CS4200-B: Compiler Construction Back-End

Eelco Visser

TU Delft

CS4200: Two Courses

CS4200-A: Front-End (5 ECTS)

- Syntax and Type Checking
- Project: Build front-end of compiler for ChocoPy in Spoofax
- Exam in October

CS4200-B: Back-End (5 ECTS)

- Analysis and Code Generation
- Project: Build back-end of compiler for ChocoPy in Spoofax
- Exam in January

Lecture Topics CS4200-B (Q2) (Tentative)

- Transformation
- Virtual Machines
- Code Generation
- Data-Flow Analysis
- Monotone Frameworks
- Register Allocation
- Memory Management

Communication

mattermost organization

#spoofax-users channel on research group Slack

- Where many Spoofax users hang out
- Ask questions about Spoofax
- (Not answers to assignments)
- No guarantee to quick response, or answer at all
- Send me an email if you want an invitation

ChocoPy: Project Language

ChocoPy: A Typed Restricted Subset of Python 3

```python
# Binary-search trees
class TreeNode(object):
   value:int = 0
   left:"TreeNode" = None
   right:"TreeNode" = None

   def insert(self:"TreeNode", x:int) -> bool:
      if x < self.value:
         if self.left is None:
            self.left = makeNode(x)
            return True
         else:
            return self.left.insert(x)
      elif x > self.value:
         if self.right is None:
            self.right = makeNode(x)
            return True
         else:
            return self.right.insert(x)
      return False

   def contains(self:"TreeNode", x:int) -> bool:
      if x < self.value:
         if self.left is None:
            return False
         else:
            return self.left.contains(x)
      elif x > self.value:
         if self.right is None:
            return False
         else:
            return self.right.contains(x)
      else:
         return True
```

ChocoPy is a programming language designed for classroom use in undergraduate compilers courses. ChocoPy is a restricted subset of Python 3, which can easily be compiled to a target such as RISC-V. The language is fully specified using formal grammar, typing rules, and operational semantics. ChocoPy is used to teach CS 164 at UC Berkeley. ChocoPy has been designed by Rohan Padhye and Koushik Sen, with substantial contributions from Paul Hilfinger.

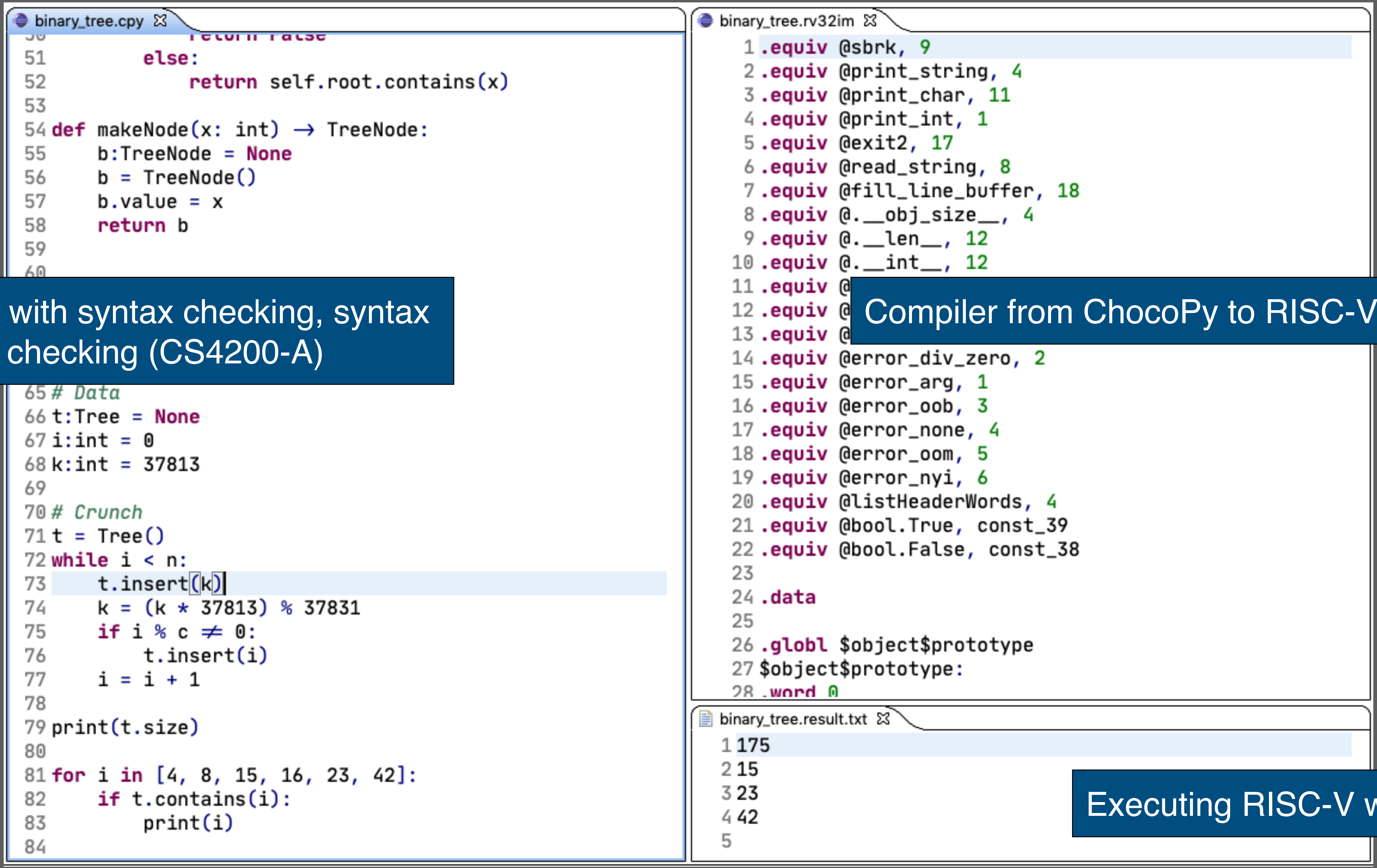
At a glance, ChocoPy is:

- **Familiar**: ChocoPy programs can be executed directly in a Python (3.6+) interpreter. ChocoPy programs can also be edited using standard Python syntax highlighting.
- **Safe**: ChocoPy uses Python 3.6 type annotations to enforce static type checking. The type system supports nominal subtyping.
- **Concise**: A full compiler for ChocoPy be implemented in about 12 weeks by undergraduate students of computer science. This can be a hugely rewarding exercise for students.
- **Expressive**: One can write non-trivial ChocoPy programs using lists, classes, and nested functions. Such language features also lead to interesting implications for compiler design.

Bonus: Due to static type safety and ahead-of-time compilation, most student implementations outperform the reference Python implementation on non-trivial benchmarks.

Source: https://chocopy.org/

A Compiler and IDE for ChocoPy



ChocoPy IDE with syntax checking, syntax coloring, type checking (CS4200-A)

Compiler from ChocoPy to RISC-V (CS4200-B)

Executing RISC-V with simulator

ChocoPy: Language Design and Implementation Documentation

ChocoPy v2.2: Language Manual and Reference

Designed by Rohan Padhye and Koushik Sen; v2 changes by Paul Hilfinger

University of California, Berkeley

November 23, 2019

Contents

ChocoPy v2.2: RISC-V Implementation Guide

University of California, Berkeley

October 31, 2019

1 Introduction

This document is intended to accompany the ChocoPy language reference manual, to serve as a guide for developers wishing to implement a ChocoPy compiler that targets the RISC-V instruction-set architecture.

Specifically, this guide assists with the task of generating RV32IM[1] assembly code for a semantically valid and well-typed ChocoPy program. This guide is not a complete specification; it is the developer's responsibility to implement the full operational semantics listed in the language manual. The design decisions described in this guide mirror the design of the official reference implementation, which is not optimized for maximum performance. Developers are free to tweak any or all of these design choices.

2 Naming conventions

The RISC-V assembly program generated for a ChocoPy program uses a single global namespace. To ensure unique naming, each such program entity is referred to by its fully-qualified name (FQN). FQNs are defined as follows. A class with name `C` has a FQN of `C`. A global variable with name `v` has FQN of `v`. A function `f` defined in global scope has a FQN of `f`. These names do not collide since they are distinct in the global namespace of the ChocoPy program as well. A method `m` defined in class `C` has FQN of `C.m`. A nested function `g` defined inside a function or method with FQN `F` has a FQN of `F.g`. A local variable `v` defined in a function or method with FQN `F` has a FQN of `F.v`. An attribute `a` defined in a class `C` has a FQN of `C.a`. As an example, consider the program:

```
class C(object):
  def f(self:"C") -> int:
    def g() -> int:
      x:int = 1
      return x
    return g()
C().f()
```

Here, the local variable `x` has a FQN of `C.f.g.x`.

[1]RV32IM is the 32-bit version of RISC-V with integer-only arithmetic, including multiplication (and division) instructions.

Nano-Pass Compiler Architecture

Nano-pass approach to constructing a compiler back-end.

Essentials of Compilation

The Incremental, Nano-Pass Approach

Jeremy G. Siek
Indiana University

with contributions from:
Carl Factora
Andre Kuhlenschmidt
Ryan R. Newton
Ryan Scott
Cameron Swords
Michael M. Vitousek
Michael Vollmer

OCaml version:
Andrew Tolmach
(with inspiration from a Haskell version by Ian Winter)

April 19, 2021

https://wphomes.soic.indiana.edu/jsiek/

https://www.dropbox.com/s/ktdw8j0adcc44r0/book.pdf?dl=1

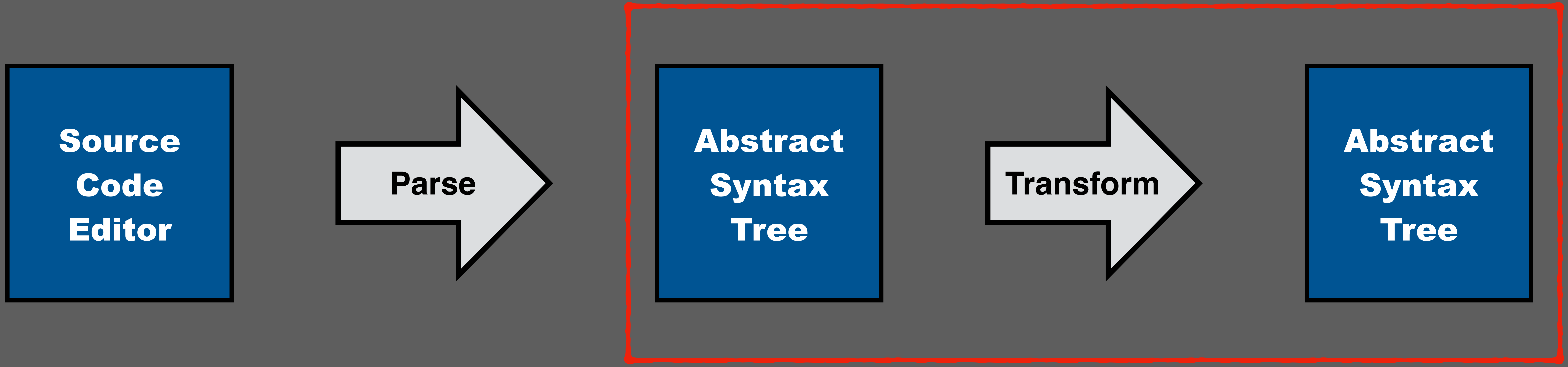
Program Transformation by Term Rewriting

Eelco Visser

TU Delft

CS4200 | Compiler Construction | November 11, 2021

This Lecture



Define transformations on abstract syntax trees (terms) using rewrite rules

Reading Material

The following papers add background, conceptual exposition, and examples to the material from the slides. Some notation and technical details have been changed; check the documentation.

Term rewrite rules define transformations on (abstract syntax) trees. Traditional rewrite systems apply rules exhaustively. This paper introduces programmable rewriting strategies to control the application of rules, the core of the design of the Stratego transformation language.

Note that the notation for contextual rules is no longer supported by Stratego. However, the technique to implement contextual rules still applies.

ICFP 1998

https://doi.org/10.1145/291251.289425

Building Program Optimizers with Rewriting Strategies*

Eelco Visser[1], Zine-el-Abidine Benaissa[1], Andrew Tolmach[1,2]
Pacific Software Research Center

[1] Dept. of Comp. Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA
[2] Dept. of Computer Science, Portland State University, P.O. Box 751, Portland, Oregon 97207 USA
visser@acm.org, benaissa@cse.ogi.edu, apt@cs.pdx.edu

Abstract

We describe a language for defining term rewriting strategies, and its application to the production of program optimizers. Valid transformations on program terms can be described by a set of rewrite rules; rewriting strategies are used to describe when and how the various rules should be applied in order to obtain the desired optimization effects. Separating rules from strategies in this fashion makes it easier to reason about the behavior of the optimizer as a whole, compared to traditional monolithic optimizer implementations. We illustrate the expressiveness of our language by using it to describe a simple optimizer for an ML-like intermediate representation.

The basic strategy language uses operators such as sequential composition, choice, and recursion to build transformers from a set of labeled unconditional rewrite rules. We also define an extended language in which the side-conditions and contextual rules that arise in realistic optimizer specifications can themselves be expressed as strategy-driven rewrites. We show that the features of the basic and extended languages can be expressed by breaking down the rewrite rules into their primitive building blocks, namely matching and building terms in variable binding environments. This gives us a low-level core language which has a clear semantics, can be implemented straightforwardly and can itself be optimized. The current implementation generates C code from a strategy specification.

1 Introduction

Compiler components such as parsers, pretty-printers and code generators are routinely produced using program generators. The component is specified in a high-level language from which the program generator produces its implementation. Program optimizers are difficult labor-intensive components that are usually still developed manually, despite many attempts at producing optimizer generators (e.g., [19, 12, 28, 25, 18, 11]).

*This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

A program optimizer transforms the source code of a program into a program that has the same meaning, but is more efficient. On the level of specification and documentation, optimizers are often presented as a set of correctness-preserving *rewrite rules* that transform code fragments into equivalent more efficient code fragments (e.g., see Table 5). This is particularly attractive for functional language compilers (e.g., [3, 4, 24]) that operate via successive small transformations, and don't rely on analyses requiring significant auxiliary data structures. The paradigm provided by conventional rewrite engines is to compute the normal form of a program with respect to a set of rewrite rules. However, optimizers are usually not implemented in this way. Instead, an algorithm is produced that implements a *strategy* for applying the optimization rules. Such a strategy contains meta-knowledge about the set of rewrite rules and the programming language they are applied to in order to (1) control the application of rules; (2) guarantee termination of optimization; (3) make optimization more efficient.

Such an ad-hoc implementation of a rewriting system has several drawbacks, even when implemented in a language with good support for pattern matching, such as ML or Haskell. First of all, the transformation rules are embedded in the code of the optimizer, making them hard to understand, to maintain, and to reuse individual rules in other transformations. Secondly, the strategy is not specified at the same level of abstraction as the transformation rules, making it hard to reason about the correctness of the optimizer even if the individual rules are correct. Finally, the host language has no awareness of the transformation domain underlying the implementation and can therefore not use this domain knowledge to optimize the optimizer itself.

It would be desirable to apply term rewriting technology directly to produce program optimizers. However, the standard approach to rewriting is to provide a fixed strategy (e.g., innermost or outermost) for normalizing a term with respect to a set of user-defined rewrite rules. This is not satisfactory when—as is usually the case for optimizers—the rewrite rules are neither confluent nor terminating. A common work-around is to encode a strategy into the rules themselves, e.g., by using an explicit function symbol that controls where rewrites are allowed. But this approach has the same disadvantages as the ad-hoc implementation of rewriting described above: the rules are hard to read, and the strategies are still expressed at a low level of abstraction.

In this paper we argue that a better solution is to use explicit specification of *rewriting strategies*. We show how

Stratego/XT combines SDF2 and Stratego into toolset for program transformation.

This paper gives a high-level overview of the concepts.

The StrategoXT.jar is still part of the Spoofax distribution.

Lecture Notes in Computer Science 2003

https://doi.org/10.1007/978-3-540-25935-0_13

Program Transformation with Stratego/XT

Rules, Strategies, Tools, and Systems in Stratego/XT 0.9

Eelco Visser

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80089 3508 TB, Utrecht, The Netherlands
visser@acm.org
http://www.stratego-language.org

Abstract. Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation systems including parsing and pretty-printing. The framework addresses the entire range of the development process; from the specification of transformations to their composition into transformation systems. This chapter gives an overview of the main ingredients involved in the composition of transformation systems with Stratego/XT, where we distinguish the abstraction levels of rules, strategies, tools, and systems.

1 Introduction

Program transformation, the automatic manipulation of source programs, emerged in the context of compilation for the implementation of components such as optimizers [28]. While compilers are rather specialized tools developed by few, transformation systems are becoming widespread. In the paradigm of generative programming [13], the generation of programs from specifications forms a key part of the software engineering process. In refactoring [21], transformations are used to restructure a program in order to improve its design. Other applications of program transformation include migration and reverse engineering. The common goal of these transformations is to increase programmer productivity by automating programming tasks.

With the advent of XML, transformation techniques are spreading beyond the area of programming language processing, making transformation a necessary operation in any scenario where structured data play a role. Techniques from program transformation are applicable in document processing. In turn, applications such as Active Server Pages (ASP) for the generation of web-pages in dynamic HTML has inspired the creation of program generators such as Jostraca [31], where code templates specified in the concrete syntax of the object language are instantiated with application data.

Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is based on the paradigm of rewriting under the control of programmable rewriting strategies. The XT tools provide facilities for the infrastructure of transformation

C. Lengauer et al. (Eds.): Domain-Specific Program Generation, LNCS 3016, pp. 216–238, 2004.

Spoofax combines SDF2 and Stratego into a language workbench, i.e. an IDE for creating language definition from which IDEs for the defined languages can be generated.

A distinctive feature of Spoofax is live language development, which supports developing a language definition and programs in the defined language in the same IDE instance.

Spoofax was developed for Eclipse, which is still the main development platform. However, Spoofax Core is now independent of any IDE.

Note that the since the publication of this paper, we have introduced more declarative approaches to name and type analysis, which will be the topic of the next lectures.

OOPSLA 2010

https://doi.org/10.1145/1932682.1869497

The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a "build" button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Documentation for Stratego at the spoofax.dev website.

Stratego

The Stratego language caters for the definition of program transformations.

Transformations operate on the abstract syntax trees of programs. Abstract syntax trees are represented by means of first-order terms.

A program is structured as a collection of modules, which may import each other.

Transformations are defined by means of named rewrite rules. Rules may explicitly invoke rules. Alternatively, rules may be invoked by strategies that define how to combine rules into a more complex transformation using strategy combinators. Context-sensitive transformations can be expressed using dynamic rewrite rules.

Starting with Stratego 2, terms and transformation strategies are (gradually) typed.

Placeholder Convention

In this reference manual we use placeholders to indicate the syntactic structure of language constructs. For example, a rewrite rule has the form

```
$Label :
  $Term -> $Term
```

in which the `$Label` is the name of the rule, the first `$Term` the left-hand side, and the second the right-hand side of the rule. This convention should give an indication of the formal structure of a construct, without going down to the precise details of the syntax definition. As a side effect, the schema also shows the preferred indentation of language constructs where that is

https://www.spoofax.dev/references/stratego/

This paper defines the formal semantics of the full Stratego language, including its scoped dynamic rules feature.

These slides document most features of the base language without dynamic rules and without giving the formal semantics.

If you want to dig deeper

Fundamenta Informaticae 69 (2006) 123–178
IOS Press

Program Transformation with Scoped Dynamic Rewrite Rules

Martin Bravenboer, Arthur van Dam, Karina Olmos and Eelco Visser*
Department of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht
The Netherlands
visser@acm.org

Abstract. The applicability of term rewriting to program transformation is limited by the lack of control over rule application and by the context-free nature of rewrite rules. The first problem is addressed by languages supporting user-definable rewriting strategies. The second problem is addressed by the extension of rewriting strategies with scoped dynamic rewrite rules. Dynamic rules are defined at run-time and can access variables available from their definition context. Rules defined within a rule scope are automatically retracted at the end of that scope. In this paper, we explore the design space of dynamic rules, and their application to transformation problems. The technique is formally defined by extending the operational semantics underlying the program transformation language Stratego, and illustrated by means of several program transformations in Stratego, including constant propagation, bound variable renaming, dead code elimination, function inlining, and function specialization.

1. Introduction

Program transformation is the mechanical manipulation of a program in order to improve it relative to some cost function C such that $C(P) > C(tr(P))$, i.e. the cost decreases as a result of applying the transformation [30, 29, 11]. The cost of a program can be measured in different dimensions such as performance, memory usage, understandability, flexibility, maintainability, portability, correctness, or satisfaction of requirements. Related to these goals, program transformations are applied in different settings; e.g. compiler optimizations improve performance [24] and refactoring tools aim at improving understandability [28, 14]. While transformations can be achieved by manual manipulation of programs, in general, the aim of program transformation is to increase programmer productivity by *automating*

*Address for correspondence: Department of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

This paper introduces a gradual type system for Stratego, which is available as Stratego2 in Spoofax3

https://doi.org/10.1145/3426425.3426928

Gradually Typing Strategies

Jeff Smits
Delft University of Technology
The Netherlands
j.smits-1@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

The Stratego language supports program transformation by means of term rewriting with programmable rewriting strategies. Stratego's traversal primitives support concise definition of generic tree traversals. Stratego is a dynamically typed language because its features cannot be captured fully by a static type system. While dynamic typing makes for a flexible programming model, it also leads to unintended type errors, code that is harder to maintain, and missed opportunities for optimization.

In this paper, we introduce a gradual type system for Stratego that combines the flexibility of dynamically typed generic programming, where needed, with the safety of statically declared and enforced types, where possible. To make sure that statically typed code cannot go wrong, all access to statically typed code from dynamically typed code is protected by dynamic type checks (casts). The type system is backwards compatible such that types can be introduced incrementally to existing Stratego programs. We formally define a type system for Core Gradual Stratego, discuss its implementation in a new type checker for Stratego, and present an evaluation of its impact on Stratego programs.

CCS Concepts: • **Software and its engineering** → **Semantics**; *Polymorphism*; *Extensible languages.*

Keywords: gradual types, strategy, generic programming, type preserving

ACM Reference Format:
Jeff Smits and Eelco Visser. 2020. Gradually Typing Strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3426425.3426928

1 Introduction

The Stratego language supports program transformation by means of term rewriting with programmable rewriting strategies [30]. Stratego's traversal primitives support concise definition of generic tree traversals. For example, the definition of `bottomup(s)` in Figure 4 defines in one line a generic bottom-up traversal that can be instantiated with a selection of rewrite rules to be applied in a particular transformation, without needing to define a traversal for each constructor in the abstract syntax. Stratego is used in the Stratego/XT program transformation tool suite [2] and the Spoofax language workbench [11] and used in production in research, education, and industry [6, 12].

Stratego is a dynamically typed language, because its language features cannot be captured fully by a static type system. While dynamic typing makes for a flexible programming model, it also exposes Stratego programmers to unintended type errors. Static typing of strategies has been considered before by Lämmel and Visser [16], Lämmel [14], and others. Lämmel and Jones [15] adopted Stratego's strategic programming in the SYB Haskell design pattern. These efforts focus on the statically typable fragment of strategies, making them unsuitable, as is, as a type system for Stratego. Furthermore, there is a considerable base of existing Stratego code, and having to convert that, at once, to statically typed code would preclude adoption of a type system.

In this paper, we introduce a gradual type system for Stratego that combines the flexibility of dynamically typed generic programming, where needed, with the safety of statically declared and enforced types, where possible. We integrate ideas for statically typing strategies by Lämmel [14] with ideas from the gradual typing literature [24, 25]. In particular, we extend conventional static types with the special type for type preserving transformations [14]. And we introduce a dynamic type in the tradition of gradual type systems to account for, as yet, untyped code. At the interface of statically and dynamically typed code, the type checker inserts dynamic type checks (through casts and proxies) to guarantee the assumptions of static code. This ensures that the type system is backwards compatible such that existing code can pass the type checker as is, and such that types can be introduced incrementally to existing code. At the intersection of typed strategies and gradual types, we find an interesting dynamic types for strategies. For example, the type unifying strategies of Lämmel [14] do not need a special type, but can be modeled with a dynamic input type and a specific result type.

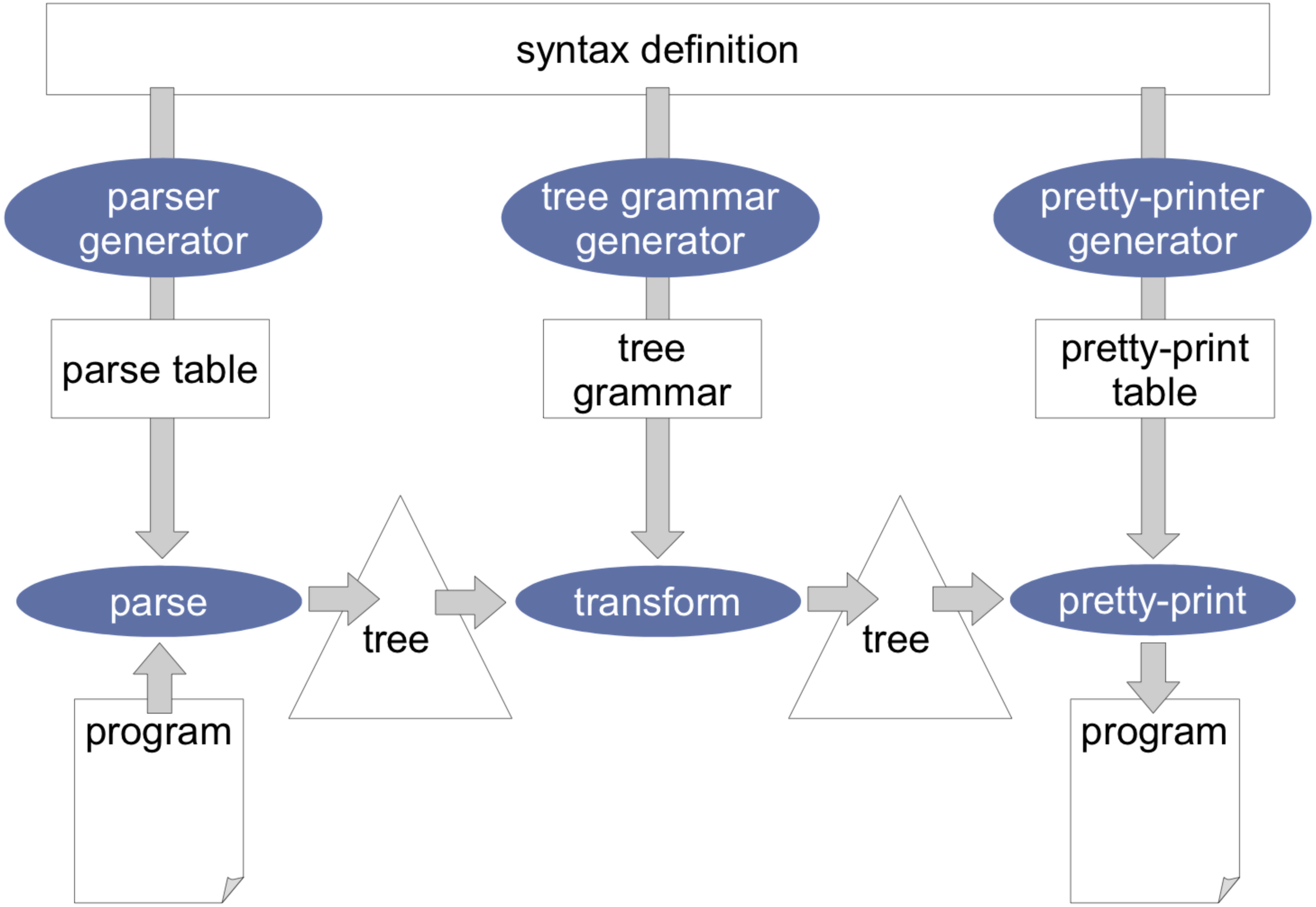
SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11.
https://doi.org/10.1145/3426425.3426928

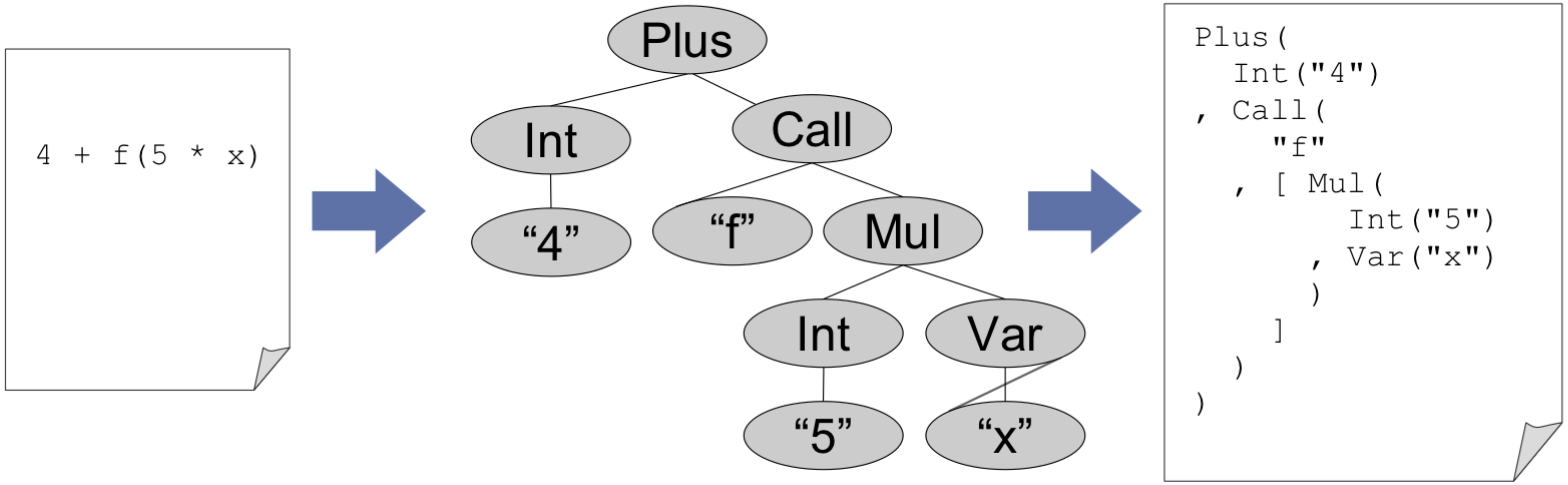
Transformation Architecture

This presentation uses the Stratego Shell for explaining the behavior of Stratego programs. The Stratego Shell is currently not supported by Spoofax

Architecture of Stratego/XT



Programs as Terms



Trees are represented as terms in the ATerm format

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

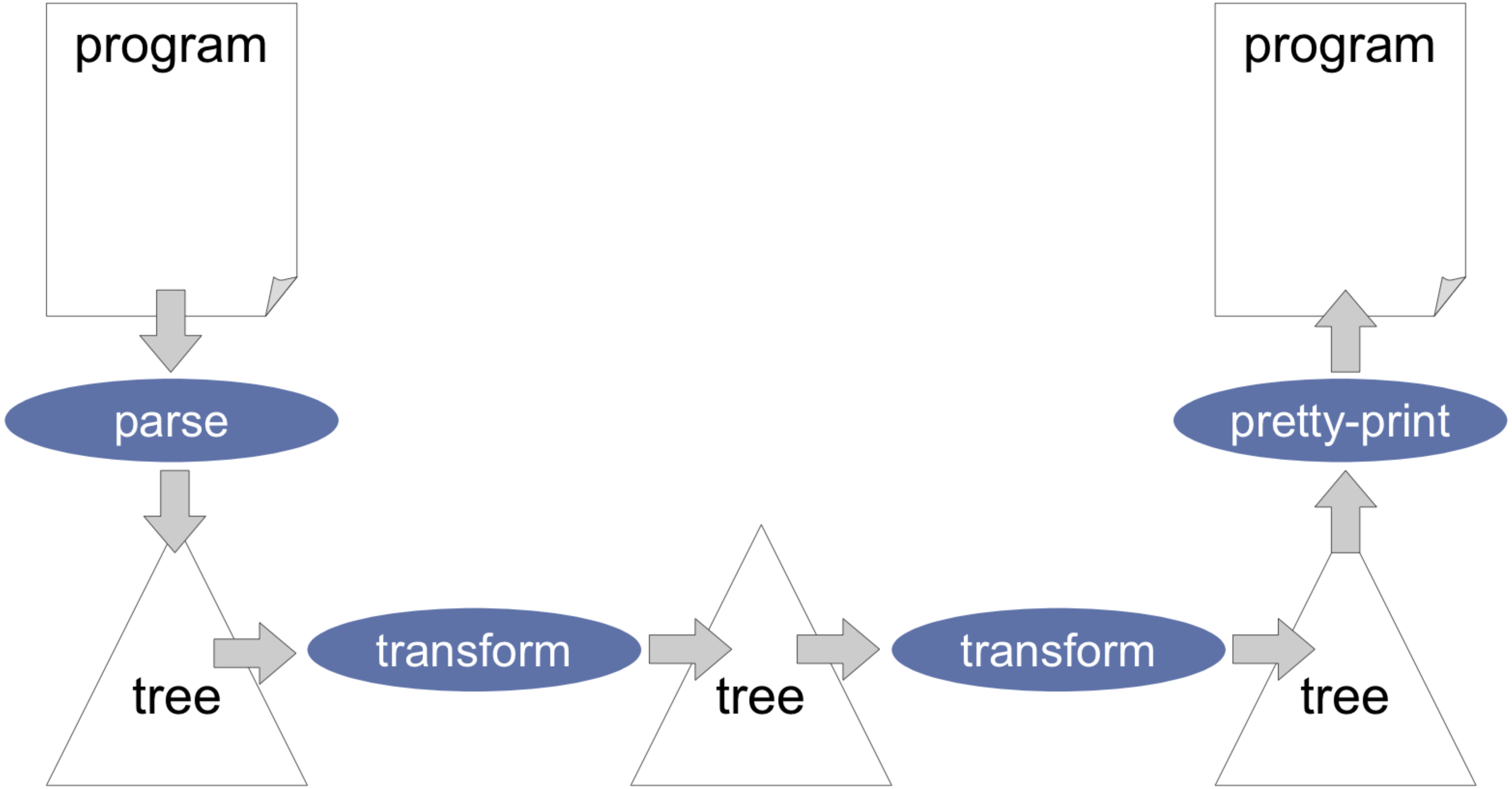
ATerm Format

Application	`Void()`, `Call(`t`, `t`)`
List	`[]`, `[`t`, `t`, `t`]`
Tuple	`(`t`, `t`)`, `(`t`, `t`, `t`)`
Integer	`25`
Real	`38.87`
String	`"Hello world"`
Annotated term	t`{`t`, `t`, `t`}`

- Exchange of structured data
- Efficiency through maximal sharing
- Binary encoding

Structured Data: comparable to XML
Stratego: internal is external representation

How to Realize Program Transformations?



Term Rewriting

Conventional Term Rewriting

- Rewrite system = set of rewrite rules
- Redex = reducible expression
- Normalization = exhaustive application of rules to term
- (Stop when no more redices found)
- Strategy = algorithm used to search for redices
- Strategy given by engine

Strategic Term Rewriting

- Select rules to use in a specific transformation
- Select strategy to apply
- Define your own strategy if necessary
- Combine strategies

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

<current term>

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>
stratego> <strategy expression>
<transformed term>
```

Transformation Strategies

A transformation strategy

- transforms the **current term** into a new term or **fails**
- may bind term variables
- may have side-effects (I/O, call other process)
- is composed from a few **basic operations and combinators**

Stratego Shell: An Interactive Interpreter for Stratego

```
<current term>
stratego> <strategy expression>
<transformed term>
stratego> <strategy expression>
command failed
```

Terms

Parsing: From Text to Terms

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

```
Let(
  [ FunDec(
      "fact"
    , [FArg("n", Tp(Tid("int")))]
    , Tp(Tid("int"))
    , If(
        Lt(Var("n"), Int("1"))
      , Int("1")
      , Seq(
          [ Times(
              Var("n")
            , Call(
                Var("fact")
              , [Minus(Var("n"), Int("1"))]
              )
            )
          ]
        )
      )
    )
  ]
, [Call(Var("fact"), [Int("10")])]
)
```

Syntax of Terms

```
module Terms

sorts Cons Term

lexical syntax

  Cons = [a-zA-Z][a-zA-Z0-9]*

context-free syntax

  Term.App = <<Cons>(<{Term ","}*>)>
```

```
Zero()

Succ(Zero())

Cons(A(), Cons(B(), Nil()))
```

Syntax of Terms

```
module Terms

sorts Cons Term

lexical syntax

   Cons = [a-zA-Z][a-zA-Z0-9]*

context-free syntax

   Term.App   = <<Cons>(<{Term ","}*>)>
   Term.List  = <[<{Term ","}*>]>
   Term.Tuple = <(<{Term ","}*>)>
```

```
Zero()

Succ(Zero())

[A(), B()]
```

Syntax of Terms

```
module ATerms

sorts Cons Term

lexical syntax
   Cons       = [a-zA-Z][a-zA-Z0-9]*
   Cons       = String
   Int        = [0-9]+
   String     = "\"" StringChar* "\""
   StringChar = ~[\"\\n]
   StringChar = "\\" [\"\\]

context-free syntax
   Term.Str   = <<String>>
   Term.Int   = <<Int>>
   Term.App   = <<Cons>(<{Term ","}*>)>
   Term.List  = <[<{Term ","}*>]>
   Term.Tuple = <(<{Term ","}*>)>
```

```
0

1

[A(), B()]

Var(“x\\”)

Let(
  [ Decl(“x”, IntT()),
    Decl(“y”, BoolT())
  ]
  , Eq(Var(“x”), Int(0))
)
```

Syntax of A(nnotated) Terms

```
module ATerms

sorts Cons Term

lexical syntax
   Cons = [a-zA-Z][a-zA-Z0-9]*
   // more lexical syntax omitted

context-free syntax

   Term.Anno      = <<PreTerm>{<{Term “,"}*>}>
   Term           = <<PreTerm>>

   PreTerm.Str    = <<String>>
   PreTerm.Int    = <<Int>>
   PreTerm.App    = <<Cons>(<{Term ","}*>)>
   PreTerm.List   = <[<{Term ","}*>]>
   PreTerm.Tuple  = <(<{Term ","}*>)>
```

```
Var(“x”){Type(IntT())}
```

Signatures

Signatures

```
context-free syntax

  Exp.Uminus  = [- [Exp]]
  Exp.Power   = [[Exp] ** [Exp]]
  Exp.Times   = [[Exp] *  [Exp]]
  Exp.Divide  = [[Exp] /  [Exp]]
  Exp.Plus    = [[Exp] +  [Exp]]
  Exp.Minus   = [[Exp] -  [Exp]]
  Exp.CPlus   = [[Exp] +i [Exp]]
  Exp.CMinus  = [[Exp] -i [Exp]]
  Exp.Eq      = [[Exp] =  [Exp]]
  Exp.Neq     = [[Exp] <> [Exp]]
  Exp.Gt      = [[Exp] >  [Exp]]
  Exp.Lt      = [[Exp] <  [Exp]]
  Exp.Geq     = [[Exp] >= [Exp]]
  Exp.Leq     = [[Exp] <= [Exp]]
  Exp.True    = <true>
  Exp.False   = <false>
  Exp.And     = [[Exp] & [Exp]]
  Exp.Or      = [[Exp] | [Exp]]
```

Signature is automatically generated from SDF3 productions

Signature declares argument and return types of term constructors

```
signature
  constructors
    Uminus    : Exp -> Exp
    Power     : Exp * Exp -> Exp
    Times     : Exp * Exp -> Exp
    Divide    : Exp * Exp -> Exp
    Plus      : Exp * Exp -> Exp
    Minus     : Exp * Exp -> Exp
    CPlus     : Exp * Exp -> Exp
    CMinus    : Exp * Exp -> Exp
    Eq        : Exp * Exp -> Exp
    Neq       : Exp * Exp -> Exp
    Gt        : Exp * Exp -> Exp
    Lt        : Exp * Exp -> Exp
    Geq       : Exp * Exp -> Exp
    Leq       : Exp * Exp -> Exp
    True      : Exp
    False     : Exp
    And       : Exp * Exp -> Exp
    Or        : Exp * Exp -> Exp
```

Stratego compiler only checks *arity* of constructor applications

Rewrite Rules

Desugaring

```
if x then
   printint(x)
```

```
if x then
   printint(x)
else
   ()
```

```
IfThen(
   Var("x")
, Call(
     "printint"
   , [Var("x")]
   )
)
```

e1

e2

```
IfThenElse(
   Var("x")
, Call(
     "printint"
   , [Var("x")]
   )
, NoVal()
)
```

pattern matching

pattern instantiation

```
desugar: IfThen(e1, e2) → IfThenElse(e1, e2, NoVal())
```

Lists of ElseIfs

```
signature
  constructors
    If : Exp * Exp * List(ElseIf) → Exp

    ElseIf : Exp * Exp → ElseIf

    IfThen : Exp * Exp * Exp → Exp
```

```
If(c, e1, [
  ElseIf(c2, e2),
  ElseIf(c3, e3),
  …
])
```

```
Desugar :
  If(c, e, []) → IfThen(c, e, NoVal())

Desugar :
  If(c, e, [ElseIf(c2, e2) | es]) → IfThen(c, e, If(c2, e2, es)
```

More Desugaring

```
signature
  constructors
    PLUS:    BinOp
    MINUS:   BinOp
    MUL:  BinOp
    DIV:  BinOp

    EQ:   RelOp
    NE:   RelOp
    LE:   RelOp
    LT:   RelOp

    Bop: BinOp * Expr * Expr → Expr
    Rop: RelOp * Expr * Expr → Expr
```

```
desugar: Uminus(e)      → Bop(MINUS(), Int("0"), e)

desugar: Plus(e1, e2)   → Bop(PLUS(), e1, e2)
desugar: Minus(e1, e2)  → Bop(MINUS(), e1, e2)
desugar: Times(e1, e2)  → Bop(MUL(), e1, e2)
desugar: Divide(e1, e2) → Bop(DIV(), e1, e2)

desugar: Eq(e1, e2)  → Rop(EQ(), e1, e2)
desugar: Neq(e1, e2) → Rop(NE(), e1, e2)
desugar: Leq(e1, e2) → Rop(LE(), e1, e2)
desugar: Lt(e1, e2)  → Rop(LT(), e1, e2)
desugar: Geq(e1, e2) → Rop(LE(), e2, e1)
desugar: Gt(e1, e2)  → Rop(LT(), e2, e1)

desugar: And(e1, e2) → IfThenElse(e1, e2, Int("0"))
desugar: Or(e1, e2)  → IfThenElse(e1, Int("1"), e2)
```

Constant Folding

```
x := 21 + 21 + x
```

```
x := 42 + x
```

```
Assign(
  Var("x")
, Plus(
    Plus(
      Int("21")
    , Int("21")
    )
  , Var("x")
  )
)
```

```
Assign(
  Var("x")
, Plus(
    Int("42")
  , Var("x")
  )
)
```

```
eval: Plus(Int(i1), Int(i2)) → Int(i3)
      where <addS> (i1, i2) ⇒ i3
```

More Constant Folding

```
eval: Bop(PLUS(), Int(i1), Int(i2))  → Int(<addS> (i1, i2))

eval: Bop(MINUS(), Int(i1), Int(i2)) → Int(<subtS> (i1, i2))

eval: Bop(MUL(), Int(i1), Int(i2))   → Int(<mulS> (i1, i2))

eval: Bop(DIV(), Int(i1), Int(i2))   → Int(<divS> (i1, i2))

eval: Rop(EQ(), Int(i), Int(i))    → Int("1")
eval: Rop(EQ(), Int(i1), Int(i2)) → Int("0") where not( <eq> (i1, i2) )

eval: Rop(NE(), Int(i), Int(i))    → Int("0")
eval: Rop(NE(), Int(i1), Int(i2)) → Int("1") where not( <eq> (i1, i2) )

eval: Rop(LT(), Int(i1), Int(i2)) → Int("1") where <ltS> (i1, i2)
eval: Rop(LT(), Int(i1), Int(i2)) → Int("0") where not( <ltS> (i1, i2) )

eval: Rop(LE(), Int(i1), Int(i2)) → Int("1") where <leqS> (i1, i2)
eval: Rop(LE(), Int(i1), Int(i2)) → Int("0") where not( <leqS> (i1, i2))
```

Application: Formatting

Tiger: Parenthesize

```
module pp/Tiger-parenthesize

imports
  strategolib
  signatures/-

strategies

  parenthesize-Tiger =
    innermost(TigerParenthesize)
```

```
context-free priorities

  Exp.And
  > Exp.Or
  > Exp.Array
  > Exp.Assign
  > …
```

```
rules
  TigerParenthesize :
    Or(t_0, t_1) → Or(t_0, Parenthetical(t_1))
    where <(?For(_, _, _, _)
            + ?While(_, _)
            + ?IfThen(_, _)
            + ?If(_, _, _)
            + ?Assign(_, _)
            + ?Array(_, _, _)
            + ?Or(_, _)
            + fail)> t_1

  TigerParenthesize :
    And(t_0, t_1) → And(Parenthetical(t_0), t_1)
    where <(?For(_, _, _, _)
            + ?While(_, _)
            + ?IfThen(_, _)
            + ?If(_, _, _)
            + ?Assign(_, _)
            + ?Array(_, _, _)
            + ?Or(_, _)
            + fail)> t_0
```

pp/Tiger-parenthesize.str

Tiger: Pretty-Print Rules

```
context-free syntax

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >
```

syntax/Control-Flow.sdf3

```
rules
  prettyprint-Tiger-Exp :
    If(t1__, t2__, t3__) → [ H(
                                [SOpt(HS(), "0")]
                              , [ S("if ")
                                , t1__'
                                , S(" then")
                                ]
                              )
                            , t2__'
                            , H(
                                [SOpt(HS(), "0")]
                              , [S("else")]
                              )
                            , t3__'
                            ]
    with t1__' := <pp-one-Z(prettyprint-Tiger-Exp)
                   <+ pp-one-Z(prettyprint-completion-aux)> t1__
    with t2__' := <pp-indent(|"2")> [
      <pp-one-Z(prettyprint-Tiger-Exp)
      <+ pp-one-Z(prettyprint-completion-aux)> t2__ ]
    with t3__' := <pp-indent(|"2")> [
      <pp-one-Z(prettyprint-Tiger-Exp)
      <+ pp-one-Z(prettyprint-completion-aux)> t3__ ]
```

pp/Control-Flow-pp.str

Application: Desugaring

Tiger: Desugaring

```
function printboard() = (
  for i := 0 to N-1 do (
    for j := 0 to N-1 do
      print(if col[i]=j then " O" else " .");
    print("\n")
  );
  print("\n")
)
```

```
function printboard() = (
  let
    var i : int := 0
  in
    while i < N - 1 do ( (
      let
        var j : int := 0
      in
        while j < N - 1 do (
          print(if col[i] = j then
                  " O"
                else
                  " .");
          j := j + 1
        )
      end;
      print("\n")
    );
    i := i + 1
  )
  end;
  print("\n")
)
```

Expressing for in terms of while++

Tiger: Desugaring Transformation

```
module desugar
imports signatures/Tiger-sig
imports …
strategies
  desugar-all = topdown(try(desugar))
rules
  desugar :
    For(
      Var(i)
    , e1
    , e2
    , e_body
    ) ->
    Let(
      [VarDec(i, Tid("int"), e1)]
    , [ While(
          Lt(Var(i), e2)
        , Seq(
            [ e_body
            , Assign(Var(i), Plus(Var(i), Int("1")))
            ]
          )
        )
      ]
    )
```

Application: Completion

Tiger: Completion Rules

```
context-free syntax

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >
```

syntax/Control-Flow.sdf3

```
rules

  suggest-completions(|completions):
    Exp-Plhdr() -> <add-completions(
                     | ( "If"
                       , If(
                           <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()
                         , <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()
                         , <try(inline-completions(|Exp-Plhdr()))> Exp-Plhdr()
                         )
                       )
                     )
                     ; fail> completions
```

completion/Control-Flow-cp.str

Combining Rewrite Rules with Strategies

Naming and Composing Strategies

Reuse of transformation requires definitions

1. Naming strategy expressions
2. Named rewrite rules
3. Reusing rewrite rules through modules

Simple strategy definition and call

- Syntax: *f* = *s*
- Name strategy expression *s*
- Syntax: *f*
- Invoke (call) named strategy *f*

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs = {e1,e2:(Plus(e1,e2) -> Plus(e2,e1))}
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Named Rewrite Rules

Named rewrite rules (sugar)

- Syntax: f `:` $p1$ `->` p_2 `where` s
- Name rewrite rule $p1$ `->` p_2 `where` s
- Equivalent to: f `= {`x_1`,...,`x_n`: (`p_1 `->` p_2 `where` s`)}`
  (with x_1,...,x_n the variables in p_1, p_2, and s)

```
Plus(Var("a"),Int("3"))
stratego> SwapArgs : Plus(e1,e2) -> Plus(e2,e1)
stratego> SwapArgs
Plus(Int("3"),Var("a"))
```

Example: Inverting If Not Equal

```
if(x != y)
  doSomething();
else
  doSomethingElse();
```

$\Rightarrow$

```
if(x == y)
  doSomethingElse();
else
  doSomething();
```

```
InvertIfNot :
  If(NotEq(e1, e2), stm1, stm2) ->
  If(Eq(e1, e2), stm2, stm1)
```

Modules with Reusable Transformation Rules

```
module Simplification-Rules
rules
  PlusAssoc :
    Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))

  EvalIf :
    If(Lit(Bool(True())), stm1, stm2) -> stm1

  EvalIf :
    If(Lit(Bool(False())), stm1, stm2) -> stm2

  IntroduceBraces :
    If(e, stm) -> If(e, Block([stm]))
    where <not(?Block(_))> stm
```

```
stratego> import Simplification-Rules
```

Composing Strategies

Rules define one-step transformations

Program transformations require many one-step transformations and selection of rules

1. Choice
2. Identity, Failure, and Negation
3. Parameterized and Recursive Definitions

Composing Strategies

Deterministic choice (left choice)

- Syntax: s_1 <+ s_2
- First apply s_1, if that fails apply s_2
- Note: local backtracking

```
PlusAssoc :
  Plus(Plus(e1, e2), e3) -> Plus(e1, Plus(e2, e3))
EvalPlus :
  Plus(Int(i),Int(j)) -> Int(k) where <addS>(i,j) => k
```

```
Plus(Int("14"),Int("3"))
stratego> PlusAssoc
command failed
stratego> PlusAssoc <+ EvalPlus
Int("17")
```

Composing Strategies

Guarded choice

- Syntax: s_1 `<` s_2 `+` s_3
- First apply s_1 if that succeeds apply s_2 to the result else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- s_1 `<+` s_2 always backtracks to s_2 if s_1 fails
- `(`s_1`;` s_2`)` `<+ s3` $\neq$ s_1 `<` s_2 `+` s_3
- commit to branch if test succeeds, even if that branch fails

```
  test1 < transf1
+ test2 < transf2
+ transf3
```

Composing Strategies

Guarded choice

- Syntax: s_1 `<` s_2 `+` s_3
- First apply s_1 if that succeeds apply s_2 to the result else apply s_3 to the original term
- Do not backtrack to s_3 if s_2 fails!

Motivation

- s_1 `<+` s_2 always backtracks to s_2 if s_1 fails
- `(`s_1`;` s_2`) <+ s3` $\not\equiv$ s_1 `<` s_2 `+` s_3
- commit to branch if test succeeds, even if that branch fails

```
  test1 < transf1
+ test2 < transf2
+ transf3
```

If then else (sugar)

- Syntax: `if` s_1 `then` s_2 `else` s_3 `end`
- Equivalent to: `where(`s_1`)` `<` s_2 `+` s_3

Composing Strategies

Identity

- Syntax: `id`
- Always succeed
- Some laws
  - $\texttt{id} \; ; \; s \equiv s$
  - $s \; ; \; \texttt{id} \equiv s$
  - $\texttt{id} \; \texttt{<+} \; s \equiv \texttt{id}$
  - $s \; \texttt{<+} \; \texttt{id} \not\equiv s$
  - $s_1 \; \texttt{<} \; \texttt{id} \; \texttt{+} \; s_2 \equiv s_1 \; \texttt{<+} \; s_2$

Failure

- Syntax: `fail`
- Always fail
- Some laws
  - $\texttt{fail} \; \texttt{<+} \; s \equiv s$
  - $s \; \texttt{<+} \; \texttt{fail} \equiv s$
  - $\texttt{fail} \; ; \; s \equiv \texttt{fail}$
  - $s \; ; \; \texttt{fail} \not\equiv \texttt{fail}$

Composing Strategies

Identity

- Syntax: `id`
- Always succeed
- Some laws
  - `id ;` $s \equiv s$
  - s `; id` $\equiv s$
  - `id <+` $s \equiv$ `id`
  - s `<+ id` $\not\equiv s$
  - s_1 `< id +` $s_2 \equiv s_1$ `<+` s_2

Failure

- Syntax: `fail`
- Always fail
- Some laws
  - `fail <+` $s \equiv s$
  - s `<+ fail` $\equiv s$
  - `fail ;` $s \equiv$ `fail`
  - s `; fail` $\not\equiv$ `fail`

Negation (sugar)

- Syntax: `not(`s`)`
- Fail if s succeeds, succeed if s fails
- Equivalent to: s `< fail + id`

Parameterizing Strategies

Parameterized and recursive definitions

- Syntax: $f\,(x_1\,,\,.\,.\,.\,,x_n\,|\,y_1\,,\,.\,.\,.\,,y_m)\ =\ s$
- Strategy definition parameterized with strategies $(x_1,\ldots,x_n)$ and terms $(y_1,\ldots,y_m)$
- Note: definitions may be recursive

Parameterizing Strategies

Parameterized and recursive definitions

- Syntax: $f\,(x_1, \ldots, x_n \,|\, y_1, \ldots, y_m)\ =\ s$
- Strategy definition parameterized with strategies $(x_1,\ldots,x_n)$ and terms $(y_1,\ldots,y_m)$
- Note: definitions may be recursive

```
try(s)          = s <+ id

repeat(s)       = try(s; repeat(s))

while(c, s)     = if c then s; while(c,s) end

do-while(s, c) = s; if c then do-while(s, c) end
```

Traversal Strategies

Term Rewriting for Program Transformation

Term Rewriting

- apply set of rewrite rules exhaustively

Advantages

- First-order terms describe abstract syntax
- Rewrite rules express basic transformation rules (operationalizations of the algebraic laws of the language.)
- Rules specified separately from strategy

Limitations

- Rewrite systems for programming languages often non-terminating and/or non-confluent
- In general: do not apply all rules at the same time or apply all rules under all circumstances

Term Rewriting for Program Transformation

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
rules
  DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN    : Not(Not(x))      -> x
  DMA   : Not(And(x, y))   -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

Term Rewriting for Program Transformation

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
rules
  DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN    : Not(Not(x))      -> x
  DMA   : Not(And(x, y))   -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

This is a non-terminating rewrite system

Encoding Control with Recursive Rewrite Rules

Common solution

- Introduce additional constructors that achieve normalization under a restricted set of rules
- Replace a 'pure' rewrite rule

  p_1 `->` p_2

  with a functionalized rewrite rule:

  `f :` p_1 `->` p'_2

  applying f recursively in the right-hand side
- Normalize terms `f(t)` with respect to these rules
- The function now controls where rules are applied

Recursive Rewrite Rules: Disjunctive Normal Form

```
dnf  : True        -> True
dnf  : False       -> False
dnf  : Atom(x)     -> Atom(x)
dnf  : Not(x)      -> <not>(<dnf>x)
dnf  : And(x,y)    -> <and>(<dnf>x,<dnf>y)
dnf  : Or(x,y)     -> Or(<dnf>x,<dnf>y)

and1 : (Or(x,y),z) -> Or(<and>(x,z),<and>(y,z))
and2 : (z,Or(x,y)) -> Or(<and>(z,x),<and>(z,y))
and3 : (x,y)       -> And(x,y)
and  = and1 <+ and2 <+ and3

not1 : Not(x)      -> x
not2 : And(x,y)    -> Or(<not>(x),<not>(y))
not3 : Or(x,y)     -> <and>(<not>(x),<not>(y))
not4 : x           -> Not(x)
not  = not1 <+ not2 <+ not3 <+ not4
```

Analysis

Functional encoding has two main problems

Overhead due to explicit specification of *traversal*

- A traversal rule needs to be defined for each constructor in the signature and for each transformation.

Separation of rules and strategy is lost

- Rules and strategy are completely *intertwined*
- Intertwining makes it more difficult to *understand* the transformation
- Intertwining makes it impossible to *reuse* the rules in a different transformation.

Analysis

Language Complexity

Traversal overhead and reuse of rules is important, considering the complexity of real programming languages:

language	# constructors
Tiger	65
C	140
Java 5	325
COBOL	300–1200

Requirements

- Control over application of rules
- No traversal overhead
- Separation of rules and strategies

Programmable Rewriting Strategies

- Select rules to be applied in specific transformation
- Select strategy to control their application
- Define your own strategy if necessary
- Combine strategie

Idioms

- Cascading transformations
- One-pass traversal
- Staged transformation
- Local transformation

Strategic Idioms

Rules for rewriting proposition formulae

```
signature
  sorts Prop
  constructors
    False : Prop
    True  : Prop
    Atom  : String -> Prop
    Not   : Prop -> Prop
    And   : Prop * Prop -> Prop
    Or    : Prop * Prop -> Prop
rules
  DAOL  : And(Or(x, y), z) -> Or(And(x, z), And(y, z))
  DAOR  : And(z, Or(x, y)) -> Or(And(z, x), And(z, y))
  DOAL  : Or(And(x, y), z) -> And(Or(x, z), Or(y, z))
  DOAR  : Or(z, And(x, y)) -> And(Or(z, x), Or(z, y))
  DN    : Not(Not(x))      -> x
  DMA   : Not(And(x, y))   -> Or(Not(x), Not(y))
  DMO   : Not(Or(x, y))    -> And(Not(x), Not(y))
```

Strategic Idioms: Cascading Transformation

Cascading Transformations

- Apply small, independent transformations in combination
- Accumulative effect of small rewrites

```
simplify = innermost(R1 <+ ... <+ Rn)
```

disjunctive normal form

```
dnf = innermost(DAOL <+ DAOR <+ DN <+ DMA <+ DMO)
```

conjunctive normal form

```
cnf = innermost(DOAL <+ DOAR <+ DN <+ DMA <+ DMO)
```

Strategic Idioms: One-Pass Traversal

One-pass Traversal

- Apply rules in a single traversal over a program tree

```
simplify1  = downup(repeat(R1 <+ ... <+ Rn))
simplify2  = bottomup(repeat(R1 <+ ... <+ Rn))
```

constant folding

```
Eval : And(True, e) -> e
Eval : And(False, e) -> False
Eval : ...

eval = bottomup(try(Eval))
```

Strategic Idioms: One-Pass Traversal

Example: Desugarings

```
DefN  : Not(x)      -> Impl(x, False)
DefI  : Impl(x, y) -> Or(Not(x), y)
DefE  : Eq(x, y)   -> And(Impl(x, y), Impl(y, x))
DefO1 : Or(x, y)   -> Impl(Not(x), y)
DefO2 : Or(x, y)   -> Not(And(Not(x), Not(y)))
DefA1 : And(x, y)  -> Not(Or(Not(x), Not(y)))
DefA2 : And(x, y)  -> Not(Impl(x, Not(y)))
IDefI : Or(Not(x), y) -> Impl(x, y)
IDefE : And(Impl(x, y), Impl(y, x)) -> Eq(x, y)

desugar = topdown(try(DefI <+ DefE))

impl-nf  = topdown(repeat(DefN <+ DefA2 <+ DefO1 <+ DefE))
```

Strategic Idioms: Staged Transformation

Staged Transformation

- Transformations are not applied to a subject term all at once, but rather in stages
- In each stage, only rules from some particular subset of the entire set of available rules are applied.

```
simplify =
  innermost(A1 <+ ... <+ Ak)
  ; innermost(B1 <+ ... <+ Bl)
  ; ...
  ; innermost(C1 <+ ... <+ Cm)
```

Strategic Idioms: Local Transformation

Local transformation

- Apply rules only to selected parts of the subject program

```
transformation =
  alltd(
    trigger-transformation
    ; innermost(A1 <+ ... <+ An)
  )
```