Crossroads
The ACM Student Magazine

Stratego: A Programming Language for Program Manipulation

by ***Karl Trygve Kalleberg***

Programming languages have a dual role in the construction of software. The language is both our substrate (the stuff we make software from), and our tool (what we use to construct software). Program transformation (PT) deals with the analysis, manipulation and generation of software. Therefore a close relationship exists between program transformation and programming languages, to the point where the PT field has produced many domain-specific languages for manipulating programs. In this article, I will show you some interesting aspects from one of these languages : Stratego.

Introduction

In most industries, the goal is to deliver a satisfying product within budget, on time. For some reason this is a rarity in the software industry. The observation that software is often late, expensive, buggy and bloated is certainly not new. Pioneer Edsger Dijkstra named this phenomenon the **software crisis** in his 1972 ACM Turing Award Lecture. In essence, it refers to the difficulty of writing correct, understandable and verifiable computer programs. The difficulty stems from complexity, expectations, and change **[1]**. This chronic crisis has been both direct and indirect inspiration for many bright ideas in the past 30 years, perhaps especially related to programming languages.

Given the central place of computer languages as both substrate and tool, it is reasonable to spend part of our collective resources searching for improvements in the way we design, implement and apply our languages. The rationale is simple: by improving our languages, we improve what are arguably the two most important technical pillars of software construction.

Comparing ourselves to other industries, we have some distance to cover. Software construction is still not a genuine engineering discipline. We cannot act in accordance with an established set of rules and expect to obtain a predicted result. Construction in other disciplines is mostly a systematic and regular activity, whereas software construction on the other hand, is anything but.

The reasons for this situation are many and complex. Given our context -- programming languages --

I will focus on how to tackle change and complexity in software using techniques and tools from the field of program transformation. I will introduce you to the domain-specific program transformation language, Stratego, and give an intuition for why transforming contemporary languages is tricky enough to warrant its own language. The journey ends with a brief discussion of the state of research in improving the representations we use for software, and how this work can significantly improve the capabilities of program transformation systems.

Ways Out

Software engineering seems to lack adequate, well founded and established methodologies for software construction. The result is that projects still fail at all stages of development. A failure may manifest itself at any phase of development: requirements analysis, specification or software design. Failures also frequently happen due to problems in other aspects of development, such as poor communication between customer and developer. For each development phase and aspect, many alternative techniques and best practices exist. These techniques and best practices are often tamed into coherent methodologies; disciplined approaches for arriving at a correct implementation for a given problem. Common to the methodologies is that they are realized on top of languages, which provide some degree of "substrate-level" support for the ideals promoted by the methodology. As we shall see, the degree of coupling between language and methodology varies greatly among the various methodologies.

Methodologies

Software development methodologies include the Waterfall Model, Agile Programming, Extreme Programming, and the Rational Unified Process. The purpose of a development methodology is primarily to provide rules of conduct which are applicable across tools, languages and teams. In a sense, they restrict the diversity we allow ourselves in expressing our software, with the aim that increased coherence and discipline results in software which is more robust, easier to maintain and more correct. Even though some of the methodologies come with custom tools that aid in the process, they are not a strict requirement for applying the methodology. Similarly, some of these methodologies promote specific language features to obtain their qualities, such as encapsulation. This does not mean, however, that the methodologies cannot be used with languages lacking such features.

Languages

Another approach to solving the software crisis is to start by focusing on the languages we express our software in, imbuing them with good qualities, while leaving out bad quirks. This approach has given rise to many philosophies for constructing computer languages. Object-oriented, aspect-oriented, procedural, functional, rule-based or even language-oriented **[3]** paradigms are described both in research and technical literature, and are taught at colleges and universities. These paradigms all focus on how the programming languages should be constructed to maximize maintainability, extensibility, security or other aspects of the development process. The challenge for the language designer is to strike a balance between well founded primitives, discipline and flexibility.

Getting this balance right on the first try is tricky, consequently the evolution of languages is prevalent. Very often, this evolution happens through cross pollination between paradigms: we see that popular languages may end up with new (old) features such as generics, nullable types, inner classes, delegates and embedded query languages. We are witnessing a creeping 'featuritis', of sorts. Guy L. Steele emphasized the need for languages to gracefully evolve in his OOPSLA'98 keynote *Growing a Language*, however the realization that one language may not be able to bind them all is not a new observation. In 1969 and 1971, symposiums were held on the topic of extensible languages ([4], [5]), which gives us an indication that this issue has been with us almost since the inception of structured programming languages.

Combined Approaches

Some approaches to solving the software crisis intimately interweave methodology with language. The Eiffel Method, based around the object-oriented Eiffel language, is one such example. Central to this approach is language support for contracts. A **contract** in this context is a formal description of the behavior of a method, and of data invariants in a class, specified by the programmer. The more detailed and accurate such a contract is, the more of the program's behavior can be analyzed and tested in advance. Such contracts can even serve as a basis for optimizations and (unit) testing.

A Common Theme: Program Transformation

Common to all approaches in the previous section, is that they need to handle the troublesome trio of the software crisis: complexity, change and expectations. The various methodologies attack these facets with different techniques. Agile and Extreme Programming embrace change by promoting the use of refactoring (discussed below) and unit testing; in an attempt to minimize the cost of changing the actual source code. The Unified Process advocates the use of modeling tools, which will generate the source code (semi-)automatically. This is based on the assumption that changing the model is less error prone and quicker than doing equivalent changes at the source code level. Intimately tied to each approach, you will find essential parts of program transformation.

But what is program transformation? A frequently used definition is that **program transformation** is the systematic development of efficient programs from high level specifications by meaning preserving program manipulations [6]. This definition may turn out to be too restrictive to capture the bulk of activities that program transformation is applied to. For example, in its strictest sense, it would not apply to refactoring, or even source code analysis tools producing reports of potential defects. A more inclusive and accurate definition (perhaps under a new name, such as program manipulation), should include provisions about the fundamental activities of program transformation: analysis, manipulation and generation of software.

Analysis

Whenever you use a compiler, you are relying on automatic analysis of your code, both syntactic and semantic. In all but the most trivial of languages, semantic analysis is required to make sense of what a given program means. Take the following C++ code fragment as an example:

```
f() {
  int a(b);
}
```

It is not possible to tell what this expression means on a purely syntactical basis: that is, given only the code fragment given above, it is not possible to know what the statement `int a(b);` means, since nothing is known about `b`. If `b` is a value (or variable), it merely says that the integer a should be initialized to the value of `b`. However, if `b` is a type, the statement is a function declaration, and `a` is to be considered a function taking one argument of type `b` which returns an `int`. The fact that several interpretations of the same piece of code are possible means that the code is **ambiguous**. The tools and techniques for program transformation I will cover later are well suited for dealing with ambiguities both syntactically (parsing) and semantically.

In a related vein, "extended" syntactic and semantic analysis is often referred to as **static analysis**. Static analysis tools search for common programming errors and code written in a bad style. As such, they can be used to support or even enforce aspects of given methodologies. For example, they can be used to search for the existence of bad design patterns (also called "anti"-patterns), methods that are too long, pairs of classes that seem too tightly coupled, classes that are undocumented, or code that violates subsystem encapsulation, and so on. Sometimes, analysis for bad coding style is referred to as detection of bad code **smells**.

Well known tools for this include the venerable C/C++ `lint`, FindBugs **[7]** and PMD **[8]**. In the Office group at Microsoft, Robert F. Crew constructed a Prolog-based analyzer of abstract syntax trees (abstract syntax trees are discussed later), which has been used to detect bad code smells in the Office products **[9]**. Static analysis has also made its way into many modern compilers. The MipsPro C/C++ compilers, the Eclipse Compiler for Java and Jikes all provide code style warnings in addition to the normal error reporting.

Source code querying is another example of where program transformation techniques are useful. Many integrated development environments (IDEs) provide functionality for structural searches in a project. Methods, fields, classes and packages may be searched for based on where they are declared and referenced. Again, the basis for this functionality is the analysis of source code.

Manipulation

Perhaps the most popular application of manipulatory program transformation is refactoring which is changing the internal structure of a program in order to increase understandability and maintainability, without changing the program's observable behavior. Most IDEs support some notion of refactoring these days. The requirement that a refactoring be behaviorally preserving, requires the presence of accurate semantic information about the code. The change is usually done on an internal representation of the source code, the abstract syntax tree, which has been obtained through parsing and semantic analysis. Changes to the AST are then automatically reflected back into the source code text.

Generation

Compared to analysis and manipulation, generation of code requires a lot less infrastructure, and is therefore a rather common activity for programmers to engage in. The generation of glue code from textual specifications, such as CORBA IDLs to actual C++ code, the generation of SQL and Java code from database schemas or generation of JavaScript code by JSPs and Servlets are all familiar examples.

In most cases, however, the generated source code is treated as pure text. This invariably results in seemingly minor mistakes: a missing '}' here or a missing '"' there. The problem is that no way exists to statically determine if the generating code is correct, and report this to the programmer. Or is there? As it happens, by employing techniques from program transformation -- the use of transformations with concrete syntax -- we can actually ensure that we never generate a syntactically invalid program ever again.

Taken together, the facilities for analysis, manipulation and generation make program transformation systems well equipped to handle complexity and change. It would be sensationalist to argue that they completely solve these aspects; program transformation is *not* the silver bullet. But if working on computer languages using program transformation is so useful, why is it not more widespread? The story is familiar. Firstly, little awareness exists about the techniques and their usefulness, partly due to education, and partly due to the lack of accessible books on the subject. Second, the availability of solid tools has been scanty. I will do my bit to rectify this by making you aware of one system for program transformation: Stategо/XT.

A Programming Language for Program Manipulation

The Stratego/XT environment is a toolkit consisting of a language for manipulating software: Stratego, and a collection of infrastructure tools for working with software artifacts, XT. This environment is used to construct program transformation systems. Both the language and its accompanying toolkit have some uncommon properties which make them ideally suited for tackling the problems of analysis, manipulation and generation of software. Let us ease into the subject by considering the diagram of a typical program transformation system created with Stratego/XT.

(I will only give a bird's eye view of Stratego/XT, to provide an intuition of its fundamental concepts. The details can be found in the full tutorial **[2]**.)

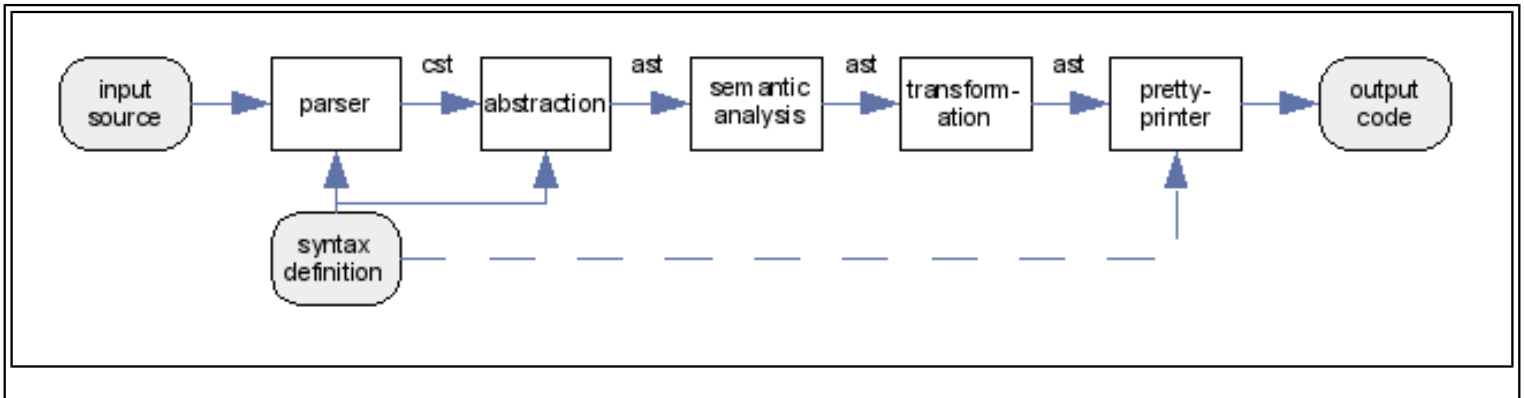

Figure 1: Anatomy of a typical Program Transformation System

For people familiar with compilers, this pipelined approach should come as a surprise. An input source file is parsed to a (concrete) syntax tree (CST). This is pruned to become an abstract syntax tree (AST) in the `abstraction` step, and is then annotated with type information and contextual links by a `semantic analysis` step. At this point, the AST has all the information embedded into it required for arbitrary transformations. The `transformation` step (which may in reality be multiple steps in sequence, or even a loop of steps which eventually terminates) results in a final AST, which is be serialized back into a textual source file.

Parsing of source code is a crucial part of any program transformation system, so we will devote a bit of time to explain it. The syntax definition (grammar) is used to derive a number of important artifacts. First, it is used by the parser to parse the textual source files. The output of this phase is a concrete syntax tree (CST): a tree representation of the source code, void of whitespace. Second, the syntax definition is used to automatically obtain an AST from the CST. The AST representation strips away all the superfluous syntax and only contains the essence of the code. Third, we use the syntax definition to derive an "inverse"-parser, or pretty-printer, that takes an AST to a source file. The grammar system used in Stratego/XT is called Syntax Definition Formalism (SDF). It allows us to construct completely modular grammars, where the level of modularity is highly scalable. As an example, SDF has been used to construct complete Java 1.4 and 1.5 grammars, and also a grammar for AspectJ. The latter is an extension to the Java grammar, which allows separate maintenance and evolution of the two.

Defining Syntax

I am not going to provide any realistic grammars in this article, but consider the following definition of arithmetic expressions with variables. The definition is split into two files, `Lexicals.sdf` and `Expressions.sdf`. They are treated as separate modules by the SDF tools, and we can compose them using the `imports` directive.

```
module Lexicals
exports
  sorts Nat Id Real

  lexical syntax
    [0-9]+ "." [0-9]+  -> Real
    [0-9]+             -> Nat
    [a-z][a-z0-9]*     -> Id
```

Figure 2: `Lexicals.sdf`

The file in Figure 2 defines three lexicals (or atoms, if you will), namely `Real`, `Nat` and `Id`. Their

allowed syntaxes are given as regular expressions.

```
module Expressions
imports Lexicals
exports
  context-free start-symbols Exp
  sorts Exp

  context-free syntax
    Real        -> Exp {cons("Real")}
    Nat         -> Exp {cons("Nat")}
    Id          -> Exp {cons("Id")}
    Exp "+" Exp -> Exp {left, cons("Plus")}
    Exp "-" Exp -> Exp {left, cons("Minus")}
    Exp "*" Exp -> Exp {left, cons("Times")}

  context-free priorities
    Exp "*" Exp -> Exp > {left: Exp "+" Exp -> Exp
                                Exp "-" Exp -> Exp}
```

Figure 3: Expressions.sdf

The file in Figure 3 defines the grammar for the arithmetic operations +, - and *. The line

```
    Real        -> Exp {cons("Real")}
```

says that Real is a basic building block in constructing an expression Exp. It also tells the abstraction step in Figure 1 how to construct an AST: by making a Real node.

We are also allowed to specify the associativity of our operators. For example, the tag left in the line

```
    Exp "+" Exp -> Exp {left, cons("Plus")}
```

tells us that the + operator is left associative. We are even allowed to declare the precedence between operators:

```
  context-free priorities
    Exp "*" Exp -> Exp > {left: Exp "+" Exp -> Exp
                                Exp "-" Exp -> Exp}
```

Parsing Expressions

Armed with the syntax definition above, we can now parse the following expression:

```
0+0.1*a
```

The abstract syntax tree constructed from this is given in Figure 4.

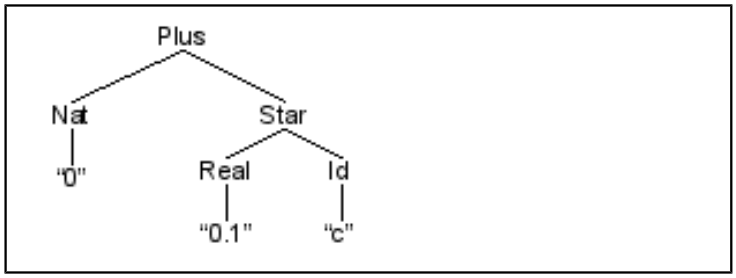

Figure 4: Abstract syntax tree for `0+0.1*a`

In the following, we will rather represent such ASTs textually. The AST from Figure 4 can be written in a notation referred to as ATerms:

```
Plus(Nat("0"),Times(Real("0.1"),Id("c")))
```

Manipulating ASTs with Stratego

To get on with the real program manipulation, we need to start modifying our ASTs. I have argued loosely that program transformation techniques are very good for working *on* source code from normal programming languages, but I have not said anything about what languages *for* program transformation should look like. In the same way that almost any language is usable for modifying XML documents, some languages, such as XSLT, are more tuned to the task than others. Stratego is a language tuned to the manipulation of ASTs.

In Stratego jargon, ASTs are called **terms**. This has historical reasons: Stratego is based on the theory of rewriting systems, and borrows much of its terminology from this field. We use **signatures** to define our terms (ASTs). Think of signatures as type declarations. A signature for our arithmetic expressions is given next.

```
module Arithmetic
signature
  constructors
    Plus  : Exp * Exp -> Exp
    Minus : Exp * Exp -> Exp
    Times : Exp * Exp -> Exp
    Nat   : String -> Exp
    Id    : String -> Exp
    Real  : String -> Exp
```

This signature is automatically derived from the syntax definition we wrote above. It tells Stratego how we may construct legal terms (ASTs): a `Plus` is a term (node) taking two subterms, both of type `Exp`. The ATerm notation above is exactly the representation Stratego uses for its terms, i.e.

```
Plus(Nat("0"),Times(Real("0.1"),Id("c")))
```

is a valid term according to the Stratego signature we have defined.

So how to we modify this term? With functions called rules and strategies. Just as Java programs are built from methods, fields, classes and packages, Stratego programs are built from signatures, rules, strategies and modules. A rule is a mini-transformation, a **rewrite**, from one term (a **left-hand side**) to another (a **right-hand side**). For example `Nat("0") -> Nat("1")` is a rule that rewrites the natural number zero to the natural number one, in our notation. We may give the rule a name, for example `Inc`:

```
Inc:
  Nat("0") -> Nat("1")
```

Looking at our term above, we see that our rule may be applicable, if we only could apply it at the right spot. How do we do that? This is where strategies come in. A strategy is a function that controls the application of rules. In our eagerness for change, we may be tempted to apply our rule all over the place, hoping that it will fit somewhere. The following strategy does just that.

```
  topdown(try(Inc))
```

The above strategy walks the term (think about the tree in Figure 4) in a top down order (preorder), trying to apply our `Inc` at all nodes. When `Inc` succeeds, it will keep the result. When `Inc` fails (because we are not at the term `Nat(0)`), the strategy will forget it ever tried. After we have applied the strategy to our term, we get the not-so-surprising result:

```
Plus(Nat("1"),Times(Real("0.1"),Id("c")))
```

The strategy `topdown` is called a **traversal**, because it traverses a term. There are other traversals in the Stratego library, such as `bottomup`, `innermost` and `outermost`. They are all used to select the order in which subterms are visited in.

The standard way of doing tree traversals in Java (or C#, and many other OO languages), is using visitors. When tree traversal becomes a frequent and basic operation in your programs, the OO-notation for visitors results in an unmanageable code explosion.

Figure 5 shows a complete Stratego program incorporating all the code above. After compilation with the Stratego compiler, it will result in a stand alone program that accepts an ATerm like the one above on `stdin` and will write a transformed ATerm to `stdout`.

```
module Example
imports lib
signature
  constructors
    Plus  : Exp * Exp -> Exp
    Minus : Exp * Exp -> Exp
    Times : Exp * Exp -> Exp
    Nat   : String -> Exp
    Id    : String -> Exp
    Real  : String -> Exp

rules
  Inc:
    Nat("0") -> Nat("1")

strategies
  main = io-wrap(topdown(try(Inc)))
```

Figure 5: A complete Stratego program

By composing many of these transformations in a series, through (Unix) pipes, we get the pipeline from Figure 1. The Stratego/XT environment comes with a whole component infrastructure that abstracts over these details, called XTC. Using XTC, each square in Figure 1 becomes an XT component, and the XT component composition language can be used to create flexible and advanced transformation pipelines.

Generic Transformations

The examples we showed previously are all conceptually very simple. It is important to realize that Stratego is also useful for heavy-duty work, such as semantic analysis of Java and C++ programs, or even their optimization and compilation. Much work has been invested into these areas. Fortunately, it turns out that essential parts of this heavy-duty work are decomposable and even generalizable.

The fruits of this labor can be found in the Stratego library. Generic transformations (think of them akin to very small OO frameworks) are now available for reuse. By plugging your language grammar and glue code into these transformations, you can get proper name scoping as well as data and control flow analysis quite easily.

Concrete Syntax

I claimed that syntax errors in code generators were a thing of the past. One powerful technique for dealing with this in Stratego/XT is the use of concrete syntax **[10]** inside the Stratego programs. Let us consider another rule, the `DouglasOperation`, which must be applied to the expression `6*7`. It will of course result in `42`. In the ATerm notation, such a rule will look like:

```
DouglasOperation:
  Times(Nat("6"), Nat("7")) -> Nat("42")
```

While this notation is very precise, it is a rather unfamiliar way of writing arithmetic, and it does not scale well to large expressions. The alternative offered by concrete syntax is to write the rule using the language for which we already have a grammar:

```
DouglasOperation:
  |[6*7]| -> |[42]|
```

Enclosed in the special brackets `|[]|` is the concrete syntax for the terms we want. With the proper directive, the Stratego compiler will know which grammar to use in order to parse the contents of such brackets, and will automatically expand `|[6*7]|` to `Times(Nat("6"),Nat("7"))` behind the scenes. More importantly, the right-hand side will also be expanded to a valid term. If a valid term cannot be obtained, i.e. the expression inside the special brackets is invalid; we will get a *compile-time* warning. This way, we know before deployment that our code generation is at least syntactically correct.

Environment

The Stratego/XT environment is similar to many other language environments, so I will not go into details. The environment provides an interactive editor (Spoofax), a compiler, an interpreter, as well as a source code documentation tools (xDoc) in the style of Javadoc.

Data Structures for Program Manipulation

As I said before, program transformation is no silver bullet, and comes with some problems of its own. Perhaps most important is the fact that there is no perfect data structure for representing and manipulating software. Programs are extremely rich and interwoven structures. One reason for this is the presence of substantial amounts of accidental complexity in computer language designs, at least when seen from our transformational viewpoint. I mentioned the ambiguity problem earlier; another is lack of orthogonality. For example, in many Algol-derived languages, such as C, C++, Java and C#, there is no difference between `for` and `while` loops. You can trivially rewrite one to the other. (`foreach` should not be considered equivalent to `for`/`while`: we cannot rewrite an arbitrary `while` to a `foreach` without resorting to cleverness of encoding and additional code generation.) Annoyance arises from always having to deal with both forms when writing transformations.

The lack of orthogonality also exacerbates the issue of non-local side-effects. If your transformation receives a term (AST) annotated with semantic information, you may quickly invalidate all this information by making one minor change. Some of this is essential and desired: name scoping is considered a desirable property. If you rename a variable, you must also rename all its uses in the lower scopes. This is straightforward inside methods and classes, but renaming a field or method in a class requires a bit more footwork.

Non-orthogonality and ambiguity are easy-to-grasp examples that are visible at the tip of the iceberg of symptoms. The fact is that few contemporary languages were designed to be easily transformable, and this sometimes comes to bite us. Coming up with a well founded way for abstracting over accidental complexity, and maintaining non-local consistency is still a research problem.

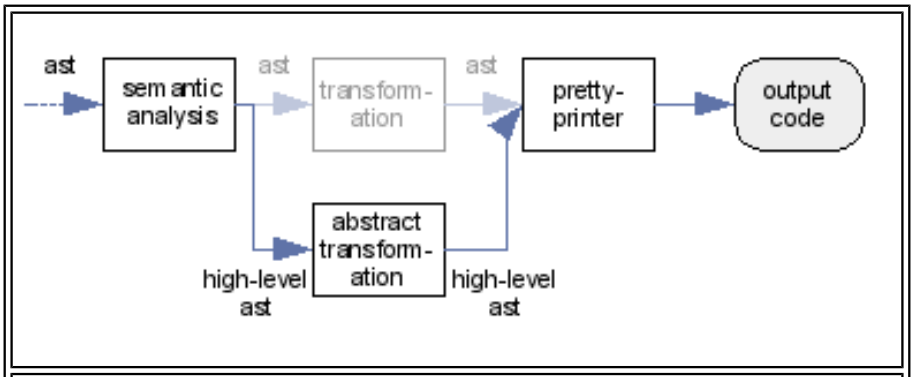

Figure 6: High-level abstract syntax trees

One way to deal with the abstraction issue -- abstracting over `for` and `while`, by considering both as (potentially) unbounded loops -- is indicated in Figure 6. After semantic analysis, a high-level AST is produced, where accidental complexity has been pruned. At this level, both `for` and `while` are now generalized into an `UnboundedLoop` term, adhering to the following signature, which is essentially a `while`:

```
UnboundedLoop: Exp * CodeBlock
```

In the case of `foreach` and many `for` loops, we can extract lower and upper bounds, as well as an induction variable. When this is possible, we can put more structure on our loop, and make it into a `BoundedLoop` instead:

```
BoundedLoop: Lower * Upper * Step * CodeBlock
```

This variant is like the Pascal-style `for` loop, and opens up for many loop transformations which are either very difficult or downright impossible on an `UnboundedLoop`.

More abstract (and simplified) transformations are expressed on this high-level AST, preferably using generic transformations where possible. There are cases where dealing with accidental complexity is required, for example when doing refactoring. When moving code around, we need to keep `for`s as `for`s, but at the same time we want to perform our visibility analysis as simply as possible. In this case, fine-grained switching between the high-level and low-level AST representations is useful, and being able to trace the origin of a given `UnboundedLoop` is important. These are all topics still under research.

Conclusion

At this point, I hope you have an appreciation for what program transformation is, why it is intimately tied to programming languages, and how it fits into the picture of software engineering by addressing two of the three causes of the software crisis, namely complexity and change.

What is more, program transformation brings with it interesting programming languages of its own. While the tiny examples of Stratego are rather sterile, they show how concisely the domain-specific notation of Stratego allows us to express operations on terms. They illustrate that Stratego is a precise language for manipulating software.

Term-rewriting, the mathematical foundation on which Stratego is based (which, out of common courtesy, I have kept out of the article), dates back decades, and is closely related to the theory behind functional programming. Both theories provide a pool of stable knowledge on which to build program transformation languages, tools and techniques.

The imperfect border of the field of program transformation is lined by a research front. Many of the topics on this front are shared between other disciplines, such as how to organize and modularize programs for program transformation. I detailed one topic which is specific to the field, namely the quest for an optimal program representation.

As with any paradigm, buying into program transformation wholesale is probably not appropriate. What you should be aware of, however, is that the field is a treasure trove of techniques for the analysis, generation and manipulation of software. This is witnessed in the many examples where program transformation techniques have been successfully transplanted into other parts of software engineering.

The Stratego/XT environment is freely available from **www.stratego-language.org**, and works on most platforms.

Acknowledgements
I wish to thank Eelco Visser for his insightful comments on this article, and Andrew David, the diligent ACM editor who helped me clean up this manuscript for publication.

References

1
Wikipedia entry. *The* ***software crisis***

2
M. Bravenboer, K.T. Kalleberg, E. Visser. ***The Stratego/XT Tutorial***

3
Language-Oriented Programming refers to the pervasive use of domain-specific languages to solve well-defined, complicated tasks, in the style of Unix' "little languages"".

4
Carlos Christensen and Christopher J. Shaw, editors, *Proceedings of the Extensible Languages Symposium*, Boston, Massachusetts, May 13, 1969 [SIGPLAN Notices 4 no. 8 (August 1969)]

5

Stephen A. Schuman, *Proceedings of the International Symposium on Extensible Languages*, Grenoble, France, September 6-8, 1971 [SIGPLAN Notices 6 no. 12 (December 1971)].

6

Paraphrased from Partsch, H. and Steinbrüggen, R. 1983. *Program Transformation Systems*. ACM Comput. Surv. 15, 3 (Sep. 1983), 199-236. DOI= **http://doi.acm.org/10.1145/356914.356917**

7

FindBugs - **http://findbugs.sourceforge.net**

8

PMD - **http://pmd.sourceforge.net**

9

R. E Crew. *ASTLOG: A language for examining abstract syntax trees*. In Proc. of the First Conf. on Domain Specific Languages, pages 229-242, Oct. 1997.

10

E. Visser. *Meta-programming with concrete object syntax*. In Generative Programming and Component Engineering (GPCE) Conference, LNCS 2487, pages 299--315. Springer, 2002.

Biography

Karl Trygve Kalleberg is a PhD student at the University of Bergen, Norway, researching improved program representations. He spent the last academic year at Universiteit Utrecht in The Netherlands, working on the Stratego/XT environment. He is funded by The Research Council of Norway.