Static analyses for Stratego programs

BSc project report

June 30, 2011

Author name	Vlad A. Vergu
Author student number	1195549
Faculty	Technical Informatics - ST track
Company	Delft Technical University
Department	Software Engineering
Commission	Ir. Bernard Sodoyer Dr. Eelco Visser

Preface

For the successful completion of their Bachelor of Science, students at the faculty of Computer Science of the Technical University of Delft are required to carry out a software engineering project.

The present report is the conclusion of this Bachelor of Science project for the student Vlad A. Vergu. The project has been carried out within the Software Language Design and Engineering Group of the Computer Science faculty, under the direct supervision of Dr. Eelco Visser of the aforementioned department and Ir. Bernard Sodoyer.

I would like to thank Dr. Eelco Visser for his past and ongoing involvment and support in my educational process and in particular for the many opportunities for interesting and challenging projects.

I further want to thank Ir. Bernard Sodoyer for his support with this project and his patience with my sometimes unconvential way of working.

Vlad Vergu
Delft; June 30, 2011

Summary

Model driven software development is gaining momentum in the software engineering world. One approach to model driven software development is the design and development of domain-specific languages allowing programmers and users to spend more time on their core business and less on addressing non problem-specific issues. Language workbenches and support languages and compilers are necessary for supporting development of these domain-specific languages.

One such workbench is the Spoofax Language Workbench. Within Spoofax, parsers are generated from SDF syntactic definitions and compilers are specified in the Stratego programming language. The Stratego programming language is a powerful tools for specifying software rewriting systems, but has one drawback: since its creation it has been untyped. The absence of a (static) type system exposes the programmer and the user to risks of runtime failures, and inherently exposes the user to the internals of the compiler. To attempt to mititage these risks and to serve as a basis for further development, this project has been dedicated to designing and implementing a set of static analyses for Stratego code.

Table of Contents

List of Figures

1 Introduction

Well typed programs can't go wrong

This is the motto behind the drive for statically typed languages. This project posed the question of whether something similar is possible for Stratego. In the current case a tradeoff had to be made between strong typing and uncompromised flexibility of the language.

This reports presents the cause, means, and effect of such a development for the Stratego programming langauge. It begins with a brief overview of the problem (Section 2) and of the organisation behind the project (Section 3). An in detail account of the anatomy of a Stratego program is given (Section 4) which should prepare the reader for a detailed discussion of the design of basis for static analysis of Stratego code (Section 5). After a brief look at two notable challenges encountered during implementation (Section 6) and a short evaluation of the results and methods (Section 7) the report finalizes with detailed suggestions for future directions (Section 8).

2 Problem definition

Stratego [9,1] is a functional language for software transformation. Together with SDF for grammar specifications and Spoofax they form a powerful and flexible language workbench built on top of the Eclipse IMP framework. With Spoofax the task of specifying textual domain-specific languages and building their compilers is significantly simplified. Within Spoofax, SDF is used to specify language syntax, whereas Stratego is used to specify editor services such as semantic error checking for the target language. Depending on the complexity of the target language, the number of lines of Stratego code to be written for the full specification of editor services and code generators can easily reach into the thousands scale. The chance of errors increases with complexity. Statically typed languages attempt to mitigate error risk by detecting potential errors statically:

> Well typed programs cannot go wrong

Stratego is not a statically typed language. In fact, Stratego is completely untyped. This exposes the language designer, and - more importantly - the user of the built compilers to errors caused by type incompatibility. One of the reasons for Stratego not having a type system is the strive for maximum flexibility. The challenge of this project is to define a number of static analyses for Stratego programs that would warn developers of potential problems, while in no way limiting the flexibility of the language:

> Make it safe, without making it more difficult.

3 Project definition

This was organized as a Bachelor of Science project, with the main beneficiary being the TU Delft, under the supervision of Dr. Eelco Visser - creator of the Stratego language. The ambition of creating a full type system for Stratego within the ten weeks allocated for this project was highly unrealistic, but this was known from the beginning. Goals - which are discussed in the following section - where laid out from the beginning.

3.1 Goals

1. Give semantics to signature language
2. Detect violations of term signatures
3. Detect rules that never succeed
4. Identify type preserving rules

The first (1) goal is actually an indirect goal - necessary for the remainder of the goals. Terms are used in the Stratego language for representation of programs as abstract syntax trees. Term signatures - definitions of terms - are generated from the SDF syntax definitions (more about signatures in Anatomy of a Stratego program).

The Spoofax editor for Stratego currently performs no static semantic analysis on the Stratego code. One benefit of this is to preserve maximum flexibility of the language. Abstract syntax tree fragments that appear in Stratego code are also not checked for consistency against their declared structure. On one hand this is beneficial to flexibility because it allows for gradual transformation of the trees where the initial and final trees are valid but at intermediate rewrite stages invalid trees can be produced. On the other hand this exposes the programmer to a high risk of error. It is better practice to define an intermediate language for internal representations. The second goal (2) of this project is to detect violations of the declared tree form thereby detecting construction of invalid abstract syntax trees. Ensuring valid ASTs increases confidence that they represent structurally valid programs and therefore reduces the risk of runtime errors.

Additionally, *rewrite rules* are used to specify abstract syntax tree transformation. It was a goal (3) of this project to identify rules that can never succeed given valid syntax trees: if a rule only matches invalid trees, it is unlikely to ever produce valid trees itself.

The final goal (4) stems from a typical Stratego usage scenario where a tree is generically traversed and term rewriting is performed at different positions. The transformation positions are dictated by the rewrite rules that are specified. The important aspect here is that the traversal itself is not handled by the rewrite rules. When performing generic traversals it is important to know that the rules applied during the traversal preserve the structure of the tree. More specifically, it may be useful for the programmer to know when a generic traversal of the

tree actually alters its structure (a typical type preserving traversal strategy is associated with registering entity declarations).

Two extra requirements were placed on the potential results of the project: - Safety markers instead of errors - Report only if sure. The first requirement states that for some classes of errors it may be beneficial to the user to receive information about safety of a particular code fragment rather than a normative error. The reasoning behind this being that descriptive safety information reduces programmer interruptions. More so, the programmer should be free to ignore any warnings.

The second requirement stems from a well known fact about strong type systems: they give false alarms. These false alarms limit the flexibility of the language. It was an a priori consensus that a strong statical type system for Stratego was unachievable (perhaps impossible without severely harming the language), instead we would limit to reporting errors only if we were sure that a code fragment actually contained an error: no false positives.

The extent to which these goals were met varies, details of these results can be found in the Evaluation section of this report.

3.2 Stakeholders

Under the assumption that the presumptive set of static analyses would actually be accepted into use, three major stakeholders are identifiable:

1. Upcoming Stratego programmers
2. Language maintainers
3. Dr. Eelco Visser

New Stratego programmers are probably the most important target group. They are the users most at risk of involuntarily creating the type of errors that this project strives to detect. The existence of tools to support them during the (steep) initial learning curve is their stake.

Existent maintainers of languages developed using Stratego would prefer if their source code continues to function - what we develop must not reject previously written bad code. They would probably prefer an opt-in system, as covered by the goals. While medium sized projects with experienced maintainers would not benefit significantly by the products of this project, it is expected that as the complexity of the projects increases, errors seep in easier.

As the creator of Stratego, Dr. Eelco Visser has a personal interest in seeing the set of features and tools associated with Stratego being developed further.

Informally, there is a fourth stakeholder:

1. Me

My stake in this is threefold: successful completion of the BSc project is a requirement (1); future plans involve further research into language workbenches and facing the difficulties associated with designing static analysis systems is a necessary experience (2); upcoming year will find me as a TA for the Compiler

Construction course at the TU Delft, where students have to develop a compiler for the MiniJava language within Spoofax. Most of them would be using Stratego for the first time (type 2 stakeholder), and their smooth learning curve is in my interest.

3.3 Planning

The initial plan with the project was to perform some initial analysis (first three weeks) and then begin implementation. This was proven to be infeasible in reality. The hope was that the problem can be divided into sub-problems and these tackled separately. After initial research it became evident that indeed some sub-problems can be defined, but these can only be approached after some supporting code would be written. Initial research and the general direction taken had to be appropriate for all sub-problems: a non-trivial requirement in the current case. Significant decisions had to be taken early on, which required a large amount of time for research.

On a weekly basis, Figure 1 shows how time was allocated to different parts of the project:

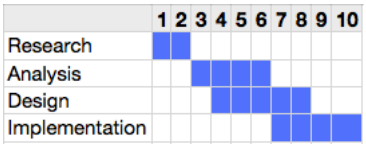

Fig. 1. Project time allocation

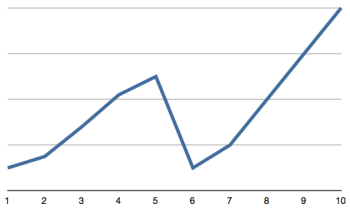

Fig. 2. Project weekly progress

Figure 2 shows how progress was actually made throughout the ten weeks. The dip in progress at approximately six weeks corresponds with the realization that although the way of working was correct important details had been overlooked which if ignored would have led to an imcomplete approach. As a consequence most of the implementation existent at that moment had to be refactored. In concordance with the way of working code maintainability is insured

by detailed documentation of the reasoning behind a specific implementation, by in-place comments and by separation of code into modules by responsibilities. At the time of this project, Spoofax testing facilities for language definitions were still in early development and as such were not used.

4 Analysis

4.1 Anatomy of a Stratego program

A Stratego program is composed of one or more modules. Each module can specify imported modules, constructor declarations, rewrite rules and strategies. This section offers an overview of how constructor declarations and rules look like and the semantic behind them. Strategies are not checked by the product of this project due to time constraints.

Constructor declarations Constructor declarations specify how the nodes in the abstract syntax tree look like. Here is an example:

```
Entity : ID * List(Property) -> Definition
```

The left hand side declares the name of the node, while the right hand side of the colon specifies the number of and sorts of arguments that the node receives. In this case two arguments of sorts ID and List of Property, respectively. The right hand side of the -> specifies the sort of this Entity node. The entire declaration can be interpreted as a function called Entity that transforms the combination of the sorts ID and List(Property) to a sort called Definition. From this interpretation we can draw the following conclusion:

the constructors reside in a different namespace than the sorts

In other words, constructor names cannot be used as sort names. Just like in polymorphic languages functions can be overloaded in Stratego constructor declarations can be overloaded:

```
Entity: ID * List(Property) -> Definition
Entity: ID * ID * List(Property) -> Definition
```

The interpretation of this is that the Entity node can either have two or three arguments. Constructor declarations in Stratego are considered overloaded even if the entire right hand side of the colon is different from other declarations of the same constructor.

A special type of constructor-less declaration is available in Stratego:

```
: Definition -> ExtendedDefinition
```

This can be interpreted as the sort Definition being equivalent with the sort ExtendedDefinition. This type of equivalent allows declared sorts to be polyvalent and is central to most generated declarations representing abstract syntax trees. Figure 3 shows the sort dependencies in the Entity language (example language generated automatically by Spoofax):

The boxes represent declared sorts, the arrows indicate dependencies of sorts and the arrow labels indicate the constructor name which represents that dependency.

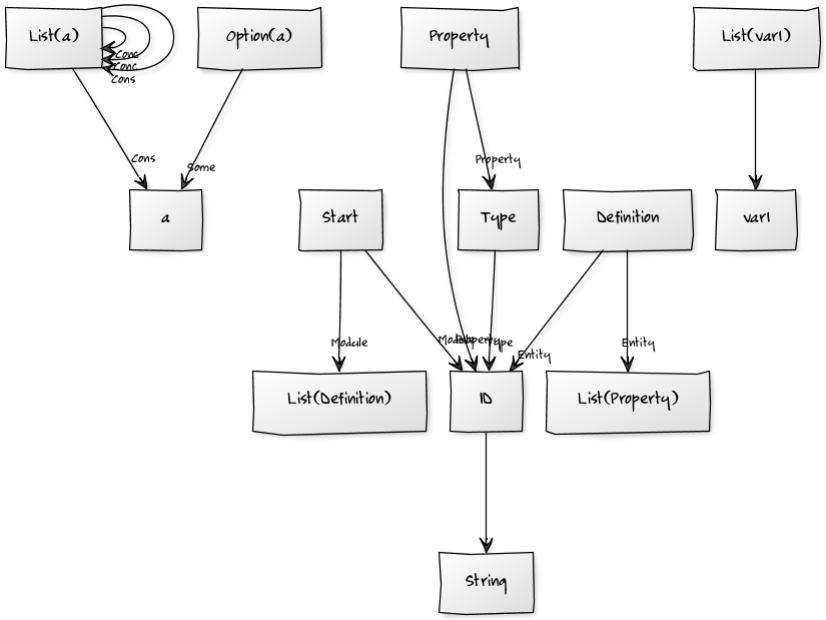

Fig. 3. Sort dependencies in Entity language

One issue with the current state of affairs in the Stratego constructor declarations is that the more we talk about them, the more semantics we actually give to them. Indeed, there really is not much meaning to the declarations currently. The constructor name is declared and can be called. There currently is only one check related to sorts and declarations that is performed: at constructor usage positions it is checked that a constructor with the used name and the given number of arguments is declared.

This was identified as a potential point of improvement, after all if you want to have valid ASTs you should have detailed specifications of what these valid ASTs look like, and you should check usages against those definitions. A significant part of this project has been focused to giving/taking meaning to/from constructor declarations. This topic will be elaborated further in the Design section of this document.

Rewrite rules The actionable parts in Stratego code are given by rewrite rules and strategies. The scope of this project does not extend to Stratego strategies, which are not analyzed, this is reserved for future work. The differences between rules and strategies are mostly limited to different syntax; during compilation all rewrite rules are re-written to strategies.

The rewrite rules in Stratego take the following form:

```
to-type:
NotExpression(t) -> BooleanType() where t => BooleanType()
```

In this case to-type is the rule name. The rule name is followed by term match - NotExpression(t), a horizontal arrow to the right followed by a term build - BooleanType(). When called on the AST (fragment) the rule will attempt to match the input AST to the term in its declaration. If the match succeeds the rule will replace the input term with the built output term. We talk about matching and building terms in the next sections.

Additionally, each rule may have one or more where clauses. These allow fine grained control for matching the input. Rules can also specify one or more with clauses which are meant to extend the term building space for more complex rules. Where and with clauses are not being analyzed as part of this project, and should be part of future developments.

In addition to the parts already mentioned, each rule may take zero or more rule/strategy arguments and zero or more term arguments:

```
to-type(sA,sB|tA,tB): _ -> tA
```

The strategy arguments allow this rule to generically apply transformations to terms using the strategy of the callers preference. The term arguments allow this rule to be given terms that are not part of the input AST and hence are not matched against. The strategy and term arguments in the rule declaration are separated by a vertical bar |.

In the example above the output term is built without any calls to other rules, but calls can be given at any depth level within the output term. Even more so, it is not compulsory that terms are explicitly built. The following is a perfectly valid Stratego rule:

```
eval: Sum(x,y) -> (x,y)
```

where the rule addi (integer addition) is applied to the two arguments of Sum.

Rules in Stratego are commonly overloaded to provide different rewrites depending on the term being matched. Overloading rules in Stratego is simply a matter of declaring new rules with the same name (and strategy and term arguments). On any given input term at most one rule fires. In other words, the first rule to match the input term is fired. The order in which overloaded rules are tried is neither specified nor guaranteed. Thus having rules that have overlapping matches can (and will) give unpredictable results (detecting these is a topic of future work).

Having had an overview of Stratego rules, the next sections discuss in some depth the meaning of matching and building terms which are the core of the Stratego language.

Anatomy of a match Two ASTs representing different programs (even those that are signature compliant) are clearly very different from each other. Writing rules that can always succeed on every possible node of an AST is impossible. Instead, Stratego rules are declared to match against the input term. The match

can be performed with fine control, ranging between matching any input to matching only a very specific shape of an AST node.

Matching is performed using the constructors declared and can also contain wildcards (_), variables are literals. Heres an example of rule containing all of these:

```
main-body: mdec@Method(main,Void(),_,body) -> body
```

In this example the rule matches declarations of void functions with the name main without looking at its arguments. Additionally, the term in the actual input term at the positions of the fourth argument of Method will be available as the variable body for the entire scope of the rule without mutability. Similarly, the variable mdec will contain the entire term if the rule matches. Applying the rule on the term

```
Method(main,Void(),[],[])
```

will match successfully and the variables mdec and body will be holding Method(main,Void(),[],[]) and [], respectively. Applying the rule on the term

```
Method(mein,Void(),[],[])
```

will fail because the string *mein* is not identical to the string *main* from the rule declarations.

Matching can be done on the contents of lists as well:

```
[x,y,z][x,y|_]
```

matching a list of three items and a list of two or more items, respectively. Matching on the middle of a list

```
[|mid|]
```

is unfortunately not possible in Stratego.

A powerful tool is using just bound variables to further refine the match. Lets imagine the hypothetical situation of wanting to rewrite head-recursive functions (as opposed to tail-recursive). One could use the following match

```
Method(name,,,body@[MethodCall(name,)|])
```

were the rule only matches on a Method which has a recursive call as the first statement in the body. A less hypothetical match would be to detect perfectly symmetrical non-final subtrees:

```
TreeNode(x@TreeNode(,),x)
```

Where only TreeNodes with identical children that are not leaves would be matched.

Anatomy of a build In the previous section we briefly discussed how terms are matched, we now look at the other stage of rewriting - building output terms. Terms are built using constructors, literals, variables and results from rule calls. Heres an example which uses all of them

```
Method(main,Void(), <to-java> args, <to-java> body)
```

where a Method term is built. To build this term the calls to to-java have to succeed and their result will be used to build the node. Similarly to matching, lists can be built by fragments:

```
[x| xs]
```

This is what one should know about building terms, in a nutshell. In the following section we put this knowledge to some use, thinking about how signatures, matches and builds can be used to learn something about the correctness of the compiler.

5 Design

We cannot test the correct functioning of rules statically, this can only be done at runtime. So then there is no checking possible. That is not entirely true. Indeed, without knowing what the input AST looks like, we cannot do much. However, if we give semantics to constructor declarations, each constructor literal in the code has a sort equivalent. We can formalize this as a definition:

Each valid node has a *valid* **sort dual**

It means that for every node literal that we encounter there must exist at least one representing sort and that if we cannot find such a dual then the literal is invalid. This flows naturally from the requirement that constructors must be declared with sorts on the right hand side. Note that because of sort equivalence declarations we cannot guarantee uniqueness of the sort dual (this is central in what will follow).

Abstract syntax trees are composed of nodes and leaves, hence we can take the above definition farther and state that:

Each valid abstract syntax tree has a *valid* **sort dual**

Again, if no such dual can be found for an AST then it means that the AST is invalid. We can extend this for rules as well, for they process and return node literals:

Each rewrite rule has an equivalent sort rewrite rule

This says that if we are given any rewrite rule we can construct an equivalent rewrite rule operating on sorts. This also implies that if such a rewrite rule cannot be constructed, or that rule fails, the original rule was either matching on or building an invalid AST node.

None of this should be anything new. All typed languages work on this assumption that each literal operation must have an equivalent type operation.

Earlier we used the words valid and invalid repeatedly, but in the context of ASTs we should clarify them:

An AST is valid if and only if it has a sort dual

At first sight this seems reasonable, but we would prefer this:

An AST is valid if and only if and only if it represents a valid program

But this no longer makes any sense for the validity of a program cannot be guaranteed solely on its syntax (Abstract **Syntax** Tree). Even more so, the program is dependent on the languages compiler, which we are trying to analyze after all (it would be the holy grail of software engineering if we could pronounce on the correctness of an as yet unwritten program). There seems to be this hard line between syntax and semantics that we cannot cross. The approach of this project attempts to play with this line. If we can capture some basic semantic

rules of the language in the constructor declarations we can make sure that our compiler obeys them. The reason we think we can blur this line slightly is that it seems that what is a semantic problem in the original program gradually becomes a syntactic problem as we increasingly abstract from it in the compiler. To bring this closer to the idea of a sort dual: what is a semantic problem in the source program should become more of a syntactic problem in its sort dual. Or, more importantly to this analysis, what is a syntactic problem in the sort dual, could translate to a semantic problem in the target program.

5.1 Overview of the analysis workflow

The static analysis normally takes place in the following order:

1. Desugar
2. Rename
3. Map
4. Project
5. Check

Each of these stages generally corresponds to a full pass of the AST. In the rename stage (2) all the variables and entities are renamed to unique names so their scopes can later be ignored. The map stage (3) stores all declarations to internal representations, while the project stage (4) performs name and sort projections from values (or variables) to their respective types. In the final stage (5) the projections are checked and errors are reported if found. Sometimes a clear separation between projecting and checking does not exist, because the projections are performed during collection of the errors.

The analysis designed here does not follow such a clear separation between phases:

1. Rename sorts & variables in signatures
2. Map sorts
3. Project sort variables
4. Rename sorts & variables in rules
5. Map & project rules to internal signatures
6. Check signatures and rules

The desugared AST is provided by the existent analysis. All constructor declarations are mapped to internal representations and once all declarations have been mapped the sort of sort variables are inferred (see next section). Given the internal representation of sort definitions, rules are mapped and sorts partially projected. There is not much mapping to do for rules without projecting sorts, hence mapping and projecting happen in the same pass of the AST. In the final phase (6) errors are collected for signature declarations, rule projections are finalized and errors are collected.

Coupling with existent analysis The designed analysis has to couple into the existent analysis performed by the Spoofax Stratego editor. Because of the experimental nature of the analyses to be designed it was decided to reduce coupling between the existent and the new analysis. It is expected that code duplication will be minimal because the two analyses should complement each other. The new analysis will hook into the constructor declaration and rule declaration strategies while the error/warning/note collection will be provided by overloading of the existing error generating rules.

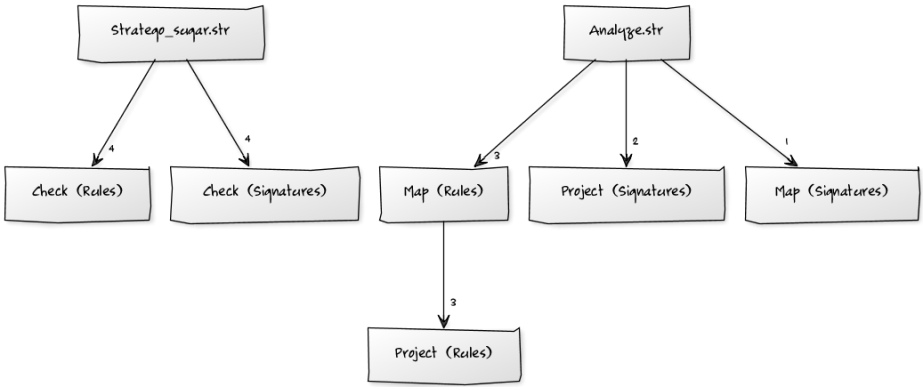

Fig. 4. Coupling with existent analysis

Figure 4 shows how calls to the new analysis take place. The arrow labels indicate the AST pass in which the call takes place. The reduced coupling above allows the two analyses to be developed separately. At a later stage, if it is proven that the new analyses are successful, the two should be integrated so that they can benefit from each other.

5.2 Sort/constructor declarations

In the beginning of this report it was said that there arent much semantics currently associated with constructor declarations. It is not that signatures do not mean anything, but that they are not used sufficiently in the static analysis. This deserves some elaboration:

```
Module : ID * List(Definition) -> Start
```

This states that the Start sort results from an ID sort and a List of Definition sorts. In other words, the node Module is of sort Start, and receives two nodes as arguments: one of sort ID and the other a List of Definition. This is the meaning carried by constructor declarations and this is what we would like to use during the analysis. However, the current analyses ignores this meaning and

simply checks that literals of sort Module receive two parameters. No further checking is performed.

Earlier we said that it would be desirable to attempt to capture some semantics in the constructor declarations. Lets look at the following declarations:

```
Number: Int -> NumExpr
Number: Double -> NumExpr
Sum: Expr * Expr -> Expr
NumExpr -> Expr
```

The arguments of the Sum node can be either Sum nodes or leafs (Number). Its typical during desugaring to perform some constant folding. The intermediate result generally corresponds to one side of the tree having been folded. We should be able to capture partially folded trees in the signatures:

```
Sum: Expr(a) * Expr(b,c) -> Expr
```

This tells us that a partially folded Sum is a combination of a 1-argument Expr and a 2-argument Expr (a very serious drawback of parameterization of sorts is discussed in the final chapter of this report). This can only mean that the left side of a Sum tree has to be a Number. But what about the variables a,b and c? Is there something we could say about them statically? By looking at the definitions, we can see that variable a can be either a NumExpr(Int) or a NumExpr(Double). Similarly variables b and c can be an Expr(Expr(x),Expr(y,w) or an Expr(Expr,Expr). There are two interesting facts that can be inferred here. Firstly, if we would replace variable c with variable b such that the declaration became:

```
Sum: Expr(a) * Expr(b,b) -> Expr
```

we could force that the right-subtree of the Sum node is sort symmetric. This can be taken farther:

```
Sum: Expr(a) * Expr(a,a) -> Expr
```

This states that the right-subtrees of a Sum node have to be of the same sort as the left-subtree node. In the context of the current definitions we can see that this could never occur:

```
Sum(Number(1),Sum(1,40))
```

should not be a valid AST because the sort of 1 is Int while the sort of the left-subtree is NumExpr(Int). Would we be able to detect this error? Yes. By simply projecting the tree nodes to sorts we should get to a contradiction, in this case:

```
Expr(Expr(Int),Expr(Int,Int))
```

is not compatible with:

```
Expr(Expr(a),Expr(Expr,Expr))
```

because the sorts Int and Expr are not compatible. Clearly this deals with invalid constructor declarations, and this is something that we can detect statically by just looking at the signatures. When we infer the types of the variable a we can see that we reach a contradiction: Int/Double is not compatible with Expr(NumExpr) or Expr(Expr,Expr).

The second noteworthy fact is that two declarations for the Sum node are overlapping:

```
Sum: Expr * Expr -> Expr
Sum: Expr(a) * Expr(b,c) -> Expr
```

In the case of Sum literal such as Sum(Number(1), Sum(Number(1), Number(40))) we would not be able to tell which of the declarations should be used to convert this to sorts. Would this overlap make sense in reality? Probably not, not unless the resulting sorts of Sum are different. We could regard this as non-overlapping:

```
Sum: Expr * Expr -> Expr
Sum: Expr(a) * Expr(b,c) -> LExpr
```

This would mean that the literal from above either be an Expr or a LExpr corresponding to two transformations paths of equal priority (priority cannot be expressed in sort declarations).

Let us look at the different example:

```
A: String * Double * Double -> E
B: String * Int * Int -> E
C: String * String * String -> C
D: Int -> D
D: Double -> D
D: List -> D
T: D(a) * E(String,a,a) -> T
```

We can see that the declaration for T is polymorphic on a, but restricted to Int or Double:

```
T: D(Int) * E(String,Int,Int)
T: D(Double) * E(String,Double,Double)
```

Any other sorts for variable a would violate either the declaration for sort D or that for sort E. That said, if we encounter an AST fragment such as:

```
T([1],A(op,6,6))
```

we would know that it is invalid because it would require us to coerce the variable a in the declaration from List to Int. This is a classic error, where after processing (folding) a singular list was returned instead of it being unpacked to

```
T(1,A(op,6,6))
```

which would be a valid AST.

We end this section with an example of how we store internally the information gathered from the declarations. For every declaration:

```
Sum: Expr(a) * Expr(b,c) -> LExpr
```

we store an internal representation:

```
ConstSig(Sum,[Expr(a),Expr(b,c)],LExpr,freevars)
```

where the ConstSig term is declared as follows:

```
ConstSig: String * List(Sort) * Sort * ConstraintSet -> ConstSig
```

The first three arguments are self-explanatory. The last argument is a representation of sort constraints that sort variables in the declaration must meet. This is a mechanism which allows us to express inferred limitation on the sorts of the variables. Sort variable inference and constraint solving are discussed in a later section.

5.3 Sort compatibility

The previous section mentioned sort compatibility and sort coercions, an explanation of how this mechanism works is overdue. All languages have a set of built-in sorts. They allow user sorts to be built on top. We define the following primitive sorts for Stratego:

- ATerm
- Int
- Double
- String
- Tuple
- List
- Option

To be noted that the sort ATerm is the parent sort of all sorts. All terms are of type ATerm. One could argue that the List and Option sorts should not be treated as built-in, as they can easily be composed:

```
Cons : a * List(a) -> List(a)
Nil : List(a)
Conc : List(a) * List(a) -> List(a)
Some : a -> Option(a) None : Option(a)
```

Anything that can be composed is not elementary (primitive). We define some compatibility between the primitive sorts. Of course each sort is compatible with itself. For the List sort there is no difference made between full lists or empty lists; the same applies for Option. Every sort is compatible with itself. At the moment it also holds that none of the sorts is compatible with another sort. This may only appear strange for Double and Int where currently there is no conversion possible between them. Normally, one should be able to use an Int in place of a Double. This covers the built-in sorts, next we discuss how we can test compatibility of user declared sorts.

As a design decision, sorts with the same name but with different arities are not compatible with each other unless otherwise specified through sort injections. Whether this approach is more beneficial than it is harmful remains to be seen. This discussion will remain the topic of future work. For example, lets have the following two sort declarations:

```
X: Int * String -> A
Y: String -> A
XX: A -> B
```

The sort argument of the last declaration refers to the sort A, more precisely it refers to either of the two declarations of sort A. The general principle here is that unparameterized sort references refer to the entire pool of declarations, whereas the parameterized sorts refer to the subset of potentially compatible declarations - specifying extra information should refine/limit the subset of sorts that are being referred.

We explain how these sort unifications work by using a few examples. Lets imagine the following declarations:

```
X: Int * Double * a -> X
X: Double * String * a -> X
Y: X(Int,Double,String) * Double -> Y
```

To test that the sort X(Int,Double,String) is compatible with any of the declarations of sort X, we first look whether there exists a declaration of X with arity 3. In this case it exists and they are the first and the second declarations. To determine compatibility between its declaration and its use, we have to test whether each of the use arguments is compatible with each of the declaration arguments, and we do this for each of the contending declarations:

```
(Int,Int),(Double,Double),(String,ATerm) (Int,Double),...
```

Clearly this is the case for the first declaration and it is not for the second. The result of testing the sort validity of the usage:

```
X(Int,Double,String)
```

is that it is valid and in this case refers to the first declaration of X. If the only declaration had been the second one, then the sort usage would have declared invalid due to sort incompatible arguments.

In the case of nested sorts, where a non-primitive sort is given as an argument to another sort, the procedure is identical. If all the sorts are compatible with the declarations then the usage is sort valid.

The examples so far have only referred to sort declarations, we now give an example from actual AST terms:

```
VariableAssignment : Variable * Expression -> Statement
Variable : String -> Variable NewExpression : String -> Expression
```

And the following usage (whether this is a build or match is irrelevant for this discussion):

```
VariableAssignment(Variable(varA),NewExpression(Oops))
```

The first step would be to convert this to a sort equivalent, beginning with the deepest terms, to obtain the following:

```
Statement(Variable(String),Expression(String))
```

And then test the sort validity of this composed sort as previously explained, resulting in a verdict that the sort is valid - confirming that the AST node is conforming to its declaration.

In case the AST node was:

```
VariableAssignment(Variable(varA),NewExpression(42))
```

after projecting it to sorts we would obtain:

```
Statement(Variable(String),NewExpression(Int))
```

which we would immediately classify as invalid because the sort NewExpression does not accept Int as an argument.

Sort injections The previous section intentionally avoided using sort injections, because the method for interpreting them is still unclear. On one hand sort injections allow us to specify sort equivalence explicitly and maintain a higher level of abstraction. On the other hand, their effects on sort compatibility can be unpredictable: in many grammars, every sort is compatible with every other sort by following paths through sort injections. This is discussed as part of future works in the last chapter of this report. Notwithstanding we will explain the current approach to determine sort compatibility in the presence of sort injections.

The last example of the previous section contained a modified signatures fragment, where we avoided the use of sort injections. The original is:

```
Variable  : ID -> Variable
VariableAssignment : Variable * Expression -> Statement
NewExpression : ID -> VariableExpression
: VariableExpression -> Expression
: String -> ID
```

We can see that for our original AST node

```
VariableAssignment(Variable(varA),NewExpression(Oops))
```

it's sort equivalent this time is

```
Statement(Variable(String),VariableExpression(String))
```

which does not immediately appear compatible with the original declaration for the Statement sort. When we encounter a situation when the two sorts under comparison are not compatible, we attempt to lift one of them through sort injections to the other sort. If this is possible we conclude that the lifted sort is equivalent to the other and can be substituted for it. In the case of the example we perform the following lifts:

```
String -> ID
VariableExpression -> Expression
```

and conclude that the original sort is compatible with the signatures, and indeed that the original AST node is compliant with the signatures.

Note that the sort injections actually represent a directed graph where the vertices are the sorts and the directed edges represent the direction of equivalence. To lift one sort to another we are actually attempting to find a path from the former to the latter. Currently we are doing this using a depth-first search. For a discussion of alternatives see the last chapter of this report.

5.4 Rule sort-signatures

We have discussed previously how sort information can be collected from the signatures. To use this information in the analysis we have to analyze rewrite rules. We start with an example; let the following signatures:

```
Sum  : Expr * Expr -> Expr
Number : Int -> NumExpr
Number : Double -> NumExpr
: NumExpr -> Expr
```

Weve said earlier that a rewrite rule takes the form:

```
eval(s1|t1): Sum(x,y) -> (x,y)
```

In this example we have no information about the sorts of the arguments s1 and t1. We could infer that the former operates on tuples of two elements. In this case the two elements are x and y. These variables can represent either Sum or Number. We can therefore learn that s1 operates on a 2-tuple of the combinations between Number and Sum. In terms of sorts these are combinations of size two of the sorts Expr. We can only assume that the strategy s1 can handle all of these combinations. More importantly we have no knowledge of the term that s1produces. Stratego syntax allows us to explicitly state what the sort signature of the argument strategies is:

```
eval(s1 : (Expr,Expr) -> Expr | t1):
Sum(x,y) -> (x,y)
```

We can now see that s1 promises to return an Expr given that it receives a 2-tuple of Expr and Expr. We can be more specific:

```
eval(s1 : (Int,Int) -> Int | t1) : Sum(x,y) -> (x,y)
```

and learn that s1 is probably a strategy performing addition on integers. An analogous explicitation is permitted for the term arguments:

```
eval(s1 : (Int,Int) -> Int| t1: Int): Sum(x,y) -> (x,y)
```

If such an explicit information is not given we add generic sorts during the desugaring stage:

```
eval(s1 : ATerm -> ATerm| t1: ATerm):
```

For each rule declaration we store internally the knowledge accumulated as a term, analogous to the way we store constructor declarations:

```
RuleSig(eval,[s1],[t1],
[(Expr(x,y),m-constr)],
[(Tuple(x,y),b-constr)],constraints)
```

The first argument is the name of the rule. The second and third arguments are lists containing the names of the strategy and term variables, respectively. Their sort signatures are stored in the constraints; a RuleSig for s1 and some sort for t1. Besides the benefit of summarizing all the sort information of the rule, this internal representation of rewrite rules allows us to check rule applications against their declarations.

In the next sections we discuss ideas for determining the sorts for term matches, rule calls and term building within rules.

Sort of a match The match part of a rule is very important for performing a static analysis. By projecting a term match to a sort match we can on one hand check that the rule fires on valid ASTs and on the other hand gather information that allows us to check the validity of the resulting term. The reasoning behind checking the former is that a rule that only matches invalid ASTs is very likely to either:

1. Produce invalid ASTs
2. Never match in real life

Given the following signatures:

```
Sum : Expr * Expr -> Expr
Sum : Char * Char -> CharExpr
Number : Int -> NumExpr
Number : Double -> NumExpr
Char : ID -> Char
: NumExpr -> Expr
: String -> ID
```

And the following rule:

```
eval: Sum(Number(x),Number(y)) -> Number( (x,y))
```

To obtain the sort of the match part, we look up the declarations pertaining to the constructors, beginning with the deepest first, in this case Number. There are two declarations for Number, one which wraps an Int and one which wraps a Double. We can conclude that Number(x) is either of:

- NumExpr(Int)
- NumExpr(Double)

The same is perfomed for Number(y). Since all the arguments have been examined, we go one level up and look at the Sum constructor itself. In this case there are two declarations which we expand to contain the argument sorts from the deeper level:

- Expr(NumExpr(Int),NumExpr(Int))
- Expr(NumExpr(Int),NumExpr(Double))
- Expr(NumExpr(Double),NumExpr(Int))
- Expr(NumExpr(Double),NumExpr(Double))
- CharExpr(NumExpr(Int),NumExpr(Int))
- CharExpr(NumExpr(Int),NumExpr(Double))
- ...

At this stage we can eliminate all the CharExpr sorts because the arguments are not compatible with declaration (Char is not compatible with NumExpr) and we are left with Expr(NumExpr,NumExpr). We result that the input sorts of this rule can be the either of the possible Expr(NumExpr,NumExpr) sorts.

Expanding the sorts for all the possible combinations of argument sorts is indeed very space consuming. This issue can be partly addressed by introducing variable constraints. The reason we have to expand the sort is because of the polymorphic NumExpr. Instead of expanding we can record that the variables x and y have certain constraints and leave the sort unexpanded:

```
Expr(NumExpr(x),NumExpr(y))
```

under the constraint that:

```
[ x is Int or Double ] AND [ y is Int or Double ]
```

This allows us to at least delay the expansion of the sort so that we only take the space expensive step of fully resolving it when absolutely necessary. Variable constraints are discussed in detail in a later section.

We have seen how constructor literals are projected to sorts. One interesting topic is how list matches are projected to sorts. Given the following match:

```
[Sum(,),Sum(,),z | xs]
```

What should its projected sort be? The approach that follows is inspired by the way the Scala type system [6] infers the types of list literals. Just like in Stratego, Scalas lists are parameterized with the type of the elements. Constructing a list:

```
List(1,2,3)
```

will result in a list parameterized with type Number. Constructing a list such as:

```
List(1,2,3)
```

will result in a list parameterized with type Any. The head and tail of the list then have the same types as the full lists: the type of the head of the first list is of type Number and its tail of list List(Number), whereas the type of the head of the second list is of type Any and that of the tail List(Any).

We would like to do something similar. Going back to our earlier list matching example, we should acknowledge that there is a wealth of sort information contained in this match. Firstly, the first two elements of the list are of the Expr sort. Secondly we could assume that the sort of the variable z is also Expr unless otherwise previously constrained. Having this we can assume that the sort of the sublist containing the first three elements is List(Expr). This allows us to assume that the sort of the xs variable is List(Expr) unless otherwise previously constrained. In an example where the elements of the list are not compatible:

```
[ Sum(,), Sum(,), z | [Number(x) | xs] ]
```

This of course is not a realistic example, because any programmer would have written

```
[ Sum(,), Sum(,), z, Number(x) | xs]
```

Notwithstanding, the two are equivalent in functionality, and hence should be equivalent in sorts. Just like before, we can conclude that the sort of the match is List(Expr). Why is that? It is because Sum is an Expr, and the sort of Number is NumExpr which can be lifted through sort injections to Expr. Thus the list is of type Expr. On the other hand, a list match such as:

```
[ hello, 42 | xs]
```

will have the sort List(ATerm) because the String and Int are not sort compatible. The type of the variable xs is thus also List(ATerm). As expected, there are some drawbacks with this approach for sorting lists, these are discussed in the future work chapter.

Sort of a call The previous section briefly explained how the sort equivalent of term matches are determined. Any rule that performs some realistic transformations makes use of other rules to build its output term. We now briefly look at the sort of rule applications. A typical rule application is of the form:

```
<addi> (x,y)
```

There are three situations that can occur. The first is the normal situation where the call succeeds on the given term. The second is where the rule does not actually match the input term. This type of failure would be detected at runtime. We can attempt to reduce the risk of this happening by trying to detect this statically. The approach is very simple. We check to see whether the sort of the given input term, in this case (x,y), is compatible with any of the input sorts stored in the internal rule signature. If at least one is compatible, it is possible that this rule will succeed.

The third possibility is that the called rule does match on the input term but that it either produces a malformed AST or fails to build its output term. To detect these situations we attempt to refine the stored sort signature of the called rule for the given input term. It is possible that given the extra information about the current term we can say more about how the rule behaves.

In general, given a rule application to a term, we first convert the term to a sort equivalent, we retrieve the rule signature and we attempt to find a path from the input sort to one or more output sorts. If at least one such path exists, we know that the rule can succeed.

Two strategies are at the very core of Stratego: identity (id) and failure (fail). In the case of the former, its signature is clear - it just preserves the input sort:

```
id : a -> a
```

For the failure strategy, things are not as straight forward. A call to it makes the clause in whose scope it is called to fail. For our purposes this type of failure is very different than our failure: it is a functional failure rather than sort failure. At this stage we just take its signature to be:

```
fail: a -> ATerm
```

so that it doesnt influence our sort checking. We discuss an alternative to this in the section dedicated to future work.

Sort of a build Building a term generally makes reference to variables and rule calls; a typical example we have seen before:

```
Number( (x,y) )
```

We infer the sort variant of built terms in a similar way to matched terms, with the exception of handling rule calls. One important point to note is that the signatures are available at the first pass through the AST, because we first process the constructor declarations and only then the rule declarations. This is not true for rules. At the first pass we have no guarantee that all declarations of a rule have been processed, and thus cannot pronounce on the resulting sorts of a call. For this we need to defer checking the rules until the checking stage of the analysis. Implementation details aside, to determine the sort of a built term we first substitute the variables in place and replace rule calls with the resulting term. Where we have more than one variant we split them up into corresponding terms. From the collection of possible terms we only keep the ones that are valid sorts, thus corresponding to valid AST fragments. If at least one valid sort exists we proclaim that the rule produces valid AST fragments.

5.5 Variable configurations as parallel universes

Weve seen earlier that signatures can be parametric polymorphic:

```
List: a -> List(a)
```

We can clearly see in this case that the variable a is completely unconstrained, namely its sort is ATerm. In other situations however, sort variables can (and should) be constrained:

```
Prog : Number(a) * Number(a) * FunExpr(b,a,a) -> Expr
AFun : String * Int * Int -> FunExpr
AFun : Int * String * String -> FunExpr
Number : Int -> Number
Number : Double -> Number
```

What should the type of the sort variables a and b be? If we say that a can be anything (ATerm) implies that we are referring to both declarations for Number and both declarations for FunExpr. Then should we say that the variable b is also ATerm? We could, but we can also do more. We see that the variable a can be only Int or Double and that the variable b can be either String or Int. There is however an error in this approach. The variable a cannot actually be a Double, and the variable b cannot actually be an Int. This is because the sort

FunExpr takes either Int or String, not Double. Since we can only give it an Int without violating its signature, b can only be Int.

This was an example where the combination of sort variables and sort usages allowed us to constrain insofar as the declarations became isomorphic although their declarations seemed to indicate polymorphism. The advantage of these constraints on variables becomes apparent. Let us modify our example such that it becomes more complex:

```
Prog : Number(a) * Number(a) * FunExpr(b,a,a) -> Expr
AFun : String * Int * Int -> FunExpr
AFun : Int * String * String -> FunExpr
AFun : Number(x) * x * x -> FunExpr
Number : Int -> Number
Number : Double -> Number
```

Granting that these examples do not represent real signatures for real ASTs, we can indulge in more exploration. Weve added a new declaration for the FunExpr sort, which takes a Number in the first position. The variable x in the declaration can be either Int or Double. How do we describe the constraints of variables a and b? Well, if a is Int, then b can be either String or Number(Int), while if a is Double b can only be Number(Double).

This is correct, but as these constraints become more complex, we will easily lose track of how they influence each other. Time to introduce some notation and some rules over the meaning of the notation.

Variable constraint arithmetic It is important to realize that earlier, when we used the word **or** we were referring to disjunctions, and when we were saying **then** or **implies** we were making conjunctions on the constrained variables. We formalize these constraints into a very simple arithmetic of sorts.

Using our simple example from the previous sorts, we write the initial variable constraints as disjunctions and conjunctions:

```
{ [a = Int] + [a = Double] } *
{ {[b = String] + [a = Int]} || {[b = Int] * [a = String]} }
```

Where + represents disjunction and $*$ represents conjunctions. We can read this from right to left easily: a can be either Int or Double, and either b is String and a is Int or b is Int and a is Int. As we can see this notation allows us to maintain integrity of all constraints. In the right side we can see:

```
[b = String] * [a = Int]
```

which leads to our first rule regarding conjunctions. If the two operands of a conjunction do not constrain the same variables, they represent a valid configuration:

```
[b = String] * [a = Int] = [b = String, a = Int]
```

The same can be said about disjunctions that do not constrain the same variables:

```
[b = String] + [a = Int] = [b = String, a = Int]
```

Using this new rule, we can rewrite the initial formula:

```
{ [a = Int] + [a = Double] } *
{ [b = String, a = Int] || [b = Int, a = String] }
```

If the operands of a disjunction do overlap we say that the disjunction represents a valid configuration if and only if the variables in the intersection are of compatible sorts. A simple example:

```
[x = Int, y = String] * [x = Int, z = List(Int)]
=
[x = Int, y = String, z = List(Int)]
```

If however the variables in the intersection are not compatible, the conjunction represents a failed configuration:

```
[a = Int, b = String] * [a = String, b = String] = FAIL
```

With this in mind we declare some arithmetic rules. Unsurprisingly we expect conjunctions to have precedence over disjunctions. This makes sense in logical formulas and it also makes sense in sort constraint formulas. Additionally we give the following unsurprising rules:

- a * b = b * a
- a + b = b + a
- a + a = a
- a * a = a
- a * FAIL = FAIL
- a + FAIL = a
- a * (b + c) = a * b + a * c
- (a + b) * (c + d) = a * (c + d) + b * (c + d)

From these rules it should be evident that we always calculate conjunctions but we cannot always calculate disjunction. Disjunctions can only be calculated if the intersection of variables is empty (as stated before) or if the all the variables in the intersection are constrained to the same sorts in the two operands.

Going back to our example:

```
{ [a = Int] + [a = Double] } *
{ [b = String, a = Int] || [b = Int, a = String] }
```

we rewrite it to obtain:

```
[a = Int] * [b = String, a = Int] +
[a = Int] * [b = Int, a = String] +
[a = Double] * [b = String, a = Int] +
[a = Double] + [b = Int, a = String]
```

which then rewrites to:

```
[b = String, a = Int] + FAIL + FAIL + FAIL = [b = String, a = Int]
```

The same can be applied to the slightly more complex example from the previous section:

```
Prog : Number(a) * Number(a) * FunExpr(b,a,a) -> Expr
AFun : String * Int * Int -> FunExpr
AFun : Int * String * String -> FunExpr
AFun : Number(x) * x * x -> FunExpr
Number : Int -> Number
Number : Double -> Number
```

Where we start with the following expression:

```
{[a = Int] + [a = Double]} *
{ [b = String, a = Int] + [b = Int, a = String] +
[b = Number(Int), a = Int] + [b = Number(Double), a = Double]}
```

and by multiplying the left side into the right side, reduce it to:

```
[a = Int, b = String] + FAIL +
[ a = Int, b = Number(Int)] + FAIL + FAIL + FAIL +
[a = Double, b = Number(Double)]
=
[a = Int, b = String] +
[ a = Int, b = Number(Int)] +
[a = Double, b = Number(Double)]
```

Which we can no longer calculate further, because they represent parallel variable configurations.

Using the method described in this section, expressing and calculating variable constraints is reduced to a realtively brainless sequence of operations, while still permitting us to express disjunctive possibilities. The examples in this section were limited to constructor declarations, but the same mechanism is used for keeping track of variables in rules.

5.6 Gradual and forward analysis

The general approach to analysis in this project is to require no input from the user. It is an opt-in system where providing extra (sort) information is optional

but yields better error detection. The reader should interpret better error detection as more error detection, since the mentality is to only report an error if absolutely sure. One of the side effects of not having a lot of explicit sort information is that the analysis has to rely heavily on inference. The relationship between inference and available information is a push-pull one: the more information you have, the more inference you can do; the more inference you can do, the less information you need. Notwithstanding this contretemps, the rule of thumb is to report an error when everything inferred contradicts the users explicit statements.

Inference requires information; to make the best of the available information we should refine inferred knowledge with every bit of relevant information we find. We can also refine inferred knowledge with other inferred knowledge, this after all is only natural. What happens if we infer knowledge from something incorrect? Refine knowledge with a bit of incorrect knowledge, and you get nothing but refined incorrect knowledge. This issue combined with the fact that programming languages - Stratego included - have scopes, yields the problem of scoping inferred knowledge. This should allow us to limit the spread of incorrect knowledge.

The point here is that we can only infer sorts gradually, but this risks polluting the information with errors from the user. In case an error is found this would propagate in all directions throughout the code. To protect the user from this, we allow inferred knowledge to only propagate forwards through the analysis. An example is warranted:

```
Sum: Number(a) * Number(a) * X(a,a) -> Expr
Number: Int -> Number
Number: Double -> Number
X: Int * String -> X
```

There is an error here, if we look at the inferred constraints:

```
{ [a = Int] + [a = Int] } *
{ [a = Int] + [a = Int] } * { [a = Int] * [a = String]
```

We can clearly see that the variable a cannot be both Int and String at the same time, resulting in the entire constraint expression failing. This is a clear-cut case where all the inferred knowledge contradicts the user statement. Where do we report the error? At the point where it occurs. Report no errors for the first and second argument, but highlight the third as a point of failure. Assuming that the constructor had a fourth argument:

```
Sum: Number(a) * Number(a) * X(a,a) * Number(a) -> Expr
```

we would report an error on the fourth argument as well, because the type of a (FAIL) is not compatible with any of the arguments of the Number sort. Assuming that there is also a rule that uses this term:

```
dup: (a,b) -> Sum(Number(a),Number(b),X(a,b),Number(a))
```

should we also report the built term as being invalid? If we were to be strict we ought to answer yes. However this would be based on an earlier detected/reported error - there is no error we can be sure about here. If we encounter a failed configuration we replace it with the most relaxed constraint - [a = ATerm] in this case - and attempt to continue our analysis. The worst case scenario is that we can no longer detect some errors (and indeed there is an additional error in the rule), but this will be detected when/if the user corrects the initial problem. We can guarantee that under no circumstance would we be reporting false positives because of this relaxation.

5.7 Reporting errors/warnings

The previous sections of this chapter have looked at how sort information can be gathered from actual Stratego code. Signatures are translated to internal representations, which in conjunction with variable constraints are used to detect erroneous AST fragments. We now briefly look at how we actually detect problems.

The simplest type of error is caused by referencing a sort that is not declared, when we just report an error on the sort reference. If all the sorts used are declared, we can move to detecting sort usages that do not match their declaration. This can be detected as a sort incompatibility between the declared arguments and the use arguments, which can also manifest as a failed variable constraint expression. Another type of error is caused by overlapping signature declarations:

```
A: a * a -> X
A: Int * Int -> Y
```

The sort arguments of the two constructors overlap in types; for an use such as A(3,4) it is unclear what sort (X or Y) should be associated with this term. Some of the signatures automatically generated from SDF-specified grammars exhibit overlaps, it is unclear whether this is desirable or not. Lacking decision we prefer to report these situations to the user as warnings.

Reporting errors for rules is not more complicated. For rules there are three sets of constraints: constraints from the strategy and term arguments; constraints from the input variants; constraints from the output variants. For every rule that has no valid sort variant for the matched term we report that the rule may only be matching invalid terms. If mediation between the constraints from the term arguments and the matched term is not possible we report that the rule is likely to always fail. If none of the build term sorts is plausible we report that the rule always violates term signatures for the AST. If mediation between match-inferred constraints and the built term constraints fails, but the built does not fail by itself, again we report that the rule violates term signatures. Rule calls that are performed when building terms are checked before the actual term building as this allows us to refine the information regarding the built term. Rule calls that do not have a matching input term are reported as always failing calls,

whereas rule calls that produce invalid terms or cause invalid terms to be built are reported as a violation of signatures for the built term.

Detecting errors once information has been gathered is much simpler than actually gathering the required information.

6 Implementation

The following paragraphs report on some of the interesting approaches and issues encountered during implementation. Due to time limitations and the increased difficulty of analyzing Stratego code generically, support for inferring sorts of rule calls could not be implemented, although, as outlined in the previous section, a mechanism for this has been thought of.

Some difficulty was encountered when implementing inference for variables used in sort declarations. These were caused by the fact that the declarations are unlikely to be reverse order of their dependencies. In other words, it is likely that we begin inferring variables from a declaration which uses sorts for which we have not yet inferred variables. One solution to this is to maintain a queue of declarations to be processed. If the declaration at the head of the queue does not have all its dependencies satisfied (already processed) we can skip and continue with the rest of the queue. We redo this until the queue is empty. A basic algorithm for this would be:

```
queue = all declarations
previousqueue = Nil
while queue not empty
        for every declaration D
                if dependencies of D satisfied then
                        process D
                        remove D from queue
                end
        end
end
```

This may work if dependencies are linear, but this not always the case. In that case the algorithm above can deadlock. Our goal is to be able to infer variables. One trick that we applied to break deadlocks and still infer as much as possible was to first detect a deadlock. When such a deadlock is detected the head of the queue is removed and placed in a secondary queue. The most relaxed constraints are stored temporarily for that declaration (all the variables are constrained to ATerm), and the remainder of the queue is processed. If this has not broken the deadlock, the removal of a new head is performed, etc. At a certain stage, all the dependencies of at least one declaration will have been removed from the queue and have received temporary constraints. That declaration can now be processed. Once the primary queue is empty we replace it with the secondary queue if not empty and repeat the process. This last step ensures that we infer real constraints for the deferred declarations. The process continues until both primary and secondary queues are empty:

```
queue = all declarations
def-queue = Nil
previousqueue = Nil
while queue not empty
```

```
        for every declaration D
                if dependencies of D satisfied then
                        process D
                        remove D from queue
                end
        end
        if previousqueue == queue then
                def-queue.add(queue.first)
        else
                previousqueue = queue
        end
        if queue == Nil then
                queue = def-queue
                def-queue = Nil
        end
end
```

Using this algorithm we resolve circular dependencies and gather as much information as possible. A separate circular dependency issue appears with singular recursive sort declarations:

```
Sum: Expr * Expr -> Expr
```

If the above is the only declaration for the Expr sort, the previous algorithm fails in breaking the deadlock. A solution to this was not implemented because it was found to not occur often in practice. A workaround is relatively easily implemented however using a similar trick to the original deadlock problem. When we encounter a recursive declaration such as the one above, we can temporarily store relaxed constraints for it and mark all of its dependencies as satisfied. We can then proceed to infer variable constraints and replace the temporarily ones with the new constraints.

With the exception of the above issue, implementation did not pose a particularly difficult challenge - most of the topics were thought of in advance. That said, there are of course quite a few catches, which are discussed in the last chapter of this report.

7 Evaluation

The goals of this project mentioned at the beginning of this report were:

1. Give semantics to signature language
2. Detect violations of term signatures
3. Detect rules that never succeed
4. Identification of type preserving rules

In the period between April 15, 2011 and June 15, 2011, a period of ten weeks, some analysis, much thinking and some implementation has been performed. Goal 1 has been fully met. In addition to the first goal a small set of analyses for the signature declarations has been provived, which should help programmers detect potentially faulty signatures caused by missing or incompatible sorts. Additionally the programmer is able to see the plausible sorts for variables in constructor declarations and rewrite rules.

We believe that the time spent in developing a method for expressing and operating on variable configurations is beneficial to future work although it should first be reviewed.

Goals 2 and 3 were partially met. The set of analyses can detect violation of term signatures and rules that never succeed on valid ASTs. As mentioned previously however, rule calls are not supported, leaving the majority of the rules unchecked. However the method proposed appears promising and it is not expected to pose a major difficulty in implementation.

Goal 4 was not met. Currently it would be trivial to detect type preserving rules as long as they do not make any rule calls. Once support for rule calls is added, identification of type preserving and type unifying rules is not expected to pose a challenge.

When the current analysis was run on the MiniJava compiler developed for the Compiler Construction course a number of warnings and errors were found. Seven warnings regarding signature declarations were found, all of which are valid. Nine errors regarding signature violations were found. Of these, six were valid. From the three invalid warnings, two were caused by yet unsupported syntactic constructs, and one was caused by a current limitation in supporting sort injections.

Additionally we have provided a rudimentary mechanism for the programmer to disable the analysis on a per-project basis.

Given the current state of the implementation, the produced code is not yet ready for re-integration into the mainstream Stratego editor for Spoofax. The main reasons for this and some recommended directions for future work are discussed in the next section.

8 Future directions

This project served as an initial attempt at providing static analyses for Stratego code. While it has achieved most of its goals, a long road lies ahead before it can fulfill its role as a helpful tool for Stratego programmers.

The first recommendations for continuing development is a review of the methodology and simplification of the code. The produced code is very complex at places. A different approach to these areas may provide simpler materializations. Areas of increased complexity are also fracture points; points with a high risk of errors. These should be reviewed before more features are layered on the existing implementation.

8.1 Better messages

Current error messages are not very informative. There are two contributing factors:

1. The location of errors messages for rules is rule-top
2. The messages are non-informative

Error messages pertaining to rewrite rule are currently reported at the top of the rule declarations. It is better if errors are marked where they occur such as a matched term or a built term.

Current messages are also not very informative. In the case of failures caused by failed variable configurations we cannot report any useful message to the user. We can illustrate this with an example:

```
Number : Int -> Number
Char : String -> Char
Plus : Number(a) * Char(a) -> Plus
```

We infer the constraints for variable a to be:

```
[a = Int] * [a = String] = FAIL
```

This problem manifests itself more when a disjunction is applied to a FAIL constraint:

```
a * FAIL = a
```

Evaluating such a disjunction removes the FAIL constraint from the expression, thus throwing away any trace its cause. An alternative to this would be to piggy back the reason for failure on the FAIL constraint:

```
[a = Int] * [a = String] = FAIL([a = Int * String])
```

In the case where the evaluation of a constraint expression fails we can retrieve the last cause for failure from the piggybacked cause. In the case the evaluation succeeds we can drop the FAIL altogether and keep the valid result.

8.2 Reduce explosions

The current implementation uses generic term explosions and implosions to deconstruct and respectively construct user declared sorts as constructors:

```
?"Tuple"#(x)
```

While these explosions are very generic and easy to program with, their flexibility comes with a significant performance penalty. Beyond the performance issues, the current implementation suffers from name collisions between internal constructors and user sorts. We illustrate this with an example. We represent variables internally as:

```
Var: String -> Var
```

yielding terms such as Var("a"). If the analyzed program also declares a constructor with the same name:

```
Var: ID -> Var
```

we can no longer distinguish between the internal Var and the user declared sort Var.

Resolution for both of these issues is the replacement of internal representations for sorts to a more generic approach:

```
SortReference: ID * List(Sort) -> Sort
```

This of course requires some rewriting of the analysis code, but it is expected to significantly simplify in a few areas.

8.3 Sort signatures are sort functions

Sort signatures should also be seen as normative structures, not only as descriptive structures. Their role is to enforce the structure of a valid abstract syntax tree. Given the following constructor declaration and rule definition we cannot currently correctly infer the type of the variable a:

```
A: x * x -> A
answer: A(a,42) -> a
```

Clearly its sort can only be Int, however we do not obtain that information. The reason is that we do not apply gradual inference refinement within the scope of one match. The approach should be to see the declaration of A as a function that can be partially applied and its result being a set of constraints. In this example we could temporarily constrain the variable x to the sort Int, thereby immediately inferring the constraint for a to sort Int as well.

In simple words, we currently infer variables in rules as if constraints for them would press down on them. Before we press down the constraints we should first push up against the signatures with literal information that we are given in the respective match or build.

8.4 Sort parameterization breaks abstraction

The advantages of allowing parameterization of sorts are obvious. It allows the programmer to refer to specific subgroups of sort declarations, and to use the sort information contained in those declarations locally. At the moment however, when wishing to restrict the polymorphic parameter of a sort one has to implicitly limit the arity of that sort as well. In the following example:

```
A: String * a * a -> A
A: String * a -> A
```

if a user wanted to declare a constructor C of one argument of sort A but limit its variable *a* to only Int he would have to write:

```
C: A(String,Int) -> C
C: A(String,Int,Int) -> C
```

Clearly this is code duplication due to lack of flexibility. If we allow sorts to be unparameterized we should also allow sorts to be polymorphic restricted while still being unparameterized.

This likely requires the introduction of some new syntax (in what form that may be):

```
C: A<Int> -> C
```

This should restore abstraction and flexibility, offer the same effect and still allow the user to parameterize sorts if so desired.

8.5 The problem of sort injections

The current approach to sort injections in testing sort compatibility is to attempt a direct match and if that fails to perform a depth-first search of sort injections in search for a compatibility point between the given sorts. Besides the performance disadvantage, the depth-first search is likely to find a more distant compatibility point then optimal. It is wiser to look for a shortest compatibility path. This would insure that the inferred sorts are "closer" to the real meaning of the syntax specification.

A second point is that sometimes the compatibility point between two sorts can only be reached by lifting both of them through sort injections and after each lift searching for a path (as opposed to the current approach of lifting either the first or the second sort). A search in this fashion is non-trivial in neither space nor time. Even stronger is the argument that if no other sort injections determine a compatibility point, it is likely that sorts are equivalent through very top level injections where everything is equivalent to the ID sort. This would expose the risk of never being able to fail a sort compatibility test.

A new way of handling sort injections is required, and it may well include the temporary removal of support for them. Further investigation is necessary in this area.

8.6 Unified algebras

The current approach to inferring the sort of a list is to try to find lowest common sort of the elements given. This is a greedy approach which works well for some languages (Scala), but it may not be sound for Stratego. Inferring the type of the list:

```
[Number(_),_,_|_]
```

to List(Number) may not be what the user had in mind. So far we have not encountered any false-positive errors because of this, but it is a possibility. A more refined approach has to be evaluated.

While still in the context of lists we can see that inferring the sort of the above to be List(Number) actually discards variable information about the size of the list. We currently have no way of encoding this information in the sort, nor should we because that would break abstraction again. Unfortunately this is not limited to lists. When we say:

```
Number(42)
```

We are not referring to the sort Number(Int) but to Number(42), this would be a generalization and thus a loss in precision.

A different approach is necessary and should probably be in the direction of unified algebras [4]. Unified algebras specify mechanisms for treating literal values as refinements for types. Dialyzer [7] - the Erlang static analysis suite - makes use of these to some extent with signification success. An approach based on unified algebras would have to weigh complexity against benefits: Stratego programs have a very limited amount of constants.

8.7 Loss of link between input and output

The current inferred knowledge about rule sorts is composed of a list of matched sorts and a list of built sorts. Once these lists are built the notion that one possibility from the matched set may correspond to a specific possibility from the built term is lost. This does not cause any false positives but it can cause us to miss detection of some real errors. Maintaining such a link between matched sorts and built sorts is intuitive because Stratego rewrite rule effectively specify this link at a term level. It is also cheaper because we can avoid building and evaluating cartesian products of these lists. An approach for achieving this has not yet been researched but it is believed to increase the accuracy of the analysis.

8.8 Sort of *fail*

We have stated earlier that the type of *fail* strategy is:

```
fail: a -> ATerm
```

However this does not seem very sound. The fail strategy does not actually return any term because it produces failure, and ATerm expresses the presence of a returned term. It is likely that introduction of a sort indicating a lack of sort - Nil - may be required in the future.

8.9 Handling of external rules

While we have shown that it is possible to perform static analyses on locally declared rewrite rules, many of these use external libraries. We have researched no mechanism for providing sort information for the rules contained therein. The Stratego library is a large code base, a large subset of which is relied heavily on by programmers. It is however nearly void of any type information and its source code is also unavailable at analysis time. As a difficulty amplifying quality, with the exception of primitive handling strategies, most of the library is composed of generic strategies. Even if sort information would be available it would likely be as specific as:

```
try(s): ATerm -> ATerm
```

Even if its source code would be available the performance penalty incurred by analyzing external rules would be very large.

There are two future areas of research. The first is the requirement to provide as concise as possible sort information for the Stratego library. Due to the large code base this cannot be done manually. The second direction is a mechanism of storing module sort information that can be accessed by dependent modules at analysis time without requiring re-analysis of the imported modules. This is also the approach that Dialyzer [5,8]- the Erlang static analyzer - takes

8.10 Supporting generic traversals

A core feature of Stratego is its collection of generic AST traversal strategies, especially the ones which handle rule failure, such as: innermost, alltd and repeat. Providing sort fingerprints for these strategies is a non-trivial task. What should the sort of this call be?

```
<innermost(substitute)> Sum(Number(...),...)
```

Although significant research has been published [10,3,2] on this topic, this project has not taken any steps towards providing an answer. It is possible that actually specifying pertinent sort information for generic traversals is impossible, and instead the most that is achievable is detecting whether the traversal preserves or alters the shape of the AST (fourth goal of this project).

Finding an answer to this in the near future is an absolute requirement for providing meaningful static analyses for Stratego programs.

8.11 Fragile rules

While examining sort information for rewrite rules we only report errors if there is no possibility of the match or the build to succeed on or produce valid terms. This guarantees that we never report false-positives. Could we increase our detection rate without disturbing the programmer too much? We could perhaps also give some information regarding the fragility of the rule. A question worth

asking is whether the ratio of compatibility to incompatibility gives an indication of the robustness of the rule. For rules under a set (by the user) threshold ratio we could warn that the rule is fragile.

9 Conclusion

This project has met part of its goals. A first set of static sort analysis for Stratego was proposed and implemented. The set of analyses can detect term signature vialotions and rules that never succeed on valid ASTs. Currently it would be trivial to detect type preserving rules as long as they do not make any rule calls. Once support for rule calls is added, identification of type preserving and type unifying rules is not expected to pose a challenge.

When the current analysis was run on a previously developed compiler for a small object oriented language, seven warnings regarding signature declarations were found, all of which were valid. Nine errors regarding signature violations were found. Of these, six were valid. From the three invalid warnings, two were caused by yet unsupported syntactic constructs, and one was caused by a current limitation in supporting sort injections.

Given the current state of the implementation, the produced code is not yet ready for re-integration into the mainstream Stratego editor for Spoofax. Many directions for future research have been discussed in the previous sections.

The way of working was adequate. With hindsight, if more time had been spent consulting with the project supervisor it is likely that some of the incorrect decisions discussed earlier could have avoided.

For the author this project has served to draw a good picture of the complexity of the problem at hand and to reiterate that complex problems are better tackled in teams.

References

1. Merijn de Jonge, Eelco Visser, and Joost Visser. XT: a bundle of program transformation tools. *Electronic Notes in Theoretical Computer Science*, 44(2), 2001.
2. Ralf Lmmel. Typed generic traversal with term rewriting strategies. *CoRR*, cs.PL/0205018, 2002. informal publication.
3. Ralf Lmmel and Joost Visser. Typed combinators for generic traversal. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 2002.
4. Peter D. Mosses. Unified algebras and abstract syntax. In Hartmut Ehrig, editor, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types Joint with the 4th COMPASS Workshop, Caldes de Malavella, Spain, October 26-30, 1992, Selected Papers*, volume 785 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 1992.
5. Sven-Olof Nystrm. A soft-typing system for erlang. In Bjarne Dcker and Thomas Arts, editors, *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, pages 56–71. ACM, 2003.
6. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala. A comprehensive step-by-step guide*. Artima, November 2008.
7. K. Sagonas. Experience from developing the dialyzer: A static analysis tool detecting defects in erlang applications. In *Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools*. Citeseer, 2005.
8. Konstantinos F. Sagonas and Daniel Luna. Gradual typing of erlang programs: a wrangler experience. In Soon Tee Teoh and Zoltn Horvth, editors, *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, Victoria, BC, Canada, September 27, 2008*, pages 73–82. ACM, 2008.
9. Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.
10. Joost Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 2003.

A Research report

Reduce, Reuse, Recycle, Type-check

A survey of typed generic programming approaches

Vlad Vergu

No Institute Given

1 Introduction

Generic programs are programs that provide specific functionality applicable on generic inputs. While their functionality is specific, their implementation is free from any application-context datatypes jail. The need for generic programming becomes evident when dealing with large forests of datatypes where some, many or all may not be known until a later stage. Use of generic programs facilitates code reuse and allows programmers to focus on the actual to-be-implemented logic rather than spending time implementing boilerplate code - for example for data structure traversals. The economic need for generic programming is clear: the invention of type safe generic programming would drastically reduce software development costs and increase maintainability of software systems.

One of currently still open issues with generic programming is the unfortunate trade-off between flexible genericity and type safeness. Type systems are needed to catch typing errors before they cause a crash and to catch logical mistakes by checking for internal consistency [54], however they require a certain level of specificity.

This survey presents some of the notable approaches to typed generic programming. Due to the abundance of literature using Haskell in the context of generic programming, most of the systems that are presented here are either Haskell compliant or are lightweight extensions on it. Although not necessary for understanding this text, a beautiful introduction to programming with Haskell is given in [24, Hudak et al, A gentle introduction to Haskell]. Although it leaves many areas unexplored, this survey limits itself to approaches that make heavy use of the Haskell type inference system.

This text begins with a short introduction to generic programming (Section 2) where it also exemplifies the need for static polymorphic type checking by use of a simple example. Section 3 provides a brief introduction to inference type checking systems with a focus on the Hindley-Milner system and its extensions into the Haskell type checking system. Sections 4 through 6 give overviews of the notable typed generic programming approaches, with sections 7 and 8 summarizing and pointing the reader to areas this survey left unexplored.

2 Generic programming

Many engineering fields have benefited for decades now from (design) reuse paradigms. This has lead to overall lower cost of development and shorter time to

market. For over four decades, the field of software engineering and its adjacent fields have been troubled by the inherent difficulty of writing software that can easily be reused. A number of approaches have been attempted with the most successful in recent years, but the level of success has been much restricted in comparison with other engineering disciplines.

Most successful past attempts at increasing software reuse have been in the area of software libraries through the use of APIs, such as Linux's X window system. While these libraries allow programmers to reuse code with variable degrees of complexity most of them perform actions on fully defined data types. Applying them to new types would require modifying the library therefore defeating the purpose of the library as a reusable item.

Generic programming [50] is an approach which offers the chance to achieve a higher level of reuse than the just mentioned libraries. The core principle behind generic programming is the decomposition of software into: data types, data structures, and algorithms [10]. These decompositions would be programmed individually as components without making (or rather minimizing) assumptions about each other. The goal being that elements of software decomposition become fully interchangeable and type abstract. In an ideal of genericity, programming (design) patterns could be captured as standalone entities and reused as parameterizable abstractions [3]. As an example of the need for type independent programming take the C function *sqrt* available in the standard libraries. Although it can be applied to most numeric types it is impossible to apply it to a newly defined numeric type such as a quad-precision floating point without redefinition. For a gentle introduction to generic programming - with many algorithmic examples - see [3].

A popular example used in literature [5,26,25] is that of determining the number of elements inside a data structure - the *length* function. This would be straight forward for both objects of type List and of type Tree. However providing an single definition that would work for both (or for that matter *all*) types requires the decomposition of the software into a traversal strategy for the structure and an element counting strategy. The function length would then become parametric polymorphic [8,5] over the structure type while the traversal functions would be ad-hoc polymorphic [58,37]. It would indeed become trivial to then use this function to determine the number of terms in an abstract syntax tree for example. The need for higher-order polymorphism with overloading will be touched upon in more detail in the paragraph regarding type systems. As mentioned earlier this is less a survey about possibilities of generic programming in general and more on the difficulties and potential solutions to performing static type checking on these. Generic programming practices are dependent on the language capabilities and in particular to whether the language has facilities for defining generic functions and data types [12].

Applications of generic programming

Probably the most illustrative example for the need of generic programming is the case for *software evolution*. The idea that any large software system

evolves through change is part of it's nature. Throughout it's evolution in the abstraction-customization cycle [3] the system will undergo numerous changes. When changes are performed to algorithms, programmers have to ensure that the new implementation conforms to the requirements. Without generic programming it is likely that multiple *clones* of the same algorithm will exist making management of changes much more difficult. The problem becomes larger if instead of algorithms some data types change. The programmer then has to ensure that all algorithms applied to that data type are maintained functional while the number of clones is likely to increase. This increase is natural the algorithms are applied to more types than the changed ones because functionality would have to preserved for those as well. If however the system would make use of generic programming the generic algorithms would naturally adapt to the changed data types and provide the same uniform and identical functionality as before the change [20]. This for one could prevent a number of the clones from occurring in the first place and make software maintenance a less complex and dangerous task. Overall the reuse of the algorithm implementation would increase and the overall maintainability of code would increase.

Stratego

A case for generic programming that has inspired this survey is that of language independent program transformation. More clearly providing language and tool based facilities for program transformation. Under program transformation we understand software refactoring, re- and reverse engineering. One example of such a suite of language and tools is Stratego [55] and the XT Transformation tools [9].

Stratego is a functional programming language that provides facilities for - among others - expression of traversal and transformational rules. Together with the XT toolset - a bundle of reusable transformation tools - the Stratego/XT pair supports development of program transformation programs, domain-specific languages and compilers.

> *Stratego/XT is a generic infrastructure for creating stand-alone transformation systems.*

For a short introduction to Stratego/XT see [4, E. Visser et al (2008)]; for a more in depth presentation see [56, E. Visser (2003)].

Need for polymorphic type-checking

As mentioned earlier, this survey was inspired by the Stratego/XT transformation tools. At their core is the Stratego functional language, which through its flexibility allows the programmer to easily write code that traverses and transforms abstract syntax trees. Without delving into the reasons we can state that at the moment Stratego is not a typed language. While this limitation allows uncompromising flexibility it exposes the program and programmer to a higher

level of risk. As an example take the following definition the strategy to determine whether the two elements of a tuple are equal:

```
/**
  * Tests whether two terms are equal.
  */
eq =
  ?(x, x)
```

In this definition, the strategy *eq* will bind the first element of the tuple to variable x and attempt to match that the second element to that. If the matching succeeds then the application of the strategy will succeed, otherwise it will fail. Take the following applications of eq:

```
<eq> (3,3)  // succeed
<eq> (3,4)  // fail
<eq> (3,"Hello") // fail
<eq> (3,3,3) // fail
```

All three applications produce code that compiles without problems, however only the first application would succeed at runtime. The second would fail due to inequality, the third due to incomparable types, the fourth due to inapplicable number of arguments. In Stratego indeed there would be no difference in the type of failure produced by the last three applications. In the presence of static type checking the last two applications would produce an error at compile time rather than at runtime. Although the Stratego compiler statically checks constructor declarations and variable binding and referencing, Stratego is not a typed language so type checking is not performed [4]. Stratego lacks a type system because no type system has been found that would be strong/safe enough without sacrificing too much flexibility. For a more detailed explanation of this reasoning see [54]. Whether it is desirable or not to have checking that is as strict as suggested or a weaker form is beyond the purpose of this text, but the simplicity of the example illustrates the essence of static type checking.

Type systems protect the program and programmer from many type related errors and allow the program to be more defensive through its definition. While they shield the programmer from common errors, type systems limit the flexibility of a language and most importantly reduce the reusability of the programs written in that language [15,5]. This is one of the main reasons for which the search for strong typing for generic programs is still ongoing. Contrary to Cardelli [5] polymorphic type checking is necessary but not sufficient for type checking generic programs [15].

3 Type systems

As outlined in the previous section, type systems are the first layer of defense against type violations, illegal instructions and memory violations. This section

attempts to present some of the type systems that have had a significant influence to what is now the Haskell GHC type system. We will also touch upon unrelated but potentially interesting type systems.

Dynamic typing

Languages like Python, PHP and Lisp are dynamically typed meaning that no typing is enforced on programs at compile time. Operations on data of different types is allowed to pass successfully during compile time regardless of types but may fail when the machine attempts to execute it. This may lead to more code flexibility and lighter compilers but the lack of type checking may increase the prevalence of crash causing bugs that could have been caught by a compiler with a type check. Agreeing with [15] that no type system (or in this case dynamic typing) allows for great flexibility, one would suggest that it is easier to provide generic programming facilities in a dynamically typed languages by foregoing on all of the type safety requirements. Sine generic programming is however defined as programming that is data-type independent it is primarily about programming with types. It would therefore be difficult to simulate generic typea since the concept of types and the inherent checks and guidance are missing [20].

Somewhere in between no typing and strong static typing lies a dynamic type system which has received very little coverage in terms of generic programming: *duck typing*. It is based on the *duck test*:

> *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*

Or in terms of variables "if it walks like a duck, and quacks like a duck, then it is a duck". In duck typing type information is determined at runtime based on partial structure information about used functions and substructures. As an example, the Python programming language makes heavy use of duck typing to determine types of variables, the code `2 * "Hello"` would produce "HelloHello" but would fail at runtime for `2 + "h"`. Duck typing allows the use of functions on unrelated data types as long as the required functions are defined on them.

Static typing

In contrast to dynamically typed languages, the compilers of statically typed languages are able to catch program errors due to function applications on incorrect types before they get the chance to cause a crash. The compiler of such languages examines the code and based on the types of variables and functions detects and reports errors. Differently than in the earlier Python example, the Java code `2 * "Hello"` would produce a compile time error indicating that the multiplication operator cannot be applied to strings. Both Java and C++ present facilities for some generic programming in the form of Java generics and C++ templates respectively but their flexibility and genericity is limited [10].

Explicit vs. implicit

Both C and Java are explicitly typed, meaning that the programmer must explicitly provide typed declarations for variables and function arguments. While this provides some level of code self-documentation and forces the programmer to adhere to this documentation it dramatically reduces the flexibility of the language required for very generic programming. Languages like Haskell and ML do not require the use of such explicit type declarations. Both use a *type inference system* that is capable of deducing the types of expressions in most of the cases. Both ML and Haskell have type systems primarily based on the Hindley-Milner [49] type inference algorithm. The core principles behind the Hindley-Milner algorithm will be presented in the next paragraphs together with some of the more relevant modifications some of which are currently implemented in Haskell.

Hindley-Milner system

The Hindley-Milner [49] type inference system, also knows as the Damas-Milner system [8], forms the basis of the most referential type inference system. The principal idea behind it is that explicit type annotations can be omitted from code in most cases and at the same time they can be inferred by the compiler. The algorithm deduces constraints for the types of functions and arguments and variables resulting in a system of constraints. After the constraints have been generated (1), some reductions (mainly by substitution [8]) are performed (2), and then a unifying solution in the form of type reconciliation for the constraints is searched for.

As an example consider the following definition for the *length* function:

```
length [] = 0
length (x:xs) = 1 + length xs
```

The Hindley-Milner algorithm would successfully deduce that the type signature of length is `length :: [a] -> Int`. A significant feature of the the algorithm is its ability to handle *polymorphism*. In the particular case of the length function the set of constraints would not constraint the type of elements in the list to anything meaning that length is not dependent on the type of the contents of the list, only on the fact that the argument is in fact a list. This type of polymorphism is known as *parametric polymorphism* [29].

One limitation of the Hindley-Milner type inference system is the inability to handle ad-hoc polymorphism or overloading. The original algorithm did not provide any way to declare functions that would have different functionality for different types. The choice of type for a particular function parameter was either a monotype or an unconstrained type. This, for example, does not allow for different definitions of a function with the same name for the equality function on different types. One would require equalNum and equalString for example for numbers and strings.(This is indeed the approach that Stratego uses, with its *add* and *addS* strategies for example). The first attempts at achieving typed

overloading were made by [36, Kaes] and [58, Blott et al] with the former being later proven to be undecidable [37]. The current Haskell implementation provides overloading based on the idea of *type classes* as originally described in [30], where type classes can be declared parametric polymorphic and then instantiated for different types:

```
class Eq a where
   (==) :: a -> a -> Bool

instance Eq Int where
   (==) = primEqInt
```

Hence providing the ability to overload function definitions. This in turn allows parametric function definitions on those types of the form:

```
elem :: Eq a => a -> [a] -> Bool
```

It is important to note that in the case of Haskell explicit type signatures are still allowed and sometimes required to resolve type ambiguities. The explicit type signatures are used to generate type constraints during the constraints generation phase [33]. Although rare, overloading ambiguities are caused when multiple instantiations are possible for the same variable, and these can only be resolved by explicit type annotations by the user [24].

Extensions of the type inference system to support arbitrary rank functions (rank-n polymorphism) - widely accepted as a requirement for many generic programming approaches - which are now available as language extensions for Haskell 98 and fully supported as part of the Glasgow Haskell compiler are presented in [34] and discussed in much of Haskell documentation.

Further discussion on some of the limitations Hindley-Milner based type inference and an approach to first-class polymorphism is given in [32].

4 Polytypic programming

Polytypic programming [26] was the first attempt at bringing genericity to Haskell in the form of PolyP [25]. Polymorphic type systems like the Hindley-Milner system allowed the definition of parametric polymorphic functions - functions that have constant behavior over multiple types, but sometimes are not sufficiently flexible. For example the equality function that is not parametric polymorphic (because it needs to inspect the structure of its parameters to determine equality [13]) cannot be implemented once and used across data types. With the introduction of type classes [30] it became possible to use overloading to provide distinct type specific implementations of the equality function. This is not sufficient for generic programming however because the users still has to *manually* give type-specific implementations. What is desired is functionality similar to what the Haskell compiler does to functions in derivable classes: automatically generate specializations of those functions for instances of the derivable

types [25]. Polytypic programming as described in PolyP aims at providing automatic specialization of polytypic functions. Possible applications of polytypic programming are discussed in [26].

PolyP is an extension to the Haskell language, which makes use of its own compiler to compile to code that can then be compiled by the Haskell compiler. PolyP adds the *polytypic* construct that is used to define functions by induction over the structure of data types. As an example the PolyP implementation of the flatten function is given:

```
flatten :: Regular d => d a -> [a]
  flatten = cata fl

  polytypic fl :: f a [a] -> [a]
    case f of
      g+h -> either fl fl
      g*h -> \(x,y) -> fl x ++ fl y
      () -> \x -> []
      Par -> \x -> [x]
      Rec -> \x -> x
      d@g -> concat . flatten . pmap fl
      Con t -> \x -> []

  cata :: Regular d => (FunctorOf d a b -> b) -> d a -> b
```

The above example highlights some of the features of PolyP: it uses functors and bifunctors to specialize catamorphisms on the data types [20]; and provides a case by case definition of functionality for each type of four types of data type definitions: sum-, product-, parametric- and recursive types. For a full explanation of the thought behind PolyP see the original paper [25].

As the example type signature of *cata* suggests, polytypic functions in PolyP can only be specified on regular data types. This constitutes a serious limitations, Jeuring et al. [25] define regular types as: a datatype containing no function spaces and whose constructors have the same arguments on the left- and right-hand side of their definition. The second limitation implies that higher rank polymorphism is not possible with PolyP and that mutually recursive data structures cannot be handled [20]. As a secondary consequence polytypic functions are not first-class citizens of the language drastically hurting flexibility.

Foregoing on the limitations, PolyP is a type safe language, and the PolyP compiler checks the correctness of the polytypic function definitions. Polytypic definition must however be explicitly typed. At times ambiguous overloading must be resolved by type annotations. The specialized code that is produced by the PolyP compiler produces type correct Haskell code. The type system is based on M. Jones' classified types [29] and higher order polymorphism [28]. The unification algorithm used is an extension of Jones' kind-preserving unification algorithm [31]. The type system is also extended with a number of types, kinds and classes, most notably the constructors for functors and bifunctors and the Regular type class [25].

As a lightweight approach and improvement on PolyP, [51] improves on some of the original limitations and provides an optional compiler for generating necessary code for data types. Most notable improvements are different definitions of bifunctors and first-class polytypic functions [51].

As further reading, Hinze's approach to polytypic programming in Haskell is presented in [14], and his views polytypic values and polykinded types are presented in [13].

5 Generic Haskell

Probably the generic programming approach that promises the best balance of flexibility and type safety so far is Generic Haskell (GH)[17]. It is strongly based on the theory of type-indexed function with kind-indexed types developed by Hinze [13]. In layman's terms it permits generic functions to be indexed by type in a similar way type-indexed values are indexed in the Haskell derivable classes. This permits the user of GH to also define his own derivable classes. Based on Hinze's work for typed-indexed types has also been integrated into GH [18]. Particular advantages for a programmer are the availability of *constructor cases* and *default cases*. The former allows a programmer to define specific behavior of a generic function for specific type constructors [6]. This could be interpreted as a form of overloading, but note that that the specificity is given on still generic data types (hence the type-indexation). Default cases allow generic functions to implicitly based on other generic functions. This for example could be particularly useful when defining functions with different functionality but same data structure traversal method - stimulating reuse of generic functions.

Although powerful, simple and elegant, GH has some limitations [20]. In terms of applicability of GH to applications of program transformation (of the Stratego family) one limitation is that generic functions cannot be applied to generalized algebraic data types, of which a notable instance would be abstract syntax trees. Functions in GH are not first-class citizens, this is a consequence of the way the GH compiler performs specialization which eliminates type arguments from code.

With regards to type checking, Generic Haskell is not strongly safe. All generic functions have to be explicitly typed and the remained of type checking is performed by the Haskell compiler on the specialized code [20]. GH only guarantees that correctly typed programs will be correctly typed after specialization as well and that errors caused by missing generic functions will be caught at compile time.

As a further limitation, specifying complex systems in GH can become a very complex endeavor due to the intrinsic recursive definitions of generic functions [47] especially in the absence of a type checker. A type checker based on the principle of dependent typing [2] has been proposed [47] but is yet unimplemented. Further extensions that attempt simplifying usage have been proposed [7].

A generic programming extension to Clean [1] is very similar to the approach at GH in that it is also based on Hinze's work on type-indexed functions with

kind-indexed types. Clean overcomes the limitations of GH's type checking system making the language fully type safe. This however comes as a compromise in expressiveness, flexibility and power of the language [20].

Much literature is available describing the theories, implementation and applications of Generic Haskell, good starting points being [17,16,20,6,27] and a soft introduction to the principles of programming in GH being [23].

6 Generalized folds

This section presents two approaches that share the same view on values of types known as *generic folds*: Strafunski and Scrap Your Boilerplate (SYB) [42]. The generic folds theory takes a different view than approaches such as Generic Haskell and PolyP, namely that values of data types are either represented by a constructor or the application of a constructor to a value [23]. Strafunski [46] is an approach to software metrics and transformation making use of generic source code traversal functions. The SYB approach centers on the idea of providing generic functions that could replace boilerplate code needed for example for company-wide database traversals.

The reader should note that although the author of this survey could not verify the similarities between the two approaches it is his belief that being based on the same core concepts of generic traversals, SYB is a generalization of the source code traversals to generic large data structure traversals. The reader should be warned that literature provides almost no support for this claim.

6.1 Strafunski

Strafunski is a suite of tools written in Haskell that allow a programmer to perform typed traversal and transformation of source code representations. It is part of a large family of software language processing tools, family of which Stratego/XT are part of as well. In fact work on Strafunski was inspired by the XT bundle with its transformation strategies and Stratego's lack of typing [45] and is therefore very closely related in design.

Strafunski makes use of functional strategies which are generic functions that can be applied to terms of any type and that allow generic (recursive) traversal into subterms. As an interesting usability feature we note that while strategies are generic they are allowed to exhibit type specific behavior. On the paradigm of generic programming this may seem odd, but a similar feature is supported in Generic Haskell. Strafunski makes use of two types of strategy combinators: type-preserving and type-unification to allow for the mix of strategies which preserve input and output types with those that preserve the types. As an example to each of them, a strategies collecting occurrences of terms with particular properties from the source code would be a type-unifying strategy, whereas a strategy that transforms (rewrites) terms would be a type-preserving strategy [45].

Strafunski is composed of a library of traversal strategies and primitive combinators - StrategyLib. In terms of typing Strafunski is as safe as the Haskell type

checker permits. All types in Strafunski are in the class Term which is a specification of the class Update. The library uses type classes to allow overloading of primitive functions on the Terms such as *explode* and *implode*. It is important to note that because strategies are Haskell functions they are first-class citizens of the language and can therefore be passed as parameters to other strategies.

Although the goals of Strafunski are much less generic that other approaches to generic programming, it is successful at meeting its goals of providing type-safe strategic programming with generic traversal of abstract syntax representations.

For a definitive reading on Strafunski see [44]. For a comprehensive survey of strategies in program transformation systems see [57].

6.2 Scrap your boilerplate

The core concept behind Scrap your boilerplate [42] is providing a library that of combinators that can be used to define traversals and queries. In SYB traversals selectively modify complex data structures, whereas queries are used to collect information from the complex data structures. The goal of SYB is to allow the programmer to define boilerplate code once and then focus on the implementation of the real code. To this purpose SYB is composed of the library of combinators mentioned and a preprocessor that generates data-type specific functions. For the sake of illustration we will give the most popular example from literature [42,41], that of increasing the salary of every employee of a company by a certain amount:

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

In the example a generic traversal of the entire *company* data structure is performed and the function *incS* is applied which actually increases the salaries. From the point of view of the programmer, the incS function is interesting and the boilerplate traversal code has eliminated. From our point of view, everywhere is generic recursive traversal of the entire data structure which - for every term it encounters - applies mkT to it. As explained in the original paper, mkT is used a type extending function, wrapping incS in a what in Stratego would be a *try* strategy and in Stratego a *tryTP* strategy. More precisely it will produce the result of incS when incS succeeds on the current given term or will act as the identity function when that is not the case [42].

Interestingly, it is precisely this type extending function which is the weakness in the typing of SYB. SYB allows generic functions to be defined on all instances of type Data, types which are in turn instances of the class Typeable. In turn the user must define instances of the classes Typeable and Data for all the user-defined classes in order to provide overloaded functions for *mkT* and *everywhere*. In order for mkT to try to apply incS to the given term it must attempt a type-safe cast (at runtime) [42] from the input term to the type Salary through the Typeable class. This could be abused by a user by defining instances of the

Typeable class that do not follow the required signatures [20]. Providing user-defined instances of the Typeable class is actually not necessary as they are very simple and can be mechanically generated [42].

With the exception of this potential typing weakness, SYB is type safe and makes use of the Haskell type inference system. A proposal on the elimination of the type-safe cast in favor of type classes in order to obtain more type safety and increase flexibility of the language was proposed in [43]. Although a work around, the addition of type classes is not sufficient as the Haskell typing system would not be able to infer the correct types. As a consequence each function on the Data and Typeable classes receives one extra type parameter with the purpose of making instantiation types explicit [20]. This puts even more stress on the need that SYB had for arbitrary rank polymorphism in order to support functions as first-class citizens.

Further improvement to SYB is presented in [22,21]. For a discussion on the related aspects of XSLT and SYB and a proposed enhancement of SYB with XPath like idioms see [41].

7 Summary

This survey began by introducing generic programs and its dependence on type safeness. Our statement at the beginning was that type safeness and generic flexibility constituted a compromise. After having looked at a number of facilities for typed generic programming the understanding of this compromise is even deeper. Perhaps the term should not even be compromise but more of a *negotiation* between being safe and being adventurous. Narrower-field approaches such as Strafunski and SYB have proven to be much safer than very wide approaches such as Generic Haskell. Some of the explored approaches relied almost entirely on type system built into their target language, while Generic Haskell and PolyP (the original) relied on preprocessors and compilers to offer type safety. The two properties all that these approaches have in common are: attempt to reach strongly typed generic programming (1) without changing the existent type systems too much (2).

Unfortunately many areas have been left unexplored (some of which are references for further exploration in the next section) and others, such as Strafunski, which would require further investigation to evaluate their type safeness versus their flexibility.

8 Further exploration

As mentioned in the previous section, this survey has focused only on approaches to typing generic programs that are heavily based on the traditional Haskell type inference system. An area left entirely unexplored is the use type annotations or scoped type annotations to attempt to reduce the dependence on type inference systems or aid them where possible [52,35,38]. Going further than allowing typed annotations by attempting to embed a DSL for typed contracts into Haskell is

[19]. Although it implies foregoing our laziness of specifying any type signatures, being flexible could lead to reaching 'negotiation' outcomes.

In the field of program transformation abstract syntax trees are widely used, structures which are instances of generalized algebraic data types (GADT). The problem of providing type inference for them is difficult; S. Jones proposes two methods with [35] and without [53] type annotations.

The Haskell type system is based on the Hindley-Milner type inference system with extensions for type classes and more recently for arbitrary rank polymorphism. In recent years new type inference systems have been proposed [48,40,39] one of which attempts to use elements from both static and dynamic checking [11].

Acknowledgements

This survey was inspired by practical experience with the Stratego/XT program transformation tools bundle [4], its untyped nature and short discussions with its brain - Eelco Visser.

References

1. Artem Alimarine and Marinus J. Plasmeijer. A generic programming extension for clean. In Thomas Arts and Markus Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2002 Stockholm, Sweden, September 24-26, 2001, Selected Papers*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–185. Springer, 2001.
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany*, volume 243 of *IFIP Conference Proceedings*, pages 1–20. Kluwer, 2002.
3. Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, pages 28–115, 1998.
4. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.
5. Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.
6. Dave Clarke, Johan Jeuring, and Andres Lh. The generic haskell user's guide, 2001.
7. Dave Clarke and Andres Lh. Generic haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany*, volume 243 of *IFIP Conference Proceedings*, pages 21–47. Kluwer, 2002.
8. Lus Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

9. Merijn de Jonge, Eelco Visser, and Joost Visser. XT: a bundle of program transformation tools. *Electronic Notes in Theoretical Computer Science*, 44(2), 2001.
10. James C. Dehnert and Alexander A. Stepanov. Fundamentals of generic programming. In Mehdi Jazayeri, Rdiger Loos, and David R. Musser, editors, *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1998.
11. Cormac Flanagan. Hybrid type checking. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 245–256. ACM, 2006.
12. Ronald Garcia, Jaakko Jrvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 115–134. ACM, 2003.
13. Ralf Hinze. Polytypic values possess polykinded types. In Roland Carl Backhouse and Jos Nuno Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2000.
14. Ralf Hinze. Polytypic programming with ease. *Journal of Functional and Logic Programming*, 2001(3), 2001.
15. Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483, 2006.
16. Ralf Hinze and Johan Jeuring. Generic haskell: Applications. In Roland Carl Backhouse and Jeremy Gibbons, editors, *Generic Programming - Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 57–96. Springer, 2003.
17. Ralf Hinze and Johan Jeuring. Generic haskell: Practice and theory. In Roland Carl Backhouse and Jeremy Gibbons, editors, *Generic Programming - Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003.
18. Ralf Hinze, Johan Jeuring, and Andres Lh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.
19. Ralf Hinze, Johan Jeuring, and Andres Lh. Typed contracts for functional programming. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.
20. Ralf Hinze, Johan Jeuring, and Andres Lh. Comparing approaches to generic programming in haskell. In *SSDGP'06: Proceedings of the 2006 international conference on Datatype-generic programming*, pages 72–149, Berlin, Heidelberg, 2007. Springer-Verlag.
21. Ralf Hinze and Andres Lh. Scrap your boilerplate revolutions. In Tarmo Uustalu, editor, *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer, 2006.

22. Ralf Hinze, Andres Lh, and Bruno C. D. S. Oliveira. Scrap your boilerplate reloaded. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2006.
23. Stefan Holdermans, Johan Jeuring, Andres Lh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer, 2006.
24. Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5), 1992.
25. Patrik Jansson and Johan Jeuring. Polyp - a polytypic programming language. In *POPL*, pages 470–482, 1997.
26. Johan Jeuring and Patrik Jansson. Polytypic programming. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer, 1996.
27. Johan Jeuring, Sean Leather, Jos Pedro Magalhes, and Alexey Rodriguez Yakushev. Libraries for generic programming in haskell. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 165–229. Springer, 2008.
28. Mark P. Jones. Ml typing, explicit polymorphism and qualified types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS 94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 1994.
29. Mark P. Jones. A theory of qualified types. *Science of Computer Programming*, 22(3):231–256, 1994.
30. Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995.
31. Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.
32. Mark P. Jones. First-class polymorphism with type inference. In *POPL*, pages 483–496, 1997.
33. Simon L. Peyton Jones. Haskell 98: Declarations and bindings. *Journal of Functional Programming*, 13(1):39–66, 2003.
34. Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.
35. Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61. ACM, 2006.
36. Stefan Kaes. Parametric overloading in polymorphic programming languages. In Harald Ganzinger, editor, *ESOP 88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144. Springer, 1988.

37. Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LISP and Functional Programming*, pages 193–204, 1992.
38. Daan Leijen. Extensible records with scoped labels. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*, volume 6 of *Trends in Functional Programming*, pages 179–194. Intellect, 2005.
39. Daan Leijen. Hmf: simple type inference for first-class polymorphism. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 283–294. ACM, 2008.
40. Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 66–77. ACM, 2009.
41. Ralf Lmmel. Scrap your boilerplate with xpath-like combinators. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 137–142. ACM, 2007.
42. Ralf Lmmel and Simon L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In Zhong Shao and Peter Lee, editors, *Proceedings of TLDI 03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*, pages 26–37. ACM, 2003.
43. Ralf Lmmel and Simon L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 204–215. ACM, 2005.
44. Ralf Lmmel and Joost Visser. Design patterns for functional strategic programming. *CoRR*, cs.PL/0204015, 2002. informal publication.
45. Ralf Lmmel and Joost Visser. Typed combinators for generic traversal. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, volume 2257 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 2002.
46. Ralf Lmmel and Joost Visser. A strafunski application letter. In Vernica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proceedings*, volume 2562 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2003.
47. Andres Lh, Dave Clarke, and Johan Jeuring. Dependency-style generic haskell. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 141–152. ACM, 2003.
48. Weiyu Miao and Jeremy Siek. Incremental type-checking for type-reflective metaprograms. In Eelco Visser and Jaakko Jrvi, editors, *Proceedings of the ninth international conference on Generative programming and component engineering (GPCE 2010)*. ACM, 2010.
49. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

50. David R. Musser and Alexander A. Stepanov. Generic programming. In Patrizia M. Gianni, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC 88, Rome, Italy, July 4-8, 1988, Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1988.
51. Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, *Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 168–184. Springer, 2003.
52. Martin Odersky and Konstantin Lufer. Putting type annotations to work. In *POPL*, pages 54–67, 1996.
53. Tom Schrijvers, Simon L. Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 341–352. ACM, 2009.
54. Eelco Visser. Language independent traversals for program transformation. In *Workshop on Generic Programming (WGP 2000)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.
55. Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.
56. Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003.
57. Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.
58. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.