermost-scope(elem-of(|[x])|["RenameVar"]) => x
        ; <new> x => y
        ; rules(RenameVar.x: |[ x ]| -> |[ y ]|)
```

Figure 9.6: Instantiation of GFPS for bound variable renaming.

difference with respect to the implementation of CSE is the direction of the propagation rule: the CSE propagation rule is defined with the construct `CSE.z : |[ e ]| -> |[ x ]| depends on xs`, while the propagation rule of forward expression substitution is defined by `FSubs.z : |[ x ]| -> |[ e ]| depends on xs`.

Bound Variable Renaming Figure 9.6 shows the implementation for bound variable renaming. The transformation is implemented as an instantiation of the GFPS framework, but it does not require dynamic rule operations at confluence points. The GFPS framework can also be used to describe this type of transformations. Dynamic rules that do not require operations over dynamic rules can be specified. More specifically the third list of rule names of the GFPS was introduced exactly for this purpose.

The implementation of bound variable renaming renames occurrences of variables on demand, i.e., keeps names of variables that have no conflicts. The strategy `innermost-scope-RenameVar` keeps a list of labels for which a dynamic rule has already been defined. By applying this strategy to all variables in turn it is possible to determine whether a renaming is needed. Renaming rules are only defined for variables that clash.

The scoped labelled dynamic rules provide a mean to handle language scoping. The implementation for bound variable renaming is presented here for two purposes: as an instantiation of the GFPS for which no rule operations is needed at confluence points, and to be part of combined transformation described in the following section.

9.3 Generic Backward Propagation Framework

A comparable strategy for generic backward propagation (GBPS) for Tiger is defined. The strategy is shown in Figure 9.7. The main difference is the order in which the constructors and their elements are visited. Similarly, the backward propagation strategy is parameterised with the strategies `transform`, `before`, `recur` and `after` to implement local modifications for tailoring a specific transformation.

The GBPS is composed of the following strategies:

`bward-prop` is the top level strategy which has the strategies `transform`, `before`, `recur`, and `after` as strategy parameters. Other parameters of the strategy are the sets of rule names to be used in the transformations.

`bward-prop-let` matches a `let` construct and introduces the scope of the dynamic-rule(s) for the declared variables. The declarations are traversed in forward direction flow, and the `let` expression is traversed in backward direction flow. The strategy `reverse-map`[2] visits the elements of a list starting from the end towards the initial element. With the `reverse-map` strategy the statements of a sequence are visited in *backwards* fashion.

`bward-prop-vardec` matches a variable declaration and initialises dependent dynamic rules(s) for that variable.

`bward-prop-assign` matches an assignment and undefines the rules which are dependent of that definition. After that, it visits the right-hand-side of the assignment.

[2] The `reverse-map` and `reverse-filter` are Stratego strategies defined in the Stratego library and they are listed in Appendix A.

```
backward-prop(transform, before, recur, after| Rs1, Rs2, Rs3) =
  let bp(transform, before, recur, after| Rs1, Rs2, RsSc, RsDF) =
            bward-prop-seq    (transform, before, recur, after)
         <+ bward-prop-let    (transform, before, recur, after| RsSc)
         <+ bward-prop-vardec (transform, before, recur, after| RsSc)
         <+ bward-prop-assign (transform, before, recur, after| RsDF)
         <+ bward-prop-if     (transform, before, recur, after| Rs1, Rs2)
         <+ bward-prop-while  (transform, before, recur, after| Rs1, Rs2)
         <+ transform <+ before; all(recur); after
   in bp(transform, try(before), recur, try(after)
       | Rs1, Rs2, <conc>(Rs1, Rs2, Rs3), <conc>(Rs1, Rs2))
  end

bward-prop-let(transform, before, recur, after| Rs) =
  ?|[ let d* in e* end ]|
  ; (transform
     <+ {| ~Rs
         : before
           ; |[ let <*map(recur)> in <*reverse-map(recur)> end ]|
           ; after
        |})

bward-prop-vardec(transform, before, recur, after| Rs) =
  ?|[ var x ta := e ]|
  ; (transform <+ before
                 ; new-dynamic-rules(|Rs, x, x)
                 ; after)

bward-prop-assign(transform, before, recur, after| Rs) =
  ?|[ x := e ]|
  ; (transform <+ before
                 ; undefine-dynamic-rules(|Rs, x)
                 ; |[ <id> := <recur> ]|
                 ; after)

bward-prop-seq(transform, before, recur, after) =
  ?|[ (e*) ]|; before; |[ (<*reverse-map(recur)>) ]|; after

bward-prop-if(transform, before, recur, after| Rs1, Rs2) =
  |[ if <id> then <id> else <id> ]|
  ; (transform <+ before
                 ; (|[ if <id> then <recur> else <id> ]|
                     /~Rs1\~Rs2/  |[ if <id> then <id> else <recur> ]|)
                 ; |[ if <recur> then <id> else <id> ]|
                 ; after)

bward-prop-while(transform, before, recur, after| Rs1, Rs2) =
  |[ while <id> do <id> ]|
  ; (transform <+ before
                 ; (/~Rs1\~Rs2/* |[ while <recur> do <recur> ]|)
                 ; after)
```

Figure 9.7: Generic backward propagation strategy.

```
dce-asg = backward-prop(dce-decl-transform
                        , dce-decl-assign-before
                        , dce-decl
                        , dce-decl-after
                        | [], ["Needed"], [])

dce-decl-transform =
  VarNeeded <+ ElimAssign

dce-decl-assign-before =
  ?|[ x := e ]|
  ; where(<Needed> |[ x ]| )

dce-decl-after =
  |[ ( <*filter(not( ?|[ () ]|))> ) ]|
  <+ dce-decl-vardecl-after

dce-decl-vardecl-after =
  (?|[  var x  ta  := e  ]| <+ ?|[ var x  ta  ]| )
  ; undefine-dynamic-rules(|[ "Needed"], x)

VarNeeded =
  ?|[ x ]|
  ; rules(Needed.x: |[ x ]| -> |[ x ]|  depends on [(x, x)])

ElimAssign :
  |[ x := x ]| -> |[ () ]|

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

Figure 9.8: Instantiation of GBPS for dead assignment elimination.

`bward-prop-seq` matches a sequence of statements and visits the elements in a reverse order using the strategy `reverse-map` .

`bward-prop-if` matches an `if` statement and visits first its branches and applies the intersection operation to the resulting rule sets. Subsequently it visits the condition expression.

`bward-prop-while` matches a `while` statement and applies the fixed-point operator over the dynamic rules.

Data-flow analysis is not required to reuse these generic transformation strategies. It is always possible to define an alternative strategy, if necessary.

Dead Assignment Elimination Figure 9.8 shows the instantiation for dead assignment elimination. This instantiation reuses the strategies for control-flow

```
dce-decl = backward-prop(VarUsed, id, dce-decl, dce-decl-after(dce-decl)
        | [], ["Used"], [])

dce-decl-after(dce-decl) =
  |[ ( <*filter(not(?|[ () ]| ))> ) ]|
  <+ |[ let <*reverse-filter(keep-declarations(dce-decl))>
         in <*reverse-filter(not(?|[ () ]| ))> end ]|
  <+ dce-decl-vardecl-after
  <+ dce-decl-assign-after
  <+ ElimIf <+ ElimWhile <+ ElimLet

dce-decl-vardecl-after =
  (?|[ var x ta := e ]| <+ ?|[ var x ta ]| )
  ; undefine-dynamic-rules(| ["Used"], x)

dce-decl-assign-after =
  |[ x := e ]|
  ; rules(Used.x: |[ x ]| -> |[ x ]| depends on [(x,x)])

VarUsed =
  ?|[ x ]|
  ; rules(Used.x: |[ x ]| -> |[ x ]| depends on [(x,x)])

keep-declarations(dce-decl) =
  ?|[ var x ta := e ]|
  ; where(<Used> |[ x ]| )
  ; |[ var x ta := <dce-decl> ]|
```

Figure 9.9: Instantiation of GBPS for dead declaration elimination.

constructs and only defines the strategies needed to propagate the information required to keep or delete assignments.

Dead Declaration Elimination The instantiation for dead declaration elimination using the GBPS is shown in Figure 9.9. It defines the strategies needed to mark variables used in the program and the strategies that decide whether a declaration is in the resulting program or not and strategies needed to clean up the code by removing empty statements.

9.4 Combining Data-flow Optimisations

The generic forward propagation strategy can apply different analyses and transformations in the same traversal by combining elements from several one-issue transformations. Figure 9.10 shows a strategy that combines constant propagation, copy propagation, common-subexpression elimination, unreachable code elimination and

```
super-opt =
  forward-prop(prop-const-transform(super-opt)
    , bvr-before
    , super-opt
    , bvr-after; copy-prop-after; prop-const-after; cse-after
    | ["PropConst", "CopyProp", "CSE"], [], ["RenameVar"]
  )
```

Figure 9.10: 'Super' transformation combining constant propagation, copy propagation, common-subexpression elimination, and bound variable renaming.

```
dce = backward-prop( dce-transform
                   , dce-decl-assign-before
                   , dce
                   , dce-asg-after(dce) <+ dce-decl-after
                   | [], ["Needed", "Used"], [])

dce-transform =
  VarUsed; VarNeeded <+ ElimAssign
```

Figure 9.11: Instantiation of GBPS for dead code elimination.

renaming. We have included renaming on the fly in this combined transformation to avoid unnecessary renaming, undefinition and shadowing of dynamic rules.

The following transformation is an example of the application of the super-optimiser:

```
let var a := foo()
    var b := a
    var c := b
 in let var a := 2
        var x := b + 1
     in b:= a;
        print(b*c + x)
   end
end
```

⇒

```
let var a := foo()
    var b := a
    var c := a
 in let var a_0 := 2
        var x := a + 1
     in b := 2;
        print(2*a + x)
    end
end
```

⇒

```
let var a := foo()
 in let var x := a + 1
     in print(2*a + x)
    end
end
```

Renaming for `a` in the inner scope is applied on the fly to avoid name capture. In this way, we avoid the need to undefine the dynamic rule generated for `b := a` in the outer scope. After this transformation, dead code elimination has been applied to the code fragment. It is important to note that this transformation could not have been achieved by a sequential composition of the individual transformations without renaming.

Dead Code Elimination Dead code elimination is achieved by combining dead assignment elimination and dead declaration elimination. The combined strategy is presented in Figure 9.11. A more complex transformation is achieved in a simpler way by combining pieces of other transformations. This combination is only possible because the optimisations share the same direction of propagation and the rules do not interfere with each other.

9.5 Summary

This chapter introduced two general Tiger frameworks for data-flow transformations. These strategies are general and mainly capture the traversal of a program for forward or backward propagation flow. Operations over dynamic rule sets are specified at top level and they allow combination of transformations when transformations do not interfere. Moreover, the strategies are parameterised at many points of the transformation process. Further tailoring can be accomplished by providing different implementations using the strategy parameters to achieve different behaviour.

10
Partial Evaluation with Strategic Rewriting

This chapter presents a strategic rewriting implementation of partial evaluation. We extend constant propagation, constant folding and removal of dead code to be applicable at inter-procedural scope. Furthermore, functions invocations are unfolded or specialised with respect to the statically known arguments of the call. An implementation of poly-variant online partial evaluation is presented. Strategies are defined in a natural way to control the process.

10.1 Introduction

Partial evaluation is a well known source-to-source program transformation which specialises programs according to part of the program's input [58, 53, 27]. The input to a program can be split into two parts: *static* (i.e., known input) and *dynamic* (i.e., unknown input). A partial evaluator uses the static part of the input to optimise a program p and generate a new program p'. The generated program p' requires only the dynamic input for its execution. The obtained program p' is known as the *residual program.* Both programs obtain the same result provided that they share the same values for the dynamic arguments. The program p' typically executes faster than p. The run-time gain is due to the static evaluation of some of the program's constructs in the original program.

Partial evaluation has two flavours: offline and online. An offline partial evaluator [74] is structured in (at least) two stages. The first stage differentiates static from dynamic expressions. This is also known as *binding time analysis*. The second stage uses the information from the first stage to determine which constructs can be statically evaluated and which must be evaluated at run-time. An online partial evaluator is structured as (mostly) one-stage process. Online partial evaluators can be more efficient than offline partial evaluators because they can exploit the obtained results for subsequent evaluations.

A partial evaluator shares implementation components of an interpreter and/or a compiler. Figure 10.1 shows the architecture of the partial evaluator. The front-end and back-end components are shared with other transformation tools. In between we find components forming the partial evaluator. It is composed of a program specialiser (*tiger-specialise*) followed by a phase that restructures the code by lifting and moving expressions to reach a canonical form of the program (*tiger-canonicalise*). This transformation was presented in Chapter 4.6.4 with the purpose of removing side-effects from expressions. Code restructuring introduces many copies of variables which are propagated with *tiger-copy-prop*. A follow-up transformation cleans the code by removing dead code.

In this chapter, we describe how-to implement a partial evaluator by a series of incremental steps. Starting with a constant propagation strategy that performs partial evaluation locally, we refine it to be applicable to an inter-procedural scope by means of function unfolding and function specialisation. The contribution of this chapter is the clean implementation of an online partial evaluator for Tiger in Stratego.

10.2 Expression Folding and Constant Propagation

The goal of partial evaluation is to evaluate program expressions at compile time. This optimisation shares its goal with many other optimisations, in particular with constant folding, unreachable code elimination and constant propagation.

Constant folding defines the reduction of an expression with constant arguments for built-in operators. We have already discussed constant folding by means of evaluation rules. Unreachable code elimination removes code that will not be executed. Rewrite rules for constant propagation and unreachable code elimination were presented in Chapters 4.6.2 and 4.6.3.

Constant propagation extends constant folding by propagating constant values assigned to variables to their uses. Constant propagation allows constant folding to statically evaluate more expressions. The implementation of constant propagation relies on dynamic rewrite rules and dynamic rule operators to handle language scoping, propagation of constant values until the variable is reassigned, and the selection of valid propagation values at confluence program points. This transformation was presented in Chapter 7 combined with constant folding and unreachable code elimination.

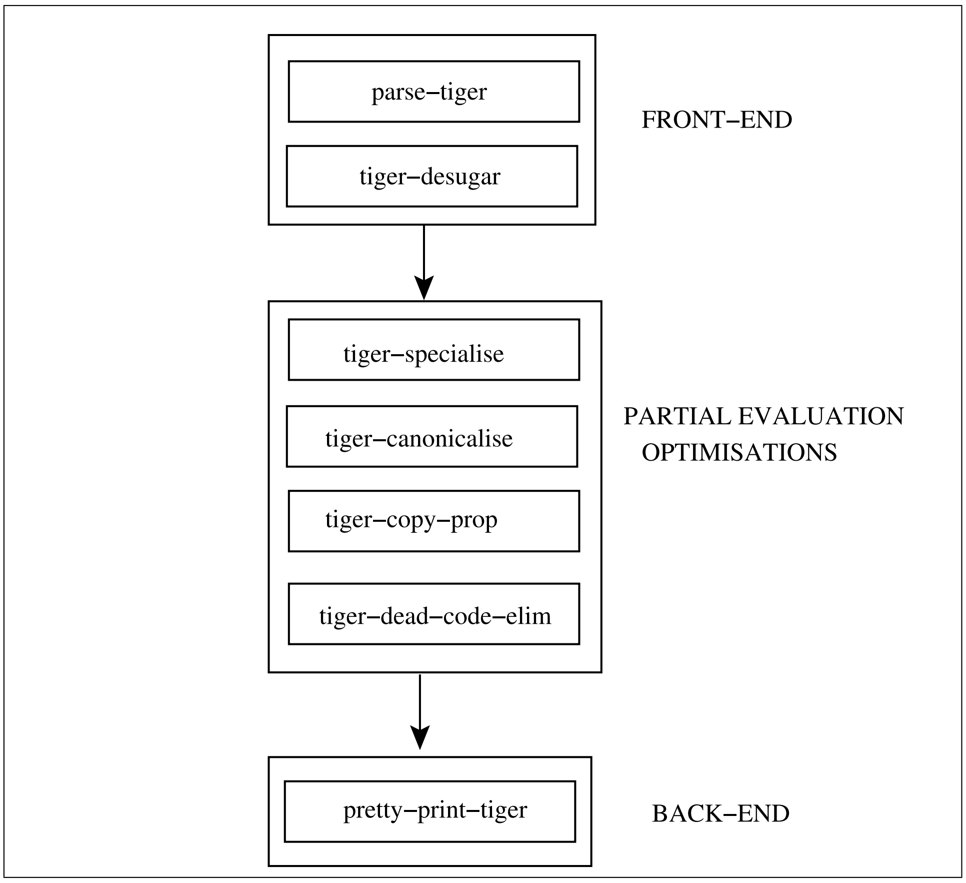

Figure 10.1: Architecture of a Tiger partial evaluator.

The implementation of partial evaluation is incremental over constant propagation. For simplicity, the basic implementation is shown in Figure 10.2.

10.3 Function Unfolding

Partial evaluation is an inter-procedural extension to constant propagation. Although the scope of constant propagation can be inter-procedural, partial evaluation adds the optimisation of function calls. There are two possible function optimisations: function unfolding (inlining) and function specialisation. Function unfolding replaces a function call with the body of its definition. This modification to the program may increase the extent of static evaluation of expressions. Function specialisation creates a new function that is specialised to the static arguments of the call. The static function call arguments are used to evaluate expressions inside the function body.

```
pe = PropConst <+ pe-assign <+ pe-declare
     <+ pe-let <+ pe-if <+ pe-while
     <+ all(pe); try(EvalBinOp)

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

pe-declare =
  ? |[ var x ta ]|
  ;  rules( PropConst+x :- |[ x ]| )

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

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

pe-if =
  |[ if <pe> then <id> else <id> ]|
  ; (ElimIf; pe
     <+ (|[ if <id> then <pe> else <id> ]|
         /PropConst\ |[ if <id> then <id> else <pe> ]|))

pe-while =
  |[ while <id> do <id> ]|
  ; (|[ while <pe> do <id> ]|; ElimWhile
     <+ /PropConst\* |[ while <pe> do <pe> ]|)

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

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

Figure 10.2: Constant propagation as a basis for partial evaluation.

In this section we consider function unfolding and in Section 10.4 we introduce function specialisation. Subsequently we examine strategies for combining these optimisations for functions in Section 10.5.

10.3.1 Replacing Function Calls

The implementation of function unfolding defines for each function definition a dynamic rule that replaces a function call with the body of its definition. This step was already shown in Section 5.5.2.

```
pe = PropConst <+ UnfoldCall; pe <+ pe-declare ...

pe-declare =
  ?|[ function f() ta = e ]|
  ; rules( UnfoldCall : |[ f() ]| -> <exprename> |[ e ]|)
```

Figure 10.3: Replacing a function call by the function body.

In Figure 10.3 the partial evaluation strategy pe is extended to unfold function calls. The strategy pe-declare matches a function definition and defines a dynamic rule UnfoldCall that replaces a function invocation by the body of the invoked function. Renaming of the variables of the function body is done to prevent name conflicts. The UnfoldCall rule is part of the partial evaluation pe strategy.

We show an example of the effect of this transformation as part of a partial evaluator:

```
let var x : int := 0                let var x : int := 0
    function g() =                      function g() =
      x := x + 1              ⇒           x := x + 1
 in g(); g()                         in x := 1; x := 2
end                                 end
```

The first version of the partial evaluator is simplistic. It only unfolds calls for functions without arguments. A more elaborated partial evaluator needs to bind function arguments to the formal parameters of a function definition. It also needs to guarantee uniqueness of variables, otherwise the transformation tool can introduce variable clashes. Function unfolding has to follow the scoping rules of the language. In Tiger, function definitions are visible inside the let statement where they are defined. The strategies for unfolding calls needs proper scoping implementation to suit the scoping rules of the language at hand.

10.3.2 Parameter Instantiation

In general functions and procedures have parameters. Formal parameters should be replaced by actual parameters. Figure 10.4 extends the previous definition of the dynamic rule UnfoldCall to perform parameter substitution. The strategy

SubstArg defines a dynamic rule PropConst to substitute the formal parameter with the actual parameter. The scope of the PropConst rule is delimited to the unfolded expression. However, naive substitution is incorrect and it can cause undesired results such as duplication of side-effects. Consider the example:

```
let var x : int := 0
    function g(y : int) =
      (x := x + 1; y + y)
 in x := g(h(x))
end
```

⇒

```
let var x : int := 0
    function g(y : int) =
      (x := x + 1; y + y)
 in x := (x := x + 1; h(x) + h(x))
end
```

In the resulting program the function h is evaluated twice and if the function has side-effects or if its implementation is not trivial, the optimisation can have undesired results. Rather than substituting parameters, the introduction of a let binding must be accompanied by new variable declarations for the renamed actual parameters retaining semantics and avoiding unwanted results. Thus, intra-procedural constant propagation determines whether it is safe to propagate values of certain parameters.

Figure 10.5 shows the substitutions of formal parameters by introducing a let which binds new variable declarations to the actual parameters and rename the unfolded function body appropriately. Binding expressions to variables and subsequent constant propagation has the same effect as substitution and does not introduce duplication of side-effects.

10.4 Function Specialisation

Function specialisation is defined as the process of partially evaluating a function with respect to the static arguments of the call. A new function definition is created with formal parameters for the dynamic arguments of the call. Figure 10.6 presents

```
pe = PropConst <+ {| PropConst : UnfoldCall; pe |}  <+ ...
pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      UnfoldCall :
        |[ f(a*) ]| -> |[ e ]|
        where <zip(SubstArg)> (x*, a*) => d*
    )
SubstArg =
  ?(FArg|[ x  ta ]|, e)
  ; rules( PropConst : |[ x ]| -> |[ e ]| )
```

Figure 10.4: Instantiation by argument substitution.

```
pe = PropConst <+ UnfoldCall; pe <+ ...
pe-declare =
 ?|[ function f(x*) ta = e ]|
 ; rules(
    UnfoldCall :
    |[ f(a*) ]| -> |[ let d* in e2 end ]|
    where <exprename>|[function f(x*) ta = e]| => |[function f(x2*) ta2 = e2]|
        ; <zip(BindArg)> (x2*, a*) => d*
    )
BindArg :
  (FArg|[ x ta ]|, e) -> |[ var x ta := e ]|
```

Figure 10.5: Instantiation by let binding.

an example of function specialisation. The function call `power` is specialised for its second argument `5`. A new function call and definition for the specialised function are introduced. The new function `power0` has only one argument and in its body a new binding for the second argument with value `5` is introduced.

Function specialisation can be *mono-variant* or *poly-variant*. A mono-variant specialiser produces a single specialised function per function definition. The poly-variant specialiser produces several specialised versions per function definition corresponding to different values of static arguments. The specialised code of Figure 10.6 is an example of a mono-variant function specialiser. In this chapter we implement a poly-variant version of a function specialiser.

The implementation of function specialisation is shown in Figure 10.7. The dynamic rule `SpecialiseCall` matches a function call and replaces it with a function call for the specialised function. The strategy `split-static-dynamic-args` takes the formal parameter and the actual parameter lists as inputs and yields a list of declarations for the static arguments and the list of actual parameters that are dynamic. The strategy `partition`[1] produces a tuple with two lists. The first list contains the elements for which the strategy parameter `BindArgValue` succeeds. The second list contains the list of dynamic arguments. Splitting arguments is done if at least one argument of the call is static. The strategy `UnfoldCall` is used to obtain the function definition corresponding to the call.

A new dynamic rule `Specialisation` is defined when the rule `SpecialiseCall` is successfully applied. The `Specialisation` rule will replace the original definition of a function with a specialised version. The new function definition is constructed with the formal arguments corresponding to the dynamic arguments of the call. The body of the new function is constructed by partially evaluating the body of the original function with respect to the static arguments of the call. Note that the rule `Specialisation` has the operator `+` which allows multiple defined rules for the same left-hand-side. This feature of dynamic rules allows us to implement a simple

[1] The strategy `partition` is part of the Stratego library and its definition is presented in appendix A.

```
let function square(x : int) : int = x * x
    function mod(x : int, y : int) : int = x - (x / y) * y
    function even(x : int) : int = mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

$\Downarrow$

```
let function square(x : int) : int = x * x
    function mod(x : int, y : int) : int = x - (x / y) * y
    function even(x : int) : int = mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1)
    function power0(x : int) : int =
      let var n : int := 5
       in if n = 0 then 1
          else if even(n)
               then square(power(x, n/2))
               else x * power(x, n-1)
      end
 in printint(power0(readint()))
end
```

Figure 10.6: Example of function specialisation.

and clean poly-variant partial evaluator.

10.4.1 Replace Function Definition

With the successful application of the rule `SpecialiseCall`, a function call is replaced with a call to a new specialised function. The new function definition for the specialised function has to be included into the reduced program. Figure 10.8 shows the inclusion of the specialised function definitions into the resulting program. Specialised function definitions are introduced in the strategy `pe-let`. The specialised functions are kept in `Specialisation` rules. A list of specialised function definitions is obtained with the strategy `bagof-Specialisation`. The strategy `mapconcat` yields a linearised list of function definitions.

The example presented in Figure 10.6 did not optimise the body of the specialised functions. By optimising the body before generating the new function definition we obtain better results.

By using optimised residual functions and allowing a function to have multiple

```
pe-declare =
?|[ function f(x1*) ta = e1 ]|;
rules(
 Specialisation+f
 UnfoldCall : ...

 SpecialiseCall :
   |[ f(a1*) ]| -> |[ g(a2*) ]|
   where <split-dynamic-static-args> (x1*, a1*) => (x2*, a2*)
       ; <UnfoldCall> |[ f(a2*) ]| => e2
       ; <newname> f => g
       ; <pe> e2 => e3
       ; rules(
           Specialisation.f :+
             |[ function f(x1*) ta = e' ]| -> |[ function g(x2*) ta = e3 ]|
         )
)
split-dynamic-static-args =
 zip; partition(BindArgValue); (not([]), unzip)

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

Figure 10.7: Generate specialisation for function calls.

specialised versions (i.e., poly-variant) better results can be obtained. In Figure 10.9 the result for the current running example after a poly-variant specialisation is presented.

10.5 Partial Evaluation Strategies

In the previous sections we have defined the basic ingredients of a partial evaluator: constant propagation, function unfolding, and function specialisation. However, there are some issues that need to be taken into account. First, indiscriminately

```
 pe-let =
   |[ let <*id> in <*id> end ]|
   ; {| PropConst, UnfoldCall, Specialisation
      : all(pe)
      ; |[ let <*specialise-functions> in <*id> end ]|
     |}

 specialise-functions =
   map(try(|[ <fd*: mapconcat(bagof-Specialisation)> ]|))
```

Figure 10.8: Replace function definitions by specialised function definitions.

```
let function square(x : int) : int = x * x
    function mod(x : int, y : int) : int = x - (x / y) * y
    function even(x : int) : int = mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
           else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

$$\Downarrow$$

```
let function square(x : int) : int = x * x
    function power0(x : int) : int = x * power1(x)
    function power1(x : int) : int = square(power2(x))
    function power2(x : int) : int = square(power3(x))
    function power3(x : int) : int = x * power4(x)
    function power4(x : int) : int = 1
 in printint(power0(readint()))
end
```

Figure 10.9: Polyvariant specialisation of power.

unfolding recursive functions will lead to non-termination. Second, multiple specialised versions of a function increase the code size of the residual program considerably. In this section we look at several strategies for controlling these issues.

10.5.1 Calls with Static Arguments

The implementation of the current partial evaluator may not terminate for recursive programs. As an example consider the case of recursive function unfolding. In order to avoid non-termination, we can unfold only functions with static arguments.

The strategy `unfold-call` shown in Figure 10.10 decides when it is safe to unfold or specialise a function call.

10.5.2 Memoization of Unfolding Calls

In the situation of multiple occurrences of a function call with the same values, our implementation of a partial evaluator will re-evaluate the function multiple times. To avoid this extra work we use memoization of evaluated functions. Memoization [71] is an optimisation technique for speeding up the execution of programs by storing the result of a function call. A memoized static function stores results of computed calls to be used later when a call with the same static arguments is encountered.

Our implementation of memoization is simple and it is shown in Figure 10.11. When an `UnfoldCall` rule is successfully applied, the result is stored using the dynamic

```
  pe = PropConst <+ pe-declare <+ ...
     <+ unfold-call
     <+ all(pe); try(EvalBinOp)

  unfold-call =
    if is-static-call
      then UnfoldCall
      else SpecialiseCall
    end

  is-static-call =
   |[ f(<*map(is-value)>) ]|
```

Figure 10.10: Static specialisation of function calls.

```
pe-declare =
  ?|[ function f(x*) ta = e ]|
  ; rules(
      Specialisation+f
      MemoizeUnfolding+f
      UnfoldCall : ...
      SpecialiseCall : ...
    )

  MemoizeCall :
    |[ f(a*) ]| -> |[ e ]|
    where <MemoizeUnfolding> |[ f(a*) ]| => |[ e ]|
          <+ <UnfoldCall; pe> |[ f(a*) ]| => |[ e ]|
            ; rules( MemoizeUnfolding.f: |[ f(a*) ]| -> |[ e ]| ) )
```

Figure 10.11: Memoization of unfolding results.

rule `MemoizeUnfolding`. The next call to the function will attempt to reuse the resulting memoized call.

10.5.3 Dynamic Conditionals

In a program with control-flow statements not all execution paths are necessarily executed. For instance consider an `if` statement for which it is impossible to statically determine which branch of the `if` will be executed. Specialisation of function calls in the branches of `if` statements under dynamic conditionals may introduce unneeded specialised functions. These functions contribute to useless code growth.

To avoid unnecessary calls we decided not to specialise under dynamic conditionals. The implementation in Figure 10.12 uses the dynamic rule `DynamicIf` to indicate that the current language construct is inside a dynamic `if` expression. This rule is defined while visiting an `if` expression in the strategy `pe-if`. The strategy `reduce-call` is modified to attempt to reuse a memoized call. If there is no mem-

```
reduce-call =
  ?|[ f(a*) ]|
  ; (MemoizeUnfolding
     <+ reduce-call-aux => e
        ; rules( MemoizeUnfolding.f : |[ f(a*) ]| -> e )
    )

reduce-call-aux =
  if not(DynamicIf) then
   MemoizeCall
  else
    SpecialiseCall
  end

pe-if =
  |[ if <pe> then <id> else <id> ]|
  ; (ElimIf; pe
     <+ {| DynamicIf
         : rules( DynamicIf : _ )
         ; (|[if <id> then <pe> else <id>]|
            /PropConst\ |[if <id> then <id> else <pe>]|)
        |})
```

Figure 10.12: Control of unfolding under dynamic conditionals.

oized rule for the current function call, the expression is reduced with the strategy `reduce-call-aux`. This strategy unfolds calls that are not in inside a dynamic `if`, otherwise the function call is specialised. The result obtained is memoized for further reuse.

The case of specialising a static `if` is covered in the initial strategy `|[ if <pe> then <id> else <id> ]|; ElimIf; pe`.

10.5.4 Loop Unrolling

So far we have only considered unfolding and specialising functions. Loops were treated uniformly. In Figure 10.13 a while loop is unrolled once with the rule `UnrollWhile`. Unrolling can be controlled using the rule `UnrollWhile` if the loop is not under a dynamic `if`. However this strategy is insufficient for controlling the rule to be applied infinitely if the while expression is not under a dynamic `if`.

A further refinement of loop unrolling is shown in Figure 10.14. Loop unrolling is done if the condition of the while loop is executed at least once, i.e., the condition can be statically evaluated.

```
UnrollWhile :
  |[ while e1 do e2 ]| -> |[ if e1 then (e2; while e1 do e2) ]|

pe-while =
  |[ while <id> do <id> ]|
  ; (|[ while <pe> do <id> ]|; ElimWhile; pe)
     <+ if not(DynamicIf) then
          UnrollWhile; pe
        else
          /PropConst\* |[ while <pe> do <pe> ]|
        end
```

Figure 10.13: Loop unrolling.

```
pe-while =
  |[ while <id> do <id> ]|
  ; (|[ while <pe> do <id> ]|; ElimWhile; pe)
     <+ if not(DynamicIf) then
          UnrollWhile
          ; |[ if <pe> then <id> ]|
          ; ElimIf
          ; pe
        else
          /PropConst\* |[ while <pe> do <pe> ]|
        end
```

Figure 10.14: Controlling unrolling.

10.5.5 Recursive Specialisation

To avoid ending up with multiple specialised residual functions with the same arguments and executing unneeded work; specialised functions are memoized and reused. Before specialising a function call, verifying whether the call has been already specialised prevents duplication of function specialisations. To determine whether a similar call has been specialised before, we require only the arguments for which the function was specialised. To isolate differences between calls with the same static arguments possibly containing differences in the dynamic arguments we replace the dynamic arguments with a name *dyn*.

Figure 10.15 presents the new strategy for specialising function definitions. The new implementation of the dynamic rule `SpecialiseCall` matches a function call and rewrites it with the new name for the specialised function. The result is memoized for further reuse with the dynamic rule `MemoizeSpecialisation`. The rule `specialise-call` matches a function and to replaces it with a call to the specialised version. The first application of the rule does the work and future applications reuse the memoized version. The `specialise-call` strategy replaces the dynamic arguments of the call with a *dyn* denoting a *dynamic* argument. With the function name for the function *f* and the list of the *static* arguments the name of the specialised version *g* is obtained. With the new function name *g* and the list of dynamic arguments of the call a function call to the specialised function is constructed. The strategy `select-dynamic-args` obtains the dynamic arguments of the call.

10.6 Postprocessing

The result of transforming a program with a program specialiser can generate dead code such as dead declarations, dead assignments and unused functions. Follow up transformations can clean the code and reduce significantly its size.

Canonicalisation Canonicalisation of expressions changes the structure of the program by lifting expressions out of statements and local declarations out of expressions. Expressions with implicit side-effects are removed by proper placement of instructions preserving the behaviour of the program. This transformation was presented in Section 4.6.4.

Copy Propagation Canonicalisation introduces many spurious variables and assignments of the form `x := y`. By means of copy propagation such 'copies' can be reduced, producing dead statements. This transformation was presented in Section 8.2.

Dead Code Elimination We remove dead code using a separate transformation that performs backwards, flow-sensitive dead assignment and dead function elimination. When propagating constants into a piece of code, expressions may be completely evaluated. However, the residual code may contain dead definitions and assignments that may obscure the evaluation. Dead code can be removed by the transformation presented in 8.6.

In Figure 10.16 we present the result of our partial evaluator for the running example. The function `power` for the static argument `5` is specialised. The first box contains the original program. In the second box we give the final outcome of the transformation. The third box shows the result after applying a series of post-processing transformations.

10.7 Summary

The implementation of partial evaluation was presented in this chapter. The partial evaluator is presented as an extension of constant propagation. The added extensions are unfolding function calls and specialisation of function calls where and when appropriate. The transformations are easily represented by rules and more importantly, control of the process is achieved by strategies in a concise and clear way. This basic schema for partial evaluation can be generalised further in the style of Chapter 9.

```
pe = ... <+ all(pe); try(... <+ specialise-call)

pe-declare =
?|[ function f(x1*) ta = e ]|;
rules(
  MemoizeUnfolding+f
  MemoizeSpecialisation+f
  Specialisation+f
  UnfoldCall : ...

  SpecialiseCall :
    |[ f(a1*) ]| -> g
    where <dynamic-args-to-fresh-vars>  (x1*, a1*) => (x2*, a2*)
        ; <UnfoldCall> |[ f(a2*) ]| => e2
        ; <newname> f => g
        ; <pe> e2 => e3
        ; rules(
            MemoizeSpecialisation.f : |[ f(a1*) ]| -> g
            Specialisation.f :+
              |[ function f(x1*) ta = e ]| -> |[ function g(x2*) ta = e3 ]|
          )
)

specialise-call :
  |[ f(a1*) ]| -> |[ g(a3*) ]|
  where <dummy-dynamic-args> a1* => a2*
      ; <MemoizeSpecialisation <+ SpecialiseCall> |[ f(a2*) ]| => g
      ; <select-dynamic-args> a1* => a3*

  dummy-dynamic-args =
   map(try(is-value <+ !|[ dyn ]|))

  select-dynamic-args =
    filter(not(is-value))

  dynamic-args-to-fresh-vars =
    zip(DynArgToFreshVar <+ ?(x,y); !([],y))
    ; unzip
    ; (concat, id)

  DynArgToFreshVar :
    (FArg |[ x ta ]|, |[ dyn ]|) -> ([FArg|[ y ta ]|, |[ y ]|])
    where <newname> x => y
```

Figure 10.15: Memoization of repetitive specialisations.

```
let function square(x : int) : int =
      x * x
    function mod(x : int, y : int) : int =
      x - (x / y) * y
    function even(x : int) : int =
      mod(x, 2) = 0
    function power(x : int, n : int) : int =
      if n = 0 then 1
      else if even(n) then square(power(x, n/2))
      else (x * power(x, n - 1))
 in printint(power(readint(), 5))
end
```

$\Downarrow$

```
let
 in printint(
      let var x0 : int := readint()
       in x0 *
          let var x3 : int := x0
              var x14 : int := let var x6 : int := x3
                                   var x13 : int := let var x9 : int := x6
                                                     in x9
                                                    end
                                in x13 * x13
                               end
           in x14 * x14
          end
      end)
end
```

$\Downarrow$

```
let var a_0
    var x14 : int
    var e_0
    var f_0
 in a_0 := readint();
    x14 := a_0 * a_0;
    e_0 := x14 * x14;
    f_0 := a_0 * e_0;
    printint(f_0)
end
```

Figure 10.16: Result of a partial evaluated program.

11
The Octave Compiler

Most array processing languages such as APL, Matlab and Octave rely on dynamic type-checking by the interpreter rather than static type-checking and are designed for user convenience with a syntax close to mathematical notation. Functions and operators are highly overloaded. The price to be paid for this flexibility is computational performance, since the run-time system is responsible for type checking, array shape determination, function call dispatching, and the handling of possible run-time errors. In order to produce efficient code an Octave compiler should address those issues at compile-time as much as possible. In particular, static type and shape inferencing can improve the quality of the generated code considerably. In this chapter we show how overloading in dynamically typed Octave programs can often be resolved by program specialisation. We discuss the typing issues in compiling Octave programs and give an overview of the implementation of an Octave specialiser.

11.1 Introduction

Octave, an open source clone of MATLAB[1], is a high level programming language and development environment which is widely used for rapid prototyping and simulation of scientific applications [70]. The language was designed to be user-friendly and thus its syntax closely follows mathematical notation. Distinguishing features

[1]MATLAB is a registered trademark of The MathWorks©, Inc.

of the language are its high level built-in data types and a rich set of mathematical functions. Octave is an array processing language too as it supports arrays as built-in data types. In keeping with mathematical notation, functions and operators can be highly overloaded. For example, a function can operate on scalars as well on matrices, and can take a variable number of arguments. Such features make array processing languages easy to use as a prototyping language. The strength of problem solving environments in general and Octave in particular is their rich set of built-in functions. The use of a domain specific language for numerical applications such as Octave provides the user with a set of high level operations and specialised libraries. Octave contains specialised libraries for different fields. However, the price to be paid for this flexibility is computational performance, especially when production quality code is needed.

Compilation for array processing languages is a topic of ongoing research [3, 21, 34, 52]. Issues that distinguish their compilation from other languages are type and shape inferencing, which become a problem when a language has overloaded operators and “ad hoc” overloaded functions as is the case in Octave [36].

This chapter describes the implementation of an Octave compiler in Stratego. A compiled Octave program can be executed faster if dynamic type characterisation, matrix reallocation, reducing type verification to determine the correct function invocation can be avoided as much as possible.

To illustrate this situation consider the expression `a = f(a)` where `a` is a matrix of integer elements. If the compiler can determine that the result of the function call does not change the type or shape of the matrix `a`, there is no need to reallocate memory for the outcome of the call. We describe our approach for statically finding type information. Most of the infrastructure can be reused to implement other tools, such as a partial evaluator for Octave.

The rest of the chapter is organised as follows. Section 11.2 describes the Octave language and some of the problems that a compiler has to solve in order to obtain code suitable for compilation. This section discusses overloading in Octave and shows how specialisation may be used to resolve overloading. In Section 11.3 the architecture of the compiler is presented. Section 11.4 introduces the Octave language, a definition of a small language representing the types of Octave with a mechanism for providing type information of intrinsic functions. Using all the introduced elements we present a partial type inferencer for Octave in Section 11.5. The approach used to type user-defined and intrinsic functions is explained. Section 11.7 briefly describes the optimisations implemented for Octave. In Section 11.6 the performance evaluation achieved with the Octave compiler is discussed.

11.2 Overview of Octave

Despite the convenient features of Octave that enables quick prototyping, the language is also a source of ambiguities. For instance, the statement `M = X * Y` performs matrix multiplication if `X` and `Y` are matrices. The same statement performs

```
function retval = reshape (a, m, n)
  if (nargin == 2 && prod (size (m)) == 2)
    n = m(2);
    m = m(1);
    nargin = 3;
  endif

  if (nargin == 3)
    [nr, nc] = size (a);
    if (nr * nc == m * n)
      retval = zeros (m, n);
      if (isstr (a))
        retval = setstr (retval);
      endif
      retval(:) = a;
    else
      error('reshape: sizes must match');
    endif
  else
    usage('reshape(a, m, n) or reshape (a, size(b))');
  endif
endfunction
```

```
  c = [ 1 2 3 ];
  s = 'hello';
  c1 = reshape(c, 3, 1);
  s1 = reshape(s, 5, 1);
```

$$\Downarrow$$

```
function retval = reshape_string_int_int (a, m, n)
   [nr, nc] = size (a)
   if (nr * nc) == (m * n)
     retval = zeros (m , n);
     retval = setstr (retval);
     retval(:) = a;
   else
     error('reshape: sizes must match')
   endif
end
function retval = reshape_matrixInt_int_int (a, m, n)
   [nr, nc] = size(a)
   if (nr * nc) == (m * n)
     retval = zeros(m , n)
     retval(:) = a
   else
    error('reshape: sizes must match');
   endif
end
```

```
  c = [ 1 2 3 ];
  s = 'hello';
  c1 = reshape_matrixInt_int_int(c, 3, 1);
  s1 = reshape_string_int_int(s, 5, 1);
```

Figure 11.1: Typing by means of program specialisation.

multiplication of scalars such as integers, reals, complex numbers or any combination of them depending on the type of `X` and `Y`. The lack of variable, type, and shape declarations in Octave makes the static identification of which underlying function to call problematic. In the standard Octave implementation it is the interpreter which, based on type checks, resolves overloading and invokes the proper operation.

In addition, certain functions calls are interpreted according to the context in which they occur. The number and type of the actual arguments in a function call determine the behaviour of a function. Furthermore, a function can yield multiple return values determined by the return context. As an example consider the statement `p = find([ 1 0; 6 0])` where a single result is expected. This function yields a vector of indices of non-zero elements of a matrix. The result is a row if the argument of the call is a row or as a column otherwise. The function argument `[1 0; 6 0]` is internally represented as a vector containing the concatenated columns, i.e., $[1\ 6\ 0\ 0]$. Thus, the outcome of `p = find([ 1 0; 6 0])` is $[1; 2]$[2]. The same function called in a context where two results are required, as in the statement `[rows, columns] = find([ 1 0; 6 0])`, gives the non-zero positions *in the matrix* as a pair of coordinate vectors for the row and column positions. Thus the result of the call is $rows = [1; 2]$ and $columns = [1; 1]$. In the statement `[rows, columns, values] = find( [ 1 0; 6 0])` three results are expected: the rows, the columns and the non-zero values. This call gives as results: $rows = [1; 2]$, $columns = [1; 1]$ and $values = [1; 6]$. These examples show how contextual information is required in order to correctly compile calls to `find`.

Octave functions are defined with the construct `function` *result* `= fname(arg1, arg2, ..., arg`*n*`) stmts` where *result* denotes the result of the function `fname`. The list of identifiers enclosed in parentheses denotes the formal arguments of the function. The `stmts` part denotes the body of the function. With this construct it is not possible to specify and restrict the type of the formal arguments, and hence to perform some form of static type checking in order to prevent runtime errors. One way to prevent errors is to include restrictive type checks to enforce certain argument types in the body of the function. An example of how overloading is specified in Octave, we take a look at the function `reshape` which is defined in Figure 11.1. The `reshape` function is used to illustrate typical code that can be written in the Octave language, and in particular the use that can be made of contextual information This function was previously part of the Octave distribution. From the signature of the function we can see that the function is specified to be called with three arguments. In fact, the `reshape` function can also be called with two arguments. That is the reason for starting by inspecting the actual value of the implicit argument `nargin` which holds the number of actually given arguments. This variable is initialised for each call by the interpreter and it can be manipulated during interpretation. The `nargin` variable can be accessed in the definition of user functions. Observe that the code performs many type checks to find out what actually to compute. Most of these checks can statically be performed based on the arity and types of

[2] Octave starts numbering elements of matrices from 1 onwards.

arguments of the of a function call. Figure 11.1 shows how the `reshape` function has been specialised for two different instances of a reshape call which has been derived by propagating known information to the function definition in order to infer the resulting type. This example shows how context dependent information can be used for type inferencing. By means of function specialisation the result of type inferencing is expressed in context independent functions, and as a consequence subsequent compiler phases can deal with non-overloaded code.

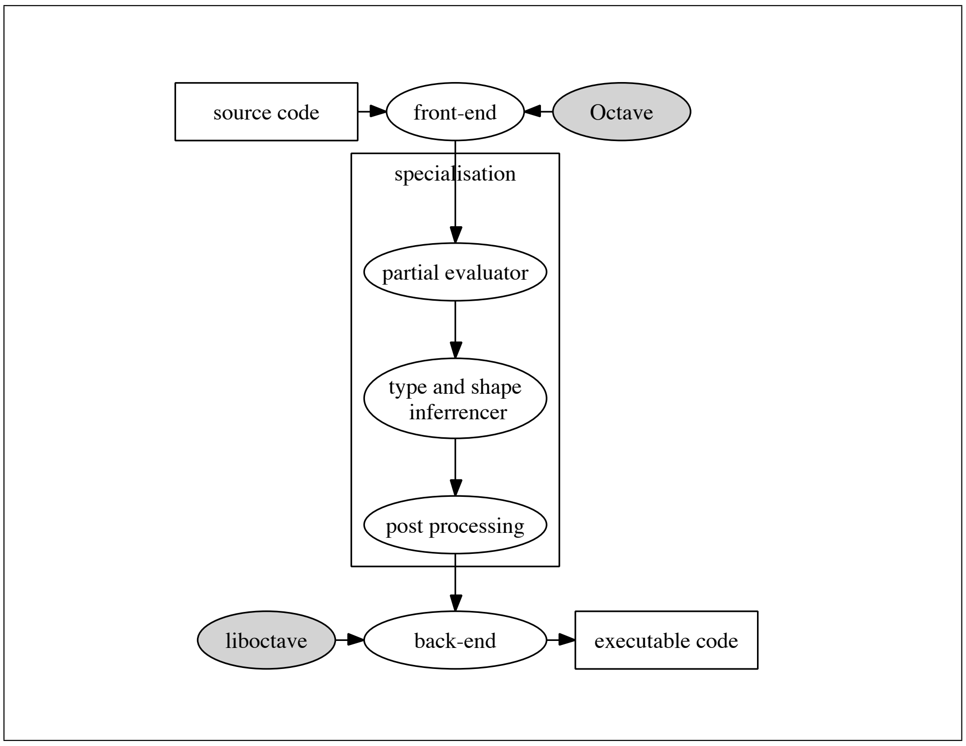

Figure 11.2: Overview of the system architecture.

11.3 Architecture of the compiler

The architecture of our `octave-compiler` is shown in Figure 11.2. It consist of three main components: the `front-end`, a series of optimising phases that are mainly based on specialisation, and the `back-end`, which reuses the `liboctave` library of the interpreter to generate executable code. The compiler is implemented in Stratego using the data-flow transformation described in previous chapters. It uses Octave for parsing.

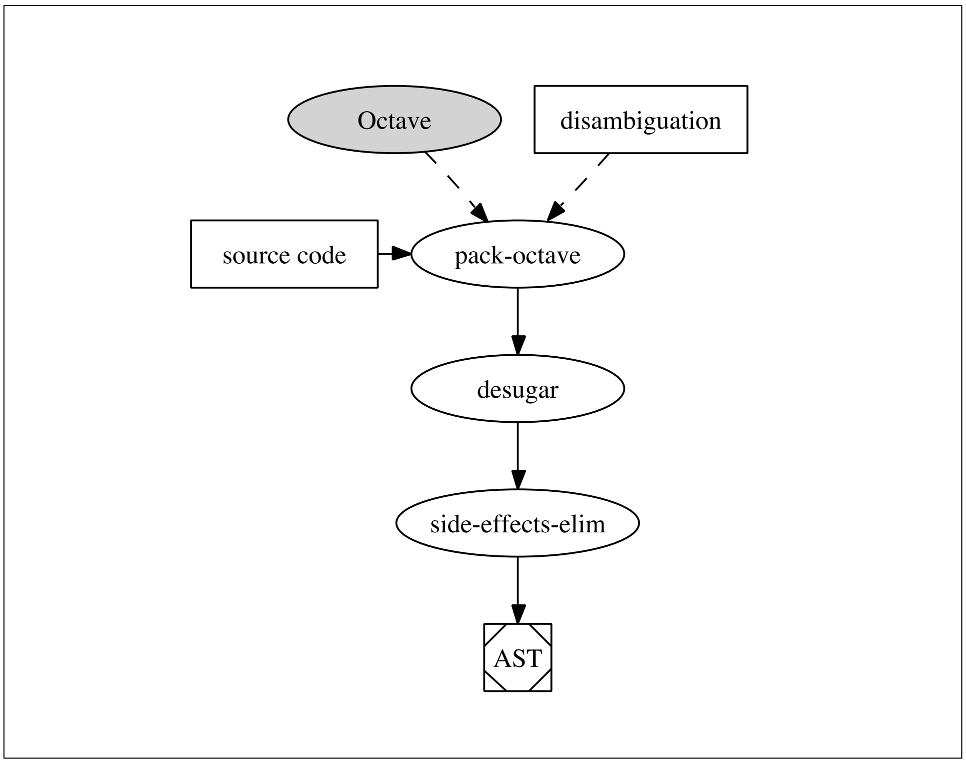

Figure 11.3: Overview of the front-end of the Octave compiler.

11.3.1 Octave Front-End

The goal of the front-end is to obtain unambiguous syntax trees for Octave programs. Octave source code is parsed using the Octave parser itself which was extended to generate abstract syntax trees in the ATerm format [14]. The obtained parse tree contains user-defined functions as well as functions that are part of the Octave distribution. The front-end is composed of the *pack-octave*, *desugaring* and *side-effect-removal* phases.

Pack-octave The octave parser works on a single file basis. Lack of variable declarations makes the Octave identifier name space ambiguous. To obtain the representation of a complete program, we need to disambiguate identifiers denoting function calls. If an identifier belongs to a function name space, the file containing the function will be parsed. The `pack-octave` component combines parsing and name disambiguation.

Disambiguation An identifier in Octave can refer either to a variable or a function call. Since Octave cannot assign to function names, we can safely assume that all identifiers occurring in the left-hand-side of an assignment refer to real variables. Without disambiguation, an erroneous interpretation can be given to a program as demonstrated in the following code:

```
function x = test(y)         function x = test(y)
   a = y + rand;               a = y + rand();
   rand(3) = a;         ⇒      rand[3] = a;
   x = rand(2) + 1;            x = rand[2] + 1;
end                          end
```

This example shows three occurrences of the identifier `rand`. The first occurrence is a function invocation without arguments. Note that for a function invocation the presence of parentheses is not required. The second occurrence denotes an assignment to the array `rand` with three elements. Although only the third element is being explicitly assigned, this statement will create an array with three elements, which is bound to the variable `rand`. The first and the second elements of the array will contain the value zero. The third occurrence of `rand(2)` denotes a subscripted expression for the array `rand` accessing the second element. The same expression in the absence of the second statement has a completely different interpretation, i.e., `rand(2)` denotes a function invocation which produces a two by two matrix of random values.

In the right box of the disambiguation example we show the resulting program where all ambiguities have been solved. The right box internally uses a pseudo Octave language which does not contain ambiguities. In the example the subscript expressions are represented with square brackets and functions calls are represented by parentheses. Square brackets are not part of the Octave syntax; however, here they are used to show that after the disambiguation phase the program represents function calls and subscripts expressions differently.

Desugaring To ease subsequent phases the parsing result is converted into a simpler representation reducing the number of constructs. Complex statements such as `switch` are desugared into semantically simpler ones as is illustrated in the example:

```
switch a              if a == 1
  case (1)             then x = 1
    x = 1;             else
  case (2)              if a == 2
    x = 2;                then x = 2
  case (3)                else
    x = 4;        ⇒        if a == 3
  otherwise                 then x = 4
    x = 5;                  else x = 5
end                        endif
                        endif
                      endif
```

11.3.2 Side-Effects Removal

Side-effects in expressions are removed by mapping them onto statements which make the effects explicit. Consider the example:

```
a = [10,11];         a = [ 10 11 ];
++a;            ⇒    a = a + 1;
b = ++a ;            a = a + 1;
                     b = a;
```

Octave provides an interactive environment which makes it possible to inspect the state of a variable at any time. If the result of an expression is not assigned to a variable the value is by default assigned to the built-in variable `ans`. This transformation also makes such implicit assignments explicit. The next example shows these type of situations:

```
r = 100 * 10;        r = 100 * 10;
r +  50;        ⇒    ans = r + 50;
ans + 40             ans = ans + 40
```

11.3.3 Type-shape Inference

Octave does not have an explicit notion of types. Type casting is transparently done when the result/context of a computation requires it. We could use the widest data type which can represent all types properly, but this leads to a waste of memory allocation for computations which require a narrow type. The goal of the Octave type inferencer is to extract as much type information as possible and to represent it explicitly. Without proper type information, the program needs dynamic type conversion.

Shape inference determines the structure and size of a matrix. Inferring exact shape information can avoid run-time memory allocation. Furthermore, shape information is vital for matrix calculations, since they impose shape restrictions on their operands. For instance the multiplication of two matrices requires the number of columns of the first matrix to be equal to the number of rows of the second matrix and for matrix addition we require that the matrices have the same shape. Moreover, single elements are considered as one by one matrices. These matrices are referred to as *scalars*.

The type-shape inferencer is implemented by traversing the abstract syntax tree following a forward traverse flow. The next example shows how specialisation can eliminate "ad hoc" function overloading. Different specialised functions are created for different type invocations.

```
function x = f(a, b)
  x = a + b;
```

```
c = 4;
r = f(c, c);
y = f([3, 4], r);
```

⇒

```
function x = f_matrixInt_int(a,b)
   x = a + b;
end
function x = f_int_int(a,b)
   x = a + b;
end
```

```
c = 4;
r = f_int_int(c, c);
y = f_matrixInt_int([3 4], r);
```

User functions inherit overloading features from operators. Arguments of the function call `f` can be instantiated with different data types. The type inferencer determines the type of the arguments and propagates this information to specialise functions calls corresponding to the types of the arguments of the call. Octave intrinsic functions are built into the interpreter and the only thing we can infer is the fact that they are called. For inferring types from intrinsic functions however, function specialisation and type inferencing do not help; our solution is discussed in Section 11.4.3.

11.3.4 Back-end

After type inferencing and specialisation, the program is translated into a C++ program. Type information is used to map to the right C++ computation, thus avoiding some of the run-time checks and function dispatching overhead performed by the interpreter. An executable is produced using the `mkoctfile` tool from the Octave distribution. The generated program reuses the `liboctave` library to produce stand-alone executables. The translation from Octave to the target language has a completely rule-based description and uses concrete syntax of a C++ subset. The description of the translation is omitted here.

11.4 Language Definition for Octave Compilation

This section introduces the languages constructs used for implementing the Octave compiler, namely the Octave language and a small language to represent type information of Octave programs. The syntax of Octave is ambiguous, and thus we start out by defining an unambiguous language that resembles Octave in Section 11.4.1. The language for representing extracted Octave types is presented in Section 11.4.2. The newly introduced type constructors make it possible to express rules and strategies in the concrete syntax of Octave.

In order to steer type inference of both intrinsic functions and user defined functions, we have defined an extensible way to make type information for Octave functions explicit. The language to express Octave types is defined in Section 11.4.3.

fd	::=	`function` e `=(`$e_1, ..., e_n$`)=` e_{n+1} `end`	`Function/4`	function definition
	$\mid$	`function [`$e_1, .., e_n$`]=(`$e_{n+1}, ..., e_m$`)=` e_{m+1} `end`	`Function/4`	function definition
e	::=	lv	`LValue/1`	l-value
	$\mid$	$str \mid i$	`Str/1`, `Int/1`	string, integer
	$\mid$	r	`Float`	float
	$\mid$	`(`e_1`:`e_2`:`e_3`)`	`Range`	range
	$\mid$	`[`$r_1, ..., r_n$`]`	`Matrix`	matrix
	$\mid$	$\{r_1, ..., r_n\}$	`Cell`	cell
	$\mid$	$pre\, e$	`UnOp/2`	pre operator
	$\mid$	$e\, post$	`UnOp/2`	post operator
	$\mid$	$e_1\, bop\, e_2$	`BinOp/3`	arithmetic
	$\mid$	$e_1\, rop\, e_2$	`RelOp/3`	relational
	$\mid$	$e_1\, op\, e_2$	`RelOp/3`	boolean
	$\mid$	$lv\, aop\, e$	`Assign/2`	assignment
	$\mid$	`[`$lv_1, .., lv_n$`] =` e	`AssignMulti/2`	multi assignment
	$\mid$	f`(`$e_1, ..., e_n$`)`	`Call/2`	function call
	$\mid$	$\{e_1$`;`...`;`$e_n\}$	`Stats/1`	sequence
	$\mid$	`if` e_1 `then` e_2 `else` e_3 `end`	`If/3`	conditional
	$\mid$	`if` e_1 `then` e_2 `end`	`IfThen/2`	conditional
	$\mid$	`while` e_1 `do` e_2 `end`	`While/2`	while oop
	$\mid$	`do` e_1 `until` e_2 `end`	`DoUntil/2`	repeat loop
	$\mid$	`switch` e cs `end`	`Switch/2`	switch
	$\mid$	`for` e `=` e_1 `do` e_2 `end`	`For/3`	for loop
	$\mid$	`try` e_1 `catch` e_2 `end`	`TryCatch/2`	exception
	$\mid$	`break`	`Break`	unconditional jump
	$\mid$	`continue`	`Continue`	unconditional jump
	$\mid$	`return`	`Return`	unconditional jump
lv	::=	`x`	`Var/1`	variable
	$\mid$	lv`.f`	`FieldVar/2`	fieldvar
	$\mid$	lv`[`$e_1, ..., e_n$`]`	`Subscript/2`	subscript
	$\mid$	$lv\{e_1, ..., e_n\}$	`CellIndex/2`	cell index
cs	::=	`case` $e_1\, e_2$	`Case/2`	switch case
	$\mid$	`otherwise` e	`Default/1`	default case
r	::=	`[`$e_1, ..., e_n$`]`	`Row/1`	list of expressions
aop	::=	`=` | `+=` | `*=` | `-=` | ...	`AssignOp`	assign operator
pre	::=	`++` | `!`	`PREINC,..`	prefix operator
$post$	::=	`++` | `'`	`POSTINC,..`	postfix operator
bop	::=	`+` | `-` | ...	`PLUS, MINUS`	binary operator
rop	::=	`<` | `=` | ...	`GT, EQ`	relational operator
op	::=	`&` | `|`	`AND, OR`	boolean operator

Figure 11.4: Abstract syntax of unambiguous Octave which includes explicit constructor and arity information for the defined language constructs.

[*x*,*y*,*z*][0 − 9\']*	->	`Var`	Octave variable
[*f*,*g*][0 − 9\']*	->	`Id`	Octave function name
[*i*,*j*,*k*][0 − 9\']*	->	`IntConst`	Octave integer
[*r*][0 − 9\']*	->	`FloatConst`	Octave float
[*s*][0 − 9\']*	->	`String`	Octave string
[*e*][0 − 9\']	->	`Exp`	Octave expression
[*e*][0 − 9\']"*"	->	{Exp";"}*	list of Octave expressions
[*x*][0 − 9\']"*"	->	{Exp","}*	list of Octave formal parameters
[*a*][0 − 9\']"*"	->	{Exp","}*	list of Octave function arguments
"*lvs*"[0 − 9\']"*"	->	{Exp","}*	list of Octave l-values

Figure 11.5: Syntax definition of Octave meta-variables.

<*s* >	->	`Exp`	congruence for expressions
<id:*s* >	->	`Id`	congruence for expressions
<multi: *s* >	->	`Exp+`	congruence for list of lvalues

Figure 11.6: Congruence strategies for Octave.

11.4.1 Concrete Syntax for Octave

Since the Octave syntax is ambiguous, we use a concrete syntax that is similar to Octave, but unambiguous. Since the changes are rather straightforward we do not describe them explicitly and which is used in the implementation of rules and strategies.

The new syntax represents function calls by the name of the function and a (possible empty) list of arguments enclosed between parentheses. In order to distinguish them from function calls, matrix accesses (subscripts) are represented by the name of the matrix and index expressions enclosed between square brackets, thus eliminating ambiguities between function calls and subscripts. Figure 11.4 presents an unambiguous grammar for the Octave language. With this new syntax, rewrite rules and strategies can be expressed in a natural way. It also turns out to be more concise than an equivalent specification based on the abstract syntax tree notation. To distinguish meta code from object code, Octave fragments are enclosed between quotation symbols: ⟦ and ⟧.

Figure 11.5 defines meta-variables used inside fragments with concrete syntax. Figure 11.6 defines congruence strategies for Octave.

11.4.2 Concrete Syntax for Octave Types

Octave data types are organised into: (1) Scalar types such as *Boolean*, *Integer, Float, Complex*, (2) *Characters* and (3) *Matrices.* Matrices are n dimensional arrays

t	::=	`bool`	`BOOL`	boolean
	|	`int`	`INT`	integer
	|	`float`	`FLOAT`	float
	|	`complex`	`COMPLEX`	complex
	|	`char`	`CHAR`	char
	|	`universal`	`ANY_TYPE`	any Octave type
ts	::=	t	`SCALAR/1`	scalar
	|	`string(`e`)`	`STRING/1`	string
	|	`matrix(`t`,dim(`$e_1,..,e_n$`))`	`MATRIX/2`	type and shape of matrix
	|	`cell{`$ct_1...ct_n$`}`	`CELL/1`	cell type
	|	`struct{`$vt_1...vt_n$`}`	`RECORD/1`	record type
e	::=	id	`Var/1`	variable
	|	i	`Int/1`	integer

Figure 11.7: Abstract syntax of Octave type and shape Information, including constructor and arity information for the defined elements.

`[t][0-9\']*`	`->`	`Type`	Octave type
`"sh"[0-9\']*`	`->`	`{Exp","}+`	Octave expression list
`"sh"[0-9\']`	`->`	`Exp`	Octave expression
`"ts"[0-9\']`	`->`	`TypeShape`	Octave typeshape

Figure 11.8: Syntax definition of Octave types meta-variables.

of scalar and string types. A String is a special matrix of characters. Boolean types are represented by integer values as in C.

We have defined a *type* language for representing the types inferred from Octave expressions. Thus, the implementation of the type inferencer combines the syntax of Octave and the the syntax of the type language, both in their concrete syntax form. Figure 11.7 presents the abstract syntax of Octave types. Notice that shape information is also part of the type. A matrix data type is characterised by the type of its elements and its dimensions. The type of the matrix `[2,3;6,7;0,1]` is represented by `MATRIX(INT,[Int(3),Int(2)])`. In Figure 11.7, the type `universal` is included to represent unprecise types. Figure 11.8 shows meta-variables for Octave types. The use of meta-variables allows to represent type terms in a brief and clear manner. The meta-variables `sh` and `sh*` are introduced to refer to the dimension of the shapes. The term `MATRIX(INT,[Int(3), Int(2)])` can be matched against the strategy `?|[ matrix(`t`, dim(3,2) ]|` using concrete syntax of types shown in Figure 11.7. The term pattern t is bound to the term `INT`.

pd	::=	`declarations` `constants` c^* `variables` v^* `functions` fs^* `mapping functions` mf^* `user functions` ufd^*	`ProgDecls/5`	program declaration
c	::=	id `:=` ts	`Constant/2`	typed constant
v	::=	id `:=` ts	`Variable/2`	typed variables
fs	::=	$id :: ts^*$	`FunctionSig/2`	function signature
mf	::=	$id :: ts^*$	`MFunctionSig/2`	mapping function signature
uf	::=	`function` $id(\{id\,;\}^*) : \{id{:=}ts\}^+$	`UFunctionDecl`	function declaration

Figure 11.9: Language for declaring intrinsics types of Octave.

11.4.3 Controlling Type Information

Some of the Octave computations cannot be typed statically. A source of imprecision stems from the intrinsics of the Octave library. We have designed an extendable scheme to provide type information for intrinsic functions, and variables and also extend the language such that type information can be given explicitly for user developed functions.

Figure 11.9 shows the abstract syntax for the scheme. The declarations are composed of five sections to declare types of constants, types of variables, types of intrinsic functions, types of mapping functions and types of user functions. A short example in which types are supplied is given in Figure 11.10. Note that several entries for the functions `abs` and `size` are displayed. Each entry describes one possible use and gives the corresponding result type.

11.5 Inferring Octave Types

The type inferencer extends the data propagation techniques developed in Chapters 7, 8, and 9 for optimisers [75, 76]. We describe how overloading in Octave is resolved by combining type-shape inferencing and function specialisation.

Expressions Type information is added to each expression using Stratego term annotations. An annotated term $t\{t'\}$ associates a term t' with the term t, while keeping the signature or the structure of the term t. Annotating expressions with their type is performed by rewrite rules which are applied in a bottom-up traversal over the abstract syntax tree. Each type rule infers the type of the expression from the operator and the inferred types of its arguments. An example of a typing rules is:

```
TiExp  :  Var("true")-> Var("true"){SCALAR(BOOL())}
```

```
declarations

constants
  e       := float
  eps     := float
  pi      := float

variables
  beep_on_error  := int
  warn_fortran_indexing := int

functions
  disp      :: universal -> string
  iscell    :: universal  -> bool
  iscellstr :: universal  -> bool
  isglobal  :: string -> bool
  size      :: universal, int -> int
  size      :: universal      -> int, int
  size      :: universal      -> matrix(int)

mapping functions
  abs       :: bool  -> int
  abs       :: int  -> int
  abs       :: float  -> float
  abs       :: complex  -> float

user functions
  myfun     :: int ->int
```

Figure 11.10: Explicit types for intrinsic functions.

The rule `TiExp` types a Boolean expression, in which the right-hand side of the rule attaches the type annotation to the expression. The symbols `::` are defined to distinguish the expression from the annotation when using concrete syntax. The previous rule with concrete syntax is represented as:

```
TiExp  :  |[ true ]| -> |[ true :: bool ]|
```

Some type inferencing rules for expressions are given in Figure 11.11. The first six rules type leaf nodes for Booleans, integers, floats, strings and complex numbers. The last three rules type some binary operations. For brevity, only a small subset of all the rules is shown; the rest of the rules are specified in a similar fashion. Observe that these rules are specified using the concrete syntax of the object language. For example, binary operations are specified using infix notation.

Matrix Expressions Matrix expressions are typed using rules which are similar to those used in typing scalar expressions. Figure 11.12 defines a rule which

```
TiExp : |[ true  ]| -> |[ true  :: bool   ]|
TiExp : |[ false ]| -> |[ false :: bool   ]|
TiExp : |[ i   ]|    -> |[ i     :: int    ]|
TiExp : |[ r   ]|    -> |[ r     :: float  ]|
TiExp : |[ s   ]|    -> |[ s     :: string ]|
TiExp : |[ e + e1 I  ]|  -> |[ (e + e1 I) :: complex ]|

TiExp : |[ e1 :: ts1 + e2 :: ts2 ]| -> |[ ( e1 :: ts1 + e2 :: ts2 ) :: ts ]|
  where  <is-numeric-type> ts1
       ; <is-numeric-type> ts2
       ; !SCALAR(<calculate-max-numeric-type>[ts1, ts2]) => ts

TiExp : |[ ++( e :: ts ) ]| -> |[ ++( e :: ts ) :: ts ]|

TiExp : |[ !( e :: matrix(t, dim(sh*)) ) ]| ->
           |[ !( e :: matrix(t, dim(sh*)) ) :: matrix(bool, dim(sh*)) ]|

 is-numeric-type =
   is-bool-type <+ is-int-type <+ is-float-type <+ is-complex-type
```

Figure 11.11: Type inferencing rules for simple Octave expressions.

```
 TiExp : |[ e ]| -> |[ e :: matrix(t, dim(i,j)) ]|
  where <calculate-matrix-base-type> e => t
       ; <calculate-nr-of-rows> e=> i
       ; <calculate-nr-of-columns> e => j

 TiExp : |[ e :: ts + e1:: ts1 ]| -> |[ (e :: ts bo e1 :: ts1):: ts2 ]|
  where  <matrix-matrix> (ts, ts1) => ts2
```

Figure 11.12: Type inferencing rules for matrix operations.

infers the type of matrix expressions. The first rule types a matrix with the strategy `calculate-matrix-base-type` which computes a type for the elements of the matrix and the strategies `calculate-nr- of-rows` and `calculate-nr-of-columns` calculate the shape of the matrix. The second rule in the figure types binary matrix operations. The strategy `matrix-matrix` enforces that the operands have the same shape.

Matrix Expansion In Octave a function or operation on scalars can be lifted so it can be applied to matrices. For example, consider the expression `abs([-2,5;3,1])`. While the function `abs` is defined only for scalar types, by lifting it can be applied to every element of the matrix. This feature in Octave is called *matrix expansion* and it applies to both user defined and intrinsic functions. Calls to mapping functions are typed by rule:

```
TiCall : |[ f(e :: matrix(t, sh)) ]| -> |[ f(e :: matrix(t, sh)) :: matrix(t1, sh)]|
       where <is-mapping-function> f; <TypeOf> (f, t) => t1
```

```
TiAssg: |[ x = (e :: ts) ]| -> |[ x :: ts = (e :: ts) ]|
  where <create-type-rule> (|[ x ]|, ts)

TiAssg: |[ lvs* = e ]| -> |[ lvs1* = e ]|
  where
   switch <get-type> e => ts
    case is-octave-value-type: <map(set-universal-type)> lvs*
    case is-list             : <zip(create-type-rule)>(lvs*, ts)
   end => lvs1*

set-universal-type: |[ x ]| -> |[ x :: universal ]|
 where <create-type-rule>(|[ x ]| , |[ universal ]|)

create-type-rule : (|[ x ]|, ts) -> |[ x :: ts ]|
    where rules( TypeOf.x : |[ x ]| -> ts depends on [(x, x)] )
```

Figure 11.13: Typing inferencing rules for assignments.

Assignments In the absence of variable declarations, assignments serve as a kind of declarations where the type of a variable is determined by the type of the assigned value, which is inferred from the right-hand side of the expression. Inferred types from variables are propagated toward their use by dynamic rules [102]. Figure 11.13 shows how dynamic rules are created at assignments. The rule `TiAssg` matches an assignment `x = (e :: ts)`, which associates the expression `e` with type-shape `ts` to the variable `x`. For this specific variable `x`, a dynamic rule `TypeOf` is generated, which annotates `x` with the type-shape `ts`. The existence of a dynamic rule `TypeOf` propagates type-shape information of a variable to its uses. A multiple assignment of the form `lvs* = e` creates multiple rules to propagate corresponding types for each variable being assigned to. When type inference fails to discover the precise type of an expression it associates the widest Octave type, denoted as `universal`.

11.5.1 Type Inference of Functions by Specialisation

In Octave function definitions are flexible. Octave function signatures define a number of incoming or returning arguments, however, a function call can provide a different number of arguments, or specify a different number of returning values. To achieve a cleaner internal representation in the Octave compiler, functions are tailored to match the function call.

There are two kinds of Octave functions. On the one hand, intrinsic functions are functions that are part of the language and are available in all Octave programs. On the other hand, scripts and user-defined functions are described in the language itself and are stored in ".m" files.

User Defined Functions Octave functions, defined by the user, inherit the following language features: operator overloading, lack of declarations and "*ad hoc*" function overloading. Another characteristic is that user defined functions can also accept and yield a different number of values depending on the context of the call.

```
ti-assign(s) =
  {| NumArgsOut:
       ?|[ lvs* = e ]|
      ; where( <length>lvs* => num; rules(NumArgsOut: () -> num))
      ; |[ <multi: id> = <s> ]|
      ; TiAssg
  |}

ti-assign(s) =
  {| NumArgsOut:
     ?|[ x = e ]|
     ; where( rules(NumArgsOut: () -> 1) )
     ; |[ <id> = <s> ]|
     ; TiAssg
  |}
```

Figure 11.14: Rules to provide context information.

When we can statically infer the types of the arguments of an overloaded function, a selection of the right type instance can be made. Once the type instances of a function call are known, by using specialisation, the type of the function result(s) can be inferred. Context information such as the number of expected results of a function call are injected in the inference process by means of dynamic rules. Values for *built-in variables* such as `nargin` and `nargout` are added to eliminate branches of the program that are not used and/or do not contribute to the required computation.

Figure 11.14 shows how context information is discovered and injected into the system. At each assignment we compute the number of the expected values, which is propagated by the dynamic rule `NumArgsOut`.

Dealing with Intrinsics To type intrinsic functions, context information is also required (recall the example on the function `find` discussed in the introduction). The question arises how to deal with functions that can yield a different results when the context of the call is different. Our solution is to include information about intrinsic functions explicitly using the language given in Figure 11.10. Although the idea to provide type information is not new [34], our approach considers functions with multiple results. The behaviour of an intrinsic function is quite irregular. Consider the following situations:

(1) Certain functions are defined only for scalar types, the number of input arguments and the number of results is fixed, and the outcome value is also a scalar type. These functions can be statically typed provided information about the argument types For example the function `abs(4.3 + 2i)` will result in `4.7424`. The `abs` function, when called with an *integer* as argument, gives as a result an *integer* number. If called with a *float* number it returns a *float* number, and when called with a *complex* number it returns a *float* number..

(2) Intrinsic functions can be typed by using information about the value(s) of the

argument of the call. The function `sqrt(`n`)` will give a *float* value as a result for n at least zero. For negative instances of n the result is of type *complex*. Functions with such a kind of behaviour are the main reason why type inference will always be imprecise for Octave. In such cases the type system gives the *universal* type as a result.

(3) Certain functions can accept different numbers of arguments and the number of results is fixed. As an example consider `zeros(4,3)`, which will yield a matrix containing integer values, namely the number of zeroes. The function `zeros` can also be called with none or more arguments. Examples of these functions are `eye`, `rand`, `ones` that can be statically typed.

(4) Certain functions can accept a variable number of input arguments and produce a variable number of return values. Some of these functions can be typed with additional context information. An example is the function `size([1 3 6])`, which can yield one, two or more return values. To infer the type for such functions we associate all possible outcome types and the selection of the proper type is delayed until the point where further information is encountered.

In order to specify different possible behaviour of intrinsic functions and variables, the language introduced for describing types of intrinsic functions enables us to deal with the described issues. The controlling type language is shown in Figure 11.9. The type information is injected into the type inferencing system using extra dynamic rules.

11.5.2 Octave Function Specialisation

In Chapter 10 we have studied how to implement partial evaluation, of which function specialisation is a special case. In this section we use type information and context information to construct a residual Octave program. Data type information inferred by the system triggers specialisation. Furthermore, part of the information available to the interpreter, is also applied during specialisation. The information includes the arity of function calls, the expected number of results, and the types and shapes of *intrinsic functions.* Thus, residual programs restrict the types that can be assigned to function arguments and results as much as possible. We have implemented an online function specialiser. It determines *static* type information during transformation and propagates the information as it is encountered. Since the transformation only propagates type information, the normal risks of online partial evaluation (such as non-termination) can be controlled. The typical disadvantage of this approach is that it may increase code size. However, as a side effect of function specialisation, branches that are not reachable are eliminated from residual programs. Such overhead is common in Octave programs. Furthermore, the obtained code is more suitable for compilation and optimisation.

Figure 11.15 shows the implementation of the strategy for type inferencing of user defined functions. The strategy `generate-specialisation-rule` matches a function definition and generates a `specialise-call` dynamic rule to match function calls to the defined function. The rule `specialise-call` it is parameterised with

```
generate-specialisation-rule =
 ?|[function x0* = f(a0*) e0 end]|
 ; rules(
   specialise-call(specialise-function,specialisation-facts| nargout) :
   |[ f(a*) ]| -> |[ g(a1*) :: ts* ]|
   where <specialisation-facts(|a0*)>a* => (a1*, facts)
     ; ( <specialised-function>(f,nargout,facts)=> |[function x1* = g(a1*) e1 end]|
        ; <map(get-type)>x1*; try(?[<id>]) => ts*
      <+
       <specialise-function(|facts,nargout)> |[function x0* = f(a0*) e0 end]|
           => |[function x1* = g(a1*) e1 end]|
       ; <map(get-type)>x1*; try(?[<id>]) => ts*
       ; rules(
          specialised-function:+ (f,nargout,facts) -> |[function x1* = g(a1*) e1 end]|
          specialised-functions:+ _ -> |[function x1* = g(a1*) e1 end]|
         )
       )
  )

specialise-function-by-type(|facts,nargout) :
 ?|[ function x0* = f(a0*) e0 end ]| -> |[ function x1* = g(a1*) e1 end ]|
   where
    alter-name-by-type(|f, facts) => g
   ; <map(create-type-of-rule)>facts => a1*
   ; <infer-octave-types> e0 => e1
   ; <map(annotate-type(calculate-type-of))>x0* => x1*

 specialisation-facts-by-type(|a*) =
    ?a1*
  ; <zip((?Var(<id>),get-type ; ?[<id>] <+ !|[ universal ]|))>(a*,<id>)
  ; !(a1*,<id>)
```

Figure 11.15: Typing Octave user defined functions.

the rule `specialise-function`, the strategy `specialisation-facts` and the number of requested results (`nargout`). To avoid repetitive work the specialisation rule is memoized in the rule `specialised-function` which uses the function name, the number of results and the specialisation facts to retrieve the result of a function specialised for those values. If there are no functions specialisation for the current function call yet, the `specialise-function` strategy is used to generate a specialised function. The resulting type(s) are obtained by extracting the type from the result expression *x1**. The dynamic rules `specialised-function` and `specialised-functions` record the results. The former avoids repetitive work by memoizing the obtained result. The later maintains the whole list of specialised functions to be added to the generated Octave program. Figure 11.15 shows the implementation of the parameter strategies of the rule `specialise-call`, `specialise-function-by-type` and `specialisation-facts-by-type` respectively. The Octave type inferencing closely follows the implementation of the specialiser discussed in Chapter 10.

Benchmark	Synopsis	M-Files
apt	Adaptive Quadrature by Simpson's Rule	`drv_adapt.m, adapt.m`
capr	Transmission Line Capacitance	`drv_capacitor.m, capacitor.m, gauss.m seidel.m`
clos	Transmission Closure	`drv_closure.m, closure.m`
crni	Crank-Nicholson Heat Equation Solver	`drv_crnich.m, crnich.m, tridiagonal.m`
diff	Young's Two-Slit Diffraction Experiment	`drv_diffraction.m, diffraction.m`
dich	Dirichlet Solution to Laplace's Equation	`drv_dirich.m, dirich.m`
edit	Edit Distance	`drv_editdist.m, editdist.m`
fdtd	Finite Difference Time Domain (FDTD) Technique	`drv_fdtd.m, fdtd.m`
fiff	Finite-Difference Solution to the Wave Equation	`drv_finediff.m, finediff.m`
nb1d	One-Dimensional N-Body Simulation	`drv_nbody1d.m, nbody1d.m`
nb3d	Three-Dimensional N-Body Simulation	`drv_nbody3d.m, nbody3d.m`

Table 11.1: MAT2C Benchmark Suite.

11.6 Performance Evaluation

This section presents some summarised results pertaining to the quality of the C++ code generated by the **octavec** compiler. Table 11.1 describes the list of programs comprising the MAT2C Benchmark Suite [51]. The programs were originally implemented for MATLAB and they are compatible with Octave.

The measurements were carried out using a Mac OS X Intel Core 2 Duo with 1GB of main memory. We have used Octave version 3.0.3 [36][3]. The C++ generated sources were compiled with GCC i686-apple-darwin8-g++-4.0.1. The ".m" source files were compiled with the transformations: constant-propagation, partial-evaluation, dead-function-elimination, dead-code-elimination. These transformations were applied to obtain code suitable for compilation.

In Table 11.2 the obtained execution times are presented. The table shows different types of execution: interpretation of the original programs (interpreted), interpretation of the programs after optimisation (specialised), and the execution times after compilation (octavec). In Figure 11.16 we compare the execution time of the Octave interpreter between the original code versus the code obtained after source to source transformations. In general the transformed code is slightly faster than the original code. The average performance improvement in the benchmark suite is 8%. The program with the most improvement is `diff`. A closer look at the generated output reveals that the program has shrunk from 122 lines of code to 64. The

[3]We only consider a substantial subset of Octave. Late language extensions such as function nesting are not yet handled.

Benchmark	**Execution time in seconds**		
	Interpreted	Specialised	Octavec
apt	4.43	4.55	3.56
capr	27.36	27.26	18.43
clos	0.7	0.5	0.17
crni	18.29	18.3	10.7
diff	19.6	10.52	5.03
dich	15.99	15.99	10.18
edit	7.34	7.62	4.25
fdtd	25.47	25.05	21.59
fiff	13.92	13.52	8.11
nb1d	32.02	27.8	11.02
nb3d	38.07	35.69	31.27

Table 11.2: Execution times for MAT2C Benchmark Suite.

original program has several checks to restrict the number of input and type of the arguments of function calls. Such checks can be optimised away with the inferred information from the actual arguments used in the function calls in the program.

Figure 11.17 shows a comparison of the programs in the benchmark between the interpreted versus the compiled execution times. The results show an average improvement of 42%. The programs with major benefit are `clos, diff` and `nb1d` with an improvement over 65%. The programs with not so prominent benefit are `apt` and `fdtd` with increase on performance of factor 15% and 19% respectively. The `apt` and `fdtd` programs contain vectorised operations for which the language and `liboctave` have optimised implementations. These programs contain little overhead to be removed.

We would like to compare our results to the compilers for MATLAB. Unfortunately MALTAB and Octave are sufficiently different. Octave is implemented in C++ and uses Object Oriented programming mechanisms to execute type-dispatching. Code generation is targeted to reuse `liboctave` and performance is dependent on the quality of its implementation.

11.7 Optimising Octave

Although the main purpose of this chapter is to discuss the implemented set of transformations to compile Octave, we have also implemented several optimisations for Octave. Data-flow optimisation such as constant-propagation, common-sub expression elimination, forward expression propagation and dead code elimination are implemented by reusing the generic data-flow optimisations presented in this thesis. Besides type-based function specialisation, we have implemented value based function specialisation.

The implementation of the Octave compiler can be improved with optimisations such as vectorisation, algebraic rephrasing at matrix operation level and memory

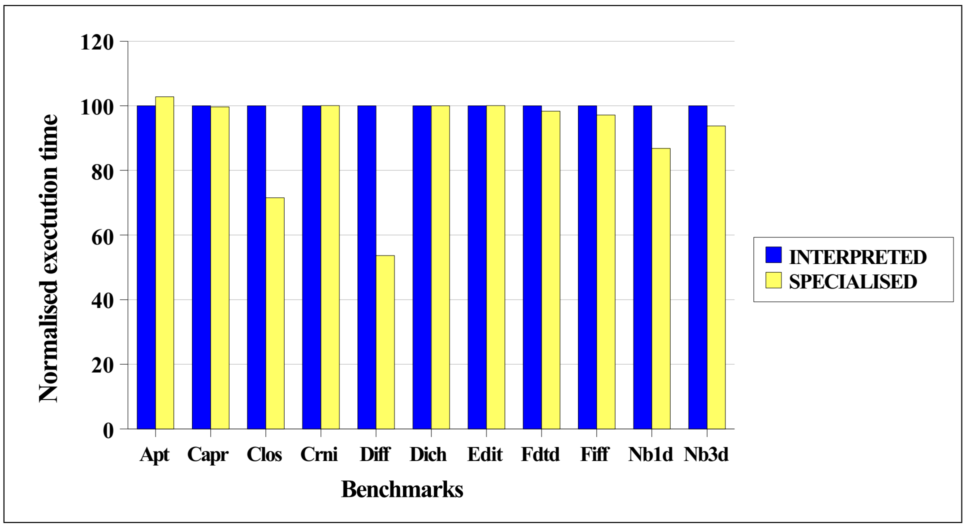

Figure 11.16: Original vs. specialised interpreted execution time

pre-allocation. Type information can be exploited to restructure the way the compiler executes computations. Optimisations from the high performance computing community are not yet applied. Transformations such as vectorisation, algebraic rephrasing, matrix restructuring operations can be added to the compiler to further improve the obtained performance gain.

11.8 Summary

In this chapter we have incorporated all implemented transformations discussed in this dissertation to be part of the Octave compiler. We have implemented constant propagation, common subexpresssion elimination, dead code elimination and forward propagation reusing the introduced propagation strategies. Partial evaluation is also part of the optimisations which shows more impact on optimising Octave programs. We have shown the developed transformations used by real programming language and enabling compilation.

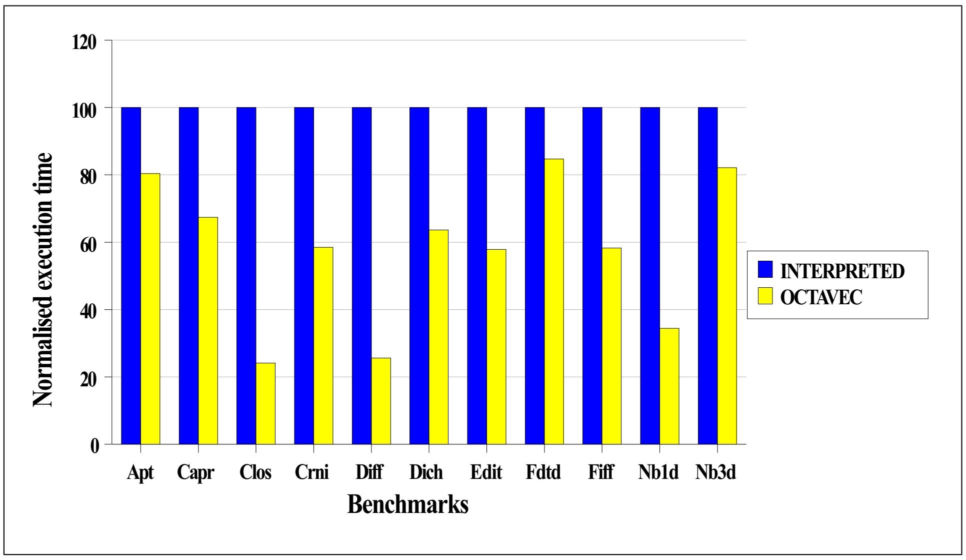

Figure 11.17: Compile vs. interpreted execution time.

12 Conclusion

This chapter concludes the thesis by discussing the contribution of this work and possible future research avenues.

The main objective for the research conducted in this thesis was to develop language support to facilitate the development of program transformations. Rewrite rules and strategies have been extended to cope with the development of program transformations which require context-sensitive information. This work contributes to the scarce set of tools that simplify the implementation of compilers or program transformation systems [50, 42].

We envisioned meta-language support to contribute to the productivity of the transformation developer, to be object language independent, and to allow the implementation of analysis and transformation tools on a high level of abstraction while enabling the combination of multiple transformations at the same time. Next we take the reader to the taken path to achieve such objectives.

Dynamic Rules

Dynamic rewrite rules play a very important role in Stratego for the development of transformation requiring context-sensitive information. Such rules abstract over complex data structures and allow access to information which is defined beyond the scope of static rewrite rules. From this perspective dynamic rules represent a mechanism for transporting information in an implicit way. Dynamic rules not only provide means to propagate information but to express modifications using such information. These rules can be applied in any defined traversal strategy, be it generic or programmed. Information that dynamic rules transport is updated according to the dynamic rule operations to reflect current state.

Dynamic Rule Operations

Initially dynamic rules were designed with the following operations: definition, update and un-definition. For the implementation of data-flow optimisations we have designed operations for coping with control-flow of imperative languages. The dynamic operations have been extended in two ways: 1) The operations can be applicable to a set of dynamic rules and 2) The extended operations are intersection, union and a combination of both. Additionally the operations can be applied to a fix point.

The designed operators extend the expression power of dynamic rules in Stratego. Such operations allowed us to write a concise implementation for constant propagation. However, for optimisations such as common sub-expression elimination which needs to relate information from multiple variables, dynamic rule operations were not sufficient. This problem lead us to the introduction of dependent dynamic rules.

Dependent Dynamic Rules

Dependent dynamic rules allow to define all the required information for the implementation of data-flow optimisations specially when the information is spread over different points of an abstract syntax tree. Finding the information to express such an optimisation is easy at the point where relations between program entities (e.g., variables) can be established. At rule application point we require to ensure that all the information specified in the rule is not modified. Dependent dynamic rules allow to specify a transformation and the list of the dependencies, i.e., all the information required for the rule to be applicable. When a dependency changes its state all rules containing such dependency can be found and the rule can be discarded, thus preventing invalid transformations.

Data-flow Propagation Strategies

Traversals are programmable strategies in Stratego and are first class citizens. A tailored traversal can be reused for several transformations requiring to be specified only once. In Chapter 9 we have defined a data-flow propagation framework which has been instantiated for different program optimisations. Once the data-flow framework has been designed, we could easily focus on specifying the enabling conditions for a transformation to take place, thus simplifying their implementation considerably.

This abstraction not only reduced the work required for implementing data-flow optimisations but allowed us to define optimisations that could coexist by sharing a single traversal. Such property enabled us not only to combine data-flow analysis and program transformations but several transformations at once. These strategies are not tied to any object language and they can be reused with little effort for other imperative programming languages.

Partial Evaluation

Data-flow optimisations were extended to implement partial evaluation. The added abstractions allow for an elegant and concise implementation of partial evaluation.

Different strategies are implemented to enhance and control the online partial evaluation of Tiger and Octave programs. The implementation of partial evaluation and function specialisation in particular are the core of the Octave compiler. The *octavec* compiler is an exercise of compilation by transformation applied to a real world language. Poly-variant specialisation of functions is cleanly implemented with the capability of having multiple definitions for left-hand sides of dynamic rules.

12.1 Reusing Abstractions

In this section we provide evidence that our objective of developing language support for the implementation of program transformation as language independent and supplying reusable building blocks for the implementation of optimisations in other languages is achieved.

Most of the transformations and the data-flow framework presented and initially targeted to Tiger have inspired and contributed to implement some of the optimisations for languages such as Octave, Stratego, the Koala compiler and AutoBayes.

Stratego Compiler

Data-flow analysis with dynamic rules are extensively used in many parts of the Stratego compiler itself. Bounds/unbounds analysis determines whether or not a variable is bound at some program point. This analysis is used in static checks and in optimisations. The Stratego inliner uses dynamic rules. Innermost fusion is a domain-specific optimisation that generates efficient code for applications of the generic strategy `innermost` [49].

The dynamic rule lifter uses dynamic rules to keep track of context variables. The C generation back-end uses dynamic rules for memoization of common terms.

Koala Compiler

Koala is a Component Model for Consumer Electronics Software Products [97]. It is developed at Philips Research. The goal of Koala is to manage the complexity of embedded software by promoting software reuse. Koala is a component definition language used to model components as C modules. There is an Open

Software version of the Koala compiler available.[1] This open version is developed with Stratego [32]. The Stratego version has successfully integrated some of the transformations developed in this thesis. Such optimisations are variable renaming, constant propagation and function inlining.

AutoBayes

Autobayes is an automatic program synthesis system for data analysis problems [40, 39]. Its input is a declarative problem description in a form of statistical model. It produces well documented code. The code derivation process uses a schema-based approach (i.e., templates) and a set of constraints. Due to this type of program derivation, the generated code has different features if compared to code written by humans. Xbayes is an optimising separate component for the Autobayes Program Synthesis System [61]. Xbayes is implemented in Stratego. This optimiser component puts emphasis on the optimisation of loops. The following optimisations are implemented in Xbayes: loop fusion, sum simplification, invariant code elimination, common subexpression with alpha equality and constant propagation.

Scoped dynamic rewrite rules target a wide range of applications involving context-sensitive program transformations. The abstractions introduced by dynamic rewrite rules are very general, i.e., they are not specific to a certain kind of programming language or program transformation, as witnessed by the successful application of dynamic rewrite rules in a wide variety of non-trivial program transformations.

12.2 Future research

There are many ways to extend the generic strategies. On the one hand, the framework can be extended to cope with unstructured control-flow constructs such as `break`, `continue` and `throw-catch` (exception handling constructs), in a way that does complete the presented strategies. Research in this direction has been conducted in [35]. To model unstructured control-flow the operations `break-`R, `continue-`R and `throw-`R over dynamic rules have been incorporated to Stratego. These operations allow to implement data-flow information in the forward direction and currently are part of Stratego. More study is required to accommodate these constructs for backward flow propagation.

On the other hand, further research can be done for combining program transformation and alias analysis. The operations on dynamic rules at confluence points have potential to model aliasing and the values associated to them. Having the possibility to use union and/or intersection at confluence points can serve for that purpose.

Other possible transformations that could be part of the presented work is vectorisation to complete a catalogue of program optimisations.

[1] XT/Koala Compiler `http://www.program-transformation.org/Tools/KoalaCompiler`.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures : A dependence approach*. Morgan Kaufmann Publishers, 2002.

[3] G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language desing and implementation*, pages 294–303, Berlin, Germany, July 2002. ACM Press.

[4] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 33–50, London, UK, 1995. Springer-Verlag.

[5] A. W. Appel. *Modern compiler implementation in ML*. Cambridge University Press, 1998.

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

[7] L. Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Technische Universiteit Eindhoven, October 1993.

[8] P. Azero and S. D. Swierstra. Optimal pretty-printing combinators(software). In *Softwareprodukt*, 1998.

[9] F. Baader and T. Nipkow. *Term rewriting and All That*. Cambridge University Press, 1998.

[10] A. Baars, D. Swiestra, and A. Löh. UUAG user manual, September 2003.

[11] R. S. Bird. Notes on recursion elimination. *Commun. ACM*, 20(6):434–439, 1977.

[12] P. Borovanský, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *First Int. Workshop on Rewriting Logic*, volume 5 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1996.

[13] J. M. Boyle, T. J. Harmer, and V. L. Winter. The TAMPR program transformation system: simplifying the development of numerical software. *Modern software tools for scientific computing*, pages 353–372, 1997.

[14] 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.

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

[16] M. Bravenboer, N. Janssen, and E. Visser. XTC: Transformation tool composition, 2004. Draft.

[17] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 2008. Special issue on Experimental Systems and Tools.

[18] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. Technical Report UU-CS-2005-005, Department of Information and Computing Sciences, Utrecht University, June 2005.

[19] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2006.

[20] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software, Practice and Experience*, 28(8):859–881, 1998.

[21] A. Chauhan, C. McCosh, and K. Kennedy. Type-based speculative specialization for a telescoping compiler for MATLAB. Technical Report TR03-411, Department of Computer Science, Rice University, January 2003.

[22] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[23] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. *In Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[24] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[25] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Commun. ACM*, 20(11):850–856, 1977.

[26] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, January 1998.

[27] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501, New York, NY, USA, 1993. ACM Press.

[28] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Programming Languages and Systems*, 23(5):603–625, 2001.

[29] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[30] K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[31] M. de Jonge. A pretty-printer for every occasion. In I. Ferguson, J. Gray, and L. Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000).* University of Wollongong, Australia, 2000.

[32] M. de Jonge. Multi-level component composition. In J. Bosch, editor, *2nd Groningen Workshop on Software Variability Modeling (SVM'04)*, 2007 -7-01. Reseach Institute of Computer Science and Mathematics, University of Groningen, Dec. 2004.

[33] O. de Moor, B. Dickens, P. Kwiatkowski, and W. Aitken. Transformation in intentional programming. In *Proc. 5th Int'l. Conf. on Software Reuse*, pages 114–123, 1998.

[34] L. A. de Rose. Compiler techniques for MATLAB programs, 1996. PhD Thesis, University of Illinois at Urbana-Champaign.

[35] B. Dumitriu. Extensible and customizable data-flow transformation strategies for object-oriented programs. Master's thesis, Utrecht University, 2006.

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

[37] T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Lecture Notes in Computer Science. ECCOP 2004 - Object Oriented Program-*

ming, volume 3086, pages 147–171. Springer Berlin/ Heidelberg, november 2004.

[38] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[39] B. Fischer and J. Schumann. Autobayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, 2003.

[40] B. Fischer, J. Schumann, and T. Pressburger. Generating data analysis programs from statistical models. In *Workshop on Semantics, Applications, and Implementation of Program Generation*, Lecture Notes in Computer Science, pages 212–229, London, UK, 2000. Springer-Verlag.

[41] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wedsley, 1999.

[42] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[43] C. Green. The design of the psi program synthesis system. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 4–18, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[44] D. Grune, H. E. Bal, C. J. Jacobs, and K. G. Langendoen. *Modern Compiler Design.* Wiley, 2000.

[45] M. Harman, L. Hu, R. Hierons, M. Munro, X. Zhang, J. J. Dolado, M. C. Otero, and J. Wegener. A post-placement side-effect removal algorithm. *Fundamenta Informaticae*, pages 2–11, 2002.

[46] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland Inc., 1977.

[47] N. Janssen. Transformation tool composition. Master's thesis, Center for Software Technology, Utrecht University, Utrecht, March 2005.

[48] D. S. João Saraiva. Generic attribute grammars. In D. Parigot and M. Mernik, editors, *WAGA'99 Second Workshop on Attribute Grammars and their Applications. INRIA rocquencourt*, pages 185–204, march 1999.

[49] P. Johann and E. Visser. Strategies for fusing logic and control via local, application-specific transformations. Technical Report UU-CS-2003-050, Institute of Information and Computing Sciences, Utrecht University, February 2003.

[50] S. C. Johnson. Yacc: Yet another compiler-compiler. Computing Science Technical Report 32, Bell Laboratories, 1975.

[51] P. G. Joisha. MATLAB2C: A MATLAB-2-C to translator. `http://www.ece.northwestern.edu/cpdc/pjoisha/MAT2C`.

[52] P. G. Joisha, U. N. Shenoy, and P. Banerjee. Computing array shapes in MATLAB. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Lecture Notes in Computer Science, pages 3–11. Springer Berlin / Heidelberg, August 2003.

[53] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentince Hall International, 1993.

[54] L. Kats, A. Sloane, and E. Visser. Decorated attribute grammars: Attribute evaluation meets strategic programming. Technical Report TUD-SERG-2008-038, Software Engineering Research Group, Delft University of Technology., 2008.

[55] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley Professional, 2004.

[56] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[57] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM Press.

[58] S. Kleene. Introduction to metamathematics, 1952.

[59] J. Knoop, O. Ruething, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27,7, pages 224–234, San Francisco, CA, June 1992.

[60] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[61] J. Kruger. Specification of loop optimizations in Stratego. improving the AutoBayes system. Master's thesis, Utrecht University, Utrecht, The Netherlands, November 2003. INF/SRC-04-53.

[62] D. Lacey. *Program Transformation using temporal logic specifications*. PhD thesis, University of Oxford, August 2003.

[63] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In *POPL 2002: 29th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 270–282, Portland, Oregon, January 2002.

[64] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for data ow analyses and transformations via local rules. Technical Report UW-CSE-2004-07-04, University of Washington, July 2004.

[65] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for data ow analyses and transformations via local rules. In *POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGPLAN Symposium on Principles of Programming Languages*, pages 364 – 377, Long Beach, California, January 2005. ACM.

[66] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* O'Reilly & Associates, 1992.

[67] B. Luttik and E. Visser. Specification of rewriting strategies. Technical Report P9710, Programming Research Group, University of Amsterdam, June 1997. Appeared in A. Sellink (editor) *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97)*, EWIC, Springer-Verlag, 1997.

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

[69] F. Martin. *Generation of Program Analyzers.* PhD thesis, Universität des Saarlandes, 1999.

[70] MATLAB: The language of technical computing. `http://www.mathworks.com`.

[71] D. Michie. Memo functions and machine learning, 1958.

[72] S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 1997.

[73] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[74] F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed λ-calculus. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–106, New York, NY, USA, 1988. ACM Press.

[75] K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies (WRS'02)*, volume 70 - 6 of *Electronic Notes in Theoretical Computer Science*, pages 156–175, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.

[76] K. Olmos and E. Visser. Turning dynamic typing into static typing by program specialization. In D. Binkley and P. Tonella, editors, *Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*,

pages 141–150, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

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

[78] V. K. Paleri. *An enviroment for automatic generation of code optimizers.* PhD thesis, Indian Institute of Science Bangalore, 1999.

[79] Program-transformation.org.

[80] A. Rodriguez. Attribute grammar extensions: Higher order attribute grammars and views. Master's thesis, Center for Software Technology, Utrecht University, Utrecht, August 2004.

[81] M. A. Serrano, C. M. de Oca, and D. L. Carver. Evolutionary migration of legacy systems to an object-based distributed environment. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 86–95. IEEE Computer Society Press, 1999.

[82] M. B. Siff. *Techniques for software renovation.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madiso, August 1998.

[83] C. Simonyi. The dead of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Reseach, September 1995.

[84] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *POPL 2004: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38. ACM, January 2004.

[85] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory based approach.* Addison Wesley, 1995.

[86] Y. Smaragdakis, S. S. Huang, and D. Zook. Program generators and the tools to make them. In *ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 92–100, Verona, Italy, August 2004. ACM.

[87] D. R. Smith and C. Green. Toward practical applications of software synthesis. In *FMSP'96,The First Workshop on Formal Methods in Software Practice*, pages 31–39, January 1996.

[88] Stratego. `http://www.stratego-language.org`.

[89] Terese. *Term Rewriting Systems.* Cambridge University Press, March 2003.

[90] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2001.

[91] E. Tilevich and Y. Smaragdakis. Binary refactoring: Improving code behind the scenes. In *International Conference on Software Engineering (ICSE)*, pages 264–273, May 2005.

[92] S. R. Tilley and D. B. Smith. Towards a framework for program understanding. In *4th International Workshop on Program Comprehension (WPC '96)*, pages 19–28. IEEE Computer Society Press, March 1996.

[93] S. Tjiang. *Automatic Generation of Data-flow Analyzers: A tool for building optimizers*. Ph.D. Thesis, Stanford University, july 1993.

[94] S. Tjiang and J. Hennessy. Sharlit: A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 82–93, 1992.

[95] A. van Dam. Extending dynamic rules. An application-oriented study into Stratego's new dynamic rules. Master's thesis, Utrecht University, Utrecht, The Netherlands, February 2004. INF/SCR-04-25.

[96] M. van den Brand, A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[97] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[98] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

[99] E. Visser. A bootstrapped compiler for strategies (extended abstract). In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 73–83, Trento, Italy, July 5 1999.

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

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

[102] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*, pages 375–396. Elsevier Science Publishers, September 2001.

[103] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

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

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

[106] M. Voelter. AspectJ-Oriented programming in Java. issue of the Java Report, January 2000.

[107] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 131–145, New York, NY, USA, 1989. ACM.

[108] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.

[109] M. P. Ward. Assembler to C migration using the fermaT transformation system. In *IEEE International Conference on Software Maintenance*, pages 67–76, 1999.

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

[111] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.

[112] N. Wirth. Program development by stepwise refinement. In *Communications of the ACM*, volume 14 - 4, pages 221–227, 1971.

[113] G. Wroblewski. *General Method of Program Code Obfuscation.* PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[114] AspectJ. `http://eclipse.org/aspectj`.

[115] E. V. Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 128–142, London, UK, 2002. Springer-Verlag.

[116] E. V. Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional programming: a host of language features. Technical Report PRG-RR-01-21, Computing Laboratory, University of Oxford, 2001.

A
Stratego Library

This appendix lists Stratego strategies used in the specification of program transformations. It presents its definitions:

```
add  = Stratego primitive for addition
addS = apply-int-to-str-tuple(add)

apply-int-to-str-tuple(s) =
  (string-to-int, string-to-int) ; s ; int-to-string

bottomup  = all(bottopmup(s)); s
conc = \ (l1, l2) -> <at-end(!l2)> l1 \  <+ \ "" # (xs) -> <concat> xs \
filter(s) = [] <+ [s | filter(s) ] <+ Tl; filter(s)
foldr(s1, s2, f)  =
   []; s1 +
    \ [y|ys] -> <s2> (<f> y, <foldr(s1, s2, f)> ys) \

int-to-string  = Stratego primitive for converting an integer into a string

map(s)    = [] <+ [s | map(s) ]
mapconcat(s) = foldr([], conc, s)
mul       = Stratego primitive for multiplication
mulS      = apply-int-to-str-tuple(mul)
repeat    = rec x(s; x <+ id)

partition(s) = partition(s, id)
partition(s1, s2) = rec part(
  \ [] -> ([],[]) \
  <+ ({[s1 => x | id]; ?[_|<part>]; !([x | <Fst>], <Snd>)} <+
     {[s2 => x | id]; ?[_|<part>]; !(<Fst>, [x | <Snd>])})

reverse-filter(s) = [] + [id | reverse-filter(s)]; ([s | id] <+ ?[_  | <id>])
```

```
reverse-map(s) = [id | reverse-map(s)]; [s | id] <+ []
string-to-int = Stratego primitive for converting string to an integer
sub       = Stratego primitive for subtraction
subS      = apply-int-to-str-tuple(add)
topdown   = s; all(topdown(s))
try       = s <+ id

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

B
Syntax Definition for Tiger

B.1 The Tiger Syntax

```
module Tiger-Condensed
exports
  sorts Exp
  context-free start-symbols Exp
  context-free syntax
    Id                                 -> Var {cons("Var")}
    StrConst                           -> Exp {cons("String")}
    Var                                -> LValue
    LValue "." Id                      -> LValue {cons("FieldVar")}
    LValue "[" {Exp ","}+ "]"          -> LValue {cons("Subscript")}
    TypeId "[" {Exp ","}+ "]" "of" Exp    -> Exp {cons("Array")}
    TypeId "{" {InitField ","}* "}"    -> Exp {cons("Record")}
    "(" {Exp ";"}* ")"                 -> Exp {cons("Seq")}
    Var "(" {Exp ","}* ")"             -> Exp {cons("Call")}
    Exp "+" Exp                        -> Exp {left,cons("Plus")}
    Exp "-" Exp                        -> Exp {left,cons("Minus")}
    LValue ":=" Exp                    -> Exp {cons("Assign")}
    Var "(" {Exp ","}* ")"             -> Exp {cons("Call")}
    "if" Exp "then" Exp "else" Exp     -> Exp {cons("If")}
    "if" Exp "then" Exp                -> Exp {cons("IfThen")}
    "while" Exp "do" Exp               -> Exp {cons("While")}
    "for" Var ":=" Exp "to" Exp "do" Exp  -> Exp {cons("For")}
    "break"                            -> Exp {cons("Break")}
    "nil"                              -> Exp {cons("NilExp")}
    "let" Dec* "in" {Exp ";"}* "end"   -> Exp {cons("Let")}

    Id "=" Exp                         -> InitField {cons("InitField")}

    "var" Id TypeAn ":=" Exp           -> Dec {cons("VarDec")}
    FunDec+                                -> Dec {cons("FunDecs")}
    "type" Id "=" Type                     -> TypeDec {cons("TypeDec")}
    "function" Id "(" {FArg ","}* ")"
                         TypeAn "=" Exp  -> FunDec {cons("FunDec")}
    Id TypeAn                              -> FArg {cons("FArg")}
                                           -> TypeAn {cons("NoTp")}
    ":" TypeId                             -> TypeAn {cons("Tp")}
    Id                                     -> TypeId {cons("Tid")}
    "array" "of" TypeId                    -> Type   {cons("ArrayTy")}
    "{" {Field ","}* "}"                   -> Type   {cons("RecordTy")}
    Id ":" TypeId                          -> Field  {cons("Field")}
```

B.2 List of Tiger MetaVariables

```
module Tiger-Variables
exports
  variables
    [s][0-9\']*           -> StrConst {prefer}
    [ijkn][0-9\']*        -> IntConst {prefer}
    [xyzfgh][0-9\']*      -> Id {prefer}
    [e][0-9\’]*           -> Exp {prefer}
    "e"[0-9\’]* "*"       -> {Exp ";"}+ {prefer}
    "a"[0-9\’]* "*"       -> {Exp ","}+ {prefer}
    "lv"[0-9\’]*          -> LValue {prefer}
    "lv"[0-9\’]* "*"      -> {LValue ","}+{prefer}
    "x"[0-9\’]* "*"       -> {FArg ","}+ {prefer}
    "bo"[0-9\’]*          -> BinOp {prefer}
    "ro"[0-9\’]*          -> RelOp {prefer}
    [d][0-9\’]*           -> Dec {prefer}
    [d][0-9\’]* "*"       -> Dec+  {prefer}
    "fd"[0-9\’]*          -> FunDec {prefer}
    "fd"[0-9\’]* "*"      -> FunDec+{prefer}
    "t"[0-9\’]*           -> Type {prefer}
    "ta"[0-9\’]*          -> TypeAn {prefer}
    "tid"[0-9\’]*         -> TypeId {prefer}
    "tf"[0-9\’]*          -> Field {prefer}
    "tf"[0-9\’]* "*"      -> {Field ","}+ {prefer}
    "f"[0-9\’]* "*"       -> {InitField ","}+ {prefer}
    "ty"[0-9\’]*          -> Ty {prefer}
    "F"[0-9\']* "*"       -> {FIELD ","}+ {prefer}
    "F"[0-9\']*           -> FIELD {prefer}

  lexical syntax
    [s][0-9\’]*           -> Id {reject}
    [ijk][0-9\’]*         -> Id {reject}
    [e][0-9\’]*           -> Id {reject}
    "es"[0-9\’]*          -> Id {reject}
    "as"[0-9\’]*          -> Id {reject}
    "lv"[0-9\’]*          -> Id {reject}
    "lvs"[0-9\’]*         -> Id {reject}
    "xs"[0-9\’]*          -> Id {reject}
    "bo"[0-9\’]*          -> Id {reject}
    "ro"[0-9\’]*          -> Id {reject}
    [d][0-9\’]*           -> Id {reject}
    "ds"[0-9\’]*          -> Id {reject}
    "t"[0-9\’]*           -> Id {reject}
```

```
"ta"[0-9\']*          -> Id {reject}
"tid"[0-9\']*         -> Id {reject}
"fs"[0-9\']*          -> Id {reject}
"ty"[0-9\']*          -> Id {reject}
```

B.2.1 Rewrite Rules and Concrete Syntax

Concrete syntax is a new capability of Stratego that allows to specify rewrite rules using concrete syntax of the object language [104]. Stratego programs can deal with specifications written in several languages; the Stratego languages constructs and the language constructs of the embedded object language(s). Concrete syntax is a neat abstraction that offers the following advantages:

1. A clean separation between the Stratego code and the code of the embedded object embedded language.
2. The embedded code is verified to be correct with respect of parsing tools used to recognise valid constructs of the embedded language.
3. It allows to write code that is concise, readable and elegant.

Thus, concrete syntax allows us to write Tiger[1] code fragments using its own concrete syntax. In order to have a clear distinction between the meta and the object languages, Tiger code fragments will be denoted between delimiter symbols.

Quotation and Antiquotation Marks. Symbols that delimit the scope of the hosted language(s) are referred between quotation marks. For Tiger these symbols are: $[\![$ and $]\!]$.

Antiquotation is a symbol or group of symbols to denote terms that are not part of the concrete syntax of the language and can be escaped and accepted by using the antiquotation symbols. For Tiger the symbol ~ precedes the quotated term.[2] In the rest of this dissertation these symbols will remain as delimiters to differentiate the object from the meta language. As examples of rewrite rules with concrete syntax consider the rewrite rules:

```
AddZero : [[ x +  0 ]] -> [[ x ]]
EvalPlus: [[ i +  j ]] -> [[ k ]] where <addS>(i ,j) => k
```

Congruence Abstractions In concrete syntax, congruence is a way to express the application of strategies to the subterms of a term. For each defined constructor, a congruence strategy is available that applies strategies to the subterms of the constructor. As an instance consider the term `If(`t_1`,`t_2`,`t_3`)` and its congruence strategy `If(`s_1`,`s_2`,`s_3`)` that is available and can be instantiated with the proper specified strategies.

[1] Tiger is the programming language used in this dissertation as the object language.
[2] Quotation and Antiquotation symbols are not fixed and can be redefined according to the developer decisions.

[*x* , *y* , *z*][0 – 9’]*	->	`Id`	Tiger variable
[*f* , *g*][0 – 9’]*	->	`Id`	Tiger function name
[*i* , *j* , *k*][0 – 9’]*	->	`IntConst`	Tiger integer
[*e*][0 – 9’]	->	`Exp`	Tiger expression
[*e*][0 – 9’]*	->	{`Exp`”;”}*	list of Tiger expressions
[*d*][0 – 9’]	->	`Dec`	Tiger declaration
[*d*][0 – 9’]*	->	`Dec*`	list of Tiger declarations
[*x*][0 – 9’]*	->	{`FArg`”,”}*	list of Tiger formal parameters
[*a*][0 – 9’]*	->	{`Exp`”,”}*	list of Tiger function arguments

< *s* >	->	`Exp`	congruence for expressions
<**s*>	->	`Exp+`	congruence for list of expressions
`<typedecs:` *s* >	->	`TypeDec+`	congruence for type declarations
`<fd:` *s* >	->	`FunDec`	congruence for function definition
`<fd*:` *s* >	->	`FunDec+`	congruence for list of function definitions

Figure B.1: (Top) Meta Variables and (Bottom) Congruence abstractions for Tiger constructions where **s** *stands for the application strategy.*

Congruence it is also available at concrete syntax level. It allows to apply strategies to subterms of the term. To denote that a strategy will be applied to a subterm, quotation marks are introduced. The delimiter symbols `<` and `>` will describe a strategy application to the subterm of a term. Several quotation symbols can be defined to describe different sort terms. As an example the construction ⟦ `if <s1> then <s2> else <s3>` ⟧ applies the strategies s_1 , s_2 and s_3 to a term `If(`t_1`,`t_2`,`t_3`)`, and it is equivalent to the congruence strategy using abstract syntax trees `If(s1,s2,s3)`. This translation is done automaticlly by the Stratego compiler, i.e, *transparent* to the user. The congruence strategy `<*`*s*`>` denotes the application of the strategy *s* to a list of subterms. Figure B.1 (bottom) shows more congruence abstractions for expressions, type definitions, function definitions, etc.

Definition of Meta-Variables In the following we show a list of meta-variables for Tiger and its equivalent abstract syntax construction. This list is not complete and it can be extended providing means to the developer to adapt the syntax to its needs. The list shows the most used meta-variables for the better understanding of this dissertation. The code that specifies these constructions is provided in Figure B.1.

B.2.2 Concrete Syntax Object Mechanics

For the realisation of concrete syntax objects, Stratego interacts with SDF. The syntax definition formalism (SDF) can be employed to fit any meta-programming language with concrete syntax notation for composing and analysing object programs [104]. The concrete syntax of a programming language is defined in the syntax definition formalism in a declarative fashion. This formalism combines the lexical definitions and the context-free syntax in a modular way. Thus, provides means to combine separated languages.

Stratego allows to embed an object language *O* by extending its syntax with the syntax of the *O* language. Expressions in the *O* language must be interpreted as data constructions and analysis patterns. To distinguish both languages, antiquotation operators are introduced to Stratego (*meta-language*) which allows the developer to indicate which are the object languages fragments. Furthermore, code fragments represented in the object language contain Stratego meta-variables. As an instance consider the term ⟦ `function` *f*`(`*x**`) =` *e* ⟧ where the identifiers *f*, *x** and *e* are not intented to match with variables of the object language, instead they constitute meta-variables. This entails the definition of the syntax of the object language *O* plus the definition of the antiquotation operators and the defintion meta-variables used in fragments of the object code. All these language elements are defined using the SDF formalism.

```
module Tiger-Congruences
exports
  context-free syntax

 "|[" Dec       "]|"             -> StrategoStrategy cons("ToStrategy"),prefer
 "|[" TypeDec   "]|"             -> StrategoStrategy cons("ToStrategy"),prefer
 "|[" FunDec    "]|"             -> StrategoStrategy cons("ToStrategy"),prefer
 "|[" Exp       "]|"             -> StrategoStrategy cons("ToStrategy"),prefer
 "Exp" "|[" Exp "]|"             -> StrategoStrategy cons("ToStrategy"),prefer
 "LValue" "|[" LValue "]|"       -> StrategoStrategy cons("ToStrategy"),prefer
 "IntConst" "|[" IntConst "]|"   -> StrategoStrategy cons("ToStrategy"),prefer
 "FArg" "|[" FArg "]|"           -> StrategoStrategy cons("ToStrategy"),prefer
 "InitField" "|[" InitField "]|" -> StrategoStrategy cons("ToStrategy"),prefer
 "Field" "|[" Field "]|"         -> StrategoStrategy cons("ToStrategy"),prefer
 "|[" Ty "]|"                    -> StrategoStrategy cons("ToStrategy"),prefer

  context-free syntax

    "<"          StrategoStrategy ">" -> Exp         cons("FromApp"), prefer
    "<"          StrategoStrategy ">" -> LValue      cons("FromApp"), prefer
    "<"          StrategoStrategy ">" -> Dec         cons("FromApp")
    "<*"         StrategoStrategy ">" -> Dec*        cons("FromApp")
    "<bo:"       StrategoStrategy ">" -> BinOp       cons("FromApp")
    "<ro:"       StrategoStrategy ">" -> RelOp       cons("FromApp")
    "<*"         StrategoStrategy ">" -> Exp ","+  cons("FromApp")
    "<*"         StrategoStrategy ">" -> Exp ";"+  cons("FromApp")
    "<*"         StrategoStrategy ">" -> FArg ","+ cons("FromApp")
    "<"          StrategoStrategy ">" -> TypeAn      cons("FromApp"),prefer
    "<"          StrategoStrategy ">" -> TypeId      cons("FromApp"),prefer
    "<if:"       StrategoStrategy ">" -> InitField   cons("FromApp")
    "<"          StrategoStrategy ">" -> Id          cons("FromApp"),avoid
    "<id:"       StrategoStrategy ">" -> Id          cons("FromApp")
    "<lv:"       StrategoStrategy ">" -> LValue      cons("FromApp")
    "<"          StrategoStrategy ">" -> Id          cons("FromApp"), avoid
    "<int:"      StrategoStrategy ">" -> IntConst    cons("FromApp")
    "<real:"     StrategoStrategy ">" -> RealConst   cons("FromApp")
    "<str:"      StrategoStrategy ">" -> StrConst    cons("FromApp")
    "<typedecs:" StrategoStrategy ">" -> TypeDec+    cons("FromApp")
```

```
"<fundec:"   StrategoStrategy ">" -> FunDec      cons("FromApp")
"<fundecs:"  StrategoStrategy ">" -> FunDec+     cons("FromApp")
"<fd:"       StrategoStrategy ">" -> FunDec      cons("FromApp")
"<fd*:"      StrategoStrategy ">" -> FunDec+     cons("FromApp")
```

Samenvatting

Een programmatransformatiesysteem is een computer programma waarvan het primaire doel het transformeren van programma's is. Er zijn verscheidene redenen om een programma te genereren uit een ander programma. Bijvoorbeeld om een geoptimaliseerde versie te verkrijgen, of om de programmacode te verduidelijken ten behoeve van de onderhoudbaarheid. De specificatie van een programmatransformatiesysteem vereist de specificatie *hoe* en *wanneer* te transformeren. De specificatie van hoe te transformeren beschrijft de modificaties van het programma. Zulke programmamodificaties kunnen elegant worden beschreven met termherschrijving. Termherschrijving is een computationeel (reken-)model gebaseerd op regels. Een herschrijvingsregel representeert een enkele stap in een modificatie van een programma. De specificatie wanneer te transformeren heeft betrekking op de identificatie van alle condities die een bepaalde transformatie toestaan. Om alle informatie te vinden die een optimalisatie mogelijk maakt, heeft een programmatransformatiesysteem de hulp nodig van programma-analysetechnieken. Hoe en wanneer te transformeren zijn respectievelijk bekend als programmatransformatie en programma-analyse. Normaliter worden deze twee taken afzonderlijk geïmplementeerd.

Voor de implementatie van programmatransformaties gebruiken we Stratego, een domeinspecifieke programmeertaal voor de specificatie van op strategische herschrijving gebaseerde programmatransformatiesystemen. Stratego onderscheidt zich van zuivere herschrijvingssystemen door gebruik te maken van een strategie (een Stratego programma) om het herschrijvingsproces te sturen. In dit proefschrift concentreren we ons op de implementatie van data-flow optimalisaties. Een data-flow optimalisatie moet informatie vergaren die geldig is voor alle mogelijke executie paden van een programma. Deze informatie wordt volledig benut om optimalisaties aan een programma uit te voeren.

Simpele herschrijvingssystemen die slechts gebaseerd zijn op herschrijvingsregels, kunnen alleen gebruik maken van informatie die beschikbaar is in de term die getransformeerd wordt. Voor implementaties van data-flow optimalisaties die informatie moeten vergaren die verspreid is over het gehele programma is dit echter niet voldoende en zij hebben dus context gevoelige informatie nodig. Stratego is uitgebreid met *dynamische herschrijvingsregels* om deze context gevoelige informatie te verschaffen. Deze dynamische herschrijvingsregels worden tijdens het herschrijvingsproces gegenereerd en hebben toegang tot informatie die beschikbaar is in de context waarin zij zijn gedefinieerd. Door het toepassen van een dynamische regel komt informatie van een andere locatie beschikbaar.

Data-flow optimalisaties kunnen gevonden worden in optimaliserende compilers. Meestal werken implementaties van deze data-flow transformaties op vaste, tussentijdse programma representaties op een laag niveau. Andere toepassingen van programmatransformatiesystemen ("refactoring tools", "code weavers", "software renovation tools", etc.) daarentegen zijn geïmplementeerd op het broncode niveau. Om een raamwerk voor data-flow transformaties op broncode niveau wijd bruikbaar te maken moet het beschikken over een groot aantal mogelijkheden tot transformaties. Dat wil zeggen dat het bijvoorbeeld niet specifiek moet zijn voor een enkele bron taal en niet beperkt moet zijn tot het aanbieden van een enkel type transformatie. Het moet voorzien in hoog niveau abstracties om de control- en data-flow van de betreffende taal te modelleren, en het moet data-flow transformaties kunnen combineren met andere programmamanipulatietechnieken zoals code generatie gebaseerd op templates. Verder moet het raamwerk kunnen abstraheren van details van programma representaties en moet bijvoorbeeld kunnen omgaan met kwesties zoals de scope van variabelen en het voorkomen van problemen zoals het vangen van vrije variabelen. Alle voornoemde eigenschappen van een transformatie raamwerk kunnen leiden tot een betere productiviteit van een transformatieontwikkelaar.

Het belangrijkste doel van het onderzoek uitgevoerd in dit proefschrift was om taalondersteuning te ontwikkelen die de ontwikkeling van programmatransformaties faciliteert. Herschrijvingsregels en strategiën om transformaties te beschrijven zijn uitgebreid om het hoofd te bieden aan programma transformaties die context gevoelige informatie behoeven. Dit werk draagt bij aan de schaarse verzameling van hulpmiddelen die de implementatie van compilers of programmatransformatiesystemen vereenvoudigen [50, 42].

Meta-taal ondersteuning werd vanaf de aanvang van dit onderzoek gezien als een mogelijkheid bij te dragen aan de productiviteit van de transformatieontwikkelaar, om objecttaal onafhankelijk te zijn en om de implementatie van analyse en transformatie gereedschappen mogelijk te maken op een hoog niveau van abstractie, terwijl tegelijkertijd meerdere transformaties gecombineerd kunnen worden. Verder beschrijven we het gekozen pad om deze doelen te bereiken.

Dynamische regels

Dynamische herschrijvingsregels spelen een belangrijke rol in Stratego voor de ontwikkeling van transformaties die context gevoelige informatie behoeven. Zulke regels abstraheren van complexe datastructuren en geven toegang tot informatie die is gedefinieerd buiten het werkingsgebied van statische herschrijvingsregels. Gegeven dit perspectief representeren dynamische herschrijvingsregels een mechanisme om informatie impliciet te propageren. Dynamische regels verstrekken niet alleen de mogelijkheid om informatie te propageren, maar ook om modificaties uit te drukken, gebruik makend van deze informatie. Deze regels kunnen worden toegepast in elke gedefinieerde doorloop strategie, generiek of geprogrammeerd. Informatie die gepropageerd wordt door dynamische regels wordt bijgehouden door dynamische regel operaties om de huidige staat weer te geven.

Dynamische regel operaties

Aanvankelijk boden dynamische regels de volgende operaties aan: definitie, bijwerken en de-definitie (vernietiging). Voor de implementatie van data-flow optimalisaties hebben we operaties ontworpen die de control-flow van imperatieve talen adresseren. De dynamische operaties zijn op twee manieren uitgebreid: 1) De operaties zijn toepasbaar op een set van dynamische regels en 2) De toegevoegde operaties zijn vereniging, doorsnede en een combinatie van beiden. Bovendien kunnen de operaties toegepast worden op een dekpunt.

De ontworpen operatoren breiden het vermogen van Stratego om dynamische regels te beschrijven uit en stonden ons toe om een beknopte implementatie voor constant propagation te schrijven. Voor optimalisaties zoals common sub-expression elimination daarentegen, waar informatie over verscheidene variabelen gerelateerd moet worden, zijn dynamische regels niet afdoende. Dit probleem heeft ons geleid naar de introductie van afhankelijke dynamische regels.

Afhankelijke dynamische regels

Afhankelijke dynamische regels bieden de mogelijkheid om alle voor de implementatie van data-flow optimalisaties benodigde informatie te definiëren, in het bijzonder wanneer de informatie is verspreid over verschillende punten in een abstracte syntax boom. Het vinden van de informatie om een optimalisatie uit te drukken is makkelijk op het punt waar relaties tussen programma-entiteiten (b.v. variabelen) kunnen worden vastgesteld. Wanneer de regels worden toegepast, moet het zeker zijn dat alle in de regel gespecificeerde informatie niet ongewijzigd is. Afhankelijke dynamische regels bieden de mogelijkheid om een transformatie te specificeren samen met een lijst van afhankelijkheden, dat wil zeggen alle benodigde informatie om de regel toepasbaar te maken. Wanneer een bepaalde afhankelijkheid van status verandert kunnen alle regels die hiervan afhankelijk zijn gevonden en genegeerd worden zodat ongeldige transformaties voorkomen worden.

Data-flow propagatie strategiën

Om een herschrijving toe te passen op een term is het noodzakelijk om deze term te doorlopen (traversal). Traversals zijn programmeerbare strategiën in Stratego en zijn manipuleerbare componenten ("first class citizens"). Een op maat gemaakte traversal kan hergebruikt worden voor verscheidende transformaties terwijl deze slechts één keer gespecificeerd hoeft te worden. In hoofdstuk 9 hebben we een data-flow propagatieraamwerk gedefinieerd die we voor verscheidene programmaoptimalisaties geïnstantieerd hebben. Nadat het data-flow raamwerk ontworpen was, was het makkelijker om te concentreren op het specificeren van de voorwaarden voor het toepassen van een transformatie, waardoor de implementatie van transformaties aanzienlijk vereenvoudigd is.

Deze abstractie heeft niet alleen het benodigde werk voor de implementatie van data-flow optimalisaties verminderd, maar stond ons ook toe om optimalisaties te definiëren die kunnen co-existeren door een enkele traversal te delen. Zo'n eigenschap geeft niet alleen de mogelijkheid om data-flow analyse en programmatransformaties te combineren, maar ook om verscheidene transformaties te combineren. Deze strategiën zijn niet gebonden aan een specifieke bron taal en kunnen met weinig moeite hergebruikt worden voor andere imperatieve programmeertalen.

Partiële evaluatie

Data-flow optimalisaties zijn uitgebreid om partiële evaluatie te implementeren. De toegevoegde abstracties staan een elegante en beknopte implementatie van partiële evaluatie toe.

Verschillende strategiën zijn geïmplementeerd om de online partiële evaluatie van Tiger en Octave programma's te sturen en te verbeteren. De implementatie van partiële evaluatie, en in het bijzonder functie specialisatie, vormt de kern van de Octave compiler. De *octavec* compiler is een exercitie in compilatie door transformaties, toegepast op een niet triviale taal. Poly-variante specialisaties van functies zijn helder geïmplementeerd met de mogelijkheid om meerdere definities te hebben voor de linkerkant van dynamische regels.

Acknowledgements

I would like to thank plenty of people who contributed in one way or another to the realisation of my thesis. I am sure of breaking tradition here; but I would like to start in a chronologic order. This is to give you an idea of my Ph.D. journey.

The person who inspired me the most and who was an enthusiastic lecturer is Hielko Ophoff. Hielko started his professional teaching career in my home city only two weeks after having arrived from The Netherlands. Hielko thanks for taught me the underlying concepts of programming languages and shown me the difference of a high level education. To Tanja Vos I want to thank her for offering me the opportunity to finish my studies in The Netherlands and thank her for having had trust in me.

To Eelco Visser goes a very special thank you. Eelco is a passionate researcher and lecturer who was always able to lift my spirit when I was less motivated. I could not have made it if I could not have had the motivation boost when I mostly needed. I learnt from you most of I know as a researcher. Doaitse, thanks for always having the time to seek for my best interest. I really appreciate your support. Furthermore, thanks for carefully reading my thesis and providing me with remarks and questions in the finishing track. Bernd Fisher, thank you for reading and providing suggestions on how to improve my thesis. Your optimistic research attitude is very refreshing.

My days at the Centrumbegouw Noord were fun and for that I must say thanks to Iris Reinbacher, Tomas Wolle, Peter Lenartz and Peter Verbaan. I also want to thank my ex-collegues Arthur Baars, Baastian Heeren, Rui Guerra and Jurriaan Hage with whom I shared office. I could not resist to include Jurriaan in the last sentence but it feels almost right. Thank you all for the nice discussions and PhD student time together. I also wish to thank the people with whom I had direct collaboration inside the Software Tools group Martin Bravenboer, Rob Vermaas, Arthur van Dam and Eelco Dolstra.

My thesis journey could have not been completed without the support and encouragement of friends. I want to thank Cecile Aurelle and Holger Walter for the friendship and nice German/French evenings were I could always feel happy in your company. Patricia Garcia and Leon Arkenstijn thanks for sharing Latin or better said South-american precious time with me and for always letting me know that I could count with you. Patricia thanks for designing the cover of my thesis. Me encanta. Inma thank you very much for sharing cultural and Spanish quality time with me.

To my "Bolivian connexion" (Alexey Rodriguez and Nicolas Avila) thank you for the friendship, fun, and support.

My thesis journey continued in Eindhoven. Many thanks goes to the TCS team: Bart, Frank, Harm, Joachim, Jos, Niels, Philippe, Ronald, Steven, Tom and Willem. Harm and Jos thank you for proofreading the preliminary version of my thesis (you did a good job at it) and for motivating me to complete this thesis. Niels thank you very much for lending me your laptop and translating into Dutch the summaries of this thesis. I hope you did not suffer any loss of precious electronic information :). To Rene K., Yanja, Razvan, Manuela and Anca, Teresa and Rene thank you for your friendship.

An added value of doing a PhD. in Utrecht was the great social network. The nights at JP and related activities will always bring me great memories. To Antti-Pekka, Eric, Linda, Ronan, Claudia, Pascal, Mireille, Marcela, Janine, David, Nick, Viktor, and Yvonne thank you guys. I am sure of leaving names behind to those people please accept my gratitude.

Finalmente deseo dar las gracias a mi familia. Papi gracias por haberme dado la inspiración y apoyo para emprender y hacer realidad mis sueños. A mi hermano Gerson gracias por el inmenso cariño y apoyo. Muchisimas gracias a los dos.

Hatse flatse!.

Karina Olmos
April 16, 2009
Eindhoven

Curriculum Vitae

Karina Olmos was born in Cochabamba, Bolivia on the 4th of October 1969. She attended the Aleman Santa Maria High School. She graduated from the Universidad Mayor de San Simón, Bolivia in 1999. In 2000 she did an internship at Philips Research Laboratories, Eindhoven, The Netherlands. Based on this internship she wrote her master thesis.

In fall of 2000 she joined as Ph.D. student the Software Tools Group in the Faculty of Wiskunde en Informatica at Utrecht University. In the summer of 2003 she got a summer job at NASA Ames Research Center in Mountain View, California, US. In 2005 she joined Philips Research in Eindhoven, The Netherlands. Since 2006 she is part of NXP Semiconductors in Eindhoven, The Netherlands.

Titles in the IPA Dissertation Series since 2005