Extensible and Customizable Data-flow Transformation Strategies for Object-Oriented Programs

Bogdan Dumitriu (0402044)
bdumitriu@bdumitriu.ro

Software Technology Group
Institute of Information and Computing Sciences
Utrecht University

Master Thesis

24th November 2006

Preface

I have always found writing text more difficult, less entertaining, more time consuming and definitely less rewarding than writing code. Hence, I would like to begin this thesis by expressing my joy that writing it is finally over. After many wasted days (doing anything else but writing my thesis) and even more wasted nights (trying to make up for the wasted days) I have eventually produced the last sentence of what has been a arduous task. But now I am a free man again!

The work for this thesis started out more than a year ago, when the main idea was "let's look into active libraries and see if there's anything good that can come out of there". Things have evolved quite a bit since then, even though not that much good came out of active libraries altogether. So, somewhere along the way, the topic of the thesis slowly shifted from active libraries towards whatever it has become in the end, which is a mixture of things, really. I've read a lot about active libraries, and then I've read some more about optimization of object-oriented programs, in order to finally end up dealing with topics quite distant from those. In fact, it was only towards the end of my work that my mind managed to finally wrap itself around this variety of topics and make me realize that they are all quite interrelated. Ultimately, if I fast-forward to today, I feel quite pleased with this course of events.

What was even better about this thesis, however, is that I got to know Stratego so much better. Stratego is a great platform and even after all this time I am still truly impressed with what so few people have managed to achieve. Confronted with accomplishments like this, I feel awe. Now that I've spent all this time getting familiar with it, I can only hope that my future activity will give me enough opportunities to come back to it over and over. The greatness of Stratego, as I would put it, lies in the fact that it helps you do so many things, so very elegantly. Development speed is unfortunately a different issue altogether, largely due to the lack of a type checker and of a debugger. I can only hope that one day such things will come to life.

Before ending this short foreword and inviting you to dive into my fascinating thesis, some really honest thank you's are in order. First of all, many many thanks to Eelco Visser for creating Stratego and leading its development for years on end, for being smart, friendly, open and helpful, for keeping things interesting, for providing valuable advice, for advising my master thesis and, most importantly, for overestimating my achievements. Right there with Eelco, I owe a big thanks to Martin Bravenboer for being awake and working 24/7, for always getting me (and all the other Stratego-ers) out of trouble, for knowing everything about everything and for his willingness to share, for giving advice and suggesting smart solutions and, why not, for keeping the standards high for me and, I dare believe, many

others. Thereafter, I'd like to thank all my colleagues from the ST lab for sharing coffee, jokes, comments, information and science and, more generally, for making my work so much more enjoyable. Last, but obviously not least, my thanks go to my parents for supporting me through this lengthy and many times difficult process of studying abroad and to my wonderful wife for being there for me all this time, both in IM and real-life mode.

Now please go on an do enjoy my thesis!

Bogdan Dumitriu
November, 2006

Contents

Chapter 1

Introduction

1.1 Motivation: towards open compilers

There are a number of aspects regarding the compilation process that have been changing steadily over the years. Compilers have evolved from compiling assembly code into machine code to compiling ever higher level languages to machine-independent code (such as Java bytecode or .NET CIL). The number of intermediate languages (and corresponding *compilation phases*) through which a program goes has increased as well, since most modern language compilers build on already existing ones. The common pattern is to write a compiler which produces source code of a lower level language and then piggyback on the compiler for that language (with C being the most widely used one). The number of such phases increases as languages drift further and further away from machine code in terms of abstractions.

While the above mentioned progress can be regarded as only partially compiler-specific (with the rest spanning also language and tool chain), there is another aspect which remains entirely within the bounds of compiler technology: *optimization*. This has been a constant preoccupation of compiler writers from the earliest of ages. In the beginning, most optimizations were geared towards limiting the amount of memory needed by a program, due to the scarcity of hardware resources. With time, the goals of optimization have widened to cover aspects like instruction locality, computing data dependencies for parallelization purposes, reducing the amount of branching in code, eliminating redundancy (e.g., loop-invariant code motion) and countless others. Nowadays a lot of work still goes into refining existing optimizations, identifying additional optimization opportunities and optimization algorithms, making more information about the code available to the optimization process, extending optimizations from basic peephole ones to whole-program ones, and so on. Considering the scope of their application (virtually all compiled software), it reasonable to think that advances in optimization technology are responsible, on average, for more software quality improvements than any other field of computer science.

Compilers have also evolved quite a lot where safety of the generated programs is concerned. This is achieved chiefly through advances in *type systems*. While type systems have uses beyond safety (such as supporting optimizations or acting as documentation to a program), it is still the case that we primarily use them for enhancing the safety of our programs. A sufficiently advanced type system will point out many kinds of errors in our programs at

compile time, which is a significant advantage according to the general observation that the earlier in the development process we identify bugs, the less costly it is to fix them. Type system have evolved along with compilers from a rudimentary int-float-bool-char state to one including enhanced notions like polymorphic types, generic types, dependent types or recursive types together with various combinations and specializations thereof. Such developments illustrate that the type systems that compilers support today are a long way from what we have begun with in the early days; and type system research is still receiving significant effort.

As summarized before, aspects like varying compilation phases, optimizations and type systems have all received proper attention both from the research world and from the industry. Significant evolutions in all these area stand as proof of this. But there is at least one path related to compiler technology that has been seriously less walked until now, namely that of *open compilers*. Open compilers are about user customization of the compilation process. To date, the interaction a user can have with the compiler is reduced to specifying a number of compiler flags which trigger or prevent the execution of specific compiler tasks. Examples of things that can be turned on or off by the user are various types of optimizations (including machine-dependent ones), the inclusion of debugging information and certain types of warnings. The user cannot, however, customize, extend or aid any of the compiler tasks, and the one we particularly have in mind is optimization.

In general, compiling a program to provably best machine code (either in terms or size or speed) is undecidable. In particular, even striving for a far less ambitious goal than that still creates enough difficulties. It is a fact that optimizers that come with today's compilers are far from being smart enough to materialize all the optimization potential that exists even at a (theoretically) easily available level. This is caused primarily by the *soundness* requirement (i.e., the compiler usually has to guarantee that the optimization is correct), which forces the optimizers to make conservative choices (about various things) whenever there is not enough information available. In simpler terms, optimizers are not allowed to guess, even if there is a 99.99% chance that the guess will be correct. This lack of information which optimizers are often confronted with can have several causes. It can be that the information is simply not there (e.g., compiled libraries for which the source code is not available), or that the optimizer is not "smart" enough to extract that information even if it is available, or that extracting the information, even if possible, would take more time than the user is willing to allow. Whichever the cause, the effect is the same: optimizers miss optimizations opportunities because they have to make conservative choices.

It is because of that that the idea of open compilers was born. An open compiler will offer the user the chance to provide that missing information to the optimizer in an easily-accessible form. This leads to the realization that one of the requirements in order to write an open compiler is that of facilitating the development of *customizable transformations* (or optimizations) so that the user can specify this missing or hard to compute information. The kind of information that is provided and the form in which it is given will vary with the purpose of the optimization, but the basic idea stays the same. The Broadway compiler [12, 13], for instance, allows the user to specify information as trivial as the "uses" and "defs" for the parameters of a subroutine. However trivial, this information is still very useful in the case of missing library source code, when the compiler cannot infer whether or not a variable will be modified or even accessed within a routine call. As a consequence, it will always have to assume "the worst", namely that it is modified, to ensure correct behavior in all cases.

Lamping et al. [17] discuss an open compiler for Scheme which would allow the user to make the decision of how to implement certain variables of interest. The concept they are promoting is to let the user benefit from all the advantages of a high-level language like Scheme for most purposes, but still have a way to influence the low-level representation of the runtime data structures pointed to by these user-chosen variables, if this is what the user needs. Kickzales et al. abstract away from this idea in a later paper [16] where they advocate reducing the opacity of traditionally black box software modules so that the user has better control over the implementation that is used when more that one is available. They no longer discuss open compilers but *open implementations*. This idea of letting the user participate in the selection of an implementation is not our goal per se, but it makes for an interesting analogy. We can parallel the optimizations to be applied to the module which used to be a black box and the user annotations for customizing them to the mechanism for reducing the opacity. In this way, we realize that the two are simply facets of the same concept.

Tourwé and De Meuter [25] consider a slightly more particular case of the general problem of missing information. This case refers to object-oriented programs, for which compilers are having a difficult time performing virtually any type of whole-program optimization. The impossibility arises from the dynamic dispatching of method calls, which prevents the compiler from knowing the particular definition of a method that a method call really refers to. The authors give the example of inlining a method body as an optimization which cannot be performed since it is not known at compile time which of the potentially many bodies of a method has to be inlined. Further on, this problem is aggravated by the fact that modern software architectural principles promote the use of objects and methods in a way that increases the degree of polymorphism in a program, making it ever harder for compilers to optimize. While some researchers (such as Dean et al. in [10]) worked out methods for finding out the precise type an object will have at runtime (with considerable success when this is actually possible) and thus replace dynamic dispatch with static binding, Tourwé and De Meuter chose the approach of an open compiler which the user can feed with information through Prolog predicates-like annotations. The information given to the compiler indicates design patterns in the source code, so that the compiler can make architectural optimizations. Using this information (and inferring additional one) the compiler performs a source-to-source transformation of the code, reimplementing the entire design pattern with different, but equivalent code such that the resulting implementation reduces the degree of polymorphism as much as possible. Then, it is assumed that a source-to-binary compiler will properly identify the lack of polymorphism and perform static binding of the method calls.

Steps towards creating an open compiler (for Java) have also been taken within the context of Stratego [4]. It is not the case that Stratego already has a working open compiler implementation. What it does have is a set of tools and libraries which will prove to be extremely valuable once work on an implementation actually begins. First of all, research on the Stratego framework has produced significant results regarding the specification of data-flow transformations, presented in [7, 20]. We will cover these later, in section 3 of this thesis. Secondly, Stratego already contains two libraries called java-front [2] and Dryad [1]. Java-front offers full parsing and unparsing support for Java 5.0, which is useful both for source-to-source and source-to-binary transformations, since we can directly manipulate an abstract syntax tree (AST) representation of the Java classes without having to worry about the rest. Dryad improves an already parsed Java AST by qualifying and reclassifying names

and typing variables and expressions. In addition, Dryad also provides a Java bytecode parser and unparser, so with a (currently unavailable) transformation from Java AST to Java bytecode AST, we already have a basic working compiler.

The final piece in the Stratego support for open compilers puzzle is represented by MetaBorg. MetaBorg is a "method for providing concrete syntax for domain abstractions to application programmers" [3]. Usually we start with a general purpose language and extend it to support a domain specific language which abstracts over an existing API. [8] discusses how Stratego can be used to easily achieve the two steps that the method implies: *embedding* the syntax definition of the first language into the second and *assimilating* the (already parsed) representation of one language into that of the other. A number of such embeddings have already been developed and are presented in [8]: Swul – embedding of a DSL for building Swing user interfaces in Java, XML in Java, Java in Java, etc. By employing SDF (Syntax Definition Formalism) and SGLR (Scannerless Generalized LR) parser (both of them part of the Stratego platform), MetaBorg can allow full modularization of the language syntax definitions (thanks to the generalized LR parser and the modularity features of SDF), can automatically generate a disambiguating parser for various language combinations (thanks to SDF's facilities for disambiguation: priority definitions, associativity specifications, reject productions or follow restrictions) and can correctly parse tokens of different languages depending on the context in which they appear (thanks to the scannerless parser).

The question that remains to be answered is how exactly MetaBorg fits in the open compiler picture. A different use of open compilers than the one we have already discussed has to be introduced in order to answer that. What we want is to let the user intervene in the compilation process not only by providing extra information for aiding compiler optimizations, but also by potentially extending the source language of the open compiler. We clearly only expect a restricted category of users (which we call meta programmers) to be interested in implementing extensions to a language, but we believe that this category is large enough to justify an effort in this direction. The extensions we have in mind are exactly the ones that MetaBorg facilitates: that of DSLs in general purpose languages, hence answering the question asked at the beginning of this paragraph. With the mechanism generically referred to as MetaBorg already available, what is left to be done is completing the Stratego framework with the proper technologies to support *extensible transformations*. More explicitly, we need to come up with a mechanism which would allow the transformations defined for a language to be easily extended so that they cover the user-defined embedding as well.

In essence, we believe that currently there is a serious need for open compilers in the object-oriented programming languages community and that researchers have to take this necessity into consideration. The road that ends with providing the software world with open compilers is long and we have merely taken a few steps on it, by exploring what an already advanced platform like Stratego still lacks for opening the way towards the creation of such a compiler. As explained in this section, we believe a technology that supports *extensible and customizable transformations* to be the core for a future open compiler (or rather that part of the core which has not been sufficiently explored yet). Therefore, our focus with this thesis has been on (1) filling some of the gaps in the Stratego platform which are necessary for and (2) somewhat experimenting with extensible and customizable transformations for an object-oriented language. We hope that our results and observations will prove valuable for further development.

1.2 Analysis and results

We define the creation of an open compiler (for Java) using Stratego as the combined eventual goal of this project and other projects that will follow it. Such a compiler has to differ from existing Java compilers in several key points. Most modern Java systems resort to just in time (JIT) compilation in order to ensure performance, which essentially means that they use a less sophisticated compiler to statically transform source code to bytecode and then transform bytecode to machine code at runtime. Our compiler should achieve performance primarily by means of static code optimizations. This does not prevent the later use of a JIT compiler at runtime, in which case the performance boost given by the static optimization would be combined with that of JIT compilation. Furthermore, we aim to differ from other compilers by making sure that the meta programmer can easily add her extensions to the language. This can be achieved by employing a MetaBorg-style model of embedding concrete syntax for other (domain specific) languages in the base language's syntax definition and parser. Finally, we want to open up the compiler internals to a level that is fully manageable by a regular user so that she can optionally intervene in the compilation process (more specifically, aid the compiler's optimizations).

Control flow While the open compiler for Java is the final goal, we are aware that the current point of development that the Stratego platform has reached could not have allowed us to achieve it within the bounds of this thesis. What we aimed to do instead was take concrete steps in that direction. To that end, we acknowledged that support for data-flow analysis (an essential component in an optimizing compiler, as ours is intended to be) is still incomplete in Stratego, so one of our goals was to contribute to improving this state of affairs. Already achieved and proved capabilities of Stratego for doing data-flow analysis included the possibility of peephole (restricted to a limited number of statements) and intraprocedural (restricted to the contents of a method) optimizations, as long as only basic control flow (sequencing, branching and iteration) was involved. However, there was no support for more involved control flow (including exception handling or handling of break's or continue's to labels). Also, before this thesis there had been no (documented) experiments with interprocedural control-flow (i.e., code with procedure/method calls), so a scheme for this also had to be devised.

In terms of results, strategies for supporting common non-sequential control flow instructions (break, continue, return and exceptions) have been successfully implemented, but they can currently support forward propagation data-flow transformations only. We have also implemented prototype support for backward propagation data-flow transformations, but this is still only marginally tested and only supports the break instruction. Regarding interprocedural control flow, we provided a couple of examples as part of the pointer analysis and generic typestate propagation work we discuss below. These examples illustrate how properties can be transferred from actual to formal arguments and from return variables to call site, how the implicit `this` pointer can be handled and how virtual dispatch can be managed.

Pointer analysis The lack of a support for complex control flow was not the only aspect of Stratego which impeded more realistic data-flow analyses. Before our work, potential implementers of data-flow analyses also had to deal with the absence of aliasing information. Aliasing indicates which variables, object fields or class fields point to the same memory

location, or *are aliased*. Such information is indispensable to most analyses, at least if we do not want them to be excessively conservative. Without aliasing information, an analysis will always have to make the conservative assumption that all variables, object fields and class fields may point to the same location. This means that every time an analysis sees, say, a modification of a variable, it has to assume that any other variable could have been changed as well (since it could be the case that the latter variable also points the memory location that was changed). It quickly becomes obvious that with such an assumption the effect of most analyses will be severely limited. Therefore, another goal of this thesis was to strengthen Stratego's support for more precise data-flow analyses by implementing a algorithm for computing alias information.

While this goal has been achieved for a prototype object-oriented language, some work still has to be done in order to map the implementation in order for it to support, e.g., the Java language. The two core algorithms that are involved do not change at all. However, some effort will be required to replace all the syntactic constructs that pertain to our prototype language with the Java-specific ones, given that a range of supporting tools are required by the core analysis (class hierarchy information, type information, code simplification). The pointer analysis is flow- and context-insensitive. This has the advantage of being fast and easier to integrate in various analyses and the disadvantage of being less specific. Analysis results are available in a user friendly format, but they currently require that the user perform a (provided) variable annotation algorithm on the analyzed code, in order to ensure a unique naming scheme for the program variables. It is improbable that this can be avoided, but better integration with the platform can be achieved.

Extensible and customizable transformations Our final objective for this thesis was to investigate how an existing data-flow transformation can be modified to cover extensions of the language (either with new language constructs or with embedded DSLs) and how user annotations can be used to aid such a transformation (extended or not). Support for extending and customizing transformations prepares the ground for open compilers by already enabling applications like active libraries and DSL embeddings. Active libraries is a term coined by contrast to the well-known static libraries of today. Their activeness stems from the fact that the library is meant to become an active component of the compilation process. The user will be given an (annotation) language which she will use to specify compilation information and/or compilation actions. These will be used by the compiler wherever the compilation process encounters code that uses the library in order to produce a user-customized result (usually for purposes of optimizing the way the library is used). The concept of active libraries will be further detailed in chapter 5. In perspective, a similar mechanism would be used when building an open compiler for allowing user intervention in the process.

DSL embeddings have already been introduced in the previous section in the context of MetaBorg. Extensible transformations should come to complete MetaBorg with a means to extend transformations along with syntax. The general idea is that we have a general purpose language *and* a number of transformations (optimizing or otherwise) defined to support all the constructs of the language. MetaBorg provides a method for extending the language with additional concrete syntax that supports domain-specific operations. What is still missing is a method for extending the transformations defined for the base language as well so that they support the additional constructs. Of course, one solution is to leave the transformations as they are and apply them only after the statements specified in the

embedded DSL have been assimilated to base language constructs, but this often results in missing optimization opportunities due to the loss of semantics that is inevitable in the assimilation process. Therefore, we consider extending the transformations to cover the new syntax as well to be a more attractive solution, since it will likely result in better optimization.

We have managed to provide some coverage of these objectives in our thesis, by producing two experiments: one as an attempt to writing a customizable transformation and one as an attempt to extending a transformation. Experimentation with customizable transformations was conducted by implementing a generic typestate propagation algorithm together with a mechanism that allows a user to specify code changes based on typestate information (pursuing some of the ideas of active libraries). Both the typestate change information and the code change information are provided by the user in a domain specific annotation language of our own design. Extensibility of a transformation was experimented by trying (and succeeding) to extend our pointer analysis such that it supports arrays in the analyzed language. The main focus in this work has been on extending both the language the we run pointer analysis over and the transformation that performs pointer analysis itself in such a way that the original code is left untouched. We report to what degree this has been possible in chapter 5 of this thesis.

Context The challenges of this thesis work have varied throughout its different goals. While some development was essentially new and based on little or no previous work (strategies for supporting non-sequential control flow), some has been an adaptation to Stratego of already existing algorithms (pointer analysis) and some has been inspired by other work, but took a rather different form and followed a different path (generic typestate propagation). Regardless of which part we have in mind, the main focus of this entire thesis was on enriching the Stratego platform with additional user support and enriching the Stratego code base with interesting solutions that can be the base for further work or even be used as is, perhaps with some adaptation. Filling in all the remaining missing pieces that will make it possible to use Stratego for developing an open compiler will still take a lot of effort, but nevertheless we hope to have made a significant contribution with the work presented herein.

1.3 Research questions

Now that we have sketched the general topics that this thesis comprises of, we can advance to defining the research questions that have guided our work. While we are confident that some of them have found their answer during this project, we are also aware that some have been considered only partially and some even not at all (particularly those related to extensible transformations). Despite this, we list them all here as they have been defined before our work had begun. In this way, the reader (perhaps coming back to these questions after reading the thesis) can get a good grasp of the extent in which we have managed to achieve our goals.

- **The Stratego model for data-flow analysis.** Related to this, we have tried to answer questions like: "how complete is the support currently offered by Stratego for managing control flow?", "what is still needed to ensure proper abstractions for dealing with all constructs of modern object-oriented languages (and Java in particular)?",

"what is the most appropriate mechanism which we can provide within Stratego for dealing with non-sequential control flow (such as exception handling or breaking and continuing)?", "can we build this mechanism in a layered way, with intermediate level abstractions and high level abstractions built on top of them?", "if so, what are these intermediate level abstractions?", "what is a good model for building interprocedural data-flow analyses in Stratego?".

- **Information for data-flow analyses.** Our efforts in this area have been guided by the following questions: "how can we use the Stratego model to implement essential analyses like pointer analysis and escape analysis?", "can we compute this information on demand or does it have to be done a priori?", "what is a proper way to represent such information (i.e., the user has to query for, say, the aliases of a variable, but a variable name is not an unique identifier, so how do we compose a key that is natural enough for the user to use easily)?", "is the accuracy of the proposed context-insensitive and flow-insensitive pointer analysis good enough for providing a real benefit to other data-flow analyses?", "for both the pointer and the alias analysis we need a call graph: can we use the information provided by Dryad to build one efficiently?".
- **Extensible and customizable transformations.** The questions that we have focused on in our experiments with extensible and customizable transformation are as follows: "how can a transformation be extended in a modular way in order to support new syntactic constructs in a language?", "do we need a new way in which to compose transformations for allowing modular extension?", "how can the user interact with transformations in particular cases?", "how can this be generalized so that virtually any transformation can be customized?", "how useful is it, in the context of customization, to provide a generic mechanism for propagating typestate?", "does the customization mechanism naturally scale to cover extended transformations as well?", "what mechanisms (except the ones covered by this thesis) are still needed for writing an open compiler for Java?".

1.4 Thesis overview

We continue in chapter 2 with discussing the setup of our work and providing some background information about the Stratego platform and, in particular, about its dynamic rules. We recommend this chapter as background material for the reader that is not so familiar with Stratego and/or its dynamic rules. In the last section of the chapter we also introduce the toy language that we have designed and implemented in order to use as a target language in the second and third parts of this thesis. Each of the following three chapters describes one of the three parts of our work. The part on the extension of the dynamic rules library with support for non-sequential control flow is covered in chapter 3. The next chapter, chapter 4, analyzes the need for alias information in Stratego and discusses an implementation of pointer analysis that we have produced in order to show how this alias information can be obtained. The third part of our thesis, discussed in chapter 5, covers the two experiments with customizable and extensible transformations that we have conducted. We end this material in chapter 6 with some conclusions and suggestions for future work.

Chapter 2

Project setting

This chapter aims to establish the context of this project by presenting a number of tools that have been employed throughout our thesis and also by laying the foundation for some of the topics that the reader needs to be familiar with before proceeding to the bulk of our work. Everything discussed in this chapter represents the ideas and work of other people; we are merely presenting them here under a common heading in order to provide you with an easily accessible reference.

2.1 Stratego/XT

Stratego/XT is a combination of a language for implementing program transformations using rewriting strategies (Stratego) and a set of readily-available transformation tools which can be used for building complex pipelines of transformations (XT). Stratego/XT has received substantial attention over the past few years and has undergone serious development, leading to an abundance of features and improvements. The research work on the Stratego/XT platform has been documented in close to one hundred papers up to this point, so it is in no way our intention to provide anything near an exhaustive presentation. Instead, we only try to offer a quick overview of some of the basic features which are important for understanding this thesis. For a more extensive presentation we refer you to [7], while for a hands-on tutorial we refer you to the Stratego manual [6].

The general transformation pipeline that is used in Stratego involves parsing a program to an internal representation (called annotated term format, or ATerm format), applying one or more transformations to this representation in order to achieve a particular purpose and finally unparsing (or pretty-printing) the transformed representation back to program text (not necessarily using the same language as the original). Stratego/XT already supports parsing and pretty-printing a wide variety of languages "out of the box" (including Java 5.0), thereby saving the user a lot of effort. The user generally has to concern herself only with writing transformations that manipulate the ATerm representation.

Such transformations are programmed in Stratego by employing a paradigm based on *conditional rewrite rules*, the application of which is controlled by *strategies*. Strategies and rewrite rules can be developed independently of one another, resulting in the possibility of creating new strategies which implement completely different transformations than the old ones by simply applying the same rewrite rules in a different manner. Stratego comes with a

large library of useful strategies which can usually be parameterized with user-defined strategies in order to achieve various results. This system makes the writing of transformations simple and attractive.

To illustrate the concepts of rewrite rules and strategies, consider this very simple example:

```
Fold : Plus(Int(x),Int(y)) -> Int(<addS>(x,y))
Fold : Times(Int(x),Int(y)) -> Int(<mulS>(x,y))
Dist : Times(Plus(x,y),z) -> Plus(Times(x,z),Times(y,z))
Dist : Times(z,Plus(x,y)) -> Plus(Times(z,x),Times(z,y))

simplify = innermost(Dist <+ Fold)
```

The first four lines in the code above show the definition of two rewrite rules and the last line shows the definition of a strategy that applies the rewrite rules. The `Fold` rewrite rule has the purpose of evaluating additions and multiplications (we implicitly assume in the definition that the arguments to `Plus` and `Times` are integer values), while the `Dist` rewrite rule applies distribution of multiplication over addition. None of them has an associated condition. Conditions guard the application of a rewrite rule and can be specified within an optional `where` clause followings the rewrite rule. The `simplify` strategy applies the two rewrite rules in fix point point manner by using the `innermost` library strategy. `innermost` repeatedly applies the strategy that is passed to it to all nodes of a tree in a bottom-up fashion, until it can no longer be applied.

Explaining the full details of the Stratego syntax is beyond the scope of this presentation, but we will briefly mention here a few things as a guide to the Stratego code used in this proposal. If you are already familiar with Stratego, you can safely skip this paragraph. The binary <+ operator indicates that first it tries to apply its left argument and, if that fails, it tries to apply its right argument. It is very often that one sees something like this in Stratego: s_1 <+ s_2 <+ $\cdots$ <+ s_n. This simply means "try to apply s_1, if that fails, try s_2, and so on." If none of $s_1, s_2, \ldots, s_n$ succeeds, the entire construct fails. An operator related to <+ is the tertiary operator formed by < and +. s_1 < s_2 + s_3 tries to apply s_1 and, if successful, applies s_2, if not, applies s_3. Sequential application of strategies is achieved with the ; operator (s_1 ; s_2 applies s_1 and s_2 in succession, only applying s_2 if the application of s_1 was successful). A strategy can take any number of strategies *and* terms as parameters (the syntax for this is $s(s_1, s_2, \ldots, s_n|t_1, t_2, \ldots, t_m)$). A strategy is normally applied to the current term. If instead we need to apply it to a different term, we can use the following syntax: <s>t. Also related to the current term is the `where` construct, which ensures that the current term valid before the code enclosed in a `where` will be reinstated before continuing execution after the `where` (`where`(s) is equivalent to `?x; `s`; !x`). `?` and `!` are two basic constructs of Stratego used for *building* and *matching* terms. A term is built by prefixing it with an exclamation mark (`![1,2,3]` builds the list `[1,2,3]`). A term is matched by prefixing either a variable (to match the entire term), or a pattern (to match parts of the term) with a question mark. For example, `?x` matches any term and binds variable `x` to it, while `?MyTerm(x,y)` only matches a `MyTerm` with two arguments, binding variable `x` to the first argument and variable `y` to the second. The => operator can be used to match a term as well after first applying the strategy which represents its left argument: `s => x` is equivalent to `s; ?x`. Finally, one other very used construct in Stratego is congruence. The underscore (`_`) can be used to match parts of a ATerm we are not interested in.

We have already seen an example of a library strategy defined by Stratego which can be

customized by the user, `innermost`. There are numerous other generic strategies defined in the Stratego library, such as `outermost` (top-down fix point application), `topdown` and `bottomup` (traverse the abstract representation top-down/bottom-up and apply a strategy at each node) or `alltd`, `oncetd` and `sometd` (apply a strategy to all topmost places where it can be applied, to exactly one such place or to some of the places). These and many others can be used and combined and different ways to achieve virtually any conceivable semantics of a transformation.

The final feature of Stratego that is important to present in this short overview refers to traversal of terms. This is useful especially when we need to apply different strategies in different branches of an abstract syntax tree. To this end, Stratego introduces congruence combinators. For each n-ary term constructor c, the strategy $c(s_1, s_2, \ldots, s_n)$ is called a congruence. Its effect is that strategy s_1 is applied to the first component of the term, s_2 to the second and so on. The entire strategy is successful if all s_i's apply successfully. To illustrate, consider the if-then-else construct in a language, represented in ATerm format as `If(condition,then-branch,else-branch)`. The `then-branch` and the `else-branch` will usually contain a statement or a lists of statements, while the `condition` will be a boolean expression. Assume we have a transformation which needs to introduce curly braces (block braces) for if-then-else's that miss them. This would transform the code on the left into the one on the right:

```
if (x < 5)
  y++;
else
  y--;
```

```
if (x < 5) {
  y++;
}
else {
  y--;
}
```

In order to achieve this, we need to apply the following `AddBlock` *conditional* rewrite rule to the `then-branch` and the `else-branch` of the `If` term, but not to the `condition`:

```
AddBlock :
  st -> Block([st])
  where <not(?Block(_))>st
```

In order to do this, we can use the congruence combinator as shown:

```
add-blocks =
  bottomup(try(If(id, AddBlock, AddBlock)))
```

The `id` strategy leaves its term unchanged, while the `try(s)` is the same as the simple application of the strategy `s`, except that it ensures success regardless of the success or failure of the application of `s`. The `bottomup` strategy ensures the traversal of the entire tree in a bottom-up fashion (we could have chosen a top-down traversal just as well, it would have worked just as well in this case). As a final note, $c(s, s, \ldots, s)$ is equivalent to `all(s)`.

2.2 Dynamic rewrite rules

Dynamic rewrite rules are the feature of Stratego that we are most interested in, mainly because the first part of out thesis is heavily related to it. We have seen "normal" (or static) rewrite rules in the previous section. These are defined for the entire scope of a Stratego

program and thus cannot be modified during its execution. However, many times when analyzing a program we need to take context into consideration, usually in order to restrict certain actions to certain contexts only. Dynamic rewrite rules have been introduced in Stratego primarily to allow easy handling of context sensitivity. Context sensitivity can be handled with static rewrite rules as well, but only in a complex and awkward way. Dynamic rewrite rules are defined and discussed at length (both formally and intuitively) in [7]. An addition to dynamic rewrite rules called *dependent rewrite rules* was introduced later and is discussed in another paper [20]. We provide here a quick summary of the aspects related to dynamic rewrite rules that are needed for understanding chapter 3. The presentation is mostly based on [7].

Defining and undefining dynamic rules Dynamic rewrite rules are identical in form with the static ones, except that they are not defined at the top level of a Stratego program, but within a special `rules` construct, which can appear in any place a normal strategy can appear. *Defining* a rule simply means writing it in a such a `rules` construct. The use of dynamic rewrite rules usually implies using some context information which is available when the rule is being defined in order to achieve context sensitivity. For example, we could use such a rule in a constant propagation transformation to associate variables with their values when an assignment is encountered:

```
const-prop-assign :
  Assign(x, e1) -> Assign(x, e2)
  where <const-prop>e1 => e2
        ; try(
            <is-value>e2
            ; rules(ConstProp : x -> e2)
          )
```

The line `rules(ConstProp : x -> e2)` defines a new dynamic rule called `ConstProp` which binds variable `x` to constant expression e2. In a top level strategy we can apply the dynamic rule `ConstProp` just as if it had been defined statically. If we apply it before it is defined, its application will simply fail, otherwise it will behave like its static equivalent.

Undefining a rule is done by adding the `-` suffix after the `:`, while only specifying the left hand side of the rule. We can now correct the code above so that whenever e2 is not a constant value, the rule `ConstProp` is undefined so that we do not wrongly propagate the old value that variable `x` had before the assignment:

```
const-prop-assign :
  Assign(x, e1) -> Assign(x, e2)
  where <const-prop>e1 => e2
        ; if <is-value>e2 then
            rules(ConstProp : x -> e2)
          else
            rules(ConstProp :- x)
          end
```

Scoping dynamic rules If we look at our `ConstProp` rule, we realize that it should no longer be active when variable `x` goes out of scope, to avoid incorrect replacement of occurrences of `x` that refer to a different variable `x`. One option would be to undefine the rules manually at the end of a scope, but this would require horrible administration in order to know which rules to undefine at the end of which scopes. Instead, Stratego offers a

scoping construct for dynamic rules which has the semantics that any changes (definitions and undefinitions of dynamic rules) made within the scoped application are automatically dropped when the scope is exited. We want to scope our `ConstProp` rule whenever we enter a block of statements in our traversal, so instead of this:

```
const-prop =
  ...
  <+ Block(const-prop)
  ...
```

which simply applies the `const-prop` strategy to the statements in the block, we can write something like this to achieve the same effect, but making sure that all effects on the dynamic rule definitions are dropped after the block is traversed:

```
const-prop =
  ...
  <+ Block({| ConstProp: const-prop |})
  ...
```

Labeling scopes Scoping is a start, but it is actually not sufficient if we want to support nested blocks (which are allowed in virtually all programming languages). This is because if a variable is defined in a scope, we need to make sure that all changes to the `ConstProp` rule that refer to that variable are undefined not when the scope where they are performed in is exited (as it happens if we use the code above), but when the scope in which the variable is defined is exited. To accommodate this, Stratego allows the labeling of scopes with random terms. Then, once a scope is labeled, we can refer to it by using its label. `ConstProp+x` labels the current scope for the the dynamic rule `ConstProp` with the term that `x` is bound to. `ConstProp.x` indicates that the action that follows (definition or undefinition of the rule) is to be performed relative to the scope labeled with `x` instead of relative to the current scope (which is the default). With this syntax, we can finally give the correct form of (a small part of) the constant propagation transformation:

```
const-prop =
  ...
  <+ Block({| ConstProp: const-prop |})
  <+ const-prop-var-dec
  <+ const-prop-assign
  ...

const-prop-var-dec :
  VarDec(x, type) -> VarDec(x, type)
  where rules(ConstProp+x :- x)

const-prop-assign :
  Assign(x, e1) -> Assign(x, e2)
  where <const-prop>e1 => e2
        ; if <is-value>e2 then
            rules(ConstProp.x : x -> e2)
          else
            rules(ConstProp.x :- x)
          end
```

The code `rules(ConstProp+x :- x)` both labels the current scope with `x` and undefines any definition of `ConstProp` for `x` that might have been valid before the variable declaration. Furthermore, `const-prop-assign` was changed to define and undefine `ConstProp` for `x`

relative to the scope where x was defined, so that all changes are dropped only when that scope is exited.

Intersection and union of dynamic rules Going back to our constant propagation example, consider what happens when we need to analyze a if-then-else in our target code. We need to make sure that (1) the rules applied in the then branch and the ones applied in the else branch do not overlap and (2) the rules valid after the if-then-else are the result of merging those generated by the then branch and those generated by the else branch. In the case of constant propagation, the merging process is done by intersecting the two sets of rules, but in the case of other transformations, we might need to merge by taking the union of the two sets of rules. Stratego offers the `/...\` binary operator for intersection and the `\.../` binary operator for union. In both cases, instead of the `...`, the user has to specify one or more dynamic rule names separated by commas. What Stratego does is clone the set of rules valid when the intersection/union operator is encountered, run each of the two argument strategies with one of the set of rules as the active one and in the end intersect or take the union of the two resulting sets, setting the new active set to the result. To illustrate, here is the code that performs constant propagation over an if-then-else construct:

```
const-prop-ifthenelse =
  If(const-prop, id, id)
  ; (const-prop
     <+ If(id, const-prop, id) /ConstProp\ If(id, id, const-prop))
```

We run the strategy `If(id, const-prop, id)` with one set of rules and the strategy `If(id, id, const-prop)` with its clone, performing constant propagation over the then and the else branch, respectively. Afterward, the rules that are common to both branches are kept, while the rest are dropped.

Fix point iteration We have a different problem if we need to analyze loops. In the case of loops, we need to make sure that the set of rules active after the loop is correct irrespective of how many times (including zero) the loop is run. For this, Stratego offers two fix point operators, both of which keep running a strategy over and over until the set of rules stabilizes. In order to do this, the set of rules that is active before the fix point operator is cloned and after each iteration the clone is updated by merging it with the set of rules obtained by running the strategy. This process continues until the cloned set of rules does not change from one iteration to the next. Merging can again be done by intersection or union, hence the two operators: `/...\*` (intersection) and `\.../*` (union). As before, `...` is to be replaced with one or more dynamic rule names.

Constant propagation over a while loop illustrates how the intersection fix point operator can be used:

```
const-prop-while =
  While(id, id)
  ; /ConstProp\* While(const-prop, const-prop)
```

The `While(const-prop, const-prop)` is the strategy that is run until the set of rules stabilizes.

2.3 TIL

TIL, or the Tiny Imperative Language, is a toy procedural language defined within the core Stratego compiler. Stratego features a parser and pretty printer for this language. TIL is particularly fit for use in unit tests for the Stratego compiler and library because this enables the independence of the unit tests from any external libraries (e.g., for parsing and/or manipulating real languages).

We mention TIL as one of the tools related to this thesis project because of two reasons. The first reason is that the preliminary task that had to be covered before the work on the thesis topics proper could begin was to develop a full unit test suite for the dynamic rules library. This library is one of the more complex ones implemented in Stratego, and the lack of a proper unit test suite made us reluctant about enhancing it, lest we should break already working things. Since the dynamic rules library is part of the Stratego library, we used TIL as the object language employed in our unit tests (for the reason discussed above). The second reason is that TIL was the base on which TOOL (Tiny Object Oriented Language) was built by the author of this thesis. In essence, TOOL includes the full TIL syntax and adds object oriented extensions to it in order to supply a toy object oriented language.

2.4 Object oriented languages. TOOL

The main aspect that differentiates an object oriented programming language like Java or C++ from an imperative one like C is the presence of virtual dispatching. Virtual dispatching (or run time dispatching) ensures that the run time type of an object is taken into consideration in order to decide which of potentially more than one method with the same signature has to be executed. Virtual dispatching will be explained at more length in section 4.3.2, along with examples. The problem that virtual dispatching introduces in a compiler is that, unlike with imperative programming, we can no longer perform interprocedural (or perhaps the right term should be inter-method) analysis quite as easily. This is because we cannot always tell at compile time what method body we need to analyze based on the call site. This creates a problem that is usually handled by analyzing all potential methods that might be called at that particular call site and performing some sort of merging of the results that makes sense in the context of the particular analysis.

Another difficult aspect that we have to deal with are exceptions. While exceptions are not strictly a feature of object oriented programming, their integration in main stream languages has taken place concurrently with the migration to object oriented languages, so the two are usually related. Exceptions are difficult to handle by the compiler because of the complicated control flow paths that they introduce. Because of this, a lot of compilers for object oriented languages actually refrain from performing any type of optimizations of program blocks that contain exceptions, which clearly reduces the efficiency of the overall optimization process. The problems introduced by exceptions in the compilation process are further debated in section 3.4.3.

Finally, when dealing with object oriented programs, we are also confronted with some new types of variables that we have to consider. Imperative programming used local and global variables along with function or procedure parameters. Object orientation maintained all these (albeit global variables were turned into static fields) and added two more: the object

which is the target of a method call (referable with the `this` variable) and instance fields, which are available in all non-static methods of a class without their being passed explicitly to the method. While this is intuitive enough for the user to not create any problems, it can pose a problem for compilers, since these variables have to be tracked in a data-flow analysis as well.

TOOL TOOL (Tiny Object Oriented Language) is a toy language developed as part of this thesis (or rather in support of this thesis), built on top of TIL (see also section 2.3). The reason why TOOL was needed instead of resorting to a real language (like Java, for instance) is that in the second and third parts of the thesis we have been trying new grounds in what the Stratego platform is concerned and, because of this, we have wanted to contain the extent of the problems we have tackled such that the time frame allotted for our work could be (somewhat) observed. Using a real language would have introduced an array of issues (complicated syntax, directory structure, name qualification, etc.) that would not have been related to our core tasks. Using a laboratory language like TOOL has allowed us to avoid most of these issues and thus keep focused on our main concerns.

However, we have created TOOL in such a way that all features of an object oriented language that were important to our research were present. Specifically, TOOL supports classes with constructors, methods, fields and static blocks, static and abstract methods, static fields, single inheritance, object construction, method calls, field accesses, the special "this" and "null" keywords, the special "super" constructor call and exceptions. TOOL does not support access modifiers, packages and multiple class definitions in a single file (and certainly many other features).

We choose to present TOOL using an example class rather than a formal syntax definition for two reasons: (1) we find it unlikely that the reader will need to use TOOL herself, so giving a language reference here makes little sense and (2) the Stratego SDF definition of TOOL is as clear a definition as any and, moreover, is the definitive one, so we refer any potential TOOL user to that one. This example shows a part of a class from one of our tests that implements a linked list. Most basic aspects of TOOL are visible in the code: class definition (class `List` inherits from `Object`), constructor definition (the `List` method), method definitions (method `put`), field definitions (fields `head`, `tail` and `size`). Statement syntax is inherited from TIL (`:=` for assignments, `;` as statement separator, the usual operators, etc.). What is perhaps important to point out is that the syntax itself does not require that all field accesses are prefixed with an object name (in our example, `this`), but we do adhere to this convention throughout our code so that our tools have an easier time separating field accesses from variable accesses. Another unusual aspect is the convention that constructors in TOOL always have to return `void` (the reason behind this being merely a simplified syntax definition).

```
class List extends Object

  field head: ListNode;
  field tail: ListNode;
  field size: int;

  // builds a new List
  def List(): void
  begin
    this.head := null;
    this.tail := null;
    this.size := 0;
  end
```

```
// adds an element at position
// index in the list
def put(index: int, elem: Object): void
begin
  if (index < 0) |
    (index >= this.size) then
    return;
  else
    var n: ListNode;
    var i: int;
    i := 0;
    n := this.head;
    while i < index do
      n := n.next;
      i := i + 1;
    end
    n.value := elem;
  end
end
```

A TOOL program is defined in a separate unit than a class, is expected to only contain a block of statements and to be defined in the class Main.tool. Here is TOOL's "Hello world":

```
begin
  println("Hello␣world!");
end
```

`println` and a couple of other functions are predefined in TOOL. Everything other function (or rather function-like definition) that is not predefined has to be a method (either static or non-static). We have developed an evaluator (interpreter) for the TOOL language that starts execution in the current directory by reading the Main.tool file and expects all classes used therein to be found in the respective <class>.tool file in the same (current) directory. The interpreter supports the following features of TOOL:

- complete procedural control-flow (including the `break` & `continue` statements);
- creation of objects (`new ClassName([args])`);
- single class inheritance (specifically, objects of derived class' type contain fields of derived class and base class(es); also method calls are resolved appropriately);
- proper handling of static and non-static fields, abstract and concrete methods, static declarations (which are run at class load time);
- (virtual) method invocation (argument passing, return values);
- field access;
- the `null` value (which has type `Any`, a subtype of all defined types);
- a couple of system functions: `print`, `println` and `string` (`print[ln]` take any number of arguments; `string` takes one argument and converts it to a string);
- the interpreter also performs some sanity checks based on type information (left hand side of assignment type against right hand side of assignment type, actual arguments type against formal argument types and declared return type against type of variable that receives the result).

The interpreter does not support the exception handling mechanism and releasing of memory.

In addition to the interpreter, we have also developed a type annotator for TOOL. The annotator takes an entire hierarchy of TOOL classes (and optionally a TOOL program) and annotates all expressions (including class fields) with their type. While running the annotator some type checking is also performed.

Chapter 3

Enhancing Stratego's dynamic rules

The first layer of work completed as part of this thesis consisted in *improving*, *extending* and *increasing the reliability* of Stratego's dynamic rules library. We increased the reliability of the library by developing a unit test suite that covers the most complex parts of the library, we improved the library by fixing bugs revealed both by the unit tests and by our code verification during the development and we extended the library by adding new strategies that help the user to easily handle various types of jumps in the control flow of a program. This chapter already assumes a certain degree of familiarity with dynamic rules (the use of which has been summarized in chapter 2) and proceeds to cover almost exclusively the way in which the new strategies are implemented and can be employed. The unit tests and bug fixes will be briefly covered as well, but the focus will still stay on the extensions of the library.

3.1 Overview

Originally, the dynamic rules library already incorporated a number of abstractions that could (and obviously still can) be successfully used for flow-sensitive transformations, as shown in [7, 20].

Basic statement sequencing is supported implicitly since dynamic rules, once defined, are automatically valid until they are undefined or until the scope in which they have been defined is exited. Thus, we do not have to explicitly ensure the flow of information from one statement to the next.

Conditional structures (if-then-else) are supported by the intersection and union operators. These operators work in such a way that whenever branching occurs in the control flow due to an if-then-else, two distinct sets of dynamic rules are computed, one for the "then" branch and one for the "else" branch. After the if-then-else, the two sets are merged either by intersection or by union and the resulting set will be the one valid from that point onwards.

Finally, loops are also supported in the current implementation of the dynamic rules library through fix point iteration. If a loop (may it be a while loop, a for loop or a do-while loop) is encountered in the control flow, the fix point operator can be used to compute a set of rules that are valid after the loop. This is done by repeatedly computing the set of dynamic rules defined by a strategy that is applied to the loop block, and merging this set with the

previously computed one (the first set of rules is the one that is valid before the loop). This is done until the set of rules remains stable from one iteration to the next. Either union or intersection can be used as the merging operation performed at the end of each iteration.

The kind of support that is available enables the writing of conceptually interesting data-flow transformations, but is not advanced enough yet to enable application to real-life source code. The reason for this is the lack of abstractions in the dynamic rules library for non-sequential control flow. What we refer to here is control flow statements that allow some type of goto behavior. In the case of Java, for instance, such statements are *break*, *continue*, *return* and *throw* (along with the entire exception throwing/catching mechanism). The user could, in principle, use Stratego for handling such behavior, but it would virtually imply writing the abstractions for dealing with such jumps herself. There is a clear need here for a general mechanism to support this type of control flow.

Hence, our goal with this first layer of work was precisely to add these missing operations to the dynamic rules library. With this additional support, we aimed to make the Stratego framework advanced enough to allow the writing of transformations which can handle all constructs of a modern language. While doing this, we also strived to meet the non-functional requirement of usability of the new abstractions. We hope to have supplied the user with abstractions which are easy enough to use that handling of non-sequential control flow is at the same high level as existing handling of sequential control flow.

Unfortunately, the complexity of the code in the dynamic rules library was (and still is) far from trivial, and that has brought it in a somewhat delicate state in which any modification came with the risk of breaking previously working code. This made ours and other development dangerous and thus unattractive. Since leaving the code alone was not an option, we acknowledged that the only alternative was to make modifications feasible again by implementing a unit test suite that would cover at least the more complex functionality of the library. While some disparate unit tests were already available, there had not been a proper organizational and development effort in order to reach a test suite that offers a clear indication of how much of the library code is actually covered. We hope that through our work we have eventually achieved this goal.

In addition to unit tests that verify correct behavior of the old code, we have also developed unit tests for our own extensions as well, such that the library is now in a reasonably reliable state, with further modifications feasible, if not attractive. It is not only the unit test suite that makes us have more confidence in the library on the whole, but also our fixing of all the bugs revealed by the unit tests and by the code scrutiny. Naturally, we have no way of knowing just how many of the problems with the library have been discovered during our efforts, but certainly important steps forward have been made.

3.2 Related work

The work we carried out through this phase (supporting non-sequential control flow) is related to efforts for building and using control flow graphs (CFGs) that accommodate interruptions in control flow (break, continue and return statements as well as exception throwing). Even though the approach we take in Stratego is different from the CFG-based approaches, there are inherent similarities between the two. In particular, all the additional edges that are introduced in CFGs had to be captured identically in the model that we developed for Stratego.

The break and continue instructions are quite trivial to handle in a CFG, therefore literature can be summarized by common sense. To accommodate a break statement we create an edge from the node representing the statement to the node representing the statement following the structure that break indicates the termination of. In the case of continue we need to do the same thing and, in addition, to add an edge from the node representing the continue statement to the node representing the condition of the loop that the continue statement interrupts.

Accommodating exceptions, on the other hand, is not such an easy job, as the rest of this section discusses.

Hennessy identifies a number of issues that exceptional control flow introduces in classical program optimization techniques [14]. He distinguishes two types of effects of exceptions, which he calls indirect and direct, and which have to be properly considered in order to ensure the correctness of the optimizations in the presence of exceptions. *Indirect effects* refer to method execution being aborted after a call to a method that generates an exception. The effect is that we cannot assume during the optimization that the statements following the point where the method can be aborted will be executed. *Direct effects* are the ones resulted from the execution of exception handlers. These obviously have to be included in the analysis of the program. Hennessy also discusses the issue of implicit exceptions. These are exceptions that can be raised without this being specified explicitly (similar to unchecked exceptions in Java). He proposes either that implicit exceptions are made explicit (cumbersome) or that the statements that could generate such exceptions (e.g., arithmetic statements) are identified and treated accordingly. Either way, he concludes that implicit exceptions severely hamper optimizations because they can occur almost anywhere and there is not much we can do about it (except design languages which use such exceptions very restrictively or even not at all).

[22] addresses the problem of constructing a CFG for Java by proposing a method for building paths in the CFG that represent the control flow paths introduced by exceptions. The analysis is straightforward, taking the various types of paths that exceptions can introduce as a starting point and determining how this has to be mimicked in the CFG. Their representation is precise, based on a local type inference algorithm, which allows the matching of throw statements with the catch clauses that handle them. The main difficulty lies with introducing edges between nodes representing throw statements, nodes representing catch clauses and nodes representing finally clauses. A throw node will have to be united either with the innermost catch node that catches the type of exception that is thrown or with the finally node that intervenes on the path to the catch node that eventually catches the exception.[1] This will happen directly, if the catch node or finally node to which the throw node has to be connected is part of the same method as the throw node. If, however, that is not the case, then an intermediate node called an *exceptional node* will be created, signaling an exceptional exit from the method. Later on, when constructing the interprocedural CFG, this node will be connected (again, directly or indirectly) with the proper catch node or finally node. The statements in a finally block are represented in a separate CFG and interaction between the main CFG of a method and that of the finally block is done by inserting call nodes (similar to those used to model normal method calls). For each finally node there will a call node per try block and per each of the catch blocks that are part

[1] If you are not familiar with how exceptions work in Java, you might be able to understand this section better by first reading the overview given in section 3.4.3.

of the same try-catch-finally construct. These will be linked with the nodes representing the last statements in the try and the catches. In addition, there will be a call node per exception type that is raised but not handled in the same try and catch blocks. A throw of an unhandled exception of type t will be linked with the call node to finally for type t. Each call node to finally has a matching return node, and these return nodes are linked (directly or indirectly, through exceptional nodes) either to a call node to the next finally up in the nesting hierarchy or to a catch for the proper type. The CFG that results represents all the "normal" paths and the ones introduced by exceptional execution.

3.3 Implementation of dynamic rules

In order to explain the details of our work regarding abstractions for non-sequential control flow, we need to first introduce the basic implementation ideas on which the dynamic rules library is built. A summary of dynamic rules have already been given in section 2.2 of this proposal, so please make sure you read that section first if you are not familiar with them.

Hash tables. Hash tables are the data structure that is used for storing, managing and applying dynamic rules definitions. A modified version of left hand side of a dynamic rule (in which all unbound variables that appear in the left hand side are replaced with the dummy term `[DR_DUMMY()]`) is used as the key in the hash table. The *logical* value such a key is mapped to is the right hand side of the dynamic rule definition. The *real* value stored in the hash table, however, is not exactly that, but a tuple that contains all the variables used in the condition of the dynamic rule (i.e., the one specified in the where clause) and the right hand side itself. This tuple is called a *closure* and contains enough information to uniquely identify the original right hand side. (The mechanism by which this tuple is effectively used to obtain the concrete value is less important for us. If you are interested in the details, you are referred to [7].)

Hash table values are lists. Instead of mapping each key to a single value, the dynamic rules library works instead by mapping keys to lists of values. This is because dynamic rules with the same left hand side can be bound to multiple right hand sides, as a result of the possibility to add definitions to a rule. For instance `rules(Foo : "x" -> 1)` followed by `rules(Foo :+ "x" -> 2)` will determine the key `"x"` to be (logically) mapped in the hash table to the list `[1, 2]`.

Scoping with hash tables. Among other things, the dynamic rules library also supports scoping (see section 2.2 for details). Scoping is implemented by creating a new hash table for each scope. The resulting model is that we have a number of rule sets (one rule set per dynamic rule name) and each rule set is composed of a variable number of rule scopes (i.e., hash tables representing scopes). Maintaining a list of scopes per dynamic rule name enables each rule to have its own scoping semantics, while representing scopes using distinct hash tables enables a good compromise in what performance is concerned. Defining, undefining and looking up of rules is slightly more expensive than it would be with a single hash table, since in the worst case we have $O(s)$ complexity (with s being the number of scopes). Nevertheless, the number of scopes is usually small, so we can still regard these as a constant time operations. The major benefit is yielded when entering and exiting scopes, since these both become $O(1)$ operations.

Intersection and union of rule sets. For dealing with conditional structures in control flow, intersection and union of rule sets was implemented (generally speaking, merging of rule sets was implemented). We explain here how the intersection operation works for a single rule set. Assume the intersection operator is called with the Foo dynamic rule as parameter (i.e., s_1 /Foo\ s_2). At this point, the rule set for rule Foo is duplicated[2], so we end up with two rule sets: the original one (say rs_1) and its clone (say rs_2). First, strategy s_1 is executed (implicitly using rs_1 as the current rule set), then the current rule set is changed to rs_2 and strategy s_2 is executed. After this process, we have rs_1 containing the effects of running s_1 and rs_2 containing the effects of running s_2. Since we are dealing with the intersection operator, it means that the user was interested in keeping only those changes which were introduced both by s_1 and by s_2. Hence, the final operation will be to *intersect* rs_1 and rs_2 and set the resulting rule set as the active rule set after the operation s_1 /Foo\ s_2.

Generalization to multiple rule sets is done by performing the same cloning/setting as current rule set/merging process for each rule set in particular (however, we still execute s_1 and s_2 only once). The union operation is identical, except that in the end the rule sets are joined by union, not by intersection.

Fix point iteration. The idea behind the fix point iterator implementation is also based on merging of rule sets (again, the merge operation can be chosen from intersection and union), except this time we do not merge the rule sets created by two different strategies, but instead we merge the rule sets created by the same strategy applied over and over, until we reach a fix point. We explain how the intersect fix point operator works for a single rule set. Let rs_0 be the current rule set for rule Foo before the call to /Foo* s. The first action to take is duplicate[3] rs_0 and save the result as rs_{ref}. Then, still with rs_0 as current rule set, we run strategy s, producing rule set rs_1 (updated version of rs_0). rs_1 is then intersected with rs_{ref}. If rs_{ref} suffers any changes, then strategy s will be run again, this time having rs_1 as the current rule set. The resulting rule set (rs_2) is intersected with rs_{ref} and the process continues in a similar fashion. The iteration ends whenever rs_{ref} suffers no changes after intersection with rs_i. When that happens, we know we have reached a fix point (i.e., nothing changes in the rule set from one iteration to the next). Since at this point rs_{ref} and rs_i (with i being the number of the last iteration) are identical, we can use any of them as the active rule set after the fix point iterator[4]. Considerations regarding generalization to multiple rule sets or using union as merge operation are exactly as before.

Rule scopes and change sets. We have indicated both in relation to intersection/union and the fix point iteration that rule sets are not exactly duplicated, since this would be extremely expensive computationally-wise. If you recall, we have explained earlier that each rule set is represented by a list of rule scopes and that each rule scope is a hash table (actually, rule scopes also contain a list of labels, but this is irrelevant for our discussion). In order to support lightweight duplication of rule sets, the concept of change sets was introduced. The point of a change set is to temporarily store all the changes that would normally be committed to the rule scopes themselves. Since an understanding of how rule

[2]The explanation details how things work conceptually. For efficiency reasons, the actual implementation avoids actually duplicating the rule set.

[3]The same comment applies as in the case of intersection and union.

[4]For the curious reader: the implementation uses rs_i.

scopes and change sets work together is essential for the following subsections, we give a more formal explanation.

Let $RS = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ be a rule set represented as a list of rule scopes and change sets ($\alpha_i, i = 1..n$ stands for either); where $i < j$ implies that α_i is more inner a rule scope or change set than α_j. Let k be the smallest value for which α_k is a change set. The semantics of change sets implies that if in this state any rule had to be added to, changed in or deleted from a rule scope from the sublist $(\alpha_{k+1}, \alpha_{k+2}, \ldots, \alpha_n)$, all these modifications would be recorded in the change set α_k instead. However, any modification to a rule scope in the sublist $(\alpha_1, \alpha_2, \ldots, \alpha_{k-1})$ would be recorded in the respective rule scope, whichever that may be, and not in the change set α_k. Depending on the kind of operations that are invoked by the user, change sets can eventually either be simply dropped or first applied and then dropped. For example, if change set α_k is not applied, then all the changes recorded in it will be lost (i.e., none of the elements in sublist $(\alpha_{k+1}, \alpha_{k+2}, \ldots, \alpha_n)$ will be changed). However, if change set α_k is applied before being dropped, then the following happens: let $\alpha_{k'-1}, k' > k$ be the next innermost change set after α_k. Then, all the changes in α_k that refer to a rule scope in the sublist $(\alpha_{k+1}, \alpha_{k+2}, \ldots, \alpha_{k'})$ will be applied to the respective rule scope, while all the changes in α_k that refer to a rule scope in the sublist $(\alpha_{k'+1}, \alpha_{k'+2}, \ldots, \alpha_n)$ will be copied to change set $\alpha_{k'}$.

Implementation-wise, change sets are represented as a pair of a set and a hash table. The set is used to store left hand sides of rules that have to be deleted, while the hash table is used for storing all other types of changes that can be made to a dynamic rule. Both the keys in the hash table and the entries in the set include an identifier for the rule scope to which the changes have to be applied.

If we go back to explaining how intersection works, we can detail now how the duplication of the rule set is realized. Let $RS = (\alpha_2, \alpha_3, \ldots, \alpha_n)$ be the rule set that is active before the intersection. When duplication of this rule set has to be done, we actually create two empty change sets (say $\alpha_{1'}$ and $\alpha_{1''}$). Then we obtain the two rule sets that are needed by prefixing RS with $\alpha_{1'}$ and $\alpha_{1''}$ respectively. Thus, we obtain $RS' = (\alpha_{1'}, \alpha_2, \ldots, \alpha_n)$ and $RS'' = (\alpha_{1''}, \alpha_2, \ldots, \alpha_n)$. Since we know that all changes to RS' will be committed to $\alpha_{1'}$ and all changes to RS'' will be committed to $\alpha_{1''}$, we can safely share $(\alpha_2, \alpha_3, \ldots, \alpha_n)$ between the two, thus avoiding any further duplication. Intersection of the two rule sets is achieved by simply intersecting the two change sets and committing the resulting change set to RS.

3.4 Strategies for non-sequential control flow

Our work in this area covers three distinct topics: library support for dealing with the break instruction, library support for dealing with the continue instruction and library support for dealing with the exception throwing/catching mechanism. For all three cases, we have only produced abstractions that can be used by forward propagation data-flow transformations. These are transformations that propagate information by analyzing a program from the beginning and advancing towards its end. The opposite type of transformations are backward propagation ones, which start the analysis at the end of a program and progress towards its beginning. We have also carried out some work in order to offer support for dealing with control flow interrupting statements in the context of backward propagation transformations, but this work was limited to supporting the break instruction. Moreover, our results in

this area should still be considered experimental, despite the encouraging successful results in our tests. We present our efforts in this direction in the last part of the section.

3.4.1 Support for the break instruction

The general semantics of the break statement in programming languages is that it interrupts the execution of some program structure (usually, but not necessarily, a looping structure) and instructs that execution should proceed with the first statement following that structure. Depending on context, a break instruction may or may not specify a label. A break that is not followed by a label is usually used to indicate that the execution of the innermost loop has to be interrupted. A break that is followed by a label indicates that the execution of the particular structure labeled with the specified label has to be interrupted. This can be either a loop or a sequence of statements. The code below shows these three patterns:

```
1  l1: {
2    ...
3    if (...) {
4      ...
5      break l1;
6    }
7    ...
8  }
9
10 l2: while (...) {
11   ...
12   if (...) {
13     break; // break l2 could have been
14   }        // used with a similar effect
15   while (...) {
16     ...
17     if (...) {
18       break;
19     } else {
20       break l2;
21     }
22     ...
23   }
24   ...
25 }
```

The break in line 5 is an example of breaking out of a sequence of labeled statements, the one in line 13 is an example of breaking out of the outer loop, the one in line 18 indicates breaking out of the inner loop (the while loop in lines 15–23) and the break in line 20 determines breaking out of the outer loop (the one labeled l2). All the break instructions in our example are part of an if-then or if-then-else structure because it makes no sense to have an unguarded break instruction. If that were the case, all the statements following it up to the specified point of interruption would be useless.

Constant propagation. Let us explain why special handling of the break statement is needed by considering a specific data-flow transformation: *constant propagation*. [7] shows how Stratego can be used to implement intraprocedural constant propagation that supports sequential flow, conditional flow and iterations without interruption of control flow for the Tiger language. Our test implementation is similar, except that it is tailored for the TIL language instead of the Tiger language. Conceptually, however, the two are identical. We

present the main strategy of this transformation below, along with support for while loops that do not contain any break statements (we do not actually check for this in the presented code; it is merely an assumption which, if broken, will determine misbehavior of the constant propagation transformation):

```
1   propconst =
2     PropConst
3     <+ propconst-declaration
4     <+ propconst-assign
5     <+ propconst-block
6     <+ propconst-if-then
7     <+ propconst-if-then-else
8     <+ propconst-while
9     <+ propconst-for
10    <+ all(propconst); try(EvalExp)
11
12  propconst-while =
13    ?While(_, _)
14    ; (/PropConst\* While(propconst, {| PropConst: map(propconst) |}))
```

The basic idea of the main `propconst` strategy is that it performs a generic traversal of the abstract syntax tree (line 10), taking specific actions at the nodes where this is necessary (in our code, variable declaration nodes, assignment nodes and so on). The action taken for the while node can be seen in lines 12–14. Line 13 matches for the while node and line 19 calls the fix point operator for the `PropConst` dynamic rule, using congruence applied to the while node as the strategy. A more detailed explanation (with examples) about how the constant propagation strategy works can be found in [7].

To see how constant propagation is supposed to work, consider the two code snippets below. The code on the left contains a while loop without any break statements inside it. The code on the right is similar to the one on the left, but with a couple of modifications among which the introduction of a break statement.

```
x := 0; z := 2;
while (file.available() > 0) do {
  if (y % 2 = 1) {
    x := x + 1;
  }
  y := file.read();
}
someCall(x, z);
```

```
x := 0; z := 2;
while (file.available() > 0) {
  if (y % 2 = 1) {
    x := x + 1;
  }
  else {
    z := 1;
    break;
  }
  y := file.read();
  z := 2;
}
someCall(x, z);
```

Let us first analyze the code on the left. Before the while loop, we have the following rule set for the dynamic rule `PropConst` (we are only interested in variables x and z): $\{x \rightarrow [0], z \rightarrow [2]\}$[5]. Running the fix point operator will have the effect of deleting the rule that rewrites x to 0. This is because during the first run of the strategy `While(const-prop, const-prop)`, as a result of the assignment `x = x + 1`, the rule $x \rightarrow [0]$ will be replaced

[5]We show right hand sides as lists because a left hand side can rewrite to multiple right hand sides. This is a general rule; note that it would make no sense to associate multiple right hand sides with the same variable in the context of the `PropConst` rule.

with the rule $x \rightarrow [1]$ *in the duplicated rule set* and the intersection of the two rule sets (the one containing the rule $x \rightarrow [0]$ and the one containing the rule $x \rightarrow [1]$) after the first run of the strategy will essentially yield that $x \rightarrow []$. The meaning is that x can no longer be propagated as a constant value after the loop. However, since nothing happens to z during the loop, the propagation rule for z will not be discarded. Hence, after the loop, the rule set for `PropConst` is $\{z \rightarrow [2]\}$, which will result in the transformation of `someCall(x, z)` to `someCall(x, 2)`.

Moving on to the code at the right, we will demonstrate that if the break statement is not handled in a special way, constant propagation will behave incorrectly. As earlier, the rule set for `PropConst` before the loop is $\{x \rightarrow [0], z \rightarrow [2]\}$. During one run of the strategy `While(const-prop, const-prop)`, the rewrite rule for z will go through a number of phases, but will always end up being $z \rightarrow [2]$ because of the assignment `z = 2` at the end of the loop. The intersection of that with the rule $z \rightarrow [2]$, valid before the loop, will result in $z \rightarrow [2]$. Consequently, the rule set active after the while loop will again be $\{z \rightarrow [2]\}$ (since the rule for x is dropped in exactly the same way), leading to the same transformation of `someCall(x, z)` to `someCall(x, 2)`. Unfortunately, in this case, the transformation is erroneous, since if the else branch of the if-then-else inside the while loop is taken, there will be a break of control flow, and the value of z that goes out of the loop is 1. Thus we have wrongly replaced z with 2, although its value after the loop could be 1 as well. We thus conclude that the correct rule set that should have been computed after the loop is $\{\}$, not $\{z \rightarrow [2]\}$. The culprit, as you suspect, is the wrong handling (or rather, the *non handling*) of the break statement.

Handling break statements. One way to handle the break statement in constant propagation would be to use no special library support and simply leave all administration to the user. Presumably, the user would have to somehow remember the values that are active before each break statement and manually merge them with the "default" ones that are valid after the structure that the break interrupts. Breaking to labels would complicate things even further, since then different sets of values would have to be merged in at different points. With enough effort, the code could be brought to a working state, but the process is likely to quickly discourage most users. Clearly, with no easy to use abstractions made available by the library, potential data-flow transformation writers are likely to have a very hard time supporting such interruptions of control flow.

Considering this, what if instead of making users put effort into writing complicated code using the wrong abstractions, we tried to come up with the right abstractions and then have them write elegant code that uses them? This, we believe, is a much more attractive prospect. Moreover, the same administration mechanism would no longer have to be reinvented and reimplemented by each separate user.

The kind of abstraction that we have come up with is one that allows the user to communicate to the library that a break statement has been encountered. When this happens, the library performs all the work behind the scenes, thus relieving the user from the administrative effort. Since such a call is dynamic rule specific, we found it most intuitive to automatically define a strategy, `break-`L, for each distinct dynamic rule L that appears in a program.[6] By default, we associate any call to `break-`L to the innermost run of a fix

[6] This is the same mechanism as the one defining strategies like `bagof-`L, `innermost-scope-`L or `new-`L for all dynamic rules L that appear in a program; see [7], [20] for details about what these and other similar

point operation. This means that we implicitly assume that the break of control flow is up to the statement immediately following the loop for which the innermost fix point operation was called.

However, this default handling does not cover all possibilities. As we have explained, most programming languages also allow breaking to labeled loops that enclose the innermost loop or breaking out of a labeled sequence of instructions. This requires further support from the library in order to allow a sensible labeling mechanism and to specify breaking to such labels. Thus we have introduced additional versions of the `dr-fix-and-intersect` and `dr-fix-and-union` strategies that take an extra label parameter and we have developed two new strategies, `dr-label-intersect` and `dr-label-union`, which function similarly to the previous two, except that they do not run a fix-point iteration algorithm, but simply apply the strategy they are passed only once. Also, we needed a version of the `break-`L strategy that takes a label as a term argument. In a similar fashion as for `break-`L, we automatically generate a new strategy called `break-to-label-`L which takes an additional label argument.

With these abstractions at hand, handling of break statements becomes trivial from a user perspective:

```
propconst =
  PropConst
  <+ ...
  <+ propconst-while
  <+ propconst-labeled-while
  <+ propconst-labeled-stm
  <+ propconst-break
  <+ propconst-break-label
  <+ ...

propconst-while =
  ?While(_, _)
  ; (/PropConst\* While(propconst, {| PropConst: map(propconst) |}))

propconst-labeled-while =
  Labeled(?label,
    dr-fix-and-intersect(
      While(propconst, {| PropConst: map(propconst) |})
      | ["PropConst"], label)
  )

propconst-labeled-stm =
  Labeled(?label,
    dr-label-intersect(propconst | ["PropConst"], label)
  )

propconst-break =
  ?Break(None)
  ; break-PropConst

propconst-break-label =
  ?Break(Some(label))
  ; break-to-label-PropConst(|label)
```

Notice that handling of a non-labeled while loop is unchanged, while that for a labeled

strategies do.

while loop simply implies a call to `dr-fix-and-intersect`[7]. Just as in the case of the non-labeled call, we pass the strategy to be run, the dynamic rule (or rules) involved and, in addition, the label. Labeled statements are handled in a similar fashion, with the exception of using `dr-label-intersect` instead of `dr-fix-and-intersect`. Thereafter, whenever we encounter a break we either call the `break-PropConst` strategy or the `break-to-label-PropConst` one, depending on whether or not we break to a label.

Writing our constant propagation transformation in this way is clean and, we hope, reasonably intuitive. It is clear that the user will no longer have a hard time writing data-flow transformations that take interruptions of control flow into consideration (given that similar abstractions are also provided for the continue statement and for exceptions, as we explain later in this section).

Implementation of support for break. Now that we have explained how the abstractions for handling break statements work from a user's point of view, we can dive into the more interesting part of how they are implemented. From an implementation point of view union and intersection of rule sets are simply two sides of the same coin. In fact, the library implements a generic mechanism that is parameterized with a merge strategy, and the intersect and union versions are simply wrappers around this generic strategy that pass the intersection and the union, respectively, as an actual parameter for the merge strategy. Therefore, whenever throughout this exposition we refer to merge, merging and the like, you should understand it in this context.

While the basic idea is essentially intuitive, a lot of weird quirks had to be put properly in place in order for the whole scheme to work. However, since most quirks relate to non-conceptual issues, we will focus on explaining the main idea. As we have described in the previous section, a fix point iteration (performed when looping structures are encountered in the analyzed code) works by creating a reference rule set and iteratively merging this with a rule set that is generated by running a user-specified strategy. This happens until merging no longer produces any changes. In case we have one or more break instructions in our loop, the result of this process has to be further modified so that it includes the effects of the additional control flow path(s) that are introduced by the break instruction(s). The way to do this is to merge the result of the basic fix point iteration (for easier reference, let us call this the *FPI rule set*) with one or more additional rule sets that each are valid right before a break instruction in the loop. So, instead of ending up with a rule set that only incorporates the effects of the sequential control flow paths through or around a loop, we now ensure that we end up with one that includes those effects as well as the ones of each control flow path that reaches a break instruction and then directly jumps out of the loop.

Normally, a break instruction is usually guarded by an if-then or if-then-else construct. If the path that ends with a break is taken, then execution will continue after the loop. If not, execution will proceed normally through the loop. The question that arises is how do we ensure that the the effects of the branch that ends with a break are not taken into consideration *within the loop*, *after the interrupted branch*. The solution is to make sure that we discard all the effects that are created by branches that end with a break. This sounds quite easy, but it was actually rather complicated to achieve within the Stratego

[7] In perspective, it would be even nicer to add syntactic sugar for calling `dr-fix-and-intersect`. One attempt could be `/label : rules\* s`. Such syntax could simply be desugared to `dr-fix-and-intersect(s | rules, label)`

model of representing rule sets.

The question of what happens when `break-`L or `break-to-label-`L are called is probably clear by now. First we have to save all the effects that have been created on the control flow path leading to the break instruction, *but* only those that were introduced by statements inspected *after* the fix point iteration began *or* after the beginning of the structure labeled with the label specified by `break-to-label-`L. Once these effects are saved, we have to make sure that they will not be excluded from further use *within* the innermost loop or labeled structure. They will only be picked up again and merged into the FPI rule set the innermost loop or labeled structure are exited.

What we have explained so far are the key points of this implementation. What follows is a more detailed description of the actual code doing this. Therefore, the reader can safely skip (and is advised to do so) to section 3.4.2, unless she is particularly interested in these details (and is already quite familiar with the implementation of dynamic rules).

For performance reasons, merging of rule sets in the dynamic rules library heavily relies on the use of change sets in order to reduce the effort of merging strictly to the bare minimum that is necessary. In principle, every time merging will be needed, two (or more) change sets are created in order to collect various changes independently, while the entire rule set up to that point is shared. We clearly wanted to apply the same principle in the case of the support for break. The problem with merging two or more change sets is that all of them have to be based on the same common rule set (or, in other words, to collect changes that apply to the exact same rule set) in order for the merging to be correct. This is not a problem with change sets that are created to deal with conditional structures or with change sets that are created to deal with iterative structures, because in each case we use exactly two change sets, both created at the same time, based on whatever rule set is active at that particular moment. In the case of break support, things get more difficult.

Change sets that are merged after a fix point iteration have to be based on the rule set that is active before we start the fix point iteration. This is not a problem for the FPI change set (the change set corresponding to the FPI rule set that we introduced above), since that one can be (and actually is) created when the fix point iteration begins, but it is a problem for change sets that have to collect all effects *from* the beginning of the fix point iteration *up to* the call to a `break-`L or `break-to-label-`L. This is a problem because a number of rule scopes and change sets may be introduced in the rule set between the two points mentioned above (beginning of fix point iteration and call to `break-`L), and we need to include all of their contents in the change set. But in order to be able to do that, we need to know exactly which rule scopes and change sets to include.

In order to be able to retrieve these rule scopes and change sets, we need to mark the position in the rule set right before the fix point iteration begins. We anticipate a bit here by mentioning that this marking has to be done both for (labeled or non-labeled) fix point iterations (what we were discussing) *and* for simple labeled structures, based on the same principle. In fact, we treat both cases together by simply performing this marking only in `dr-label`[8] and making sure that `dr-fix-and-merge`[9] wraps its own execution in a call to `dr-label`. Marking this position has to be done separately for each of the dynamic rules that are specified in the call to `dr-fix-and-merge` or `dr-label`. With this mechanism,

[8] `dr-label` is the generic implementation on which both `dr-label-intersect` and `dr-label-union` are based.

[9] `dr-fix-and-merge` is the generic implementation of the fix point iteration

every call to `dr-fix-and-merge` or `dr-label` introduces a new marking for a label (or the implicit, empty string, label in the case of non-labeled loops) that we can later use in calls to `break-`L or `break-to-label-`L in order to generate the correct change sets. Remember, the markings tell us the position in a rule set from where we need to gather all the effects and put them all in a single change set.

In this context, here is how a call to `break-`L or `break-to-label-`L functions: first, we retrieve the marking for the appropriate label (implicit or specified), we combine all the effects in the rule scopes and change sets starting from the marking into a single change set, we save this change set for later retrieval and, finally, we mark the the latest change set (i.e., not the one we just created, but the regular, last one in the current rule set) as ignored. We have to do this in order for its effects to no longer be considered active within the current structure. There are two things that can happen to a change set that is marked as ignored:

- it is (eventually) merged with a change set that is not marked as ignored (this is the case when one branch of an if-then-else ends with a break and the other one does not *or* when we only have an if-then) and in this case the merge will result in a copy of the change set that is not marked as ignored, and the ignored change set will simply be discarded;
- it is (eventually) merged with a change set that is also marked as ignored (this is the case when both branches of an if-then-else end with a break) and in this case the merge will randomly return one of the two change sets and discard the other. In this situation (after the merge) we end up committing a change set that is marked as ignored, which will have the effect of marking the next change set in the rule set as ignored. This behavior makes sense because if both branches of an if-then-else end with a break, it is similar (from an ignoring change sets point of view) with having a break after the if-then-else. Sometimes, propagating the "ignored" marking upwards may eventually lead to marking the first change set of the structure (loop or labeled sequence of statements) that we are currently dealing with as ignored. This translates to the fact that all possible paths out of that structure end with a break, so we can simply ignore this change set when performing the final merge after the structure (given that we will never exit the loop in a "normal" way). This final merge is the one where we are introducing the saved effects of all the paths that end with a break. So, instead of merging all these effects into the FPI rule set, we drop the FPI rule set altogether and replace it with the result of the merge of all the saved rule sets that correspond to control flow paths ending with a break.

The final merging itself is straightforward: we simply retrieve all the corresponding[10] change sets generated and saved as an effect of calls to `break-`L or `break-to-label-`L and merge those with the FPI change set. The resulting change set will be committed to the main rule set afterwards. There is one more particular case here, when there are no saved change sets to retrieve, and the FPI change set is marked as ignored. This situation occurs when we have all paths in a loop or labeled structure ending with a break *and* none of those breaks breaking out of this loop or labeled structure, but out of an enclosing one. In this case, we propagate the ignored marking upwards to the next change set, as usual, just as if we had

[10] This merge is performed in the `dr-label` strategy which, among other parameters, takes a label as well. When we retrieve the change sets, we need to only retrieve those that were associated with the particular label passed to `dr-label`.

had a break following the entire loop or labeled structure.

3.4.2 Support for the continue instruction

The continue instruction is in many ways related to break, therefore we will be presenting it by building on the similarities and pointing out the differences. Thus, it is advisable that you read the previous section before attending to this one. There are two difference that distinguish continue from break. The first one is that continue indicates just the interruption of the *current iteration* of a loop, not of the loop altogether. Hence, execution resumes back at the beginning of the loop, not after it. The second one is that continue can only be used within a loop, and not also within a labeled structure (as was the case for break). This is due to the fact that iterations do not make sense in the context of a labeled structure. Just like break, continue may or may not be followed by a label. A label or the absence thereof is interpreted as in the case of the break instruction. The following example illustrates:

```
1  l1: while (...) {
2    ...
3    if (...) {
4      continue; // continue l1 could have been
5    }           // used with a similar effect
6    while (...) {
7      ...
8      if (...) {
9       continue;
10     } else {
11       continue l1;
12     }
13     ...
14   }
15   ...
16 }
```

The continue in line 4 will determine the execution to jump to line 1. The same holds for the continue in line 11 (which is an example of continuing to a label that specifies a loop other than the innermost one). Finally, the continue in line 9 causes the execution to resume at line 6.

Handling continue statements in constant propagation. We go back to our constant propagation example in order to explain how the abstractions for continue are to be used. Just like break, continue has to be handled explicitly in the constant propagation transformation. Otherwise, we end up propagating wrong information due to our ignoring of the additional control flow paths introduced by the continue statements.

The abstractions for continue as well as their use is virtually identical to the break case. We automatically make available two strategies for each dynamic rule, `continue-`L and `continue-to-label-`L, that are designed to be used for indicating the presence of a simple continue or of a continue to a label, respectively. These two strategies also have to be used within the context of a call to `dr-fix-and-intersect` or `dr-fix-and-union`, either with or without a label argument. This is because `dr-fix-and-merge` and `continue-[to-label-]`L work hand in hand. The former needs to put the proper mechanism in place so that the latter can perform its action. The reverse is also true, namely that the former uses the effects of the latter in order to modify its behavior.

Including support for continue in our constant propagation strategy reduces to the two extra calls that we have to make to `continue-`L and `continue-to-label-`L whenever we encounter a continue or a continue to label. Everything else (particularly the handling of labeled and non-labeled loops) stays the same.

```
propconst =
  PropConst
  <+ ...
  <+ propconst-continue
  <+ propconst-continue-label
  <+ ...

propconst-continue =
  ?Continue(None)
  ; continue-PropConst

propconst-continue-label =
  ?Continue(Some(label))
  ; continue-to-label-PropConst(|label)
```

Implementation of support for continue. From an implementation perspective, the support for continue still bears some resemblance to the one for break, but here the differences start to show. The main distinction is triggered by the fact that now we need to incorporate the effects collected by each call to `continue-`L or `continue-to-label-`L at the beginning of each fix point iteration, not just at the end of it. This is because we need to integrate those effects in the computation of the fix point of the rule set itself, and not first compute a fix point of the rule set which does not include those effects and then perform a final merge between them (as we are doing in the case of break). The reasoning behind that lies in the semantics of the continue instruction: it determines the program to jump back at the beginning of the loop. Therefore, effects obtained in iteration $i - 1$ are in place at the beginning of iteration i, if the path leading to a continue is taken. So, we need to make sure the the rule set that we use throughout the loop includes all the effects of paths that end with a continue as well. We did not have to do this in the case of break because there we knew that if a path leading to a break was taken, then the loop would no longer be run, so the effects active before the break only had to be considered for what came after the loop.

This main aspect we have just discussed is in fact entirely encoded in the implementation of `dr-fix-and-merge`, since it is there where the fix point iteration is performed, so it is also there where the incorporation of the additional effects generated on paths leading to continue's has to take place. Because of this, the implementation of `continue-`L and `continue-to-label-`L does not differ too much from that of `break-`L and `break-to-label-`L. In fact, the only difference is that we need to save the effects under different headings so that we are later able to tell apart those that were generated by break statements and those generated by continue statements.

Just as we did for break, we also need to take care that the right rule sets are used for transforming the code that follows the continue statement(s), and is still within the loop. More clearly, a continue will introduce effects that need to be considered at the beginning of a loop, but they still do not have to be considered in the part of the loop that follows that continue. As with break, we expect that a continue statement will be guarded by some

condition, so that it makes sense to have some code following it[11]. The solution is similar to the one we used with break: we cancel the effects introduced in the branch leading to a continue so that they are no longer active in the code that follows that branch.

The discussion continues with some details that will probably only be interesting to somebody looking to understand, change or continue the implementation itself. Otherwise, you should probably skip to the next section.

The entire process is similar with that for break, but there is one aspect that requires explaining. If you remember from the previous section, when a break was encountered, we marked the change set that was active at that point as ignored. We do the same thing for the case of continue. However, we make a distinction between the two types of ignore markings for a change set. Of course, there is only one way one can ignore effects (i.e., one drops them), but nevertheless it is important for us to be able to tell not just that a change set is ignored but also why it is ignored. Hence, the two different ignore markings.

This reason is relevant only for the cases when an ignore marking is propagated[12] up to the level of the very first change set that is created by a fix point iteration (the one we used to call the FPI change set in the previous section). It is there that the information about the reason why we are ignoring the change set becomes useful. We will proceed to discussing the implications, but before that, a word of advice: please bear in mind that throughout the remainder of this section we are only talking about loops in which *all possible paths through the loop end either with a break or with a continue*. This is the only situation in which the FPI change set will end up being marked as ignored.

If the change set is ignored due exclusively to breaks, then we can cut the fix point iteration short since we know that there will never be more than one run of the loop when the program is executed.[13] This is because a change set that is ignored exclusively due to breaks indicates that all paths through the loop end with a break, so whichever of them is taken, the loop will be interrupted. On the other hand, a change set may be ignored due to continues. In this case, we still need to continue the fix point iteration, but we will handle the merge of the FPI rule set and the effects saved as a result of calls to `continue-`L or `continue-to-label-`L differently. In particular, we need to ignore the FPI change set when we perform this merge, and only consider the effects collected on the paths that end with a continue.

You might have noticed a subtle distinction above between "ignored *exclusively* due to breaks" and "ignored due to continues". The reality behind this is that it is enough to have a single path that ends with a continue in a loop and even if all the others end with a break, we still have to maintain the "softer" ignore of the change set, i.e. that due to the continue. Even if $n-1$ paths through the loop end with a break and just 1 with a continue, we still cannot cut the fix point iteration short. In order to accommodate this, whenever we perform a merge in which one of the involved change sets is marked as ignored because of a break and the other one is marked as ignored because of a continue, the resulting

[11] Of course, there is nothing wrong with having no code after a (guarded) continue statement in a loop. We simply do not have a problem in that case. What we are discussing is the case when there is some code following the continue statement *and* that code stands a chance at being run, i.e., it is not in the same block as the continue, but at a lower nesting level.

[12] See the explanation for break to find out how and when this propagation occurs.

[13] Please mark the distinction between the fix point iteration, which is the one written in Stratego and performed in the body of `dr-fix-and-merge` and the run of the loop, which refers to the loop written in the object language.

change set will end up being marked as ignored due to a continue. Hence, we propagate the "softer" ignore, to keep things correct.

3.4.3 Support for exceptions

While handling of break and continue shares many similar aspects, exceptions need a completely different set of abstractions. Since the details of the exception throwing/catching mechanism differ slightly from one language to another (for instance, the C/C++ one does not have the equivalent of Java's finally clause), we will focus our presentation on the semantics of the Java variant. Actually, not just the presentation, but the library support for exceptions itself is built on the semantics of exceptions in Java. Thus, we will begin with an overview based on the language specification [11].

Exceptions in Java The mechanism for exception handling in Java is composed of a statement (`throw`) for throwing exceptions and a construct (try-catch-finally) for catching them. We say that a catch clause catches an exception if the declared type of the clause is either the type of the exception or a supertype thereof. try-catch constructs can be nested and the basic semantics is that an exception thrown in the body of a try block can be caught by any catch clause associated with it or by one associated with a try block that encloses the original one. A try-catch construct can also optionally contain a finally clause. The code specified by the finally block is always executed, regardless of whether an exception is thrown or not. Of the two types of exceptions that exist in Java, *checked* and *unchecked*, it is only required that the former are caught by some catch clause. The latter may be caught as well, but it does not result in a compile-time error not to do so. Unchecked exceptions arise due to a violation of the Java semantics by the evaluation of an expression (e.g., ArrayIndexOutOfBoundsException, NullPointerException or ArithmeticException), due to an error in loading or linking classes (e.g., ClassNotFoundException), due to unavailability of resources (e.g., OutOfMemoryError) or due to internal errors of the virtual machine. All unchecked exceptions have to be instances of subclasses of `RuntimeException` or `Error`. Everything else (conventionally, instances of subclasses of `Exception`, but not of `RuntimeException`) is a checked exception. All the checked exceptions that a method can throw have to be specified in the signature of the method; overriding methods cannot add exceptions to the list of exceptions of the overridden method.

Of course, what we are particularly interested in from all this are the control flow implications of the exception throwing/catching mechanism, thus we explain it here in a bit more detail. An exception thrown by a `throw` statement determines the abrupt interruption of the execution of the immediately enclosing try block. If one of the catch clauses of the try block declares that it catches an exception of the same type as that of the one thrown (or a supertype thereof), then execution proceeds with the contents of that particular catch block. If this is not the case, the exception is propagated to the next innermost enclosing try block, even across method calls, and the same process is repeated. If no catch clause catches the exception, the method `uncaughtException` of either the uncaught exception handler of the thread (if one is set) or the `ThreadGroup` the thread is part of is called.[14] Before the exception is propagated up from any try block along the way, the code in the

[14] This can only happen with unchecked exceptions. Checked exceptions are always guaranteed to be caught by a catch clause since otherwise the program would not have compiled successfully.

optional finally clause of that try-catch is first executed. If this code itself throws an exception, then the propagation of the original exception is stopped, and the newly thrown exception is propagated from there on.

This complex mechanism raises several issues:

- exceptions can be thrown anywhere in the body of a try. If for checked exceptions and some unchecked exceptions an analysis of the code in the body of the try could be made to see exactly which statements might throw an exception, for certain unchecked exceptions (e.g., internal Java Virtual Machine errors) this is impossible. Even assuming we disregard the latter, we will still have a difficult time figuring out which statements can throw exceptions and what exceptions they throw, since exceptions that are instances of subclasses of `RuntimeException` do not have to be explicitly declared in the `throws` clause of a method declaration.
- an exception thrown in the body of a try can be caught by a catch clause corresponding to try-catch higher up in the nesting hierarchy.
- if we have finally clauses, control flow is even more complicated, since we could potentially execute some of the statements of a try block, then those in the finally block, and finally those of a catch block some levels higher in the nesting hierarchy (and even a number of extra finally blocks along the way). Moreover, if any code in one of the finally clauses throws an exception, then control flow will continue with the catch clause that catches this new exception.
- if in the case of break and continue, interruption of control flow could not be interprocedural (or intermethod), in the case of exceptions it can, complicating things even further.
- the structure of the program is not enough to be able to tell where control flow resumes after the throwing of an exception. One needs type information as well to be able to identify the catch clause that catches a certain exception. Since we do not want to mix dynamic library code with type inference code, it must be left to the user to figure out what catch clause matches a particular `throw` statement.

Handling exceptions in constant propagation. As you have probably already got used to by now, we proceed with our explanation in a top-down fashion, starting from showing how the abstractions we devised can be used and only thereafter explaining how they have been implemented. We continue our running example of the constant propagation transformation as our proof-of-concept implementation that employs the new abstractions.

The support for exceptions has to allow the user to perform transformations over try-catch-finally structures in such a way that every additional control flow path introduced by exceptions is implicitly considered. In the case of break and continue all we had to do was merge, at the proper program points, additional rule sets that collected the effects active right before a break or continue. Here, we have to do the same thing for `throw` statements and, in addition, prepare the right environment before each catch clause, before the finally clause as well as after a try-catch or try-catch-finally block. This is complicated due to the fact that many different rule sets have to be save, restored and combined at various program points.

To illustrate, we will just dive into one example. The rule set active at the end of a try block has to be saved in order to be reinstated at the beginning of the finally block (if there is one) or after the try-catch block (if there is no finally block). However, in each of these cases, the rule set will not simply have to be reinstated, but also mixed with the rule sets active at the end of each separate catch clause in that try-catch block (and potentially even more rule sets active before `throw` statements that throw exceptions that are only caught by a catch clause higher up in the nesting hierarchy). This complicated scenario is caused by the fact that (1) if the try block executes without throwing exceptions, execution has to proceed with the finally block, (2) if there is an exception that is caught by one of the catch clauses, then the catch clause is first executed, then the finally clause and (3) if there is an exception that is not caught by any of the catch clauses in this try-catch-finally block, then the finally clause if directly executed, and then the exception is thrown on. Hence, all the rule sets for the three cases have to be mixed together into a single rule set that will be instated at the beginning of the finally block.

A number of strategies that we detail below have been created in order to support this and a number of other scenarios.

- `dr-init-exception-block(has-finalize | catch-tags, rulenames)`: this strategy has to be called by the user at the beginning of a try-catch[-finally] block in order to allow the library to set up some internal structures. `has-finalize` encodes whether there is a finally block associated with the try-catch block, `catch-tags` is a list of strings (presumably, but not necessarily, the type names that catch clauses declare) and `rulenames` is the list of rulenames involved in the exception handling mechanism.
- `throw-`L`(eq | tag)`: this (automatically generated) strategy is similar in function to our `break-`L and `continue-`L strategies. It is meant to be used when `throw` statements are encountered in the code in order to signal this to the library. The `tag` will be compared with the ones passed as part of the `catch-tags` list to `dr-init-exception-block`. The comparison will be performed using the `eq` strategy which is passed to `throw-`L.

  You can see this as the equivalent mechanism of labels that we used for break and continue. There, we had a label passed to `dr-label` or to `dr-fix-and-merge` and a label passed to `break-to-label-`L or `continue-to-label-`L. The latter would be matched (by simple equality) against the list of all the labels declared up to that point. Here we have a similar mechanism, except that each level can declare any number of tags (since there can be any number of catch clauses associated with a try block) and matching is customizable. This last difference is introduced to accommodate the fact that a catch will match exceptions that have the same type as *or a subtype* of the type declared by the catch clause. `eq` should take a pair in which the first element will be the throw tag and the second will be the catch tag. If a subtype relationship has to be tested for, then a strategy like `subtype-of` could be passed.
- `dr-complete-catch-(intersect|union)(s | rulenames)`: instead of running a strategy (that presumably performs some transformation) over each catch clause directly, the strategy should instead be wrapped in a call to `dr-complete-catch`. This will ensure that all the administration code will be run before and after the transformation proper. An important aspect from a user perspective is that the order in which `dr-complete-catch` should be called over catch clauses has to be the same order

in which the catch tags were passed to `dr-init-exception-block`. The easiest and most natural way to achieve this is by simply using the proper program order in which the catch clauses appear. Whether the intersect or union version has to be used depends on the semantics of the transformation.

- `dr-complete-finally-(intersect|union)(s | rulenames)`: this strategy is designed to serve the same purpose for the finally block that the previous one serves for the catch blocks. `dr-complete-finally` should be called after `dr-complete-catch` has been called for all catch blocks.

- `dr-complete-exception-block-(intersect|union)(|rulenames)`: this is the strategy that ends the scheme for dealing with exceptions. Its purpose is to perform cleanup of the administrative structures and generate the proper rule set that will be used after the entire try-catch[-finally] block.

We provide underneath a typical example of how these strategies can be employed in order to manage exception throwing and catching in a program. With support for exceptions, our intra-procedural version of the constant propagation transformation is finally complete.

```
propconst =
  PropConst
  <+ ...
  <+ propconst-throw
  <+ propconst-try-catch
  <+ propconst-try-catch-finally
  <+ ...

propconst-throw =
  ?Throw(TypeName(t))
  ; throw-PropConst(eq | t)

propconst-try-catch =
  ?Try(tblock, cclauses)
  ; dr-init-exception-block(fail
                          | <get-catch-tags>cclauses, ["PropConst"])
  ; <propconst>tblock => tblock'
  ; <map(
      dr-complete-catch-intersect(propconst | ["PropConst"])
    )>cclauses => cclauses'
  ; dr-complete-exception-block-intersect(|["PropConst"])
  ; !Try(tblock', cclauses')

propconst-try-catch-finally =
  ?Try(tblock, cclauses, fblock)
  ; dr-init-exception-block(id
                          | <get-catch-tags>cclauses, ["PropConst"])
  ; <propconst>tblock => tblock'
  ; <map(
      dr-complete-catch-intersect(propconst | ["PropConst"])
    )>cclauses => cclauses'
  ; <dr-complete-finally-intersect(propconst
                                 | ["PropConst"])>fblock => fblock'
  ; dr-complete-exception-block-intersect(|["PropConst"])
  ; !Try(tblock', cclauses', fblock')

get-catch-tags =
  map(?Catch(_, <id>, _))
```

Notice the sequence in which the strategies we presented above are called in the example; first we initialize the structures, then we run the main transformation over the try block, then we run it (wrapped in `dr-complete-catch`) over each catch clause in turn, then, if we have one, over the finally clause (this time wrapped in `dr-complete-finally`) and, in the end, call the cleanup procedure. Also notice our use of the types declared in the catch clauses as catch tags. Our handling of throw statements is straightforward: we extract the type of the exception thrown and use that as the tag to pass to `throw-PropConst`. We use sheer equality to compare tags; in a real-language scenario this would have to be replaced with a strategy that would decide if one type is a subtype of another. Also, you may have noticed that in our TIL language, we do not throw exception objects, but exception classes. This is a simplification which allows us not to require a type annotation that would otherwise be necessary. In a more realistic context, you could imagine the line `?Throw(TypeName(t))` replaced with something like `?Throw(_{TypeName(t)})`. For this to work, a type annotation strategy would first have to be run in order to annotate expressions with their type.

Implementation of support for exceptions. We will base our discussion of the implementation on two examples. The first one will be very simple and straightforward, in order to help us cover the basics, while the second one will be rather complicated, so that it gives us the chance to point out a number of scenarios that all have to be covered by our implementation. You should regard the code in both these examples as only the skeleton of what would normally be found in reality. Thus, we emphasize strictly the structures and statements that are involved in exception handling. You could imagine that there are gaps virtually everywhere in between the lines of the examples which would normally be filled with a lot of additional statements, structures and so on. In fact, if you do not imagine such additional code present, all the rule sets that we will be mentioning would be empty, so there would not be much use to our effort. We have not included such code ourselves in order to avoid having the reader lose focus of what we are explaining.

Our first example is a typical try-catch-finally structure with a few `throw` statements. It is a skeleton of code that is very commonly used in practice. Since this is a smaller example, we have also illustrated with ... all the gaps where there would normally be additional code. These do not only stand for statements, but also for conditional or repetitive structures that could enclose the represented code. If you recall, in TIL we throw exception classes, not exception objects.

```
try {1
  ...
  if (...) then ... 2 throw E1; end
  ...
  if (...) then ... 3 throw E2; end
  ...
  if (...) then ... 4 throw E1; end
  ...
5}
catch (e: E1) {6
  ...
7}
catch (e: E2) {8
```

```
  ...
9 }
finally { 10
  ...
11 }
```

We begin running a transformation over the try block with a random rule set, (1). This rule set gets changed as the analysis proceeds, depending on what the user-defined strategy does. Three different rule sets (2), (3) and (4) will be active before the three `throw` statements and saved as a result of calls to `throw-`L. Rule set (5) will be active at the end of the try block. (6), the rule set that has to be instated at the beginning of the first catch clause is a merge between rule sets (2) and (4). This will suffer some changes as the analysis runs over the catch clause and become rule set (7) at the end of the clause. Similarly, rule set (8) will be equal to rule set (3) and will turn into rule set (9) by the end of the second catch clause. The rule set active at the beginning of the finally clause is a merge of rule sets (5), (7) and (9) and will become rule set (11) by the end of the finally block. This last one will also be the rule set active after the try-catch-finally block.

Let us now proceed to our second example, which, as you are about to see, illustrates additional difficulties. The code consists of four nested try-catch and try-catch-finally structures that are full of exception throwing statements. The exceptions are caught either at the same or at some higher level. We will only point out a number of special cases, given that the more basic administration is similar to that explained in our previous example.

```
try {
  try {
    if (...) then 1 throw E1; end
    if (...) then 2 throw E3; end
    try {
      if (...) then 3 throw E1; end
      if (...) then 4 throw E2; end
      if (...) then 5 throw E3; end
      try {
        if (...) then 6 throw E1; end
        if (...) then 7 throw E1; end
        if (...) then 8 throw E2; end
        if (...) then 9 throw E3; end
      10 }
      catch (e: E1) { 11
        if (...) then 12 throw E2; end
      13 }
      finally { 14
        if (...) then 15 throw E2; end
        if (...) then 16 throw E3; end
      17 }
    18 }
    catch (e: E2) { 19
```

```
      if (...) then 20 throw E1; end
      if (...) then 21 throw E3; end
    22 }
  23 }
  finally { 24
    if (...) then 25 throw E3; end
  26 }
27 }
catch (e: E1) { 28 ... 29 }
catch (e: E3) { 30 ... 31 }
finally { 32 ... 33 }
```

Here are the more interesting cases encountered in the code above:

- as a basic case, rule set (11) will be a merge of rule sets (6) and (7).
- rule set (14) will be a merge of rule sets (10), (13), (8), (9) and (12). (10) and (13) are the rule sets at the end of the red try and catch blocks, respectively, while the others are all rule sets active before the throwing of an exception that is caught by an outer catch clause. We discussed before why these need to be included in the computation of the rule set for this finally.
- rule set (19) will be a merge of rule sets (4), (15) and (17). While (4) and (15) are straightforward, (17) is a bit more interesting. If the finally clause for the red try-catch-finally had not existed, then (19) would have been a merge of (4), (15) and (8). However, since the finally clause does exist, if the exception E2 is thrown in the red try, then execution will continue with the code in the red finally clause and only then will it reach the green catch clause we are discussing now. Hence, we merge rule sets (4) and (15) not with rule set (8), but with rule set (17) in order to obtain a valid rule set (19).
- rule set (24) is obtained by merging rule sets (23), (1), (2), (3), (5), (16), (17), (20) and (21). (23) is the rule set at the end of the try block. All the others are rule sets active before throwing exceptions that will be caught by an outer catch clause ((1), (2), (3), (5), (16), (20) and (21)). The only exception is again (17), which (by a similar rationale as we made at the previous point) now appears as the transformation that rule set (9) has suffered before reaching this point.
- we compute rule set (28) as simply equal to rule set (26). If the blue finally had not been there, then rule set (28) would have resulted as a merge of rule sets (1), (3) and (20), but given that the blue finally is actually there, all these rule sets are collected at the beginning of the finally, and rule set (26) - that at the end of the finally - will be a replacement for all of them.
- rule set (30) is achieved by merging rule set (26) (in this case, replacing rule sets (2), (5) and (17), with (17) being in turn the replacement of the original (9)) with rule set (25).
- finally, rule set (32) is the merge of rule sets (27), (29) and (31). Here we no longer have rule sets corresponding to thrown exceptions, since they have all been caught before having reached the black finally.

In very, very few words, the implementation of the support for exceptions can be described as the code that makes sure that all these rule sets are saved, computed (by properly merging everything that has to be merged) and put in place at all the right program points. You can imagine that trying to explain how all this works down to the last detail would result in chaos, so we will only attempt to provide the reader with some guidelines to reading the code herself.

Perhaps the most important aspect that needs to be clarified for a potential reader of our code is how we manage the correspondence between the many rule sets that are saved and the points in the program where they have to be retrieved. In order to do this, we make use of a structure that mimics that of the nesting of try-catch and try-catch-finally structures. For easier reference, let us convene to call this data structure the exception stack. The exception stack is a list of entries that look like this: `TryBlock(pos, id, ctags, ftag)`. We add a new entry for each nested try-catch or try-catch-finally that is encountered. The `pos` stores the position in the active rule set of the very first change set created after the try-catch structure has been encountered. This position is needed for reasons that have already been explained when discussing the support for break. The `id` is a unique identifier that acts as a label for the entire try-catch[-finally] structure. We will discuss its use further below. `ctags` is a list of catch tags and `ftag` is either `Finally(id)`, if we have a finally clause, or `Finally("")`, if we do not. Again, the `id` is a unique identifier that functions as a label for the finally block. Returning to `ctags`, this is a list of terms of this kind: `Catch(id, tag)`. The `tag` is the user-specified tag, while `id` is an internal unique identifier, similar to the ones for the entire structure and for finally blocks.

The exception stack functions both as a labeling system and as a provider of structural information regarding the nesting of try-catch structures. The main purpose of `dr-init-exception-block` is to initialize and add the proper entries to the stack when the analysis encounters a new try-catch structure.

The behavior of `throw-`L is then quite similar to that of `break-`L, but adapted to the peculiarities of the exception handling mechanism. Thus, `throw-`L will need a more complex mechanism for determining the identifier with which the saved change set[15] has to be associated. This identifier will be one of the following two, whichever comes first in our exception stack, starting from the last entry: (1) the identifier of the catch tag that matches the tag passed to `throw-`L (using the user-supplied comparison strategy as match operator) or (2) the identifier of the first finally clause (`Finally("")` does not count, since its meaning is that there is no finally clause for a particular try-catch structure). This is in accordance with the control flow: if a finally clause comes before the catch clause that matches the exception, then execution will proceed with the code in the finally clause. However, even if a finally clause comes first, we still retrieve the identifier of the catch clause that will eventually catch the exception and save that along with the generated change set. We need it in order to allow that the change set (with some additional modifications introduced by the analysis of intervening finally clauses) will eventually be associated with the proper catch clause.

`dr-complete-catch` will look at the first catch tag of the last entry in the exception stack,

[15] Incidentally, the position (`pos`) saved in our exception stack is used to compute this change set so that it can be merged correctly at a point in a program where a different base rule set is active. The position is essentially a marker that separates that base rule set from the part that was added to it during the handling of the try-catch structure.

retrieve all the change sets associated with it, merge the rule sets obtained by appending the retrieved change sets to the base rule set and set the result as the active rule set. Then the user-provided strategy is run. Afterwards, `dr-complete-catch` will save the resulting rule set (because all the rule sets at the end of catch clauses have to be merged together with the one at the end of the try block later on, either at the beginning of the finally clause, if there is one, or at the end of the entire try-catch structure), remove the first catch tag from the last entry of the exception stack and reinstate the rule set that was active before the beginning of the strategy.

`dr-complete-finally` will look in the exception stack at the id for finally (we assume the strategy is only called if we have a finally block), retrieve all the change sets associated with this id, retrieve all the saved rule sets at the end of the try block and each of the catch blocks and merge everything together to obtain the rule set to be instated. With this resulting rule set in place, just like in the case of `dr-complete-catch`, the user-provided strategy will be run. Thereafter, modifications in the exception stack and in the set of saved change sets are made so that it looks like a finally clause had never existed. We do this so that we can provide a single `dr-complete-exception-block` strategy that behaves in the same fashion regardless of whether we have a try-catch or a try-catch-finally block.

The last part of `dr-complete-finally` is a bit more interesting. Exceptions that have to be caught by an outer catch, if thrown, will trigger the execution of the code in the finally block. Because of that, in `throw-`L, we associate the change sets active before the `throw` statements that throw such exceptions with the id of the finally block. If you remember, we retrieved and used those those change sets in the merge at the beginning of `dr-complete-finally`. However, we also have to ensure that the throwing of these exceptions is properly propagated upwards. Hence, we have to act just as if we had separate `throw some-exception;` statements at the end of the finally block for all the exceptions thrown up to this point and not caught yet. So, in effect, we perform the same process that takes place in `throw-`L (and that we have already explained) for all these distinct exceptions as the last part of `dr-complete-finally`.

The last one of the library strategies that deal with exceptions was `dr-complete-exception-block`. The job of this one is fairly easy. It will simply have to drop the last entry in our exception stack (since the handling of the corresponding try-catch[-finally] block is over) and prepare the rule set that has to be active at this program point. This rule set is a merge of the rule sets at the end of the try block (for the case when execution of the program carries out without exceptions being thrown) and those at the end of every catch block in the current try-catch structure (for the case when an exception has been thrown and caught by one of them and thus triggered the execution of the code in that catch clause). In case we have been dealing with a try-catch-finally block, then `dr-complete-finally` will have already done this merge and replaced all those rule sets with a single one – the one that resulted at the end of running the user strategy over the finally block. In this case, the job is already done, so nothing more happens.

As a final note, we should mention that management of rule sets used after `throw` statements follows the same pattern that we used for break and continue. The change set that was active before the call to `throw-`L will be marked as ignored and discarded as soon as this makes sense. This is because a `throw` statement also determines an interruption of control flow, so we know that if execution reaches the code *after* a `throw`, then it is sure that the innermost branch leading to the throw *has not been taken*. As such, we can safely

discard the effects introduced by this branch.

We have a special case related to ignoring of change sets due to `throw` statements.[16] If an ignore marking reaches the very first change set that was set as active before running the user strategy over the finally block, it means that all paths through the finally block end with a `throw` statement. What this amounts to in reality is the the throwing of any exception from within the try-catch-finally structure to which the finally block in discussion pertains will be surely interrupted and replaced by one or another of the exceptions thrown within the finally block itself. If this is the case, we make sure to no longer run the code that simulates `throw-`L's at the end of `dr-complete-finally`, since the exceptions for which that step usually takes place would in this case never make it past the current finally block.

3.4.4 Support for break in backward propagation transformations

Backward propagation transformations differ from forward propagation ones in that they progress against the natural control flow. This makes all the strategies we described earlier in this section inapplicable. In short, all these strategies imply saving a certain already achieved state of a rule set and remerging it at a later point in a program. In the case of backwards propagation, it makes no sense to save an already achieved state before, say, a break statement, because this state is created by statements that follow that break in a loop and which actually never get to be executed, should the break statement be reached in the normal flow. What we need to do instead is restore a state that was active right before statements that would no longer be executed due to the break are analyzed.

Consider the following loop:

```
while (...) {
  ...
  if (...) then break;
  ...
}
...
```

If execution reaches the break statement, the code that would be executed after that is the one following the loop. If we look at this in reverse, the same execution path should be followed: code leading to the end of the loop, then code preceding the break statement. In terms of rule sets, this means that the effect of a call to a library strategy that handles break statements for backwards analysis should restore the rule set that was active before beginning to analyze the while loop. The automatically generated strategy name for triggering this behavior is `break-bp-`L (L stands for the name of the dynamic rule for which the rule set is to be changed). Just like `break-`L, `break-bp-`L is only a proxy for the real library strategy.

So this is what has to happen, in principle. However, many times backward propagation transformations are not entirely backwards. It often is the case that some smaller or larger part of the analysis is still forward. The problem this incurs is that the changes introduced by the part of the analysis pertaining to the forward propagation *have to be kept intact*. So we have a series of changes that have to be reverted so that we end up with the state of

[16] If you are having trouble following this discussion, please have a look at the sections about break and continue first. We assume knowledge introduced there.

the rule set before backwards analysis over the loop began, but we also have some changes to the rule set that were made after analysis over the loop began that nevertheless have to be maintained active.

To illustrate, consider the dead code elimination transformation described in [7]. This is essentially a backwards propagation transformation that includes a small forward analysis part when examining sequences of statements. Whenever a sequence of statements is encountered, the sequence is first traversed forwardly in order to detect all the variable declaration statements. Each such declaration introduces a new rule scope for reasons that make sense in the context of dead code elimination. Apart from creating such rule scopes, also during forward analysis, entries are added to those rule scopes in order to create a new state for the analysis. These entries added during forward analysis, even though added after analysis of a loop has begun, still have to be active kept after a call to `break-bp-`L.

Presented with this issue, we had to come up with a little trick that would allow us to distinguish the changes that had to be kept and the ones that had to be reverted. The trick is to ask the user to "mark" the point up to which changes have to be kept and starting from which changes have to be reverted in a way that we can distinguish at the point where `break-bp-`L is called. This marking takes the form of the introduction of an additional change set. Changes performed before the introduction of the additional change set will be kept, changed performed afterwards will be reverted. Introduction of a change set has the effect that all changes performed thereafter will be stored in the introduced change set (as opposed to before, when they are stored in a rule scope). In order to allow the user to easily introduce and then properly commit a change set, we have added a new strategy to the dynamic rules library called `dr-transaction`. `dr-transaction` simply takes a strategy and one or more dynamic rule names, creates a change set in each of the rule sets associated with the dynamic rule names, runs the user-provided strategy and commits the change sets in all the rule sets.

This following code snippet illustrates how `dr-transaction` is used in the context of the dead code elimination transformation in order to separate changes *pertaining to a rule scope* that have to be kept from changes that have to be reverted by a call to `break-bp-`L:

```
1  dce-stats-decl =
2    ?[DeclarationTyped(x, _) | _]
3    ; {| VarNeeded, VarUsed :
4         rules(
5           VarNeeded+x :- Var(x)
6           VarUsed+x   :- Var(x)
7         )
8         ; dr-transaction(
9             [id | dce-stats]
10            ; try(ElimDecl)
11          | ["VarNeeded", "VarUsed"])
12     |}
```

The effects created in lines 4–7, so before the call to `dr-transaction`, will be kept, while the ones created during the run of the strategy from the lines 9–10 will be reverted. Please note that in both cases we are only discussing about the effects that *pertain the to rule scope being defined*. This same process happens for each additional rule scope that is introduced.

Assuming that the user adheres to this convention, the effect of calling `break-bp-`L will

be as follows. The entire rule set (for a specific dynamic rule) will be split in the part that has been created within the fix point iteration over the loop containing the break statement (which we shall call the fix point part) and the rest (which we shall call the base part). The intent is to make the entire rule set look like the base part, but still maintaining the changes that appear in the rule scopes of the fix point part. This is achieved by changing the last change set of the fix point part to include entries that cancel the changes present in all the change sets of the fix point part. After this, the rule set will be identical (in behavior, not in structure) to the base part, with the additional changes that exist in the rule sets of the fix point part.

Discussion As we mentioned at the beginning of this section, the support for break in backward propagation transformations is still to be regarded as experimental. It still seems slightly unintuitive for a regular user to be able to grasp. Or rather, the effort required for understanding how rule sets have to be set up in order for `break-bp-`L to work might be too great. We should also mention that we only had dead code elimination in mind when implementing this, so we are not entirely sure that the mechanism will generalize flawlessly to cover other backward propagation transformations as well. Finally, it is clear that once the idea is further refined (and perhaps made better and easier to use), it will have to be extended in order to provide support for breaking to labels, continue statements and exceptions. We do not envisage this to be an easy task, especially in what exceptions are concerned.

3.5 Unit tests and bug fixes

Preceding and integrated with our work on the strategies for non-sequential control-flow, we have also developed a comprehensive unit test suite for verifying proper behavior of the more complicated strategies that make up the library. This was necessary for raising our level of confidence that the changes we have made to the code base did not break any of the previously existing functionality. Likewise, we also wanted to ensure that the new strategies we have developed properly handled all the test cases we could come up with.

The unit test suite consists of unit tests for checking the following aspects of the library:

- proper management of dynamic rules with combinations of rule scopes and change sets, together with proper lookup of rules when various combinations of rule scopes and change sets are involved (dr-scope-tests.str)
- intersection tests (all combinations of one level of intersection and selected combinations of two levels of intersection) (dr-propconst-tests-1.str)
- fix point iteration tests with intersection as the merge operation (several hundred combinations of fix point intersection and simple intersection and some tests with nested fix point intersection) (dr-propconst-tests-1.str)
- tests for breaking out of non-iterative and iterative structures, with and without using explicit labels to break to, with and without multiple nesting levels of fix point iteration (dr-propconst-tests-1.str)
- tests for continuing iterative structures, with and without using explicit labels to continue to, with and without multiple nesting levels of fix point iteration (dr-propconst-

tests-1.str)

- tests for combinations of breaking and continuing within iterative structures, with and without explicit labels and multiple nesting levels (dr-propconst-tests-2.str)
- exception throwing/catching tests with and without multiple nesting of try-catch-finally statements, checking for all the scenarios that we have discussed in the section on the support for exceptions (dr-propconst-tests-2.str)
- a limited number of tests for basic union, fix point iteration with union as the merge operation, breaking & continuing within loops as well as exception throwing, all using union as the merge operation (dr-varunion-tests.str)

The creation of this unit test has also lead to the discovery to a number of bugs (about five) in the implementation of the dynamic rules library. Most (if not all) of these bugs were related to the fact that in various strategies defined by the library the lookup of values was faulty for some particular cases. To correct this, aside from multiple small corrections, we have had to (1) introduce a new strategy, `dr-lookup-rule-in-scope`, that looks for a definition of a key in a particular scope only, regardless of whether or not that definition is currently shadowed by a different one and (2) modify the strategies `dr-lookup-rule` and `dr-lookup-rule-pointer` to incorporate the contents of the removed sets of change sets into their lookup. We will discuss these two in turn.

`dr-lookup-rule-in-scope` was necessary because in many places lookup was performed which logically speaking required this behavior, but however used `dr-lookup-rule` or even `hashtable-get` (for looking up in the topmost change set or rule scope only), which was wrong. `dr-lookup-rule-in-scope` descends into the rule set and retrieves the definition defined in a particular scope only, ignoring potential shadowing of that rule scope by higher ones. All the lookups in the library that wrongly used a different lookup strategy were modified to use `dr-lookup-rule-in-scope` instead.

Turning to the modifications of `dr-lookup-rule` and `dr-lookup-rule-pointer`, this was the correction of a fairly straightforward omission of including the effects of the entries in the remove sets of change sets into the lookup. In short, entries in the remove sets of change sets indicate the removal of a key from a rule scope. When performing lookup, all the keys that appear as removed in one change set or another do not have to be returned as the result of the lookup because, at an intentional level, they do not exist anymore.

Chapter 4

Pointer analysis

In this chapter we discuss the second layer of our work which consists of making alias information about the variables of a program available to the general user. This is intended to enable users of the Stratego platform to write more realistic data-flow transformations. Without alias information, a user had two choices: either she assumed that no two variables in a program can ever point to the same memory location or she assumed that any two variables in a program can point to the same memory location. The first variant is over optimistic, while the second is excessively pessimistic. Since using the latter variant yields almost all transformations useless, most users went for the over optimistic variant, which is unfortunately unrealistic. In order to bridge the gap between these variants, alias information is necessary and this is what our work for this part of the thesis has been about.

This is not to say that we have solved the problem entirely. There are two main issues that remain. In the first place, pointer analysis is language dependent. This means that each language that has to be supported needs its own implementation made available. However, since the base algorithm is unique, supplying a parameterized generalization is possible with some effort. Thereafter, all that would be needed for each language is the filling in of the particulars. The second issue is that our analysis is flow- and context-insensitive. While this is appropriate for many cases, it is not appropriate for all. Therefore, a pointer analysis that takes flow and/or context sensitivity into account will still be a welcome addition to the platform.

4.1 Overview

There are two main features of modern languages that make the lives of compiler writers difficult: exceptions and wide-spread use of pointers. We have already explained in the previous chapter why the complex control flow introduced by exception mechanisms complicates optimizations of programs. In this section, we will focus on the second of the two aspects: *pointers*.

Languages like Java and C# have integrated pointers into the language so much that it is impossible to avoid using them. This means that even the most basic of programs written in such languages is still almost certain to employ pointers. Basic compilation is not affected by the presence of pointers, since pointers boil down to memory addresses, and there is no inherent difficulty in dealing with memory addresses. As soon as we want to turn our

compiler into an optimizing one, however, we run into trouble when pointers are involved. The reason for this is that virtually all data flow analyses stop working as they used to when pointers were not in the picture. If the same memory location can be referred by different variables, then our analyses can no longer use variable names as unique identifiers of memory locations. This creates a problem because a program is specified in terms of variable stores and loads and not in terms of memory location stores and loads. Without pointers, we had a one-to-one mapping between the two, so we could use them interchangeably. In the presence of pointers, this mapping no longer holds, leading to the need of very conservative assumptions in order to ensure the soundness of analyses.

To explain why this is, consider the following example:

```
void foo() {
  BitSet x = new BitSet();
  BitSet y = x;
  y.set(3);
  if (x.isEmpty()) {
    System.out.println("some text");
  }
}
```

A pointer-unaware analysis could conclude that, since between the `x = new BitSet()` and the test `x.isEmpty()` the variable `x` is not changed by any other instruction, the test can be safely eliminated since we know at compile time that it will always evaluate to true. This would yield the following optimized version of the program above:

```
void foo() {
  BitSet x = new BitSet();
  BitSet y = x;
  y.set(3);
  System.out.println("some text");
}
```

However, this pointer-unaware optimization is wrong in the context of a language like Java, since Java uses pointers, so the simple fact that the literal variable `x` is not modified is not a guarantee that the contents of the memory location `x` points to is not modified. In fact, in our example code, it is the case that the memory location is modified through the variable `y`, which is an alias of `x` (or, alternatively, which points to the same memory location as `x`). As you can see, the presence of pointer invalidates otherwise valid optimizations.

We can deal with this in two ways. The easier, but essentially inefficient one, is to make the most conservative assumption of all, namely that any variable in the program (including instance variables, class variables, local variables and method arguments) can be an alias of any other variable. Such an assumption makes our analysis perfectly sound, but utterly useless, since any store to any variable in our program would have to be interpreted as a potential store to any other variable. This means that we can only perform optimizations between stores to variables. While making this assumption is an option, it is clearly a poor one.

The alternative to this is to collect pointer information from the program in order to have more precise knowledge about what variables may point to what other variables. This process of collecting information about pointers is called *pointer analysis*. In fact, pointer analysis comes in more shapes and sizes, depending of what kind of pointer information we are looking for and how we represent it. We distinguish *alias analysis* as the pointer

analysis that creates pairs of variables which can be aliases of one another: (x, y) indicates that x and y could point to the same memory location. A more space-efficient alternative to alias analysis is *points-to analysis*. Points-to analysis computes what memory location(s) a variable can point to. We know that two variables can be aliases of one another if they can point to the same memory location. *Shape analysis* goes beyond simple aliasing information and computes what "shape" the data structures represented in memory have (graph-like, tree-like, list-like, etc.). Finally, *escape analysis* computes whether or not a heap location allocated in a method (or thread) can escape that method (or thread). A heap location can escape a method, for example, if that method returns a pointer to it or if it passes a pointer to it to a different method. Knowing that a heap location cannot escape a method (or thread) can enable certain optimizations. Since all these analyses have to analyze pointer assignments (or equivalent code which translates to pointer assignments) in order to compute their result, they are all referred to as pointer analyses.

Pointer analyses can be *context-sensitive* or *context-insensitive*. Context-insensitive analyses do not consider the calling context of a method (i.e., it does not make a distinction between calls to the same method based on the caller). This has the effect that there will be a unification of pointer information among all the call sites of a method, resulting in the transfer of information from each call site to all others. This will generate less precise aliasing information, since it will determine a potential aliasing relationship between variables that could never point to the same memory location. Notice that this is not wrong, but simply less precise (or conservative). What would be wrong is for the analysis to conclude that two variables cannot be aliased when in fact they can, according to the program. Context-sensitive analysis, by contrast, does consider calling contexts, thereby producing more precise points-to information[1].

Another distinction of pointer analyses refers to whether or not we consider control flow. Based on this, we have *flow-sensitive* and *flow-insensitive* pointer analysis. Flow-sensitive analysis differs from the flow-insensitive one by computing separate pointer information for each program point. Flow-insensitive analysis computes a single "batch" of information (e.g., a single points-to graph) for the entire program. The information computed by flow-insensitive analyses is valid for all points of a program, whereas each "batch" of information computed by flow-sensitive analyses is only valid at the particular program point with which it is associated. Naturally, flow-sensitive analyses are more precise, especially given that the points-to information of a program changes a lot during its execution. By computing only one "batch" of information per entire program (as is the case with flow-insensitive analysis), it is clear that there will be a lot of unnecessary unification of points-to sets, which leads to loss of precision (to put things into perspective, the least precise case is that when we have a single points-to set, i.e., all we know is that any variable can point to any other variable).

Both in the case of context-insensitive vs. context-sensitive and the case of flow-insensitive vs. flow-sensitive, the gain of precision that corresponds to the sensitive versions translates to a loss of efficiency. Thus, the balance that has to be found both in terms of context and flow sensitiveness is that between the precision of the results and the cost (both of time and space) of the analysis. A more precise analysis will take more time and space to compute and will not be scalable to large code bases. A less precise analysis, while less costly, will open fewer optimization opportunities. Depending on each particular situation,

[1] We use the terms "aliasing information" and "points-to information" interchangeably since any of them can be directly derived from the other.

combinations of these variants can be used.

Another important aspect which distinguishes pointer analyses is whether or not they take type information into consideration. It is generally a good idea to consider type information because it incurs little extra cost, but it can significantly increase precision. Type information is useful to discard spurious aliasing. If we know that two variables have incompatible types, then we can conclude that they can never alias each other.

Pointer analysis is required for many kinds of data-flow transformations. Constant propagation cannot be performed with much efficiency if we do not have pointer information since each store to a variable disallows any further propagation. Dead-code elimination cannot be done without pointer information either because, for example, we cannot assume an assignment to a variable is useless just because that particular variable is not used further on in the program. It could be that the memory location modified by the store through that variable is accessed later on in the program through another, so removing the assignment would yield incorrect results. Even common subexpression elimination can be incorrect if we do not consider pointers. For instance, if we have a sequence of assignments like `x = a * b; c = 5; y = a * b;`, it would be incorrect to replace it with `x = a * b; c = 5; y = x;` if we do not know for a fact that `c` is *not* aliased with any of `x`, `a` or `b`. Without alias information, we have to assume that, for all we know, *it may be aliased*, so we have to conservatively disallow replacing `a * b` with `x`.

These are just a few examples of data-flow optimizations which have to consider pointer information in order to behave correctly. Like them there are many others. This generalized need for pointer information makes us conclude that if Stratego is to be used for writing data-flow transformations that can apply to real programs, it clearly needs to provide pointer information. This is why, as part of this thesis, we have put effort into making this information available by looking for and implementing a strategy that computes such information.

The issue of dynamic dispatch When performing any kind of interprocedural data-flow analysis (as the points-to analysis is), one of the things we need to do is figure out which body of a method (or procedure) to inspect when we encounter a method (or procedure) call in the analyzed code. While this is not too difficult in procedural programming, things get more complicated when we have to deal with object-oriented languages. Such languages use the dynamic dispatch mechanism at run time in order to decide which method to call, which makes it not so straightforward to find out which of potentially many method bodies will actually be executed. Consider the following Java code, for example:

```
A o;
n = System.in.read();
if (n == -1) {
  o = new B();
} else {
  o = new C();
}
o.someMethod();
```

```
abstract class A {
  abstract void someMethod();
}
class B extends A {
  void someMethod() {...}
}
class C extends A {
  void someMethod() {...}
}
```

There is no way of telling at compile time if method `someMethod` of class B or the same method of class C will be called at the call site `o.someMethod()` in the code on the left. In

such cases, we have no other option than to consider all possibilities and merge the results (this will translate to a loss of precision for points-to analysis, for example).

In order to find out what code we have to analyze, it is necessary that we have access to class hierarchy information. This enables us to search all descendants of a base class for matching methods. Section 4.3.2 discusses this in more detail.

4.2 Related work

Pointer analysis has been an active field of research for over 20 years, with a huge amount of work being put in. In a paper from 2001 [15], Michael Hind gives a measure of "over seventy-five papers and nine Ph.D. theses" that have been dealing with this subject. According to the same paper, there are a number of reasons why this problem is not yet considered solved. All these reasons are really different facets of the same core one: there is so much variability that has to be dealt with in what pointer analysis is concerned, that all this work has not yet managed to cover all there is cover.

Among the open issues that still remain are things like how to improve scalability and precision at the same time (i.e., find ways in which we can achieve better precision, but still be able to apply the techniques to very large code bases), offering pointer analysis options based on the needs of the client analysis that uses its results, implementing demand-driven analyses and tuning the analyses to deal with incomplete programs.

Efforts in pointer analysis far precede the birth of the Java programming language, so much of this research is related to analyzing C/C++ programs. Although some of the results are still applicable, there is much difference between the approaches. The main aspects of Java which make pointer analysis distinct than in C/C++ is the absence of multi-level dereferencing of pointers and the lack of function pointers. We have focused our interest on pointer analyses that were implemented specifically for Java, since our efforts were in the direction of supporting Java-like languages.

Rountev et al. [21] took a constraint-based approach for doing points-to analysis for Java. They model the semantics of assignment statements and method calls (the two relevant types of statements that any points-to analysis has to consider) using *annotated inclusion constraints*. Annotated inclusion constraints refer to set-inclusion relations that can optionally be annotated ($L \subseteq_a R$). To illustrate, the constraint $ref(o, v_o, \overline{v_o}) \subseteq v_r$ indicates that reference variable r refers to object o (why object o is represented by $ref(o, v_o, \overline{v_o})$ is less important). Annotated constraints are used when object fields or method calls are involved. For instance, the annotated constraint $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$ indicates that field f of object o_1 ($o_1.f$) points to object o_2. In the case of methods, annotations are used to identify the compile-time method that is called. Annotated inclusion constraints can be represented as a directed multi-graph[2], with the annotations labeling the edges. First, the entire program is traversed in order to generate the system of constraints. Then, a generic algorithm is applied to the multi-graph representation of the constraints in order to compute its transitive closure. The edges in the resulting graph represent the the points-to relations of the program. The analysis of Rountev et al. is context- and flow-insensitive.

Whaley and Rinard propose a combined pointer and escape analysis in [29]. The analysis is flow-sensitive and context-sensitive and its result is a so-called *points-to escape graph*,

[2] A multi-graph is a graph which allows multiple edges between any pair of nodes.

which is a quadruple consisting of a set of outside edges (representing references to object allocation sites that are outside the current analysis scope), a set of inside edges (references within the current analysis scope), an escape function (indicates for each node what unanalyzed methods it escapes to) and a return set (what objects are returned): $\langle O, I, e, r \rangle$. One such graph is associated with each program point (since this is a flow-sensitive analysis) and there are two program points for each statement, before and after it executes. The nodes of the graph can be of several types: inside nodes (objects created in the analyzed scope), thread nodes (thread creation sites, subtype of inside nodes), outside nodes (objects created outside of analyzed scope) and class nodes (one per class, as a container for static fields). Outside nodes further subdivide in parameter, global, load and return nodes. This complex structure is used to represent all types of statements that are interesting for pointer analysis. For instance, an object creation site (`l = new cl`) represents the new object with an inside node and creates an edge form the node representing `l` (which could be a parameter node or a global node, for example) to the newly created node. The intraprocedural part of the analysis implies creating and deleting edges at each analyzed statement. A pass through a method will start with an initial points-to graph which contains parameter nodes and global nodes along with edges pointing to them as well as an escape function returning the empty set for any node and an empty return set. Statements are then analyzed in accordance with the control-flow graph, updating and storing the points-to graph accordingly at each program point. In the end, an exit points-to graph will be obtained. Interprocedural analysis uses a mapping algorithm to produce the graph for an invocation site, taking into consideration the graph before the site as well as the entry and exit graphs for the invoked method. After the analysis of the entire program is completed, we have accurate points-to and escape information for each program point.

Sălcianu set out in his master thesis [24] to provide the correctness proof for the analysis of Whaley and Rinard discussed above. To make reasoning easier, a reformalization of the original method was needed, which actually resulted in a significantly distinct formalism. The idea and type of the analysis have remained similar, nevertheless. The representation was switched from a quadruple to a five-tuple, $\langle I^a, O^a, L^a, S^a, U^a \rangle$[3], consisting of inside edges (I^a) and outside edges (O^a) just like before, and the abstract state of local variables (L^a, maps each variable to the set of nodes it might point to), the set of threads started by the analyzed scope (T^a) and the set of nodes that escape to unanalyzed methods (S^a). The last two components are used for modeling sources of escape information. The rest of the algorithm is similar to the one described above (by defining what edges and nodes are created or deleted by each statement and how mapping is performed at call sites). The bulk of the thesis, however, focuses on proving the correctness of the proposed method, which is of less interest to us.

Research on pointer analysis has also been done within the Soot framework for analyzing Java programs [26]. Ondřej Lhoták has begun work on pointer analysis as part of his master thesis [18] by creating Spark (Soot Pointer Analysis Research Kit). Spark is a generic implementation of pointer analysis, the goal of which is to compare different approaches within the same conditions, in order to provide results about how things like taking type declarations into consideration during the analysis or performing points-to graph simplification steps influence speed, memory consumption and precision of the results. The method proposed in [18] and then further refined in [19] implies the construction of a *pointer assignment graph* in order to represent the initial points-to relationships in the program, the

[3] The superscript ^{a} stands for the program point for which the graph is computed.

optional simplification of this graph and then the propagation of the information along the edges. Since this is in fact the analysis we chose to adapt to Stratego, we postpone the details for later.

The same group of people working on Soot have come up with an interesting idea afterwards, in order to allow better scaling of the original method. In the work described above they were using a hybrid set representation to represent points-to sets in the pointer assignment graph (points-to sets are sets containing object allocation sites which are associated with certain nodes of the graph). These hybrid sets use an array of pointers representation if the set contains 16 elements or less and a bitset representation otherwise. Even with this optimization, the algorithm still requires a lot of memory for large code bases. In this newer approach [5], the authors propose the use of Binary Decision Diagrams (BDDs) in order to represent points-to sets. After proper tuning of the representation, the benchmarks show that the analysis time is somewhat larger, but the memory requirements drop to 4 up to 6-7 times less, which allows the analysis of programs that were previously not analyzable. The BDD representation is so efficient because it allows maximal sharing of set elements representation. This means that in order to represent sets $S_1, S_2, \ldots, S_n$ we need roughly the same space as if we were to represent their union. This clearly saves a lot of space when there is considerable overlap between the sets (as is the case for points-to sets).

[28] also proposes the use of BDDs for pointer analysis, but for a different purpose than in [5]. Their approach is to obtain a context-sensitive pointer analysis by running a context-insensitive one over an extended call graph in which a different node is used for every call to a method (meaning that calls to the same method will be represented by edges to clones of the same node). It is clear that even for reasonably sized programs, such an extended graph would explode beyond representation capabilities, making it impractical, if not impossible, to compute with it. The context-insensitive analysis that is implemented is actually the one in [5], to which type information is added to improve precision. The entire algorithm (including the cloning of the call graph nodes in order to turn the context-insensitive analysis into a context-sensitive one) is represented in Datalog. Datalog is a language for specifying database relations with Prolog-style predicates and the paper explains how automatic translation from Datalog to a BDD implementation can be done. This automatic translation is used to transform the Datalog representation of the pointer analysis into BDDs. What is achieved is that the huge extended call graph is now represented and manipulated using BDDs, thus making the whole process feasible.

4.3 Setup for pointer analysis

In this section we discuss the infrastructure that had to be put in place in order to implement pointer analysis. We have chosen to implement our pointer analysis for a toy language that is very similar to Java, mainly in order to avoid all the clutter that is usually associated with a real-life language. Since this was the first implementation of a pointer analysis in Stratego, and hence we had little previous experience to build on, we tried to minimize the risks involved by allowing ourselves to iron out some of the difficulties inherent to real languages. We made sure, however, to retain (or rather build) in our toy language all those features which is important that pointer analysis takes into consideration.

The toy language we developed to perform pointer analysis over is called TOOL and was presented in section 2.4. Aside from an interpreter for TOOL (also discussed in the afore-

mentioned section), the need also arose for a type annotator and a simplifier. We detail these a bit in the first part of this section. Furthermore, in order to allow pointer analysis to properly handle dynamic dispatching, information about TOOL class hierarchy was necessary. We turn to this in the second part of the section.

4.3.1 Analysis of TOOL programs

The main aspect of a language that pointer analysis has to deal with are assignments. Everything pointer-related happens as an effect of assignments. Two variables, method parameters or object fields can only end up pointing to the same memory location as the result of an assignment. Argument passing is simply a more concise form of assignment(s), but in essence the same: each actual argument is assigned to its corresponding formal argument. Also related to argument passing in the context of object-oriented programs is the assignment of the target of a method call (i.e., `obj` in `obj.method(...)`) to the special variable `this` that is available in the body of the method. Despite the mechanism, this is in fact nothing more than a trivial assignment. Finally, another special form of assignment is that of the assignment of the result of a method to a variable. Since a method may contain multiple return statements, it follows that any of the expressions returned may end up being assigned to the receiving variable.

To sum up, when dealing with pointer analysis, all that we are interested in are assignments, in one of the many forms discussed above. Even so, assignments can come in various shapes and sizes and we need to be able to contain this variability and reduce it a number of basic types of assignments. Otherwise, we would be forced to support a very wide range of syntactic variants of what is essentially the same process in reality. In fact, there are only four basic types of assignments to which all this variability can be reduced to. These are $p :=$ new object, $p := q$, $p := q.f$ and $p.f := q$. Every other form of assignment can to be reduced to sequences of these.

For this reason, we need to run a transformation over the TOOL programs that we analyze which simplifies all complex assignments to these four base cases. Furthermore, we want to restrict argument passing and assignment of return values to assignments of the type $p := q$ only. This means that we can only accept simple variables as arguments to method calls and as expressions in return statements. We summarize this and other simplifications that we perform on TOOL code below.

- any assignment that is not in the form $p := q$, $p := q.f$ or $p.f := q$ will be transformed into a series of assignments, each in one of these forms. However, we do not simplify assignments of boolean or arithmetic expressions, regardless of the nature of the left hand side. This is because these types of assignments do not deal with pointers; we know, in fact, that the target of the assignment is bound to be of a non-pointer type. For instance, `x := (y+4)/5` will not be simplified because it does not change pointer relationships in any way.

  However, assignments of the form $p.f_1.f_2 \dots f_n := q.g_1.g_2 \dots g_n$ have to be adjusted on multiple levels. The general rule is that either the left hand or the right hand side of the assignment may contain one level of dereferencing at most, but not both sides. For the assignment above we have two options: we can either reduce it to one of the form $p.f := q$ or to one of the form $p := q.f$. We randomly chose the latter in our simplification. Hence, the original assignment will become $temp_1 := p.f_1 \dots f_n$;

$temp_2 := q.g_1.g_2 \ldots g_{n-1}$; $temp_1 := temp_2.g_n$. Naturally, simplification has to continue recursively for the first two statements from above. However, the third statement is now in an acceptable form.

- any method call (including constructors) will have to be in the form $o.m(p_1, p_2, \ldots, p_n)$, where p_i are simple variables (no expressions, no field dereferences, no method calls). Therefore, another of our simplifications is that any argument passed to a method call that is not a simple variable will be assigned to one and replaced in the call by that variable. To illustrate, `obj.method(p.f, q.g`$_1$`.g`$_2$`, r.foo())` will be replaced with `temp`$_1$ `:= p.f; temp`$_2$ `:= q.g`$_1$`.g`$_2$`; temp`$_3$ `:= r.foo(); obj.method(temp`$_1$`, temp`$_2$`, temp`$_3$`)`. The second statement will have to be further simplified to meet our criteria.

  A series of method calls will likewise have to be simplified by breaking them into separate multiple calls, the result of each being assigned to an additional temporary variable. If you consider `obj.m1(...).m2(...)`, this will have to be transformed into `temp := obj.m1(...); temp.m2(...)`.

- finally, any return (or throw) statement that does not return (or throw) a simple variable will have to be modified so that it does so. This code: `return p.f.foo();` will have to be changed to `temp:=p.f.foo(); return temp`. Naturally, the first statement will need further simplification.

In order to be able to perform all these simplifications, we needed type information for our variables and expressions. If not for more, at least for the fact that all the additional variables that have to be introduced as part of the simplification transformation also have to be properly declared, and their declaration involves specifying their type. As it is, we need type information for each expression that appears on the left or right hand side of an assignment, in a method call, or as part of a return (or throw) statement. Not only do we need the type for an entire expression, but we need it for all subexpressions that compose the expression as well, especially in what successive field dereferences are concerned. For example, in the case of the expression $p.f_1.f_2 \ldots f_n$, we need the type of the entire expression and that of subexpressions $p.f_1$, $p.f_1.f_2$, $\ldots$, $p.f_1.f_2 \ldots f_{n-1}$ as well.

Due to this necessity, we have implemented a type annotator for TOOL programs which supports all elements of the TOOL syntax: local variables, method parameters, class fields (static and non-static, inherited fields), method calls, and, generally speaking, all types of statements and expressions. The type annotator is run over TOOL programs prior to simplification in order to produce the type information needed by the simplifier.

4.3.2 Class hierarchy analysis

The last prerequisite for running pointer analysis over a TOOL program (and its accompanying collection of classes) is structural information about the class hierarchy. This is in fact not something that we need specifically for pointer analysis, but rather something that is necessary for virtually any interprocedural[4] analysis. The typical analysis will start from the main function of the program and will inevitably (with the exception of some very simple programs) reach method calls. In the case of an intraprocedural analysis we can either

[4]We the coined term *interprocedural* in our discussion, despite the fact that in OO programming we are dealing with methods, not procedures.

ignore method calls or handle them in some conservative way, whichever of the two makes sense given the semantics of our transformation. However, in the case of interprocedural analyses, handling of a method call implies applying our analysis over the body (or bodies, even) of the method that is being called. In order to be able to do this, we naturally need to retrieve the right method(s).

Resolving methods The issue of identifying which method body (or bodies) to analyze is trickier than would seem at first sight. First of all, we need type information regarding the target of the method call, since otherwise we do not know what class (or set of classes) to scan for finding the right method. Fortunately, we already have the type information available from before, when we needed it for the simplification. It is perhaps worth mentioning that the type that we use is the declared type of the variable.

Once we know the type of the target object, we can proceed to determining the method (or methods) that we have to analyze. The first case is when the method is declared (as abstract) or defined (concretely) in the class indicated by the declared type of the target object (for simplicity, let us call this class the *target class*). In either of these two cases, it does not suffice to simply return that method as the result of our search. If we have an abstract method declared in the target class, it is clear that we need to search for (and return) a concrete definition in one of the subclasses that will actually get executed.

However, even if we do have a definition proper in the target class, we still have the dynamic dispatch mechanism that makes it possible that other method bodies might get executed as well. It could be the case that even if the declared type of an object is, say, class Foo, the object might be initialized with an instance of a (direct or indirect) subclass of Foo, say Bar. If both Foo and Bar define the method that we are looking for, the actual method that gets executed is the one defined in class Bar. It follows that the actual body that will end up being executed at run time might be that defined in the target class *or* it might be any of the bodies defined in (direct or indirect) subclasses of the target class. As a result, in this case, we need to return *all non-abstract methods that match the signature of the called method, defined in either the target class or in any of its subclasses*.

The second case we have to deal with is when the target class does not declare or define the method we are looking for. In this case, the method that might end up being executed is either one defined in one of the subclasses of the target class or (and here we have a distinction from the previous case) one defined in a superclass of the target class. We only need to consider the very first definition that appears in the inheritance chain, as we go upwards from the target class.

To sum up the two cases, the result will always be a union of two components. The first component is the same for both cases and it comprises of the matching methods defined in any of the subclasses of the target class. The second component of the union will be either (1) the (single) definition of the method in the target class itself or (2) the (single) definition of the method in the first superclass of the target class as we go upwards in the hierarchy. Either the first or the second component of this union may be empty, but never both at the same time. That would be a program error.

Each interprocedural analysis will deal with this set of methods in its own way, once the set is obtained. Most analyses will want to examine each member of the set and perform some sort of merging of the results that makes sense in the context of the analysis. How we deal

with this set in the context of the pointer analysis will be discussed in the following section.

Things are a bit easier in the case of constructors (which are still a special case of method call). Whenever we have to retrieve the body that gets executed as a result of a statement like `new C(...)`, we simply look for a matching constructor in class C. The dynamic dispatch mechanism does not intervene here, so that constructor is the only method (or, if you prefer, constructor) body that has to be returned.

When analyzing the body of a constructor, it is sometimes the case that calls to the `super` method are encountered. Such calls are the syntactic means of calling the constructor of the superclass of the current class. Calls to `super` can also be resolved quite easily, since they always result in a single (constructor) method which is always defined in the superclass.

Building the hierarchy It should be clear by now that resolving sets of methods implies many queries related to class hierarchy structure. It is necessary that we can easily navigate from a given class upwards in the hierarchy chain to its ancestor classes as well as downwards in the hierarchy chain, to all its subclasses. Given that we do not want to scan files over and over again in order to retrieve hierarchy information, some sort of in-memory structure clearly has to be built.

The fact that we need to be able to retrieve all subclasses of a class makes an incremental approach to building this in-memory structure of little use. This is because there is no way of knowing which classes are subclasses of a class without scanning all the class files in the project. So, even though an incremental approach to building the structure would make sense in the case of navigating upwards (given that each class specifies which class it extends, so we could read in the relevant classes directly), it cannot be accommodated when navigating downwards.

Since a full scan of all the classes present in a project has to be done anyway, the only method for improving efficiency that we can foresee is the setting up a caching mechanism. Such a mechanism could be employed for libraries of classes which are reused across different projects. Each library could be accompanied by an additional file that specifies the class hierarchy structure. Although we have not experimented with this ourselves, it is only a matter of writing the in-memory structure to disk and reading it back into memory from the disk every time it is needed.

We return now to what we do in the particular case of TOOL. Since TOOL only supports a flat directory structure (i.e., all the class files are in the same directory), we simply read in all the class files and store the inheritance relationships among them using a dynamic rule. For each (class, superclass) pair we add an entry of the type SuperClass(c) $\rightarrow$ sc and an entry of the type SubClass(sc) $\rightarrow$ c. In the subclass relationship, we actually append entries to a list, given that a class may have multiple subclasses.

With this information in memory, we can navigate from a class to its superclass(es) by using the SuperClass entries and we can collect all subclasses of a class by using the SubClass entries. In order to support interprocedural analysis of TOOL programs (and, in particular, the pointer analysis) we have built a number of strategies on top of the class hierarchy information, the most important of which we list below:

- `get-class`, `get-superclass`, `get-ancestors`, `get-all-subclasses`: return the code of a class, the name of the superclass of a class, the names of the ancestor

classes of a class and the names of all subclasses of a class, respectively.

- `get-method`: takes a class name, a method name and a list of argument types and returns a single matching method. The definition that is returned may be the one from the class itself (if there is such a definition) or, if not, that from the first superclass as we go upwards in the hierarchy in which the method is defined. When looking for the matching method, matching of argument types is done by taking inheritance relationships into consideration. Assuming S is a subclass of T, then a requested argument type of S and a declared argument type of T will result in a match.

  The `get-method` strategy only returns non-abstract and non-static definitions. Similar strategies exist for retrieving abstract methods and static methods.

- `get-all-methods`: just like `get-method`, it takes a class name, a method name and a list of argument types as arguments, but unlike `get-method`, it returns all possible methods that might end up being executed at run time. We have described the methods that form this set earlier on when discussing the difficulties introduced by dynamic dispatching.

- `get-field`, `get-static-field`: looks for non-static and static fields, respectively, defined either in the class passed as an argument or in any of its superclasses. These strategies (and some of the others, as well) are used in the type annotation strategy, for example.

4.4 Pointer analysis algorithm

The eventual purpose of our efforts for this part of the thesis project was implementing pointer analysis for our prototype object oriented language (TOOL) and making its results readily available to Stratego users. We have investigated a number of approaches to performing pointer analysis, and in the end we have decided to adopt the one described in [18, 19]. This is a context insensitive and flow insensitive analysis, which produces a single instance of pointer information that is valid for the entire length of the program. The information obtained after running the pointer analysis is meant to be used as a resource for other data-flow transformations.

We have chosen this particular method for a number of reasons. To start with, ours has been the first attempt of implementing a pointer analysis in Stratego and as there is no previous experience to build on, we preferred choosing an analysis which was clearly described and rather straightforward to implement. We have considered the process of implementing such an analysis in Stratego complicated enough even without adding more complexity by choosing a more demanding analysis. Secondly, there is a version of this same analysis (presented in [5]) which reduces the space requirements by using Binary Decision Diagrams (BDDs) for representing points-to sets (see section 4.2). We expect that a future effort can start with the base implementation that is now available and change it to the improved BDD version. This will first require binding Stratego with an existing C library that allows manipulation of BDDs and then replacing of the data structures used by our analysis with different ones based on BDDs. Finally, we chose a context insensitive and flow insensitive analysis to ensure a reasonable efficiency so that it can be immediately used (i.e., without requiring further optimization) in our data-flow transformations. Granted, its precision will suffer from both types of insensitivity's, but we regard this as an acceptable compromise.

As a side note, flow sensitivity may not even be desired at all in Stratego transformations if we consider how dependent dynamic rules work. We expect that the primary use for the information generated by the pointer analysis will be to define triggers for the undefinition of dynamic rules using the dependent dynamic rules mechanism. To illustrate, consider again the case of constant propagation. We want a constant propagation rule for a particular variable to be undefined when that variable *or any of its potential aliases* is changed. For this, we need to use a list of potential aliases that is not only valid at the point in the program where we define the triggers for undefining the rule, but which is valid at any point of the program. From this perspective, we regard program-wide information more useful than information that is only valid at a discrete points of the program. The former is the case when we perform flow insensitive analysis, while the latter is the case when we perform flow sensitive analysis.

The original analysis that we adopted, described in [18, 19], is designed so that it can accommodate a lot of variability in the algorithm. This is because it is built to allow experimentation with various implementations of pointer algorithms. Since our goal is different, we have fixed a number of aspects and have only focused on implementing support for those. We chose to use a subset-based approach in propagating points-to sets (as opposed to a equality-based approach), to use a class hierarchy structure that we build before running the algorithm, to do a *field-sensitive* analysis (i.e., consider fields separately for each instance) and not to simplify the pointer assignment graph. We maintained some variability in the propagation algorithm by implementing both the iterative propagation the worklist-based propagation algorithm. If you want to achieve a better understanding of the alternatives, we refer you to the original papers [18, 19].

We only want to detail the choice between the subset-based and the equality-based approach. The basic idea of subset-based pointer analyses is that an assignment like $p := q$ will have the effect that all objects that q points to will be pointed to by p as well, but not the reverse. In other words, the set of objects pointed to by q will be a *subset* of the set of objects pointed to by p. In equality-based approaches, the two sets will always end up equal. You have probably guessed that the subset-based approach is more precise.

The pointer analysis, as we have implemented it, consists of one preparatory step and two main steps. Let us first explain the two main steps and then we will come back to the preparatory step. The first thing we have to do is analyze the entire program and build a *pointer assignment graph* for it. We start from Main.tool, which contains the "main" function and recursively follow all method calls to whichever code they lead us. The graph we build will be a representation of all the assignments that exist in the program and which we discussed earlier on in this chapter (basic assignments, implicit assignments of actual to formal arguments and return expressions). Once the graph is built, the second step will deal with propagating memory locations along the edges until no more propagation can be done. At that point we have information about what variables/class fields/method parameters can point to what memory locations, which is what we wanted. This leaves us with the preliminary step. Its purpose is to create unique encodings for all the variables, method parameters and memory locations created in the program. We need this unique encoding both for the pointer analysis itself and for the end user. In both cases, it is important to know if `x` refers to local variable `x` in the main program or to argument `x` of method `foo` of class `Bar` or to the index of the for loop in method `bar` of class `Foo`.

We discuss these three steps at more length in the following subsections. We also have a

final subsection about how we transform the points-to information (given a variable, we can tell what memory locations it may point to) into alias information (given a variable, we can tell what variables it may be an alias of).

4.4.1 Variable annotation

Three reasons have generated the need for an annotation scheme. First of all, we needed a way to name memory locations that program variables can refer. Memory locations can be created by `new` statements (e.g., `new Foo(...)`) or by string constants (e.g., `str := "some string"`). Memory locations are relevant because they are what variables point to. Two variables may be aliases of one another if they may point to the same memory location. We need to be able to identify each particular memory location, so we need to annotate each of them with a unique identifier.

Secondly, we needed to be able to tell identically named variables apart. As we all know, scoping allows the reuse of variable names, so generally there will be no single `x`, `y` or `i` across a number of methods and classes. Since we need to treat different instances of variable `x` separately, we need to know which occurrences refer to which declarations of the variable. More precisely, as we will detail in section 4.4.2, each declaration of a variable will have a node representing it in the pointer assignment graph. It is thus important to be able to identify what nodes an assignment like `x := y` has to connect. For this, all variables of a program need to be uniquely annotated so that all occurrences of a variable that refer to the same declared variable are annotated with the same identifier.

Finally, the third reason is quite similar to the second, except that we view it from an end user perspective. In order for the user to be able to actually use the results of the pointer analysis, she needs to have access to a strategy that, when passed a variable, will return all possible aliases of that variable. Both the variable and its aliases need to be identified uniquely across the entire program, for similar reasons as before. The user interaction also introduces an additional constraint, namely that the annotation scheme needs to be reproducible (i.e., if we run it twice, it yields the same results). Otherwise, there would be a mismatch between the annotations introduced by the analysis and those introduced by the user, and obviously the two would not be compatible.

The annotation scheme that we use for all the purposes detailed above is as follows:

- we annotate all new memory locations (`new` statements and string constants) that are created in a method with the identifier `NewId(cn, ms, idx)`. `cn` stands for class name, `ms` stands for method signature (a pair containing the method name and a list with the types of the method's arguments) and `idx` is an index that starts at 1 and increases for each newly created memory location, in program order.
- we annotate the special `this` variable with `VarId(cn, ms, 0)`, the n formal arguments of a method with `VarId(cn, ms, `i`)`, with $i = 1, n$ and the local variables of the method with `VarId(cn, ms, `idx`)`, with idx starting at $n + 1$ and increasing for each newly declared local variable in program order. Variables declared in catch clauses are handled no differently from regular local variables. Just like before, `cn` and `ms` stand for class name and method signature, respectively.
- we have two special cases: the main program and static blocks. Both are handled just like normal methods, except that we use a non existent class name (for the main

program) and hand-crafted method signatures (for both). In the case of the main program we use `Main` as the class name, `main` as the method name and an empty list of arguments; in the case of a static block, we use the regular class name of the class it appears in, `Static.Block` as the method name and an empty list of arguments.

We provide an example below of how the variables and memory locations created in the method `toString()` of class `Object` are annotated. To clarify the rather strange `__temp_13` variable name, remember that we perform simplification on the code before annotating the variables. The original method only had one line: `return "N/A"`.

```
MethodDec([], "toString", [], TypeName("String")
  , Block([
      DeclarationTyped("__temp_13", TypeName("String"))
        {VarId("Object", MethodSig("toString", []), 1)}
    , Assign(Var("__temp_13")
               {VarId("Object", MethodSig("toString", []), 1)}
           , String("N/A")
               {NewId("Object", MethodSig("toString", []), 0)})
    , Return(Some(Var("__temp_13")
                    {VarId("Object", MethodSig("toString", []), 1)}))
    ])
)
```

4.4.2 Pointer assignment graph

The pointer analysis uses a graph structure called a *pointer assignment graph* in order to represent the program under analysis. We begin by describing what the nodes and edges of this graph represent and then we proceed to explaining how the graph is built. Since there will be a lot of terms coming up, please make a mental note that in what follows, we use the terms *memory location*, *allocation node* and *element in the points-to set* to refer to essentially the same concrete thing: a representation of the memory space occupied by the object or objects created at an allocation site of the program. Allocation sites are statements of the type `new SomeClass()` or string constants. These representations are memory locations (since objects are allocated at some location in memory), are what an allocation node represents (see below), and are also the elements of points-to sets.

There are two functions that the nodes of the pointer assignment graph have to accomplish: a representation function and a "holder" function. Some nodes represent either memory locations created by the program or various types of variables used by the program. Some nodes hold points-to sets that are associated to them. These points-to sets will eventually have to be filled with memory locations. The fact that a memory location (eventually, after the propagation phase) appears in the points-to set of a node means that the variable that node represents may point to that memory location. If two nodes have common entries in their points-to sets, it means that the variables they represent may be aliases of each other.

The four types of nodes that may appear in the pointer assignment graph are as follows:

- **allocation nodes**: each such node stands for a set of run time objects. Each allocation site of the program (i.e., each `new SomeClass(...)` or string constant) will be represented by such a node. Allocation nodes stand for sets of objects and not

necessarily just single objects because the same allocation site may allocate an indefinite number of objects (consider an allocation site in a loop or in a method that gets called multiple times, for instance). Allocation nodes mainly have a representation function and, secondly, a trivial holder function (trivial in that allocation nodes will always have points-to sets with one entry only: the allocation node itself).

- **variable nodes**: these nodes are used to represent a set of memory locations that hold pointers to objects. One such node will be created for and associated with each local variable, each formal method argument and each static field (which can be regarded as global variables). These nodes have both a representation function and a holder function.

- **field reference nodes**: they are used to represent pointer dereferences (e.g., $p.f$). Each field reference node will pair a link to a variable node with a field name. This type of node only plays a representation function; it will not, like all other types of nodes, have a points-to set associated with it.

- **concrete field nodes**: this type of node only comes into play during the propagation of points-to sets (described in the next subsection) and it is complementary to the field reference node. Concrete field nodes do not play a representation function (i.e., they do not represent a concrete occurrence in the program, like the other types of nodes). Each concrete field node will only be used to hold the points-to set of a pair consisting of a link to an allocation node and a field name. This points-to set holds the memory locations that the field (second member of pair) of the object(s) represented by the allocation node (first member of pair) may point to.

Edges in the pointer assignment graph represent assignments. If two nodes are connected by an edge, it means that at some point in the program there has been an assignment involving the variables they represent. With the introduction of edges, the distinction between the representation function and the holder function of nodes can be clarified. Since edges only play a representation function themselves, they will only connect nodes that also have a representation function. In other words, concrete field nodes will never be connected among themselves or with any other type of node. In fact, their definition as nodes in the graph is really just so that we do not deal with other data structures beside the graph. They could just have well been separated completely.

Just as with nodes, there are also four different types of edges that can appear in the pointer assignment graph. We described them below.

- **allocation edges**: connect an allocation node to a variable node and indicate that the variable may point to the memory location(s) represented by the allocation node. Remember, an allocation node may just as well stand for a single object or for a set of objects, all created using the same program statement. There is no way of knowing, and it is in fact irrelevant in the context of pointer analysis.

- **assignment edges**: an edge between two variable nodes, representing an assignment of one variable to another. Since assignment edges represent assignments, they allow all elements in the points-to set of the source node to flow into the points-to set of the destination node. Therefore, an edge from $r(q)$ to $r(p)$ indicates that there exists an assignment $p := q$ in the program (we use $r(x)$ to indicate the node representing the program variable x). So, all the elements in the points-to set of $r(q)$ will have to

appear in the points-to set of $r(p)$ as well. Or, in terms of subsets, the points-to set of $r(q)$ will be a subset of the points-to set of $r(p)$.

- **store edges**: these edges link a variable node to a field reference node. An edge from $r(p)$ to $r(q.f)$ is used to represent assignments of the type $q.f := p$. By contrast with assignment edges, store edges will not determine a flow of pointers from the points-to set of the source node to the points-to set of the destination node, given that the destination node is a field reference node in this case, and that field reference nodes do not have points-to sets associated with them (remember, they only have a representation function). To explain how elements flow along store edges, let us come back to our example.

  We had $r(p)$ as the source node and $r(q.f)$ as the destination node. To refresh the reader's memory, a field reference node (which $r(q.f)$ is) is made up of a reference to a variable node ($r(q)$, in this case) and a field name (f, in this case). A concrete field node is made up of a link to an allocation node and a field name. In this context, elements will flow from the points-to set of the $r(p)$ node to the points-to sets of all the *concrete field nodes* that associate elements from the points-to set of $r(q)$ with the field name f.

- **load edges**: load edges link a field reference node to a variable node. An edge from $r(p.f)$ to $r(q)$ is used to represent an assignment of the type $q := p.f$. Flow of elements along load edges is again different from everything we have seen this far. It is, in a way, the reverse process than that taking place in the case of store edges. For the edge from $r(p.f)$ to $r(q)$, elements will flow from all the points-to sets of concrete field nodes that associate elements from the points-to set of $r(p)$ with the field name f into the points-to set of $r(q)$.

Stratego graph representation We used dynamic rules in order to represent the nodes and the edges of the graph. Dynamic rules representing the nodes bind a unique identifier for the node to the points-to set for that node. We use as unique identifiers the identifiers with which we have annotated the memory locations and variables of the program beforehand (we have discussed this in section 4.4.1). For allocation nodes we use the memory location identifier, for variable nodes we use the variable identifier, for field reference nodes we use the variable identifier - field name pair and for concrete field nodes we use the memory location identifier - field name pair.

Edges are also represented using dynamic rules, which bind the unique identifier for the source node to the unique identifier for the destination node. The identifiers used are the same ones as in the case of nodes. Since the same source node can be connected to multiple destination nodes, we use the mechanism of appending right hand sides to a dynamic rule with the same left hand side, and later using `bagof-DynRule` to retrieve all the right hand sides.

As a side note, for supporting this type of representation, we needed to add an additional strategy to the dynamic rules library, `dr-all-keys(|rulename)`. This strategy returns all the keys (left hand sides) defined for a dynamic rule. The syntactic sugar variant of this strategy is `all-keys-`L, which is generated automatically for each user-defined dynamic rule L. We use `all-keys-`L when we need to process all nodes or all edges of a certain type. This is required during the next step, the propagation of points-to sets, described in

the following subsection. Without this strategy, the only way of retrieving all the nodes or edges we define during the graph building process was to store all the keys separately. This was obviously inconvenient.

Building the graph The idea behind how the pointer assignment graph is built is to add nodes and edges that represent all the assignments that we have discussed at the beginning of section 4.3.1. The creation is driven by the various types of variable declarations and assignments, which introduce nodes and edges in the graph, respectively. In the case of store or load edges, the field reference node involved may not exist. In that case, it is created on the spot. In all cases where this makes sense[5], we look at the type of the variables that we encounter, so that we do not consider any variables (local variables, method arguments, class fields, etc.) which have basic types (int or bool, in the case of TOOL) in our analysis. Such variables are not pointers, so they are not the object of our analysis.

Most of the process of adding nodes and edges should be clear from the description that we made earlier, therefore we will only focus here on the more delicate issues, while only quickly mentioning the obvious ones.

Each `new` statement or string constant determines the addition of an allocation node, each variable declaration or method argument declaration triggers the addition of a variable node, each field access generates a concrete field node. Static fields generate variable nodes. As explained, allocation edges, assignment edges, store edges and load edges are created for statements that follow the patterns $p :=$ new Class(...) (or $p :=$ "..."), $p := q$, $q.f := p$ and $q := p.f$, respectively.

The more interesting part comes when we have to handle method calls. We will unify constructor calls and method calls in this discussion, since their handling is similar. The first thing that we need to do for a method call is find out what method bodies may end up being executed as a result of that method call. We use the infrastructure that we have already discussed in section 4.3.2 in order to get a list of possible target method definitions. Such method definitions include, among others, the formal arguments and the body of the method. For each method definition that is returned, we perform the following steps:

1. first, we create variable nodes for the formal method arguments and assignment edges between corresponding actual and formal arguments. If the method call is `obj.m(a, b, c)` and the method is defined with the formal arguments `x`, `y` and `z`, then assignment edges are drawn from $r(\texttt{a})$ to $r(\texttt{x})$, from $r(\texttt{b})$ to $r(\texttt{y})$ and from $r(\texttt{c})$ to $r(\texttt{z})$.

2. if the method called is not static, we need to handle the special `this` pointer, which is a pointer to the target object of the method call. So, during the call `obj.m(a, b, c)` from above, `this` will be an alias of `obj` (i.e., it will point to whatever `obj` is pointing). This is, in effect, equivalent to a (virtual) assignment `this := obj`, which means that we need to create an assignment edge between $r(\texttt{obj})$ and $r(\texttt{this})$. Before we can do this, however, we have to create a variable node for the special `this` pointer. If you recall, we annotate each method's `this` pointer differently, which means there will be a distinct nodes representing `this` for each different method. In the case of static methods, we simply skip this step.

[5] Some examples are assignments involving variables with basic types, passing of arguments with basic types or returning values with basic types from methods.

Handling of the special `this` variable differs in the case of constructor calls, since in constructor calls we do not have a target object which to assign to `this` (as we do in other method calls). However, due to our simplification, the result of any constructor call is bound to be assigned to a variable. It is this variable that will point to the newly created object, so this is the variable that will take the place of the target object from the normal method calls. As an effect of this observation, we create an assignment edge from the node representing the variable to the node representing `this` in the constructor body.

3. the third step is simply running the pointer analysis over the body of the method, which does not imply any particular difficulties.

4. the last step is only necessary when the result of the method call is assigned to a variable (i.e., `x := obj.m(a, b, c)`). In this case, we collect all the return statements that appear in the method and extract the returned variables out of those statements. To accommodate the new aliases that are created, we need to add an assignment edge between each of the nodes representing those returned variables and the node representing the variable that receives the result of the method call. In this way, we ensure that we properly cover any of the potential assignments that might take place (of whichever variable is returned at run time to the receiving variable).

There is another issue related to the handling of method calls, which comes up with virtually any interprocedural analysis. We need to avoid, during the analysis, the creation of the infinite loop that can be triggered by direct or indirect recursion in the target code. Given that the pointer analysis is a context insensitive analysis (so we only need to analyze each method once, regardless of the context in which it is called), we can resort to a simple trick. We can cache the information we need for each method. In addition to giving us faster access to the information we need for each method call, this solves the infinite loop problem implicitly as well. How? Well, as soon as information about a method is cached, we no longer do anything (except retrieve the cached information), so we avoid the danger of restarting the analysis of the same method. All we need to ensure is that we cache the method before we start the analysis of its body. With many analyses this is not possible since we need the result of running the analysis over the method body in order to have something to cache. In the particular case of pointer analysis, however, all we need to cache are the formal arguments and the variables returned by the method. This information is available immediately, without having to run pointer analysis over the body of the method first.

The last interesting approach that we want to mention is related to the handling of exceptions. Exceptions introduce the difficulty that the variables thrown by throw statements (if you recall, we simplify the code beforehand in order for all throw statements to throw simple variables, not expressions) are caught by some catch clause and given a new name. What we would need, then, is to create assignment edges between the nodes representing the variables thrown and the nodes representing the variables declared in the matching catch clause. Instead of doing this, we took the more simplistic approach of creating a special variable node that represents all exceptions. Every time we see a throw statement, we create an edge from the node representing the thrown variable to the special exception node and every time we see a catch statement, we create an edge from the special exception node to the node representing the variable declared in the catch clause. In this way, we effectively make all exceptions of the program aliases of each other. For handling exceptions,

we adopted this rather conservative solution (also used in [18, 19]) because variables used in exception throwing/catching are rarely important for optimizations. Therefore, putting in a lot of effort for obtaining very precise alias information regarding them is unlikely to pay off.

As a final mention, we also make sure to run the pointer analysis over the static blocks defined in a class, as soon as that class is first encountered in the code. This happens either when an instance of that class is first created or when a static field of that class is accessed.

4.4.3 Propagation of points-to sets

In the previous step we have created a pointer assignment graph which is a special representation of all the assignments in the analyzed code. However, if variables would be aliased only by direct assignment, things would be pretty easy. In reality, variables may end up being aliased also by transitivity. Because of this, this step in the pointer analysis will propagate memory locations from one points-to set to another according to the rules we explained when discussing the types of edges that exist in the graph. Propagation takes place until a fix point is reached.

Initially, all the points-to sets are empty. We start propagation by putting all allocation nodes in the points-to sets of their successor (all allocation nodes have one successor only). Five different algorithms are presented in the original work for propagating the elements of points-to sets along the edges: an iterative one, a worklist one (with an incremental version) and an alias edge one (with an incremental version). We chose to implement the worklist propagation algorithm (the second fastest, slower only than its incremental variant) and, for verification purposes, the simple iterative propagation algorithm.

Before explaining the algorithm, let us again remind you how the four types of edges are handled in any version of the propagation algorithm. The easiest are allocation edges: we simply put the allocation node from the origin of the edge in the points-to set of the variable node, before running the fix point iteration of the algorithm. Assignment edges are also easy to handle: we add all the elements in the points-to set of the node at the origin of the edge to the points-to set of node at the destination. Store edges ($p \rightarrow q.f$) are a bit more tricky. A store edge indicates that all the elements in the points-to set of p theoretically need to be in the points-to set of $q.f$. However, $q.f$ is a field reference node, so it represents the field f of all objects that q may point to. This means that what we actually have to do is propagate the elements in the points-to set of p to the fields of all these objects. For this purpose, we need to introduce one concrete field node for each allocation node from the points-to set of q. Afterwards, we can propagate the elements from the points-to set of p in the points-to set of all these concrete field nodes. The reverse process happens during the handling of load edges ($p.f \rightarrow q$): elements from the points-to set of all the concrete field nodes associated with any allocation node from the points-to set of p will flow to the points-to set of q.

First, we have implemented the simple iterative propagation algorithm as a reference implementation. This algorithm simply applies the propagation rules we have explained above for all assignment edges, store edges and load edges in turn, in an repetitive fashion, until no more changes are produced. At that point, the fix point is reached, and the propagation is completed. The iterative approach is simple, but also very slow. In order to boost the performance of the propagation, we need to add some complexity to the algorithm in order

to eliminate useless traversal of edges. The result is the worklist propagation algorithm, which we discuss in the remainder of this section.

What makes the worklist propagation algorithm perform better than the iterative propagation algorithm is the use of a list in which we save the variable nodes whose points-to sets have been changed by a previous propagation step. By maintaining this worklist, we no longer have to traverse all the edges of the graph over and over again (as we had to in the iterative version), but only consider those that involve nodes that appear in the worklist. This is the basic idea of the worklist version. The exact details we discuss below.

The initialization phase of the worklist propagation algorithm, as with the iterative one, involves propagating allocation nodes along allocation edges to the points-to sets of variable nodes and adding all these variable nodes with non-empty points-to sets to the worklist. Thereafter, an iterative process starts, which constantly picks a node from the worklist and processes it. The worklist is updated with variable nodes whose points-to sets get updated during the processing of other nodes. Processing of a variable node involves handling the following types edges: assignment edges and store edges originating from it (to propagate the new elements in the points-to set of the processed node), store edges ending in a field reference node that has this node as its base (to update the concrete field nodes associated with the allocation nodes that were added to the processed node's points-to set) and load edges originating from a concrete field node that has this node as its base (to propagate the points-to sets of the concrete field nodes associated with the allocation nodes that were added to the points-to set of the processed node).

In addition to the basic handling, there are also some special cases that have to be treated additionally. One such case is when q is in the worklist, but p is not, p has allocation node a in its points-to set, and q also had the same a added to its points-to set. This makes p and q possible aliases, so any updates done to the concrete field nodes based on a as an effect of processing store edges ending in $q.f$ (during normal processing of q) should also propagate through the load edges originating in $p.f$ (since p points to a as well). However, this does not happen, since p is not in the worklist. Therefore, we need to additionally process the load edges outside the main loop. The same case can also happen in reverse, so we need to similarly process the store edges as well.

The following is a verbatim copy of the worklist algorithm, as it is summarized in [18]:

```
process allocations
repeat
  repeat
    remove first node p from worklist
    process each assignment edge p -> q
    process each store edge p -> q.f
    process each store edge q -> p.f
    process each load edge p.f -> q
  until worklist is empty
  process every store edge
  process every load edge
until worklist is empty
```

The only aspect left unexplained is what nodes get added to the worklist during the iteration. We enumerate these below:

- when processing the assignment edge $p \rightarrow q$, q gets added to the worklist if any elements are propagated from the points-to set of p to that of q.
- when processing store edges, whether of type $p \rightarrow q.f$ or of type $q \rightarrow p.f$, we only (potentially) modify the points-to sets of concrete field nodes. Since we only keep variable nodes in our worklist, there is nothing we can add here. The same holds for when we process all store edges in the outer loop.
- when processing the load edge $p.f \rightarrow q$, we add q to the worklist if any elements are propagated into its points-to set. The same holds for when we process all load edges in the outer loop.

4.4.4 Supplying alias information

The propagation of the points-to set concludes the pointer analysis algorithm, as described in [18, 19]. The conclusion of this step leaves us with a graph that contains nodes with associated points-to sets. However, this information in this form is of little use since users are usually interested not in what memory locations a variable (object field, method argument) may point to, but in what the aliases of that variable (object field, method argument) are. In order to transform the points-to information into alias information, an additional transformation is necessary.

The point of this transformation is to generate a data structure which can be queried using a variable name (or, more realistically, an annotation which uniquely identifies a variable, a method argument or an object field) in order to get a list of variable names (or, again, annotations) in return. As you have probably guessed by now, we encode this data structure as a dynamic rule that binds a variable annotation to a set of variable annotations. If you recall, only variable nodes and field reference nodes represent actual variable expressions that appear in the program. Concrete field nodes do not have an equivalent. They merely store the memory locations that can be pointed to by fields of nameless objects. The user, however, is only interested in the variable expressions that appear in her program, since her analysis will be performed only in terms of those expressions. As such, the alias lists we return to the user only have to contain annotations that uniquely identify local variables, method parameters or field references.

The idea of the conversion can be explained fairly easily if, for the time being, we only consider variable nodes. Each variable node points to a number of memory locations. In order to find out which variables are potential aliases, we need to see which variables have the same memory locations in their points-to sets. For this, we can create a structure that maps each memory location to all the variables that point to it. This structure can be filled in by considering each variable node and mapping each memory location in its points-to set to the variable the node represents. In the end, each memory location will end up being mapped to a list of variables. Once this is achieved, we know that all variables that appear in the same list of a memory location are aliases of one another. So, we iterate over each list and we create our final structure: the one that maps each variable to all its aliases. For each list we process, we add the entire list to the list of aliases of each of its elements.

Now we can return to the general case and explain how field reference nodes and concrete field nodes fit into the picture. What we have to do is find out all the memory locations that a particular field reference node may point to, indirectly, through concrete field nodes.

A field reference node is made up of a (link to a) variable node and a field name. We look at all the memory locations in the points-to set of the variable node and, by matching them with the field name, we get the concrete field nodes that we have to examine. (Remember, the concrete field nodes are pairs of a memory location and a field name.) In this way, we obtain a large points-to set for each field reference node, which we can then treat just as if it had belonged to a variable node. So, we traverse the points-to set and map each memory location in the set to the field reference, just like we did for variable nodes. Thus, the structure mapping memory locations to variables has been extended to map memory locations to field references as well. The last step is exactly the same, except that we will not only have variables in the lists that are mapped to memory locations, but also field references.

In this way, we have obtained the alias information we needed: given a variable or a field reference (or, rather, the annotations that uniquely identify them), we can supply the list of its alias variables and field references.

Chapter 5

Extending and customizing transformations

In the last part of our work, we set out to perform some initial experimentation with two different concepts that will most certainly constitute a direction of further research: extensible and customizable transformations. As we were discussing in the introduction to this thesis, the road to implementing open compilers in Stratego has to go through the intermediate stage of discovering ways to make Stratego transformations (1) customizable and (2) extensible. As soon as a proper model for doing this materializes, development of open compilers proper will have had all its prerequisites met. We hope that our work described herein will lead to a better understanding of the issues involved and serve as valuable example for future efforts.

To establish how this part fits in with the rest of our thesis, we remind the reader that the first two parts of the thesis have dealt filling in two very important gaps in the support that the Stratego platform offers to its users for writing data-flow transformations. This last part comes not to provide further support for writing data-flow transformations, but rather to illustrate approaches for making data-flow transformations customizable and extensible. The user of Stratego will benefit from this last part more in the form of models that she can adopt, and less in the form of reusable code.

Concretely speaking, we have conducted one experiments. The first one was aimed at showing how transformations can be written so that the user may customize them without writing any Stratego code. The idea of this experiment is largely the same one used by the Broadway system (see section 5.2.3), however transposed into a Stratego setting. In short, we provide an annotation language that the user can write method annotations in, and with the help of which she can influence compilation. The system allows the user to make compilation optimization decisions on the basis of typestate information. Typestate is propagated by the system based on rules specified by the user.

Our second experiment dealt with extending transformations. Our goal was to extend the TOOL language that we used in the previous chapter so that it would support arrays. As a consequence, all the transformations written for TOOL (such as type checker, simplifier, pointer analysis and so forth) had to be extended as well so that they properly support arrays. What we tried to achieve was the extension of both the language syntax and the transformations over the language with minimal intervention on the original syntax definition

and transformation code. While we have not managed to achieve zero intervention (which is the eventual goal), we believe we came pretty close. The results, in our opinion, make the prospect of future work in this direction very promising.

In this chapter to provide some further discussion on the concepts of by extensible and customizable transformations as well as a more detailed description of our experiments.

5.1 Overview

We perceive extensible and customizable transformations as two roughly separate issues at this point, with the only connection between them being that they are both needed in order to sustain open compiler development. Since there is little other common ground that the two share, our discussion of them will be separated into two distinct parts, one that deals with extensible transformations and another one that deals with customizable transformations. The fact that customizable transformations may be extended just like any other type of transformation brings no added added value to the overall understanding, so we see little point in pursuing this track.

The idea of *extensible transformations* is part of a more general setup of an ongoing project called Transformations for Abstractions, led by Eelco Visser, and will likely be researched more extensively within this project. The idea of *customizable transformations*, on the other hand, comes from a trivial observation that an essential feature of open compilers is the ability to involve the user in the compilation process and particularly in the optimization process. The following two subsections discuss these ideas in turn.

5.1.1 Extensible transformations

Transformations for Abstractions is a recent project proposal of Eelco Visser, which aims to investigate a number of issues related to how transformations written for general purpose programming languages can be (preferably automatically) extended to cover domain specific abstractions introduced by domain specific languages. The project will also be conducted within the confines of the Stratego platform and will build on the efforts concerning MetaBorg [8] (also see section 1.1 for a short presentation). While MetaBorg focuses on the extension of the *syntax* of general purpose languages with domain specific ones, it does not propose any methods for doing the same with the *transformations* already defined for the general purpose language. The approach taken by Transformations for Abstractions will be likewise guided by existing results that either tried to parameterize transformations so that the same generic transformation can be specialized to obtain various concrete transformations [20] or tried to reuse basic rewrite rules to recreate the higher level transformation in a version that covers the abstractions as well.

A number of aspects will be investigated throughout the project on Transformations for Abstractions, as defined in the project proposal: definition of domain abstractions, open extensibility of transformations, design of open transformations, independent extensibility of transformations and automatic derivation of transformation extensions. These are goals of a project that will span over three years, so we by no means tried to address them all within this thesis. In retrospect, our work may be viewed as a simple design model for extending transformations.

It is common practice nowadays to only loosely integrate domain specific abstractions with their host language. This practice has its roots in the lack of technologies that allow the extension of the syntax and compiler of a language. While extending libraries is easy enough for even the most inexperienced of programmers to be able to do it with their eyes closed, extending the syntax (and, implicitly, the compiler) of a language implies the creation of a team which makes this its full time job. Because of this, most domain specific abstractions are encoded as libraries, while only a small number of them actually take the form of a DSL.

The poor integration of even this small number of DSLs with their host language leads to various problems, ranging from missing optimization opportunities to difficult interaction (in both directions) between the code written in the DSL and the rest of code. If we are to change this state of affairs, we need to work towards allowing easier extension of a language's syntax and compiler. Easier extension may only be achieved by designing and producing compilers that allow seamless integration of independent modules that contain new syntax and compiler extensions. A DSL extension comes with new syntax (that of the DSL) and with code that has to be run in order to compile the DSL syntax to the host language syntax. The writer of this extension needs to be able to plug in the syntax and code that define the extension into the core compiler such that it is automatically available (the syntax) and automatically executed (the code) without having to change the core compiler in any way. The purpose of our example is that of providing an example of how this integration can be approached.

5.1.2 Customizable transformations

Customizable transformations are also about allowing users to extend transformations, but in a somewhat different way than with extensible transformations. In fact, the users are not really changing the transformations (as it was the case with the extensible ones), but they are feeding them with information which otherwise would have to be inferred or, even worse, assumed unknown and usually handled in a conservative manner. This information that the users are feeding the transformation with is especially useful with optimizing transformations, as discussed in the introductory section of this thesis.

In some cases, the transformations that we want to allow users to customize make no sense in the absence of user information. In these cases, we are not trying to make a transformation perform better with the aid of user input, but we are trying to set up a context in which a user may define her own optimization rules, usually in a more accessible language than that used for writing the compiler, and have them enacted by a language-provided generic transformation. This would be the transformation that the user is customizing or, perhaps more appropriately, *driving*. In fact, our work on customizable transformations is an example of this type of use. We believed that this would serve an illustrative purpose better than, say, an example in which we would show how to provide information to, e.g., constant propagation about which argument of a method is used in a read-only manner and which is not.

Our example of a customizable transformations is inspired from work on typestates [23] and active libraries (especially in their Broadway incarnation [12, 13]). The idea behind *typestates* is to allow the implementation of transformations that can check and even impose various semantic properties at compile time. The paper we discuss in the related work section focuses on three such properties: not using a variable that has not been

initialized, not dereferencing a pointer which has not been assigned a memory location yet and not exiting a program (on any path) without deallocating all allocated memory. The analysis that allows this requires knowledge about the so-called typestate of each variable, so that it can verify whether or not certain preconditions involving these typestates are met. It also needs information about how to propagate and change the typestates along the examined paths of the program. Broadway adopts a similar approach, albeit following a different purpose. Its purpose is to allow for user-customized optimizations which are based on properties about the program that can be computed at compile time. The properties and the typestates are similar concepts. Both typestates and Broadway will be discussed at more length in the related work section.

5.2 Related work

We discuss related work regarding *extensible transformations* in the section on Transformations for Abstractions and related work regarding *customizable transformations* in the sections on typestate and active libraries.

5.2.1 Transformations for abstractions

An experiment in the direction of Transformations for Abstractions using Stratego has been documented in [27]. A tiny core language, consisting only of variable assignments, function calls and blocks of statements is considered for extension with a number of modules introducing various features in the language, like integer arithmetic, expression blocks or regular expressions. Each of these implies a syntax extension of the core language *and* extensions of the various transformations defined for the core language. The paper concentrates on the latter of the two, which it illustrates with a number of different transformations: local to local, local to global and global to local transformations, context sensitive transformations, data-flow transformations and combinations thereof.

Desugaring of user-friendly syntactic constructs to basic constructs of the language is a simple local to local transformation which can be *extended* by simply defining new cases for a rewrite rule which is applied by a fix point strategy. A local to global transformation like assimilation of the embedded language of regular expressions can be *extended* using dynamic rewrite rules for collecting transformations and application of the dynamic rewrite rules for making the global changes. *Extension* of context-sensitive transformations like bound variable renaming is also achieved with the aid of dynamic rewrite rules, but used in a different way than in the previous case, i.e. to propagate context throughout the transformation. Further transformations like evaluation, constant propagation and function specialization similarly employ interacting strategies and dynamic rewrite rules in order to achieve *extension*. Usually, the core language defines a basic strategy which first invokes a specialized strategy and, if that fails, falls back to generic handling. Extensions *extend* the definition of the specialized strategy in order for their customized behavior to be included in the overall transformation.

In essence, Visser collects in [27] a number of models for modularizing transformations and presents them from the novel perspective of extending a base transformation with special cases introduced by language extension. This work provides valuable experience and sufficient validation in order to use it as a starting point in our own experiments.

5.2.2 Typestate

Typestate was first mentioned as a proposed programming language concept in a paper from 1986 by Strom [23]. It is presented as a mechanism complementary to that of type checking, meant to ensure better "security" of programs at compile time. The idea of typestates is to help prevent so-called *nonsensical* sequences of statements in a program (i.e., those sequences that are syntactically well-formed and type correct, but semantically undefined). The examples of such sequences in the paper are limited to the use of uninitialized variables or pointers, but generally speaking the concept is useful for tracking other properties as well (such as whether objects are locked and unlocked properly).

As described in [23], a set of typestates is defined for each type in a language, with the elements in the set forming a lower semilattice. An example of such a semilattice is one containing the values $\perp$ and I (initialized), with $\perp \prec I$, that can be used to track whether a variable has been initialized or not. Typestates are associated with variables of a program, the typestate of each variable being drawn from the set of typestates that is defined for the type of the variable. Each set of typestates contains the typestate $\perp$, which is the initial value for all variables of the program.

Each program operation that involves variables may determine a change in the typestate of those variables. These changes are specified through preconditions and postconditions for all of an operation's variables (e.g., the `Thread.start` operation in Java changes the typestate of a thread variable from "not started" to "started") . Operations refer to both predefined operators and library functions. As such, we can annotate all the functions in a library module with preconditions and postconditions, so that typestate information can be propagated over calls to such functions as well. In addition to pre- and postconditions, we also have to be able to specify (either as part of the language that supports typestates itself or as user-defined information) how any typestate can be coerced to the one immediately below it in the lattice. In this way, the compiler can try to add such statements wherever it would be correct and necessary to do so in order to ensure typestate correctness (e.g., if an operation requires that one of its operands is not allocated, but the operand's typestate indicates that it is, then a coercion from I to $\perp$ could be automatically introduced).

After all variables are initialized to $\perp$, a typestate propagation algorithm can be applied in order to compute the values of variables' typestates at each program point. An analysis is carried out along with the propagation process in order to determine if the entire program is typestate correct or at least typestate consistent. A typestate correct program is one in which the preconditions of all the program statements can be met across all control flow paths *and* all variables are in the typestate $\perp$ at the end of the program. The notion of typestate consistency refers to a program which can be converted to a typestate correct one by introducing additional code to lower the typestate of variables at certain points. By lowering the typestate of a variable we are essentially deallocating resources, so a typestate lowering from I to $\perp$ in our example lattice could be achieved by releasing the memory held by the variable (using a `free x` in C++, for example). In order for the analysis to be able to perform this coercion, it is required that information about how this can be done for each type is available, as explained earlier.

This notion of typestate is a bit restricted, since it is always associated with a concrete program variable. We will see this same notion extended in Broadway to more generic properties with which library functions can be annotated.

5.2.3 Active libraries

The concept of active libraries was introduced in [9] as a manifestation of generative programming. Generative programming is a new paradigm of creating particular software systems through a process of specification, derivation and composition which draws from a common set of elementary components forming a software family. The idea is to shift the system developer's work from an environment defined by general-purpose programming languages and traditional APIs to one of high level specifications expressed in domain-specific concepts. Such specifications define how (already existing) components of a software family should be modified and composed in order to yield a unique end-system. All the steps leading from specification to realization are to be conducted automatically, by a generative programming engine. This novel way of creating software, although promising, is still a long way from being a practical reality, since developing the infrastructure for achieving even part of the goal is quite an endeavor.

And this brings us back to the relation with the active libraries. As discussed, generative programming requires a natural way for a user to specify domain-specific concepts. Resorting to (multiple) domain-specific languages is one possible answer, while turning to active libraries is the other. Combinations of the two are also possible. Quoting [9], active libraries can be used to cover a lot of ground through the capabilities they provide:

> They may generate components, specialize algorithms, optimize code, automatically configure and tune themselves for a target machine, check source code for correctness, and report compile-time, domain-specific errors and warnings. They may also describe themselves to tools such as profilers and debuggers in an intelligible way or provide domain-specific debugging, profiling, and testing code themselves. Finally, they may contain code for rendering domain-specific textual and non-textual program representations and for interacting with such representations.

Of the three types of active libraries discussed in [9] (those that extend a compiler, those that extend some sort of integrated development environment with domain-specific support and those which act as meta programs) and summarized in the quote above, we are particularly interested in the first type. Active libraries integrated with Stratego are most likely to fit the paradigm of a compiler extension rather than any of the other two. Additionally, Stratego already has extended support for inspection and transformation of the analyzed source code, thus providing a solid base for extension in the aforementioned direction.

Broadway

Our rather extended presentation of Broadway from this section is meant to set the background for one of the experiments of this chapter and, secondarily, to illustrate the concept of active libraries with a concrete example. Given that Broadway was a large project in its own, not all the aspects of Broadway presented herein have found their way into our customizable transformation example. Nevertheless, we believe to have extracted some of the more important parts and then combined them with ideas related to typestate propagation in order to yield a interesting example.

Broadway is a project developed by Samuel Z. Guyer and Calvin Lin at the University of Texas at Austin. Broadway consists of an optimizing compiler and an annotation language

which support a novel technique for optimizing software libraries. This technique is validated experimentally by applying it to the (production-quality) PLAPACK parallel linear algebra library. The authors claim that the observed performance increase of 26% for large matrices and 195% for small ones was made possible by having the compiler use the additional information given through annotations. Both the technique and the results are discussed at length in [12, 13].

The Broadway system is geared towards the user who is in no way a professional compiler writer, but rather a regular library developer and in the same time domain expert. This type of user commonly has a thorough understanding of how a certain library is best used in various circumstances and would like to be able to put this knowledge to *systematic* use. To this end, Broadway offers a number of features through its annotation language.

Compilers have a hard time performing many of the traditional optimizations (constant propagation, dead-code elimination and the like) when library calls are involved. One reason for this is that often the library's source code is not available for inspection. In such cases, optimization algorithms have to "assume the worst" in order to ensure the soundness of the optimization process. Such conservative assumptions lead inevitably to missing opportunities for optimization. And even when source code is available, it can be the case that code from other (unavailable for inspection) libraries is called from the library functions. Another hindrance to compilers is that analysis of source code is not always enough for making certain decisions. And sometimes it can be the sheer cost (in terms of time consumption) for getting all the information needed for an optimization that makes it infeasible. Broadway's annotations allow the user to aid this optimization processes by letting her specify variable dependency information for each library function. The compiler can then incorporate this information in its data-flow and pointer analysis and use it to make more aggressive optimizations.

The dependency information is expressed in a simple language. The pointer structure is specified using the `on_entry` and `on_exit` annotations, where the "`-->`" can be used to represent the "points-to" relation between input objects and internal library structures. Two additional annotations, `access` and `modify` allow the library developer to specify the "uses" and the "defs" of the library routine. The values listed in the `access` set are those which are only read by the routine, while the ones listed in `modify` are those whose value is also changed by the routine. With this extra information, the compiler's analysis will usually yield a larger amount of optimization opportunities.

The second type of improvement that Broadway proposes is the truly domain specific one. Libraries are used by different categories of users, ranging from beginners to experts. Most, if not all, libraries have their own quirks in terms how they should be best used, which the user has to be aware of if she is to achieve both performance and safety in usage. But since users are so diverse, it is unlikely that they will all manage to gain full understanding of how to utilize the library. Therefore, it is preferable that a mechanism exist to smooth away this potential lack of expertise. What Broadway does is propose such a mechanism.

Concretely, the annotation language provides a means to specify and track any number of *properties* across library calls. A property is created using the `property` keyword, and all its possible values are specified. For example, the following could be used to track whether or not an array is sorted (including information about how it is sorted, ascendingly or descendingly):

```
property Sorted : { Yes { Asc, Desc }, No }
```

Once such a property definition is established, its value for any particular input object and/or object an input object points to (either directly or indirectly) can be maintained based on a set of user-defined rules. For this purpose, the `analyze` annotation has been introduced. An analysis allows non-conditional or conditional assignment of values. If we were to write an annotation for a hypothetical `array_reverse` routine, here is how it would look like (the name of the argument to the routine is assumed to be `array`):

```
analyze Sorted {
  if (Sorted : array is-exactly Asc) { array <- Desc }
  if (Sorted : array is-exactly Desc) { array <- Asc }
  default { array <- No }
}
```

Finally, maintaining these properties is useless if they are not part of the optimization process. The way Broadway achieves this is through the annotations for actions. There are three actions which can be specified by the user: code replacement, inlining and reporting. These actions can be guarded with conditions based on the property values which result from the analysis process. The conditions are evaluated at each call site of a library routine, and it is the particular call to the routine that either gets replaced or inlined. The compiler ensures the syntactic correctness of the expanded code. To illustrate, assume that we have two routines, `array_print` and `array_print_sorted` which print an array as is and print an array after first sorting it, respectively. Then, the annotation for the `array_print_sorted` routine would look as such:

```
when (Sorted : array is-exactly Ascending) replace-with
%{ array_print(array); }%

when (Sorted : array is-exactly Descending) replace-with
%{
  array_reverse(array);
  array_print(array);
}%
```

Based on all these annotations, the Broadway compiler takes an input program, performs a data-flow and pointer analysis, using all the extra information and eventually runs a number of traditional optimizers as well as all the user-specified transformations and reporting.

5.3 Customizable transformation: typestate propagation

Our experiment with customizable transformations consisted in creating a simplified Broadway-like system which should allow users to play around with the concept of active libraries. The reader should be aware, however, that the main goal of this experiment has been exploring some of the mechanisms used for writing customizable transformation and not so much the creation of a complex, robust system for typestate propagation itself. In other words, you should regard the system which we have created and which we describe in the first subsection simply as an "excuse" for experimenting with customizable transformations.

5.3.1 Description of transformation behavior

Similarly to Broadway, the purpose of our system is to allow users to annotate methods with information that can be then used by our transformation in order to either give warning or even error messages or to modify the code *that calls the annotated methods*. The user's part consists of providing two different types of information: typestate propagation rules and code rewrite or message generating rules. The system's part is to propagate typestate based on the user-rules, evaluate user conditions for triggering code rewrite or message generation and, if the case, perform the rewrite of code or the generation of error or warning messages. We proceed to explaining this system in more detail by presenting the annotation language and an example of its use.

Annotation language

The object-oriented language that we have used as an object language here is the same toy language TOOL that we have developed for implementing pointer analysis. A description of it can be found in section 2.4. In order to let the user annotate TOOL methods, we obviously had to introduce additional syntax into the language, syntax that would be usable strictly before a method definition in order to annotate that particular method. We called the resulting language ATOOL (standing for Annotated TOOL).

Integrating additional syntax in an existing syntax definition is fairly easy to achieve with Stratego, generally speaking, given the modularity of the SDF system. The only difficulty that we have encountered was related to the fact that while the annotation language had to be integrated in TOOL, TOOL also had to be integrated in the annotation language, because annotations may also include snippets of TOOL code (with some additional meta variables) following the `%replace` directive. This ciclicity made things less straightforward that we would have liked, but in the end a solution was possible, with the aid of an SDF tool that allows prefixing all the non-terminals of a grammar with some prefix in order to obtain a prefixed version of that grammar. In this way, we included a prefixed version of TOOL in the annotation language and then the resulting language into the original TOOL grammar to obtain the ATOOL language.

We present the syntax of the annotation language in figure 5.1. What you can see there is only the annotation language, not the entire ATOOL language. ATOOL is simply TOOL in which method definitions may be annotated with annotations starting with `@spec`. To guide you through the grammar, a method specification (or annotation) may consist of a set of *transitions* (these would be the typestate propagation rules) and/or a set of *rules* (these would be the code rewrite and message generation rules). A *transition* consists of an optional *condition* and an *property assignment*. A *rule* is composed of a *condition* and of an *action*. Before we can explain *condition*s, *property assignment*s and *action*s, we first need to explain properties.

The properties we are talking about are in fact typestates and represent user-defined values that are associated with variables of the program. Each property consists of a name and a number of values, organized in a semilattice. The bottom value represents the unknown value for that property. The other values are increasingly more specific versions of the the values that precede them in the lattice. The syntax we use for representing properties is similar to that of Broadway, i.e. `property PropertyName = { v1, v2, ..., vn },`

<method-spec>	::=	"@spec" "{" (<transitions> | <rules> | <transitions> <rules>) "}"
<transitions>	::=	"@transitions" "{" <transition>+ "}"
<rules>	::=	"@rules" "{" <rule>+ "}"
<transition>	::=	(<cond> ":-")? <assign> ";"
<rule>	::=	<cond> ":-" <action> ";"
<assign>	::=	<prop> "=" <id> | <prop> "=" <prop> | "%unset" <prop>
<action>	::=	"%error" <string> | "%warning" <string> | "%delete" | "%replace" "{" <TOOL-statement>+ "}"
<cond>	::=	<cond> "|" <cond> | <cond> "&" <cond> | "!" <cond> | "(" <cond> ")" | <prop> "==" <id> | <prop> "%>" <id>
<prop>	::=	(<this> | <id> | <ret-val> | <field-access>) "#" <id>
<field-access>	::=	(<this> | <id> | <ret-val> | <field-access>) "." <id>
<this>	::=	"this"
<ret-val>	::=	"$$"
<id>	::=	sequence of letters, digits and underscores
<string>	::=	sequence of characters enclosed between " and "

Figure 5.1: Syntax of the annotation language.

where each value $\mathtt{v}_i$ may be either an identifier or an identifier followed by another list of values.

To illustrate, the code on the left in figure 5.2 defines the property called `FileStatus` which defines the semilattice on the right. Variables that refer to files may be in the typestate `Open` or `Closed`, those that are `Open` may be either open for reading (`OpenRd`) or for writing (`OpenWr`) and those that are `OpenWr` may alternatively be in the more specialized typestate `OpenAp`, open for appending.

Selecting properties of variables in the annotation language is realized using the # selector. var#$prop$ selects the value of property $prop$ associated with the variable var. Variables that may be used in *condition*s and *property assignment*s should only be from the set formed of the formal arguments of the method, the special `this` variable or the special `$$` variable. When propagating typestate, the properties of actual arguments will be used (in conditions or assignments) or changed (in assignments) where formal arguments are specified in the annotations, the properties of the target object of the method call will be used/changed where the special `this` variable is specified in the annotations, and the properties of the variable that receives the result of the method call will be used/changed where the special `$$` variable is specified in the annotations.

Turning back to the annotation language, *condition*s are comparisons of variable properties with predefined property values, possibly aggregated using boolean operators. The == operator indicates exact equality, while the %> operator indicates the property should be

```
property FileStatus = {
  Open {
    OpenWr { OpenAp },
    OpenRd
  },
  Closed
}
```

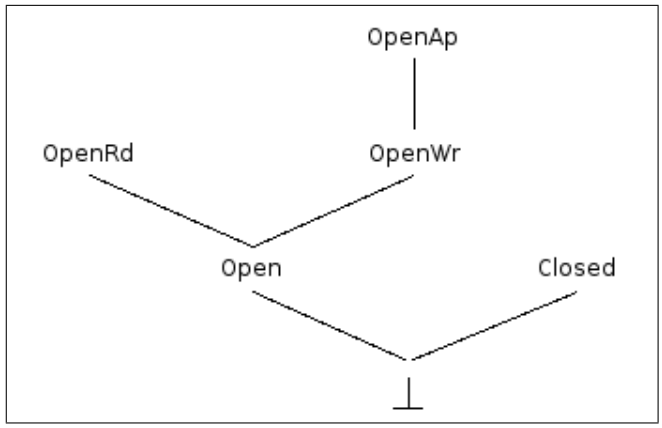

Figure 5.2: Example of property definition

either the exact specified value or any value that is a specialized version of the specified value (referring to our example, `x#FileStatus %> OpenWr` will evaluate to true if the `FileStatus` property of variable `x` is either `OpenWr` or `OpenAp`). *property assignment*s change or undefine the typestate associated with a variable. Finally, an *action* may take four forms. Two of the forms are for giving an error or a warning message, the third is for completely removing the method call which calls the annotated method and the fourth is for replacing the same method call with the code specified between { and } in the `%replace` directive. All normal variables in this code are assumed to be placeholders for some actual variables, so they will be replaced accordingly. The need for fresh variables can be indicated by the use of *meta variables* (represented by identifiers preceded by the $ sign), and this will trigger the generation of (properly matched) fresh variable names in the generated code. If a non-meta variable is present in the replacement code and does not match either of the formal arguments or the special `this` or `$$` variables, it will be left as-is, with the potential damaging effect of name capture.

Example

To help the reader better understand the annotation language as well as the way our system works, we turn to an example in this section. Although this example is quite simple, it nevertheless allows us to illustrate most of the features of the language and most of the system's behavior. Our example will define a basic `File` class that has some of the usual operations that can be found in any similar implementation. The actual implementation of the methods is irrelevant; we only focus on the annotations that come with the method definitions. This will provide the reader with a number of sample annotations. Then, in order to show how the system works as well, we will show a few snippets of code that use the methods of the `File` class and show the effect that our transformation has when run over them.

The first task that has to be performed by any library writer before turning to writing annotations for her library methods is deciding on the properties that will be used as typestate throughout the annotations. All properties have to be defined in a single file, together with their possible values and the ordering relationships between these values. For our example,

we only use two properties, `FileStatus` and `FilePos`, to keep track of the status of a file (open, closed, etc.) and the position of the pointer in the file. We define these two properties in the `properties` file (the one the system implicitly assumes to be containing the property definitions) as shown in figure 5.3:

```
property FileStatus = { Open { OpenWr { OpenAp }, OpenRd }, Closed }
property FilePos = { Beginning, End }
```

Figure 5.3: The contents of the properties file of the example

Now that the properties are established, we can advance to presenting the bulk of this example, which consists of the annotated `File` class, presented in figures 5.4, 5.5 and 5.6. We proceed to discussing a couple of the annotations presented in these three figures, while inviting the reader to browse through the rest of the example herself.

The `FileStatus` property of the objects that will point to instances of the `File` class is maintained quite easily. Upon creation, `this#FileStatus` is set to `Closed`, when opening a file it is set to one of the `Open`, `OpenRd`, `OpenWr` and `OpenAp`, depending on the `open` method called and then it is again set to `Closed` when the `close` method is called. By comparing the annotation for the `open(int)` method to that of, say, `openForWriting` one can realize how weaker (less information) and stronger (more information) typestate values come to be used. In the former case, we cannot be sure, at compile time, if the file will be opened for reading or for writing, so we can only assign the weaker `Open` typestate to `this#FileStatus`. In the latter case, we know that the file is opened for writing, so we can assign a more specific typestate, `OpenWr`.

Another interesting thing to notice is the use of conditions in typestate propagation. For instance, in the annotation for the method `open(int)`, before we assign the typestate `Open` to `this#FileStatus`, we first check if the existing typestate is not already more specific than `Open`. In the case that it is, we do not want to assign the (weaker) `Open` so that we do not unnecessarily lower the specificity of the typestate. However, this is only correct if we assume that a call to `open(int)` has no effect if the file is already open (which we do assume in this case).

Related to the fact that a call to any of the variants of method `open` does not do anything if the file is already open, we can now also explain the behavior or the `%warning` and the `%delete` actions. In the case of `open(int)` we choose to only give a warning to the user of the `File` class when she tries to call `open` on a file that is known to already be open. By contrast, in all the other variants of `open` (`openForReading`, `openForWriting` and `openForAppending`) we choose to remove the call altogether by specifying the `%delete` action to be executed if `this#FileStatus` has the value `Open` or a more specific version thereof. What action we perform is strictly our choice as designer of the active library. Note that here we take different actions for roughly similar cases strictly so that we can illustrate more diversity. In a real case, consistency in the types of actions that are performed would probably be preferred.

An example of how replacement of code can be performed can be seen by looking at the annotation for the copy constructor of the `File` class, `File(File)`. Normally, if a file object is created based on an existing one, we (assume we) need to set the the object we are creating the file name, the type of operation the file is opened for and the current position in the file to similar values as those of the file object passed as argument. However,

```
@class File
  field fileName: String;

  @spec {
    @transitions {
      this#FileStatus = Closed;
    }
  }
  def File(fileName: String): void
  begin this.fileName := fileName; end

  @spec {
    @transitions {
      this#FileStatus = file#FileStatus;
    }
    @rules {
      file#FileStatus %> OpenWr :-
        %warning "Writing to the same file using different
                  file descriptors is probably a bad idea.";
      file#FileStatus == Closed :- %replace {
                                     $$ := new File(file.fileName);
                                   };
    }
  }
  def File(file: File): void; // copy constructor

  @spec {
    @transitions {
      ! this#FileStatus %> Open :- this#FileStatus = Open;
      this#FilePos = Beginning;
    }
    @rules {
      this#FileStatus %> Open :- %warning "File is already open. No effect.";
    }
  }
  def open(op: int): void; // opens a file for reading or for
                           // writing, depending on value of "op"
  @spec {
    @transitions {
      ! this#FileStatus %> Open :- this#FileStatus = OpenRd;
      this#FilePos = Beginning;
    }
    @rules {
      this#FileStatus %> Open :- %delete;
    }
  }
  def openForReading(): void; // opens file for reading
```

Figure 5.4: File.tool (part 1)

```
@spec {
  @transitions {
    ! this#FileStatus %> Open :- this#FileStatus = OpenWr;
    this#FilePos = Beginning;
  }
  @rules {
    this#FileStatus %> Open :- %delete;
  }
}
def openForWriting(): void; // opens file for writing

@spec {
  @transitions {
    ! this#FileStatus %> Open :- this#FileStatus = OpenAp;
    this#FilePos = End;
  }
  @rules {
    this#FileStatus %> Open :- %delete;
  }
}
def openForAppending(): void; // opens file for appending

@spec {
  @transitions {
    this#FileStatus == OpenRd :- this#FileStatus = OpenWr;
    this#FileStatus %> OpenWr :- this#FileStatus = OpenRd;
    this#FileStatus == OpenRd :- this#FilePos = End;
    this#FileStatus %> OpenWr :- this#FilePos = Beginning;
  }
}
def reverseOp(): void; // changes the operation that can be
                       // performed on the file from read to
                       // write and vice-versa
@spec {
  @transitions {
    this#FileStatus = Closed;
  }
  @rules {
    this#FileStatus == Closed :- %warning "Closing already closed file.";
  }
}
def close(): void; // closes file
```

Figure 5.5: File.tool (part 2)

```
@spec {
  @transitions {
    %unset this#FilePos;
  }
  @rules {
    this#FileStatus %> OpenWr | this#FileStatus == Closed :-
      %error "Cannot read from file that is open for writing or closed.";

    this#FileStatus == Open :- %warning "File might not be open for reading.";
  }
}
def read(): int; // reads a character from the file

@spec {
  @transitions {
    ! this#FileStatus == OpenAp :- %unset this#FilePos;
  }
  @rules {
    this#FileStatus == OpenRd | this#FileStatus == Closed :-
      %error "Cannot write to file that is open for reading or closed.";

    this#FileStatus == Open :- %warning "File might not be open for writing.";
  }
}
def write(value: int): void; // writes a character to the file

@spec {
  @transitions {
    this#FileStatus == OpenRd | this#FileStatus == Open :-
      this#FilePos = Beginning;
  }
  @rules {
    this#FileStatus == OpenRd & this#FilePos == Beginning :- %delete;

    this#FileStatus %> OpenWr | this#FileStatus == Closed :-
      %error "Cannot reset file that is open for writing or closed.";

    this#FileStatus == Open :- %warning "File might not be open for reading.";
  }
}
def reset(): void; // resets the current position to
                   // mark or to beginning of file
```

Figure 5.6: File.tool (part 3)

in case the new file is created based on a file that is closed, we know for a fact (as an active library developer) that the current position in the file is irrelevant and that the file does not have to be opened. In this case, we can instruct the transformation to replace a call to `new File(f)` with a call to `new File(f.fileName)`, which, say, we know (as developers) is faster and, in this case, equivalent. The special `$$` is a placeholder for the variable that is assigned the result of the method call (in this case, the method is actually a constructor, so the variable will be the pointer to the newly created object). When replacing code, the `$$` will be replaced with the proper variable name. As a side note, typestate can also be propagated for the special `$$` variable as well, even though this does not appear anywhere in our example.

Finally, the `FilePos` property allows us to illustrate how typestate for a variable can be unset. The `FilePos` property for the file gets set to `Beginning` when the file is opened for reading or writing and to `End` when it is opened for appending. However, as soon as we read the first character from a file, the position will no longer be at the beginning, and it will not be at the end, either. So, we need to unset the `FilePos` property. We do this with the `%unset` directive, as you can see in the annotation for the `read` method.

The last interesting case we would like to point out appears in the annotation of the `reset` method. The first rule from the `@rules` sections specifies that the call to `reset` can be skipped (i.e., removed) if the `FileStatus` is `OpenRd` (because it is only allowed to call `reset` for files that are opened for reading) and if the `FilePos` is `Beginning`. The idea is that if we know (at compile time) that the position is sure to already be at the beginning of the file, calling `reset` has no effect, so the call can safely be removed.

As we were anticipating earlier, we now get to show how the system works by means of a few code snippets that use the `File` class. We present these one by one, with the code snippet on the left and the result on the right. The result is sometimes another code snippet and sometimes an error or a warning message.

The first example shows the effect of the the annotation for the copy constructor, which determines the rewrite of the constructor call to a different one if the `FileStatus` property for the file object passed as parameter (in this case, `f`) is `Closed`. If we start from the beginning, what happens is that `f#FileStatus` is set to `Closed` as an effect of the annotation for the regular constructor that is called in line 1, thus making the precondition for rewriting the code in the second and third lines true. There is no inherent difference between what happens for lines 2 and 3; the only reason for showing both is to illustrate that even if the receiver of the object is the same as the object passed as parameter (as is the case in line 3), the transformation still works. Another thing to notice is how the `$$` variable specified in the annotation is properly rewritten to `g` and `f`, respectively, for the two cases.

Finally, this first example is also useful for pointing out how mapping occurs when propagating typestate, checking conditions and rewriting code. The variable `this` used in the annotation of the regular constructor is actually a placeholder for `f`, in this case, since it is `f` that will play the role of `this` while running the constructor. Similarly, the formal parameter `file` used in the annotation for the copy constructor is a placeholder for `f`, since `f` is the actual variable passed as a parameter. You might have noticed that this is valid even for the use of `file` in the replacement code, which is a feature for allowing the use of the actual arguments when specifying replacement code. These cases illustrate the general idea that absolutely all variables used in annotations (including the special `this` and `$$` variables)

are placeholders for some other, actual, variables. When running the transformation, these latter variables will be detected in the calling code and used instead of the placeholders appropriately.

```
f := new File("test.txt");
g := new File(f);
f := new File(f);
```

$\Longrightarrow$

```
f := new File("test.txt");
g := new File(f.fileName);
f := new File(f.fileName);
```

In the next code snippet we show how an error gets generated because of an improper use of the library code. The call to `openForWriting` will determine the typestate of variable `f` for property `FileStatus` to become `OpenWr`, and the rules for the method `read` say that an error is to be generated if `this#FileStatus` is either `Closed` or `OpenWr` or a more specific version thereof. Since `f#FileStatus` (`this` is a placeholder for `f` in this case) has the value `OpenWr`, the precondition holds, so the error message is generated when the third line is reached.

```
f := new File("test.txt");
f.openForWriting();
f.read();
```

$\Longrightarrow$

```
Error: Cannot read from file
       that is open for
       writing or closed.
```

The next example shows how unnecessary code gets to be deleted. The precondition for the delete rule specified in the annotation of `reset` indicates that the call to `reset` is to be deleted if `this#FileStatus` has the value `OpenRd` and `this#FilePos` has the value `Beginning`. Since both conditions are met, with respect to the variable `f`, the call to `reset` in the third line is removed from the resulting code.

```
f := new File("test.txt");
f.openForReading();
f.reset();
```

$\Longrightarrow$

```
f := new File("test.txt");
f.openForReading();
```

In the following code snippet we illustrate the fact that if more rules for actions are specified in the annotation for a method definition, their preconditions are examined in the order in which the rules are defined and the first action whose precondition evaluates to true will be performed. The example itself refers again to the annotation of the `reset` method, which contains two rules for actions. We have seen the first one applied in the previous example, but now its preconditon no longer holds, since the value of `f#FileStatus` is `Open` as an effect of the call to `open(1)` in line 2. Since the first rule does not apply, the transformation proceeds with evaluating the condition of the second rule in the annotation, which checks if `this#FileStatus` is equal to `Open`. Since this is true for variable `f`, the warning will be produced. We still print the resulting code (which is identical to the original one, in this case) under the warning to illustrate the distinction between warnings and errors: warnings still allow the transformation to go on and complete; errors stop it altogether.

```
f := new File("test.txt");
f.open(1);
f.reset();
```

$\Longrightarrow$

```
Warning: File might not be
         open for reading.
f := new File("test.txt");
f.open(1);
f.reset();
```

Our last example simply shows another case of typestate propagation. We present two code snippets here, one showing code that will result in an error and one showing code that will be successful. The distinction is the introduction of the line `f.reverseOp()` in the second

piece of code, which transforms the value of `f#FileStatus` from `OpenRd` (which it holds after the call in line 2) into `OpenWr`, according to the propagation rules in the annotation of method `reverseOp`. If the value is `OpenWr`, then the call to `write` in the last line is legal, if the value is `OpenRd`, then it is not. Hence, the two different outcomes.

```
f := new File("test.txt");
f.openForReading();
f.write(1);
```

$\Longrightarrow$

```
Error: Cannot write to file
       that is open for
       reading or closed.
```

```
f := new File("test.txt");
f.openForReading();
f.reverseOp();
f.write(1);
```

$\Longrightarrow$

```
f := new File("test.txt");
f.openForReading();
f.reverseOp();
f.write(1);
```

The annotations for the `File` class provide numerous other opportunities for typestate propagation. Here, we have just shown a few which served to better explain the functionality of the system. The reader is encouraged to experiment with many other combinations of method calls in order to further extend her understanding of the system, if this should be necessary. Our working implementation is available for use in the Stratego/XT repository.

5.3.2 Transformation implementation

We can now proceed to explaining how the Stratego transformation whose behavior we have presented in the previous section works. In principle, the mechanism itself that allows user customization of the transformation is not very complicated; what makes it complicated, though, is the nature of the transformation at hand. If we refer to the core mechanism, it simply involves reading in the user annotations together with the rest of the TOOL classes and making sure that the strategies employed by the transformation properly use this user-supplied information. The nature of our sample transformation, however, makes this not so easy a task. The inherent difficulties lay with the fact that this transformation is interprocedural, that the transformation sometimes needs to change the original code in a non-trivial way (based on the user replacement directives of our annotation language) and that the typestate propagation requires the application of merging strategies that are not provided by the dynamic rules library. We focus in this section on describing the solutions to these (and other) more delicate problems.

Supporting tools

The fact that this is an interprocedural analysis and the fact that we want to easily manage syntactic complexity creates the same need for supporting tools as in the case of pointer analysis. Fortunately, since we are working with the same language, TOOL, makes it possible to reuse the tools that we have developed for pointer analysis. We only had to slightly adjust them so that they would support the extended syntax that allows annotations to method definitions.

The tools that we use are the type annotator, the class hierarchy builder and the simplifier, with the same functionality that we have already described in section 4.3. We use the class hierarchy builder (and the additional strategies defined therein for retrieving methods, fields and class hierarchy information) in order to accommodate retrieval of methods for analysis

and handling of the dynamic dispatch mechanism, just as we needed in the case of pointer analysis. We need the simplifier so that we do not have to deal with very many possible syntactic variants of essentially the same code and the type annotator is required to be run before the simplifier. For a more extensive explanation of all these tools and the reasons why they are needed, we advise you to have a look at the aforementioned section 4.3.

While the type annotator and simplifier hardly had to suffer any changes, the class hierarchy builder had to be somewhat adjusted to support ATOOL (as opposed to TOOL only) and the additional needs of the typestate propagation transformation. In particular, we added a new strategy for retrieving annotated methods only (as opposed to non-annotated ones) and a new strategy for replacing a method body with a different one in the class that contains it, so that we could deal with code changes triggered by the user annotations.

The implementation proper

The customizable typestate propagation transformation is built on the rather classical scheme of Stratego data-flow transformations, where we provide special handling for relevant structures and we descend without doing anything over the irrelevant ones, using the all-traversing `all` strategy. We thus begin with a presentation of the skeleton of our transformation and, more importantly, of the reasons that led to it, while later we move on to explaining the special handling bits.

Method calls The overall approach we took to writing this transformation was largely imposed by the required distinctions in handling the interprocedural aspect of the transformation.

The first distinction is drawn between method calls that invoke methods that are annotated and method calls that invoke methods that are not annotated. A method call invoking an annotated method will determine the transformation to process the information supplied in the annotation and to ignore the contents of the method body. We do this because we assume that an annotated method requires no further transformation or production of warning or error messages. If the method call invokes a non-annotated method, then the body of the method will get to be analyzed.

However, despite our initial attempt to allow typestate propagation to cross method call boundaries, we quickly realized that this would be very difficult to achieve. As a result, whenever a method call `obj.m(`p_1`, `p_2`, ..., `p_n`)` is encountered (and assuming that method `m` is not annotated) we need to "forget" (or, more precisely, undefine) any typestate information we have for parameters p_1, p_2, ..., p_n, as well as for the target of the method call, namely `obj`. We provide an explanation for this further below, but before doing that let us explain why we still run the transformation over the code in the body of methods invoked by such method calls, given that we erase typestate information after that anyway.

The first reason is that we still need to collect all the methods that get called anywhere in the program so that we can later (re)apply the transformation to each of them in particular (this will become clear when we discuss the second distinction later). However, in order to do this, it would be enough to simply collect all the method calls with a straightforward application of the `collect` strategy (and do this for the bodies of collected methods as well, recursively). This would be much faster than running the full transformation over

the method bodies. Why we run the transformation instead of doing this brings us to the second reason: while we cannot keep typestate information for the actual parameters and the target of a method call, we can still compute typestate information for the variable that receives the result of the method call by running the full transformation over the method body. To summarize, when we encounter a statement like `x := obj.m(`p_1`, `p_2`, ..., `p_n`)`, we have to undefine all typestate for variables `obj` and p_1, p_2, ..., p_n, but we can (and do) compute the typestate for variable `x`. In order to compute the typestate for `x`, we have to run the transformation over the body of method `m`. The exact details will be covered later.

We still owe the reader an explanation as to why we cannot propagate typestate information for the parameters and the target of a method call after the method call, if the method is not annotated. Let us assume that we wanted to do this. The first step would be to transfer the typestate information from the actual to the formal parameters of the method, since inside the method body all computation is performed in terms of the formal parameters. No difficulty here, we can just assign the typestate of each actual parameter to its corresponding formal parameter. Normally, our intention would be to perform the reverse operation as soon as the analysis of the method body was over (i.e., assign the typestate of the formal parameters back to the actual parameters). However, this would not be correct because as soon as a formal parameter is reassigned to point to a different object, somewhere in the body of the method, any further changes in the parameter's typestate should not be reflected back to the actual parameter. So, the correct typestate for the actual parameter after the method call should be whatever typestate value the formal parameter had before the reassignment. This means that we would have to somehow "lock" the typestate of the formal argument so that it can not suffer any changes as soon as it is reassigned. The contradicting factor is that we would also need to *not "lock"* the typestate at the same time, because it might be needed to (directly or indirectly) influence the typestate of other formal parameters later on. Adding to this is the fact the we would need to somehow be able to tell, *for each separate path through the method*, if a reassignment of any of the formal parameters is the first one along that path (since the first on each path is the only one we would be interested in). Confronted with the need to introduce a large amount of extra complexity in order to deal with all these problems, we have taken the easier, more conservative, path of simply undefining the typestate of all actual parameters of a method call.

Returning to our main discussion, the second distinction in the handling of method calls is left to explain. This distinction refers to how our transformation is run over bodies of unannotated methods. The first case, which we have already discussed above, arises when all we want is to obtain typestate information for the variable that receives the result of the method call and collect information about what methods are called. This is a context specific analysis because we initialize the typestate of the formal arguments and of the special `this` variable with whatever is the typestate of the actual arguments, and of the target object, respectively. Every time a method is called, it will be reanalyzed with a different set of initial values for its parameters. Although we do not currently do this, we could avoid repeating the analysis of the same method using the same initial typestate values by caching the results. What is also specific to this first case is that user rules for generating warnings and errors or for changing the code *are not applied* (to avoid confusion, typestate propagation rules are applied, however, otherwise there would be no point to the analysis). The reason why user rules are not applied is that the analysis is specific to a

particular calling context, so we cannot give any messages or change the code, because these actions would not necessarily hold for the other calling contexts as well.

The second case of running the transformation over a method comes to deal with this, by not assigning any initial typestate values to the formal parameters. Hence, this makes the analysis context free, and thus we can now correctly apply user-specified rules. As a result, during this case of running the transformation over the method body, we give warning or error messages and change the code if the preconditions for doing this are met, as per their specification in the user annotations.

To distinguish between these two cases when running the transformation, we had provide the transformation strategy with a parameter (called `alter-code`). If `alter-code` succeeds, then we apply the user rules, if it fails, then we do not apply the rules.

The transformation skeleton The main steps of our transformation can be summarized as follows:

1. first of all, we read in the file containing the definitions of all the properties (by default, we look for a file called `properties` in the directory where all the classes are) and store this information in memory in a form that maintains the ordering information.
2. secondly, we build the class hierarchy in memory using the class hierarchy builder. The code in each class is annotated with type information and simplified before being stored in memory.
3. then, we run typestate propagation (with user rule application) over the body of the main method. This will also trigger the running of the transformation over the bodies of all called methods (without user rule application). This, in turn, will store information about all methods called throughout the program. If the body of the main method gets changed as a result of applying user rules, it will be replaced in memory with its changed version.
4. finally, we retrieve the list with all the methods that get called and run typestate propagation (this time, with user rule application) over each of their bodies. Any method whose body gets changed as a result of applying user rules will be replaced in memory with its changed version.

The transformation will either stop due to an error message that has to be given or it will proceed until the end, with the possible generation of warning messages. If it finishes successfully, then the in-memory class hierarchy will reflect any code changes that might have been triggered by user rules.

Examining properties Before we can proceed to propagating typestate for variables we need to know the names and values of the all the properties that can be appear in the user annotations. As explained in section 5.3.1, these have to be specified by the user in a per project file. In the beginning of the transformation, we parse this file and build, for each property, an in-memory representation of all its values, for each value remembering its parent as well, aside from its name. In the end, what we end up with is a list of all properties (which is useful, e.g., when we need to transfer *all* typestate from one variable to another) and information which allows us to (1) find the most specific common denominator of any

pair of values defined for the same property and (2) decide if a value is a more specialized version of another value (this is useful during the evaluation of the %> operator).

Symbolic merge The next item on our agenda of things to explain is the so-called symbolic merge operation. If you are familiar with how dynamic rules works, you know that the two operations that can be applied to merge two sets of dynamic rules are intersection and union. These two operations suffice in a lot of circumstances, but however are not fit for our purposes here. The peculiarity of our transformation is that the outcome of merging two different typestates is their most specific common denominator, according to the user-specified property semilattice. To refer to our earlier `File` example, if one branch of an if-then-else calls `openForWriting` and another one calls `openForReading`, generating the typestates `OpenWr` and `OpenRd` respectively, we want the typestate after the if-then-else to be `Open`, since this is the most specific common denominator of the two typestates.

Unfortunately, the dynamic rules library does not currently provide any support for customizing the merge operation applied when mixing two rule sets together, so we had to build our own specialized merge strategy. Figure 5.7 shows almost all the code related to this, the only part missing being some additional code in the last strategy, `dr-symbolic-merge-rulesets`, which deals with ignoring of rule sets (this code is specific to the support for break/continue that we discuss in chapter 3, but which has little relevance here).

The strategy the we use throughout our transformation is `merge-TypestateOf`, which takes two strategies as parameters. This strategy is the equivalent of the strategies defined in the dynamic rules library that get called as a result of the use of s_1 `/rules\` s_2 or s_1 `\rules/` s_2, except that it does not use intersection or union as the merge operation, but it uses a custom strategy, defined by our transformation, called `ts-merge`. The implementation details of these strategies is probably only understandable to those familiar with the implementation of the dynamic rules library. In principle, `dr-symbolic-merge` is the rough equivalent of `dr-fork-and-merge` from the library, while `dr-symbolic-merge-rulesets` is the rough equivalent of `dr-merge-rule-sets2` from the library. Most of this code is merely copied (with some adjustments) directly from the library, with the sole purpose of our being able to replace the intersection or union operation with our own merge operation. This happens in the call `<merge(|prop)>(v1, v2)`. Most of the remaining code is dynamic rule set related administration.

The only other thing we should probably mention is the use of the string `"0"` when a key does not have an associated right hand side. This `"0"` string is simply what we use for representing the $\perp$ element for any property. We use it here when no binding for a key is found (which means that in that rule set the typestate for a particular variable is not set) so that our merge operation always has two values to merge. If one or both of the values are $\perp$, the final result of the merge will be $\perp$ as well.

Propagating typestate Typestate of variables is saved using the `TypestateOf` dynamic rule. When a variable declaration is encountered, its typestate is not initialized to any value. In the beginning, we toyed with the idea of asking the user to specify an initial value for each property that we could use to initialize typestate. However, we quickly realized that initializing the typestate of all variables, for all properties declared for a project, would be an overkill, given that a large majority of the variables will never end up having any typestate

```
merge-TypestateOf(s1, s2) =
  dr-symbolic-merge(
    s1, s2,
    ts-merge,
    TypestateOf,
    aux-TypestateOf(|(), (), ()) <+ aux-TypestateOf(|(), ()),
    redef-TypestateOf
    | "TypestateOf"
  )

redef-TypestateOf =
  ?(x@(_, label), y)
  ; (<eq>(y, "0")
     < rules(TypestateOf.label :- x)
     + rules(TypestateOf.label : x -> y)
    )

dr-symbolic-merge(s1, s2, merge, call, aux, redef | rulename) =
  where(
    dr-get-rule-set(|rulename) => rs2
    ; dr-start-change-set
    ; dr-set-rule-set(|rulename)
  )
  ; restore(s1, where(dr-discard-change-set(|rulename)))
  ; where(
      dr-get-rule-set(|rulename) => rs1
      ; <dr-start-change-set>rs2
      ; dr-set-rule-set(|rulename)
    )
  ; restore(s2, where(dr-discard-change-set(|rulename)))
  ; dr-symbolic-merge-rulesets(merge, call, aux, redef | rulename, rs1)
  ; dr-commit-change-set(|rulename)

dr-symbolic-merge-rulesets(merge, call, aux, redef | rulename, rs) =
  where(
    !rs => rs1@[ChangeSet(_, rmset1, tbl1) | _]
    ; dr-get-rule-set(|rulename) => rs2@[ChangeSet(_, rmset2, tbl2) | _]
    ; <apply-rm-set(|rmset1)>tbl2; <apply-rm-set(|rmset2)>tbl1
    ; <iset-union(|rmset1)>rmset2
    ; <union>(<hashtable-keys>tbl1, <hashtable-keys>tbl2)
    ; map({scpid, key, prop, v1, v2 :
        ?(scpid, key@(prop, _))
        ; (<dr-lookup-rule-in-scope(|key, scpid)>rs1; ?[<aux> | _] <+ !"0") => v1
        ; (<dr-lookup-rule-in-scope(|key, scpid)>rs2; ?[<aux> | _] <+ !"0") => v2
        ; !(key, <merge(|prop)>(v1, v2)); redef
      })
  )
```

Figure 5.7: The symbolic merge operation.

at all. Instead, we assume the user will provide an unconditional typestate assignment rule in the annotations for class constructors.

Typestate is assigned to a variable in one of two cases: (1) the variable appears as an actual parameter, target object or receiving variable in a method call that invokes an annotated method *and* the conditions in the annotations are met for the variable's typestate to be initialized/changed and (2) the variable appears on the left hand side of an assignment, in which case all typestate for the variable on the right hand side will be copied to the variable on the left hand side. This also means that any typestate that exists for the variable on the left hand side will be undefined unless it also appears for the variable on the right hand side.

Typestate is also passed from actual parameters to corresponding formal parameters, when calls to unannotated methods are encountered. Passing of typestate from returned variables to receiving variables is somewhat more involved and will be discussed later.

All control flow structures, if-then, if-then-else, while-do and for are handled using the strategy we presented earlier, `merge-TypestateOf`. This can be understood easily by analogy with strategies like constant propagation or copy propagation, discussed in [7]. The only difference is that here we do not merge rule sets using intersection, but using our custom `ts-merge` strategy inside `merge-TypestateOf`. To illustrate, we present below the code for handling if-then-else structures:

```
ts-prop-if-else(alter-code) =
  IfElse(ts-prop(alter-code), id, id)
  ; merge-TypestateOf(
      IfElse(id, ts-prop-statements(alter-code), id)
      , IfElse(id, id, ts-prop-statements(alter-code))
    )
```

Return statements We have already explained most of what happens in the case of method calls. The only thing that is left unexplained is how return statements are handled or, more generally, how typestate flows from returned variables to the variables that receive the result of method calls. To relate the explanation to the distinctions we were making regarding the handling of method calls, we point out that we are discussing only calls to unannotated methods which are analyzed with the purpose of propagating typestate across them, not within them.

As a method may have more than a single return statement, we need to make sure that the typestates of all returned variables are merged together in order to obtain a most specific common denominator that we can assign to the receiving variable. Furthermore, each variable has a typestate associated with it for each property defined for the project. This means that for each returned variable we have to store not one, but any number of typestates (or property values), depending on the number of properties we have. In order to manage this, we introduced a new dynamic rule that stores an association list that associates property names to lists of values for those properties. Each return statement that we encounter will add a new element in each of the lists associated with properties. After traversing the entire method body, we retrieve this association list and, for each property, merge all the values in order to obtain a single one. We thus end up with a list of (property, value) pairs. It is this list that we use in order to transfer typestate to the receiving variable.

Dynamic dispatch So far, when discussing method calls, we have ignored the problem of dynamic dispatch. To refresh your memory, this problem refers to the fact that we cannot know at compile time the exact method body that is going to be executed, so we need to analyze all of them and merge the results. A more extended presentation of this problem when writing interprocedural data-flow transformation for object oriented programs is given in section 4.3.2.

Here, just as in the case of pointer analysis, when a method that may trigger the execution of one of more possible method bodies is called, the only option we have at hand is to retrieve all these method bodies, analyze them and combine the results into a single one. In essence, for each candidate method we perform the same steps that we have already presented before (passing of typestate from formal to actual arguments, definition of association list, filling in of the association list when analyzing return statements, reducing the association list and passing of the resulting typestates to the receiving variable) and in the end (or, rather, along the way) merge the results. The strategy that helps us do this is defined below as a version of `merge-TypestateOf` that applies a parameter strategy `s` to a list of terms:

```
merge-list(s) :
  [] -> []

merge-list(s) :
  [x] -> [<s>x]

merge-list(s) =
  ?[_,_|_]
  ; merge-TypestateOf([s | id], [id | merge-list(s)])
```

With this strategy at hand, and assuming that the strategy that analyzes a single method is called `process-method`, all that we have to do in order to implement our solution is call `merge-list(process-method)` over our list of candidate methods.

Avoiding infinite cycles Like with all interprocedural analyses, we are also confronted in this case with the issue of the possible formation of infinite loops if recursion (direct or indirect) exists in the analyzed code. Since we follow method calls, recursion in the analyzed code will determine our transformation to keep switching back and forth between analyzing two or more method bodies. This is the case, of course, if we do not take provisions against the onset of this phenomenon.

The solution with this transformation was to keep an in-memory call chain that we update whenever we start analyzing a new method. Before analyzing a method, we first check to see if the method is not already present in the call chain. If it is, then we do not call `process-method` over its body anymore to avoid the formation of an infinite loop. As we were briefly mentioning at an earlier point, we currently have no caching mechanism in place, so there is no result of running the analysis over the method call that we can retrieve. This being the case, in addition to skipping the call to `process-method`, in order to keep the typestate propagation correct, we also undefine all typestates for the variable receiving the result of the method call, if any, for the actual arguments and for the target object.

Processing annotations The last aspect about our transformation that we want to get into is how user annotations are processed. User annotations appear as method annotations,

so they get processed any time a call to an annotated method is encountered in the analyzed code. As we have already explained, bodies of annotated methods are not processed, so the only thing we have to do for calls to such methods is look at the annotations and act upon them.

The first thing that we do when processing the annotation for a particular method is perform the substitution of placeholder variables from the annotation with "real" variables, i.e., variables that are in scope at the place of the method call. If you recall, we use formal arguments and the special `this` and `$$` variables as placeholders in annotations. These will have to be replaced with the actual arguments of the method call, the target of the method call and, if it exists, the variable that receives the result of the method call, respectively.

Once we have the proper variables filled into the annotation, we can proceed to checking the preconditions and, for each transition and/or rule for which the precondition holds, perform the specified action. In the case of transitions, the action in question is changing the typestate of a variable; in the case of rules, the action may be one of the following: giving an error message, giving a warning message, deleting the method call that invokes the annotated method or replacing the same method call with some different code.

The interesting part about transitions and rules is that all preconditions have to be checked using the same typestates that hold before performing any changes to the typestate, as an effect of running transitions. In other words, we cannot directly perform the typestate changes specified on the right hand side of transitions because that would influence the typestate values used in the remaining preconditions, and this would not be in accordance with the semantics of our annotations. What we need, then, is a sort of a delayed typestate change, after all the preconditions have been checked.

The most elegant solution we found to this problem was to create an empty change set before starting to process all transitions and rules of an annotation, and store it in memory, but not add it to the current rule set. Later on, every time a typestate transition has to be performed, we retrieve this change set from memory and set it on top of the active rule set. With this change set in place we perform the typestate transition, and then we immediately remove it from the top of the active rule set. In this way, the next preconditions will still be using the same typestates during their evaluation. After all transitions and rules have been processed, we retrieve the change set that contains all the typestate changes from memory, set it on top of the active rule set, and commit it. It is only now that the typestate transitions become "visible".

While the preconditions of all transitions are checked and each transition with a precondition that evaluates to true is performed, things are different in the case of rules. As we discussed when we were presenting the system, only one rule from an annotation, at most, will be run. This rule is the first one, in the order in which it appears in the annotation, whose precondition evaluates to true. Furthermore, you might recall that rules are only applied in certain contexts, even if their preconditions hold. As a result of these two aspects, processing of rules actually means checking of preconditions of rules until one evaluates to true (or until we run out of rules) and returning that rule without applying it. Only at a later point in the transformation, where we have information about whether or not the rule has to be applied, does the rule get to actually be applied (or, otherwise, it is dropped).

Evaluation of conditions, changing typestate and performing actions do not raise any interesting issues, so we see little point in discussing them here.

This concludes the presentation of our customizable transformation, a presentation focused less on code (the sheer code can be found in the Stratego/XT repository), and more on giving a guide to understanding our approach to this transformation and, while doing this, hopefully providing some valuable ideas for a continuation of this direction of research.

5.4 Extensible transformation: arrays in TOOL

As it will become clear, our second experiment has been considerably smaller in scope than our first one. It simply set out to discover how plausible it would be to extend a language and the transformation(s) defined for it so that they all support an addition to the language, *assuming no special preparations for accommodating the extension have been made*. In other words, if the original writer has not provided any "hooks" in the transformations defined for a language, how intrusive would an extension have to be in the current Stratego state of the art?

To provide the answer to this question, we have set out to extend the TOOL language with support for arrays and to extend all the transformations that we had defined for performing pointer analysis in order to accommodate this new language feature. The focus in this approach has been on keeping our extension as unintrusive as possible. We regard this to be a key aspect for allowing independent extensibility of transformations. This concept refers to the possibility of combining extensions that have been developed independently in order to obtain a larger, composed language extension. If any of the candidate exceptions were to require the changing of the core code, it would break the contract that the core code offers to the other extensions, possibly rendering them erroneous.

Despite the fact that we have not managed to not change the core code at all, our conclusion after this experiment is that completely unintrusive extensions are perfectly possible, at least if the core code offers some provisions for accommodating them. In our case, we have had to introduce a couple of hooks ourselves in some places or to apply rather unnatural, messy and performance-hindering tricks in some other places, all in order to overcome the fact that the original code had not been properly prepared for accommodating extensions.

In the remainder of this section, we give a report of this successful attempt to extend TOOL and pointer analysis performed on TOOL with array support. When discussing the various tools that we have extended, we assume a thorough familiarity with their descriptions from the previous chapter. If you have not read that chapter, the purpose and functionality of each of the tools whose extensions we discuss below will be largely unknown, thus making it quite difficult to understand our explanations. We therefore suggest that you have a look at chapter 4 before proceeding through the rest of this section.

Language syntax

The first step in the process was to extend the TOOL language itself with a new module for parsing array syntax. SDF is perfectly fit for an unintrusive extension, given its modularity, so this first task was an easy one. We defined two new .sdf files, TOOL-arrays.sdf and TOOL-ext.sdf. In TOOL-arrays we listed all the new productions which deal with parsing syntax that declares, creates and accesses array structures. TOOL-ext was simply the file that aggregated all the syntax modules together in order to define the entire language.

One minor difficulty that we can mention for this step is that a new rule for compiling TOOL-ext.sdf had to be added to the Makefile of the project. In our case, we simply added the extra lines in the original Makefile, but in a more general setting, all the core Makefile's could have a directive to include all the files with a certain extension (say *.ext) in the directory, and then each separate extension could provide its own Makefile rules in such *.ext files, and copy them to the proper directories so that they are included.

Another difficulty arose from the way in which we parsed TOOL files throughout our project. We had a utils.str file in which a parse-tool strategy was defined, which read in the TOOL parse table and parsed the argument TOOL file using it. The parse table was unfortunately hard-coded in the strategy, so the only solution was to replace it with the new parse table that also included the support for arrays. A solution to this problem could be the use of configuration files for specifying the parse table to use. In this way, the extension could replace the original configuration file with a different one that would mention the extended parse table. If more extensions are to be used together, a new parse table that includes the extra modules from all extensions has to be defined additionally and the configuration file has to specify the path to this resulting parse table.

Extending transformations

Before we dive into discussing each transformation in turn, let us first present some aspects related to extensions that appear regardless of the concrete transformation at hand.

The most recurring problem during the writing of the extensions for transformations was generated by the fact that our module for array syntax introduced a new constructor for the array type, called `ArrayTypeName`. As a result, while the type `String` would be parsed to `TypeName(String)`, the type `String[]` would be parsed to `ArrayTypeName(String)`. Originally, throughout our transformations, whenever we needed Stratego code to match for type names, we would use something like `?TypeName(t)` or `?TypeName(_)`.

The effect of using the `TypeName` constructor for matching was the failure of the matching, and usually of the whole strategy that contained the matching code, every time the term to match would be an `ArrayTypeName`. As far as our extensions were concerned, this effect was sometimes helpful and sometimes troublesome. It was helpful when the failure of a strategy would ensure the run of the strategy with the same name defined in the extension. It was troublesome when it caused the failure of a strategy for which a different version was not needed in the extension. We discuss solutions used for each case as we proceed with presenting our transformations.

Jumping to another aspect, it is important to mention the extension mechanism that we have used in order to extend all transformations. Two features of the Stratego platform make this mechanism possible. The first one is represented by the fact that any number of strategies (and dynamic rules) with the same signature can be defined, and at run time they will be called in turn in a predefined, but random, order, until one executes successfully. The second feature is that if two (or more) such strategies are defined in multiple files, it is sufficient to import all these files in a single file in order to make all the different definitions available *and* have the code that was originally designed to call just one (or some) definitions to also call the additionally imported ones. These two features allow extensions to (1) define more cases for already existing strategies that are called by the core code by simply introducing new definitions for the same strategies in separate files and

(2) making the definitions be considered in the compilation process by defining a file that imports both the files containing the original code and those containing the extra definitions. This mechanism can be understood very easily using an example.

Consider the following given code, in file foo.str:

```
module foo

strategies

  main =
    foo

  foo =
    fail
```

If we compile and run foo.str, it will naturally fail. Now assume that we want to provide an the extension to this code without changing it at all. We define our extension in file bar.str:

```
module bar

strategies

  foo =
    id
```

As you can see, we provide an additional definition of the `foo` strategy, this time one that succeeds. Of course, if we compile foo.str as it is, there will be no change, since it does not include the new definition in file bar.str. However, if we create a new file, foo-ext.str, in which we import both the original foo.str and the extension bar.str like this:

```
module foo-ext
imports
  foo
  bar
```

and we compile foo-ext, *using the* `main` *strategy defined in foo.str* (as you can see, foo-ext.str does not define a different `main` strategy), running the resulting program will be successful. Why? Because, in foo-ext the call to `foo` in the `main` strategy (the one defined in foo.str) will have both definitions of `foo` visible, so the compiler will make sure that both are called, in turn, in some random order. If the original definition is called first, it will fail, and then the second definition will be called, ending the program successfully. If the definition from the extension is called first, it succeeds, and execution ends successfully. So, as you can see, in either case the program will run successfully.

All our extension follow the same patterns. They use strategy names that already have a definition in and are already called by the core code to define additional cases. Then, by importing the original code in the file that defines the new cases (this is a bit different than in our example; it is as if bar.str would import foo.str directly), it makes the new definitions visible to the core code. All that remains is to ensure that the new definitions are defined in such a way that they provide coverage for all the cases that the core code rejects, due to the extended syntax that it is unaware of.

This mechanism has one drawback, however, in that it does not work if the original code has side effects in certain unfortunate places (a very frequent case of side effect in Stratego being the changes made to dynamic rules). The problem arises if side effects get triggered before the original code gets the chance to fail. If this is the case, even if a version of a strategy definition fails and a different version (provided by the extension) of that strategy gets executed, the side effects that have been generated before the failure of the first version will stay in place, polluting the environment of the transformation. However, if the core code is written with regard to all the possible failure points that might be triggered by various extensions, the side effects could be reverted before the strategy fails completely, thus leaving the environment clean. Fortunately, with our extension, all side effects were being triggered after the failure points, so they did not become an issue.

Type annotator The first transformation that we had to modify was that which annotates TOOL programs with type information. According to the general scheme already described above, we created a new file called `tool-tc-ext.str` in which we provided a number of additional definitions for strategies that were already used and given a definition in the core TOOL type annotator (tool-tc.str), as the mechanism to incorporate the code of the extension in the overall transformation.

There were a number of cases for which we had to provide additional handling, and we detail them below:

- *array access*: type annotation for array accesses refers to annotating an expression like `a[idx]` with the type of the array `a`. If `a` has the type `ArrayTypeName(String)`, then `a[idx]` has to be annotated with `TypeName(String)`. The only problem here is that there was no clear strategy that we could choose to provide an additional definition for. In this case, as in many other cases to come, the only option was to choose the strategy that made most sense and piggyback on it. Here, we chose the strategy that handled variables, as this seemed closest to our purposes, given that array accesses can be used in exactly the same places in which a variable can be used.

  In the core code, the strategy for annotating variable uses was called `tc-var`, so in the extension we introduced a new definition for the strategy `tc-var`, in which we perform the type annotation described above. When the extended type annotator runs over a term like `ArrayAccess(_, _)`, the original definition of `tc-var` will fail, causing the definition given in the extension to be run. This latter one will be successful and produce the proper annotation.

- *creation of arrays*: an expression like `new String[]` has to be annotated with the type `ArrayTypeName(String)`. The most natural strategy to overload for this case was the strategy that annotates creation of objects, called `tc-new-instance` in the original code. The new definition we provide in the extension simply annotates terms like `NewArrayInstance(_, _)` with the proper type.

  However, if we take a closer look at the original definition of `tc-new-instance`:

```
tc-new-instance :
  NewInstance(type, args) -> NewInstance(type, args'){TypeName(type)}
  where <tool-tc>args => args'
      ; <map(?_{<?TypeName(<id>)>})>args' => argtypes
      ; <get-method(|type, argtypes)>type
```

we notice that before we perform the annotation, we also run a check to see if an appropriate constructor exists in the class of which we are creating a new instance (basically, a bit of type checking in addition to type annotation). In order to check for the existence of the constructor, we first need to extract the types of the actual arguments passed to it. Unfortunately, this is done by using the expression `<?TypeName(<id>)>`, which leads to a failure if one of the arguments passed to the constructor is an array. The problem is that the type of such an argument would be `ArrayTypeName(_)`. In order to solve this issue, we have to introduce an additional definition for `tc-new-instance` (the third one overall, the second one in the extension), which makes sure that array types can also get passed to constructors. This definition looks like this:

```
tc-new-instance :
  NewInstance(type, args) -> NewInstance(type, args'){TypeName(type)}
  where <tool-tc>args => args'
      ; <map(?_{<?TypeName(<id>) <+
                 ?ArrayTypeName(<!Array(<id>)>)
               >})>args' => argtypes
      ; <get-method(|type, argtypes)>type
```

We cannot simply extract the type from the `ArrayTypeName`, as we do in the case of `TypeName`, because then there would be no distinction (when matching argument types, in the `get-method` strategy) between simple types and array types. For this reason, we wrap the `Array` constructor around the type we find in `ArrayTypeName` (which you can see in the code above). Similar handling is also ensured for the declared types of the method's formal arguments, so that matching of argument types can be performed correctly.

- *assignments*: assignments that involve arrays or array accesses actually get handled correctly by the original code, so there is nothing we have to do for type annotation in what assignments are concerned. However, aside from type annotation we also perform type checking when analyzing assignments, in order to check that the type of the expression on the right hand side of the assignment is assignable to the type of the variable on the left hand side. In order to do this, the original code matches for `TypeName` constructors like this:

  ```
  tc-assign =
    ?stm
    ; Assign(tool-tc => _{TypeName(tt)}, tool-tc => _{TypeName(at)})
    // check that at is assignable to tt ensues
  ```

  This will obviously fail when either the right or the left hand side is an array, thus generating the need that we provide an additional definition for `tc-assign` in the extension. In this definition, we match for `ArrayTypeName` in addition to matching for `TypeName`.

If you find our solutions too complicated (we refer especially to those for dealing with failures triggered by the fact that the original code only matches for `TypeName`, and not also for `ArrayTypeName`), please recall that we are trying to come up with solutions that *do not change the original code in anyway*. As you can imagine, this seriously limits our pool of options.

Class hierarchy builder The extension for the class hierarchy builder is much less complicated. In fact, if it were not for one single issue, all we would have to do is to create a file in which we would just import all the extended versions of the other tools (extended TOOL syntax, extended tool annotator, extended simplifier and extended variable annotator) together with the original code for the class hierarchy builder. Importing of the extended versions is necessary so that the exact same code from the original class hierarchy builder gains access to the definitions from the extensions as well. The file tool-ch-ext.str would only contain this code:

```
module tool-ch-ext
imports
  tool-ch
  TOOL-ext
  tool-tc-ext
  tool-simplify-ext
  var-anno-ext
```

However, there is one issue, related to the fact that some code in the original class hierarchy builder matches for `TypeName(_)` only. Exactly the same problem that we had before. That code is in the body of the strategy `get-method`, and it is the code that extracts the types from the formal argument declarations of methods, in order to match them against the types of the actual arguments. It is, in fact, the counterpart code of that which we were discussing earlier, related to creation of new instances in the type annotator. At some point in the body of `get-method` this line appears:

```
?MethodDec(mod, method, <map(?Arg(_, TypeName(<id>)))>, _, _)
```

Whenever one of the arguments has the type array, this will fail. One solution, similar to that which we used before, would have been to provide an alternative definition of `get-method` in the extension of the class hierarchy builder. However, given that the code of the strategy `get-method` is quite large, it would be impractical for each extension (ours and potentially others) to redefine it just so that it can change the line above to this:

```
?MethodDec(mod, method, <map(?Arg(_, TypeName(<id>))
                           <+ ?Arg(_, ArrayTypeName(<!Array(<id>)>)))>
          , _, _)
```

Therefore, we decided on an intrusive approach this time. The change that we made to the original code simply introduced a "hook strategy" that can be used by extensions to make their code be run by the original code. To illustrate, the line above was changed to something like this:

```
?MethodDec(mod, method, <map(?Arg(_, TypeName(<id>))
                           <+ ext-extract-arg)>
          , _, _)
```

and a default definition for `ext-extract-arg` was added:

```
  ext-extract-arg =
    fail
```

This hook has no effect on the original code, since `ext-extract-arg` is defined to fail, but as soon as an extension provides a new definition for `ext-extract-arg`, it will get to be run by the main code, without having to change the main code in order to achieve this effect.

We believe that this small intrusion into the main code is justified by the observation that transformations can be written with some attention to potential extensions, and then such hooks could be provided in all the sensible places by the original creator of the transformation. Granted, this does require a certain degree of anticipation on the part of the original developer, but this does not make it a less viable option.

Finally, all that was left to do was provide the proper definition for `ext-extract-arg` in the extension to the class hierarchy builder, `tool-ch-ext.str`:

```
ext-extract-arg =
  ?Arg(_, ArrayTypeName(<!Array(<id>)>))
```

TOOL simplifier The extension of the TOOL simplifier required somewhat of a larger amount of code because a number of new syntactic structures made possible by the introduction of arrays had to be simplified. However, the details related to the extensions mechanism are quite straightforward.

The original code of the simplifier is an application of the `outermost` strategy with a number of simplification rewrite rules being called:

```
tool-simplify =
  outermost(
    MethodReturn
    <+ MethodTarget
    <+ MethodArgs
    <+ NewInstanceReturn
    <+ NewInstanceArgs
    <+ RightFieldChain
    <+ LeftFieldChain
    <+ FieldAssign
    <+ VarReturn
    <+ ThrowVar
  )
  ; elim-vblocks
```

In order to plug in our extension, the only option we had was to provide additional definitions for some of the rewrite rules that get called by the original code and bring those definitions in scope. This is, in fact, no different than what we did earlier, when we provided new definitions for strategies. Strategies and rewrite rules are actually quite similar in Stratego, given the fact that rewrite rules get desugared to strategies.

We added new definitions for `NewInstanceReturn` (to cover simplification of creation of new arrays), `RightFieldChain` (to cover simplification of assignments like `x := exp[idx]` in which `exp` is not a simple variable), `LeftFieldChain` to cover simplification of assignments like `exp[idx] := x` in which `exp` is not a simple variable) and `FieldAssign` (to cover simplification of assignments like $\texttt{a[idx}_a\texttt{]} := \texttt{b[idx}_b\texttt{]}$. While the first three seem pretty natural overloads, in the case of the last one, that of `FieldAssign`, the name of the rewrite rule and the action that it performs are somewhat incompatible. This does not introduce any technical errors, but it may introduce confusion among those looking at the code. However, in the lack of an extra entry in the original code that would call an, say, `ExtensionRule` as a predefined hook for extensions, there is no other unintrusive solution but to piggyback on one of the rewrite rules that do actually get called.

Another very important aspect related to the extension mechanism, for this and all other extension files, is that we always need to pull in (i.e., import) the extended versions of all the modules that the original code uses. For example, in the case of tool-simplify.str, the imported modules are `tool-ch` and `TOOL`. Given this, in tool-simplify-ext.str we need to define an imports statement as follows:

```
module tool-simplify-ext
imports
  tool-simplify
  tool-ch-ext
  TOOL-ext
```

This ensures that all the code from tool-simplify.str will see the additional definitions in our extension and call them appropriately if needed. We stress that the code in tool-simplify.str does not require any changes in order to call the extended definitions as well; all that is necessary is that they are made visible by bringing them all in the same scope.

Variable annotator The extension to the variable annotator does not introduce any extension-related techniques that we have not already seen or discussed, therefore we will cover it expeditiously.

A new definition for the `anno-new` strategy is given in the extension, in order to properly annotate the instantiation site of array objects. These new allocation sites will add to the pool of memory locations that the pointers of the program may point to, so they need to be given unique annotations similar to those given to instantiation sites of objects. Our new definition for `anno-new` does exactly that.

The other issue that had to be dealt with is the all-familiar one, by now, of the original code having a line that matches for `TypeName(_)`. In the case of the variable annotator we handled this similarly to the way we did for the type annotator, namely by providing a new definition for the entire strategy that contains the culprit line, and adjusting the line in question so that is matches for `ArrayTypeName(_)` as well.

Pointer assignment graph builder Building of the pointer assignment graph also had to be extended in order to handle arrays. We stray a bit from the extension techniques that we used here in order to first explain how the pointer analysis was adapted to deal with arrays. The approach is rather conservative, and consists in assuming that all slots of an array are actually referred to through one single field. We assume each array to be an object that has a single field, and that that same field is accessed (written or read) regardless of the expression that constitutes the index of the array. In effect, `a[1]`, `a[i]` and `a[some-expression]` will all be assumed to be referring to the same memory location, `a.array` (the virtual field name was randomly called "array"). As soon as array accesses are handled as field accesses, there is nothing else left to be explained regarding how pointer analysis works when arrays are present. For every array access that appears on the left or right hand side of an assignment, we generate the proper load or store edge, just as if it were a field access. From there on, the algorithm behaves the same; in particular, the propagation algorithm does not have to be changed at all in order to accommodate arrays.

Coming back to extension-related issues, the pointer assignment graph builder only brings back issues that we have already seen in yet a another incarnation. We overloaded five strategies, `build-pag-new`, `build-pag-field-access`, `build-pag-assignment-alloc-edge`,

`build-pag-assignment-store-edge` and `build-pag-assignment-load-edge`, all of which are part of the original code for building the pointer assignment graph. The new definitions that we introduced ensure that allocation nodes are generated for array creation sites, that allocation edges are created when a new array is assigned to an array variable, that field reference nodes are added for the virtual "array" field of each array and, finally, that store and load edges to and from these field reference nodes are introduced when a variable is stored into a slot of an array and when a variable is loaded with the contents of the slot of an array, respectively.

The problem of the original pointer-assignment-graph having (two) locations in which matches for `TypeName(_)` appeared was handled in the same way that we did for the class hierarchy builder, i.e., in a slightly intrusive way. We added a hook strategy called `ext-pag-extract-arg` in the original code, that fails by default. We modified the two problematic locations in the original code so that they call `ext-pag-extract-arg` if the default handling fails. The only thing left thereafter was to create a definition for `ext-pag-extract-arg` in the extension that would match for an `ArrayTypeName`.

Pointer analysis In order to finalize our extension, we created a file that we called `pointer-analysis-ext.str` in which we imported the original `pointer-analysis` module and the module containing the extension of the pointer assignment graph builder, `pointer-assignment-graph-ext`. All the rest of the extension code was already being imported by `pointer-assignment-graph-ext`. Once this was done, we could compile the resulting code and run the pointer analysis over TOOL code that used arrays.

As a final word, we believe that extending pointer analysis with support for arrays was a worthwhile experience that suggested some approaches that could be taken for achieving extensions of transformations that do not need to modify the original transformations in any way. We find these results fascinating, especially if we consider that the original pointer analysis code was written without any regard to potential extensions. This further accredits the idea that if code is written with extensibility in mind, independent extensibility of it is already within reach using Stratego.

Chapter 6

Conclusion

This thesis has presented work that has spanned, with some interruptions, over a period of six months. Although we have originally set out to achieve a larger number of goals and, consequently, to answer a larger number of research questions, the inherent difficulties that appeared along the way have reduced the scope of our research to that presented herein. Even so, we believe that we have managed to cover quite a lot of ground, providing some useful code and some interesting ideas that we hope others will find helpful. Even though this thesis has been composed of a number of seemingly loosely related parts, we still have the more general goal of supporting and improving the techniques for data-flow transformations in Stratego that brings them all together. We have, according to our knowledge, also provided some of the first transformations that deal with object oriented programs in Stratego, which may be seen as an achievement in its own. All in all, we have found this work interesting and rewarding, and we can only hope that others will find some use of it as well.

6.1 Dynamic rules library

The first part of our work has dealt with developing strategies that would help writers of data-flow transformations to more easily handle statements and constructs that determine an interruption in the control flow of a program. We have named the break and the continue statements and the exception throwing/catching mechanism. This goal has been fully and successfully achieved for forward propagation transformations and we have also covered some initial ground in the direction of supporting backwards propagation transformations. As part of this process, we have also managed to increase the correctness of the dynamic rules library by identifying and fixing a number of rather subtle bugs that had made their way into it. For better reliability of the library code, an extensive unit test suite was developed as well.

Future work

There are a number of opportunities for future work that have been opened by this first part of the thesis. For starters, the strategies for supporting the handling of interruption of control flow have yet to be passed through the reality check by developing and checking the correctness of a number of distinct data-flow transformations that employ them. We have

run all of our tests in the context of the constant propagation transformation, but different transformations also need to be extended with handling of break, continue and exceptions in order to further validate the correct functionality of the library support.

Secondly, more library strategies have to be written in order to allow the break, continue and exceptions support to use a combination of intersection and union when merging. Currently, we provide strategies that allow either intersection or union to be used as the merge operation, but not both. Aside from these additional strategies, intelligent syntactic sugar would also be welcome as a more user-friendly interface to the code we have developed. While the code is perfectly usable as it is now, there is nevertheless a discrepancy between the friendly syntax used for calling the fix point operation or the fork-and-merge operation and the less intuitive strategy calls that have to be used for dealing with support for break, continue and exceptions.

Yet a different direction of future work is given by the fact that our support for exceptions has been developed and tested strictly assuming a Java-like semantics of the exception throwing/catching mechanism. While this is good news for those who plan to write transformations for the Java language, it might be a problem for those whose target language is different. Because of this, an investigation of the semantics of exceptions in other languages and the extension of the support in order to accommodate the different semantics might be worthwhile.

Finally, efforts can be made in the direction of creating mature support for handling the interruption of control flow in backward propagation transformations. Our prototype implementation for break support can be seen as a starting point, but there still is a long way to go before support equivalent to that available for forward propagation transformations will become a reality.

6.2 Pointer analysis

During the next part of this thesis we have tried to address the issue that data-flow transformations in Stratego do not have access to alias information. This information is vital if we want the transformations to be of any use in real life. Since for Stratego there has been little effort in this direction before our thesis, we have only managed to reach the point where we successfully run a pointer analysis algorithm over a toy programming language that we developed ourselves, called TOOL, and then we use the results to generate alias information. The pointer analysis that we have implemented is a flow- and context-insensitive one, described in the master thesis of Ondřej Lhoták [18], with some adaptation and particularization according to our needs. Our limited battery of tests have shown that the results computed by the algorithm are correct in that all aliases produced for a variable have been hand-checked as true potential aliases. We expect the analysis to be complete as well, according to a theoretical understanding of the implemented algorithm, but we do not claim to have proved this in any way.

Future work

Just as the first part, this second part also opens multiple directions for future work, that we discuss below.

To start with, we have mentioned that our implementation was of a flow- and context-insensitive analysis. An alternative direction would be to implement flow- and/or context-sensitive variants of pointer analysis as well. If the information we compute is valid for the entire program, alias information produced by a flow-sensitive analysis is only valid at discrete program points but, in turn, more accurate at each of these points. A context-sensitive analysis would differ by taking calling context into consideration. In this way, we would have different instances of the formal arguments of a method for each call site where that method is called, instead of having the same set used for all call sites, creating fake aliases.

Another direction of future work was already mentioned in the chapter about pointer analysis. It referred to improving the performance of the algorithm we have implemented by replacing the use of simple arrays for representing points-to sets with the use of binary decision diagrams (BDDs). The main benefit of this replacement is that BDDs offer maximal sharing of the information that appears in the points-to sets, which makes them orders of magnitude more space-efficient. Time efficiency is also improved because adding and removing elements from a set represented as a BDD is much faster than from a simple sequential array.

Moving on, we already mentioned the fact that our pointer analysis only runs over TOOL programs. This does not bring much help to Stratego users, since most of them deal with real languages. Because of this, it is obviously necessary that the pointer analysis be adapted to support languages like Java or PHP. What could be even more interesting as future research is how to extract the common parts of the implementation such that extending the analysis to cover a new language becomes a question of specializing a generic strategy.

A final problem that we would like to mention is the existence of simplification of the analyzed code as a necessary step of pointer analysis. While this makes the pointer analysis easier to write, it also makes the use of its results more unattractive, because most people do not want to work on a simplified version of the original code. However, the task of eliminating the simplification step is made quite difficult by the fact that field dereferences may contain any number of intermediate fields which, if not simplified, creates an exponential growth in the number of possible aliases with every new level of dereferencing. However, perhaps a different approach than ours could make this more easily manageable.

6.3 Extensible and customizable transformations

In the last part of our thesis, we have described two experiments, the first of which had the goal of producing a customizable transformation and the second of which set out to extend a transformation in a non-intrusive fashion. The customizable transformation was a transformation that read user annotations for TOOL methods and, based on these annotations and further logic, propagated typestate information along a program and used it in order to evaluate conditions that guarded the application of custom user actions. The extensibility experiment was designed to extend the TOOL language syntax and all the transformations that comprise our pointer analysis in order to provide support for arrays. Our main goal was to combine this support with the original syntax and transformations such that these would not suffer any changes. Both experiments have been successful and, in our opinion, provide interesting pointers for those who want to further pursue this track.

Future work

As we mentioned in chapter 5 as well, our experiments in these two directions took place in the context of and preceded what is expected to be a full-fledged research project called Transformations for Abstractions. The scope and goal of this project are defined at large in a proposal by Eelco Visser, that we have also briefly summarized in section 5.1.1. Issues such as "definition of domain abstractions", "mechanisms for open extensibility of transformations", "methods and patterns for design of open transformations", "constraints for independent extensibility of transformations" and "derivation of transformation extensions from definitions of abstractions" outline the main research direction of the aforementioned project. We believe that these topics provide sufficient indication of where future work related to our experiments is headed.

And, to end in a somewhat circular manner, we would also like to mention open compilers as a more distant possibility of future work that we hope will be reached in the years of Stratego development to come. If you have trouble understanding how open compilers are related to our work in this last part of the thesis, we invite you to restart reading this material with the... introduction.

Bibliography

[1] Dryad home page. http://www.stratego-language.org/Stratego/TheDryad.

[2] Java-front home page. http://www.stratego-language.org/Stratego/JavaFront.

[3] Metaborg home page. http://www.stratego-language.org/Stratego/MetaBorg.

[4] Stratego home page. http://www.stratego-language.org/Stratego.

[5] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to Analysis Using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114. ACM Press, 2003.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. *Stratego/XT Tutorial, Examples, and Reference Manual*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 2006.

[7] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundamenta Informaticae*, 69:1–56, 2005.

[8] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[9] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative Programming and Active Libraries (Extended Abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[10] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[12] Samuel Z. Guyer and Calvin Lin. An Annotation Language for Optimizing Software Libraries. In *2nd Conference on Domain Specic Languages*, pages 39–52, Austin, Texas, USA, 1999.

[13] Samuel Z. Guyer and Calvin Lin. Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries. *Proceedings of the IEEE*, 93(2), 2005.

[14] John Hennessy. Program Optimization and Exception Handling. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 200–206, New York, NY, USA, 1981. ACM Press.

[15] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[16] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail C. Murphy. Open implementation design guidelines. In *International Conference on Software Engineering*, pages 481–490, 1997.

[17] John Lamping, Gregor Kiczales, Luis H. Rodriguez, and Erik Ruf. An Architecture for an Open Compiler. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*, pages 95–106, November 1992.

[18] Ondřej Lhoták. Spark: A Flexible Points-to Analysis Framework for Java. Master's thesis, McGill University, December 2002.

[19] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[20] Karina Olmos and Eelco Visser. Composing Source-to-Source Data-Flow Transformations with Rewriting Strategies and Dependent Dynamic Rewrite rules. In Rastislav 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.

[21] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to Analysis for Java Using Annotated Constraints. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 43–55, 2001.

[22] Saurabh Sinha and Mary Jean Harrold. Analysis of Programs with Exception-Handling Constructs. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 348–357, November 1998.

[23] R E Strom and S Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[24] Alexandru D. Sălcianu. Pointer Analysis and its Applications for Java Programs. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2001.

[25] Tom Tourwé and Wolfgang De Meuter. Optimizing Object-Oriented Languages Through Architectural Transformations. In Stefan Jähnichen, editor, *Compiler Construction: 8th International Conference, CC'99. Proceedings*, volume 1575 of *Lec-*

ture Notes in Computer Science, pages 224–258, Amsterdam, The Netherlands, 1999. Springer-Verlag.

[26] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - A Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[27] Eelco Visser. Transformations for abstractions. In Jens Krinke and Giulio Antoniol, editors, *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 3–12, Budapest, Hungary, October 2005. IEEE Computer Society Press. (Keynote paper).

[28] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[29] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.