– Is it easy for humans to write code using this syntax?
– It depends on how you define "human".
– Magne Haveraaen asking Valentin David

3 Strategic Term Rewriting

This chapter recalls some basic elements of term rewriting theory and some supporting parts of universal algebra. It proceeds by discussing a programming paradigm called strategic programming which supports the separation of data traversal concerns from data processing logic – allowing each part to be implemented and reused separately – and how strategic programming, in the form of strategic term rewriting, helps expressing reusable term rewriting systems. The chapter describes a calculus for strategic term rewriting called System S calculus. This calculus provides the basic abstractions of tree transformations and term rewriting: matching and building terms, term traversal, combining computations, and failure handling. The strategic term rewriting language Stratego, that implements the System S calculus, is described.

3.1 Term Rewriting

The field of term rewriting studies methods for replacing subterms of terms with other terms. Techniques from this field are attractive for program transformation and analysis because every computer program can be represented as a term. The (abstract) syntax tree of a program can be directly treated as a term. The mathematical machinery of term rewriting may be brought to bear on analysis and transformation problems.

Term rewriting theory [Ter03] makes use of basic notions known from universal algebra [Coh81], a field of mathematics which seeks to describe any mathematical object by its operations. Objects and operations are described formally using signatures. In term rewriting, one talks of sorts and constructors in lieu of objects (types) and operations.

3.1.1 Algebraic Signatures and Language Signatures

In both universal algebra and term writing, terms are defined over signatures. Signatures may be considered analogous to the context-free grammars used to describe the structure of text. Both context-free grammars and signatures describe properties of (potentially) recursively defined tree structures. A standard definition of an algebraic signature is given below.

Definition 1 Algebraic Signature.

An algebraic signature Σ *is a pair* (S, Ω) *of sets, where* S *is a set of* sorts *and* Ω *a set of* operations. *Each operation is a* $(k+2)$*-tuple,* $k \geq 0$*, on the form*

$o : s_1 \times \ldots \times s_k \rightarrow s$

where $s_1, \ldots, s_k, s \in S$*,* o *is the* operation name *and* $s_1 \times \ldots \times s_k \rightarrow s$ *its* arity*. The sorts* $s_1, \ldots, s_k$ *are* argument sorts*, and* s *the* target sort*. When* $k = 0$*,* $o :\rightarrow s$ *is a* constant symbol*, or just* constant.

The following example of an algebraic signature declares the four basic arithmetic operations.

```
signature Arithmetic
sorts Int
ops
  plus : Int × Int → Int
  minus : Int × Int → Int
  divide : Int × Int → Int
  times : Int × Int → Int
```

In this dissertation, algebraic signatures will be used to describe abstract data types. For example, the above signature partially describes the data type *Int* and some of its operations (*plus*, *minus*, *divide* and *times*). All operations (and terms involving operations) will be written in *italics* in the main text.

In several traditions of program transformation based on term rewriting there is second role for signatures: they may be used to declare the abstract syntax of programming languages, akin to document type definitions commonly found for markup languages like XML [BPSM+] and SGML [sgm86]. Signatures used in this capacity are referred to as *language signatures* in this dissertation. They have some minor and subtle differences compared with the algebraic signatures.

The language signatures described here follow the tradition introduced by the Stratego rewriting language. Operations are referred to as *constructors*. In the main text, constructors (and terms involving constructors) will be written in `MixedCase`. Constructors must always start with an uppercase letter. A more important difference between the two uses of signatures is that in signatures describing languages, the argument sorts of constructors follow the abstract grammar of the subject language they define. Consider the signature definition for a minimal language L that supports variables, assignment and addition operations on floating point and integer numbers:

```
signature L
sorts Var Exp Stmt String
```

```
3 constructors
4   Var : String → Var
5         : Var → Exp
6   Int : String → Exp
7   Float : String → Exp
8   Plus : Exp × Exp → Exp
9   Assign : Var × Exp → Stmt
```

Line 4 declares that variable terms are of sort `Var`. Line 5 is an injection which declares that every term of sort `Var` is also a term of sort `Exp`, i.e. `Var` is a subsort of `Exp`. The `Int` and `Float` constructors describe literals of integers and floats, respectively. In the abstract syntax, a `Plus` term is constructed from two terms of sort `Exp`. Assignments are statements (of sort `Stmt`) which assign the result of expressions to variables.

3.1.2 Patterns and Terms

Universal algebra defines the notion of terms over signatures, a traditional definition of which is given in Definition 3. These terms may contain variables.

Definition 2 (Variables).
Given a signature $\Sigma = (S, \Omega)$ with an associated family $V = (V_s)_{s \in S}$ of disjoint infinite sets, an element $x \in V_s, s \in S$ is a variable *x of sort s.*

Algebraic terms may be recursively constructed from variables and the application of operations to the result of operations or to variables.

Definition 3 (Algebraic Terms).
Given a signature $\Sigma = (S, \Omega)$ and an associated set of variables X, the set of (algebraic) terms for Σ, $(T_{\Sigma(X),s})_{s \in S}$ are defined by simultaneous induction:

1. $X_s \subseteq T_{\Sigma(X),s}$
2. *if* $o :\rightarrow s \in \Omega$, *then* $o \in T_{\Sigma(X),s}$
3. *if* $o : s_1 \times \ldots \times s_k \rightarrow s \in \Omega$, $k \geq 0$ *and if* $t_i \in T_{\Sigma(X),s_i}$ *for* $1 \leq i \leq k$, *then* $o(t_1, \ldots, t_k) \in T_{\Sigma(X),s}$.

An element in $T_{\Sigma(X),s}$ is called a $\Sigma(X)$-term of sort s, or just a term. $\mathrm{Var}(t)$ *denotes all variables occurring in the $\Sigma(X)$ term t. If* $\mathrm{Var}(t) = \emptyset$, *$t$ is called a* ground term.

Every valid algebraic term for a given signature must respect the sorts of the signature, i.e. the arity of each operation. Algebraic terms may contain variables. The terms for language signatures, and their nomenclature, behave slightly differently from algebraic terms.

p ::=	$c(p,\ldots,p)$	constructor application
|	str	string literal
|	r	real number
|	i	integer number
|	x	variable
c ::=	identifier	constructor name
x ::=	identifier	variable name
|	_	wildcard

Figure 3.1: Syntax definition for Stratego (language) patterns. The number of patterns p in a constructor application must correspond to the numeric arity of the constructor named c. Wildcards are "open holes" in patterns, akin to nameless variables.

The syntax for Stratego language terms is described in Figure 3.1. When language terms, or just terms, are constructed, the language signature is assumed to be single-sorted. Only the numeric arity must be respected, i.e. only the number of arguments, irrespective of the sorts. This is done for practical convenience. Term rewriting approaches, including that of Stratego, use step-wise substitution of subterms when going from one signature to another. It is useful to allow intermediate terms which are not valid according to either the source or the target signature, without having to explicitly declare a "super-signature" which defines all possible constructor combinations.

Another difference between universal algebra and the nomenclature used in strategic rewriting is the meaning of the word "term". Language terms are always ground terms. A language term containing variables will be referred to as a *pattern*, often written p. Variables in patterns always start with lower case letters, e.g. `x`. Consider the example term and pattern:

`Plus(Int("0"),Int("1"))`	`Plus(x,y)`
(term)	*(pattern)*

The kind of term expression – pattern or ground term – is easily recognised from the syntax since all constructors start with an uppercase letter and all variables start with a lowercase letter.

A pattern p may be matched against a term t. This matching is purely syntactical. It succeeds if and only if there exists a valid variable substitution $\sigma(p) \equiv t$. The variables $Var(p)$ of p will be bound to their corresponding subterms in t, e.g:

$$\langle \text{match}\,\texttt{Plus(x,y)}\rangle\ \texttt{Plus(Int("0"),Int("1"))} \Rightarrow \sigma : [x \mapsto \texttt{Int("0")}, y \mapsto \texttt{Int("1")}]$$

Conversely, a pattern p may be instantiated into a term t, by replacing all its variables x with terms:

$$[x \mapsto \texttt{Int("0")}, y \mapsto \texttt{Int("1")}] : \langle \text{build}\, \texttt{Plus(x,y)} \rangle \Rightarrow \texttt{Plus(Int("0"),Int("1"))}$$

Patterns are used in program transformations to check for structural (syntactic) properties and to construct new program fragments. By combining pattern matching and pattern instantiation into one (potentially named) unit, the rewrite rule is obtained.

3.1.3 Rewrite Rules

Rewrite rules are the units of transformation – or the atomic building blocks, if you will – in term rewriting systems. Each rewrite rule describes how one term can be derived from another term in a single step.

Definition 4 (Rewrite Rule).
A rewrite rule $R : p_l \rightarrow p_r$*, with name* R*, left-hand side pattern* p_l*, right-hand side pattern* p_r*, and* $p_l, p_r \in T_{\Sigma(X)}$, reduces *the term* t *to* t' *if there exists a* $\sigma : X \rightarrow T_\Sigma$ *such that* $t = \sigma(p_l)$ *(*p_l *matches* t*) and* $t' = \sigma(p_r)$ *(*p_r *instantiates to* t'*). The term* t *is called the* redex *(reducible expression) and* t' *the* reduct.

In the context of System S and Stratego, the term variables are variables in the Stratego program, and the substitution σ corresponds to a variable environment ε. This is clarified in the next section. A set of rewrite rules R is said to induce a *one-step rewrite relation* on terms, written as follows:

$$t \rightarrow_R t'$$

This says that t reduces to t' with one of the rules in R. Composing these in sequence, i.e. $t_0 \rightarrow_R t_1 \rightarrow_R \ldots$ gives a *reduction sequence* with $\rightarrow_R$, where R is repeatedly applied to the root of a term.

Definition 5 (Conditional Rewrite Rule).
A conditional rewrite rule $R : p_l \rightarrow p_r$ where c*, with* c *being a logical expression in some logic, specifies that* R *is only applicable if, for some* σ*,* p_l *matches* t *with* σ *and* $\sigma(c)$ *holds (evaluates to true).*

3.1.4 Rewriting Strategies

The rewrite sequence, as defined above, repeatedly applies the rules of R to the root of a term, i.e. to the top-level constructor and its subterms. The definition does not describe how rules may be applied to subterms. Nor does it say anything about the order in which the rules in R of are applied for each step – it may be the case that multiple rules are applicable.

Other definitions for rule application exist in term rewriting theory, but for program transformation, a flexible and precise way of programming both the application location (inside a term) and the order of (rule) application is necessary. In this dissertation, the System S calculus is used for this purpose.

3.2 System S – Strategic Term Rewriting

Strategic term rewriting extends basic term rewriting with additional constructs that accurately control the application strategies for sets of rules. These constructs are used to control the order of rule application, traversal over term structures, and how to handle rule application failures.

The System S core calculus is a formalism for strategic term rewriting. It provides the basic abstractions of tree transformations and term rewriting: matching and building terms, term traversal, combining computations and failure handling. It was first introduced by Visser and Benaissa [VBT98, VB98]. The programming language Stratego is directly based on this calculus.

This section contains a slightly modified formulation of the same core calculus which is more in the style of [BvDOV06]. The definitions given herein are only those necessary for later chapters. Compared to the original description, non-deterministic choice, $s_0 + s_1$ and the `test` operator have been dropped. These are now replaced by a guarded choice combinator. The `some(s)` traversal primitive has been eliminated. A syntax of System S is shown in Figure 3.2. For the rest of this section, the word "program" is taken to mean the transformation program. Terms are used to represent subject programs.

In Chapter 5 and Chapter 7, the System S calculus and Stratego is extended with additional constructs that improve the capacity for expressing language independent transformation programs.

Basic Definitions

The operational semantics of System S is specified using the notation described below. The semantics describes the behaviour of strategies. Rewrite rules are encoded as strategies (shown later), but are provided with syntactic sugar to give them their familiar notation.

The domain of strategy applications is the set of terms extended with a special failure value $\uparrow$. The notation $\underline{t}$ is used to indicate terms from this extended domain; the notation t still refers to terms. Consider the following assertion:

$$\Gamma, \varepsilon \vdash \langle s \rangle\, t \Rightarrow \underline{t}'(\Gamma', \varepsilon')$$

s ::=	`id`	identify
|	`fail`	failure
|	$?p$	match term
|	$!p$	build term
|	$s; s$	sequential composition
|	$s < s + s$	guarded choice
|	`where`(s)	where
|	$\{x, \ldots, x \colon s\}$	new variable scope
|	`one`(s) | `all`(s)	generic traversal operators
|	$f(f, \ldots, f \vert p, \ldots, p)$	strategy invocation
x ::=	identifier	variable names
f ::=	identifier	strategy names
c ::=	identifier	constructor names

Figure 3.2: Syntax for System S. The definition of term patterns p was given in Figure 3.1. The semantics of strategy invocation is defined in [BvDOV06].

It states that the strategy s applied to term t in context of the system state Γ (used to model dynamic rules) and variable environment ε evaluates to the term t' in a new system state Γ' and a new environment ε'. The variable environment takes on the role of the σ substition previously described for rewrite rules.

Strategies may fail. This is noted with the following assertion:

$$\Gamma, \varepsilon \vdash \langle s \rangle\, t \Rightarrow\uparrow (\Gamma', \varepsilon')$$

Changes to state and variable bindings are preserved in the case of failure.

Variables A variable environment ε is a finite ordered map $[x_1 \mapsto \overline{t_1}, \ldots, x_n \mapsto \overline{t_n}]$ from variables to terms or failure. A variable x may occur multiple times in ε, in which case the first (leftmost) binding is applicable. The application of an environment – a variable lookup – is defined as picking out the first binding for x (if any):

$$[x_1 \mapsto \overline{t_1}, \ldots x_n \mapsto \overline{t_n}](x) \begin{cases} t_i' & \text{if} \quad x_i \equiv x \wedge \forall j < i : x_j \not\equiv x \\ \uparrow & \text{if} \quad \forall j \leq n : x_j \not\equiv x \end{cases}$$

The variables in ε fulfil the role of algebraic term variables. The instantiation $\varepsilon(p)$ of the pattern p yields a (language) term, i.e. a ground term, by replacing every variable x in p with its bound term from ε. This is identical to variable substitution with σ with the exception that the pattern variables are variables of the System S calculus (i.e. variables in the Stratego language).

Environments ε are used in the matching process of patterns p. It is convenient to have a notation stating that the only difference between environments ε and ε' are the bindings for the variables of p. The notation $\varepsilon' \sqsupseteq \varepsilon$ declares that the environment ε' is a refinement of the environment ε. This means that if $\varepsilon = [x_1 \mapsto \overline{t_1}, \ldots, x_n \mapsto \overline{t_n}]$, then $\varepsilon = [x_1 \mapsto \overline{t'_1}, \ldots, x_n \mapsto \overline{t'_n}]$ and for each $i : 0 \leq i \leq n$, $\varepsilon(x_i) = \varepsilon'(x_i)$ or $\varepsilon(x_i) =\uparrow$ and $\varepsilon'(x_i) = t$ for some term t. $\varepsilon' \sqsupseteq_p \varepsilon$ declares that the environment ε' is the *smallest refinement* of the environment ε with respect to a term pattern p if $\varepsilon' \sqsupseteq \varepsilon$ and for all x not in p, $\varepsilon'(x) = \varepsilon(x)$.

Algebraic Properties The notation $e_1 \equiv e_2$ is used to describe algebraic properties of the defined constructs and to define syntactical shorthands. These equations are universally quantified unless otherwise stated.

3.2.1 Primitive Operators and Strategy Combinators

System S provides a handful of *primitive operators* on terms. The most basic of these are identity (`id`) and failure (`fail`) operators. Applying the identity operator to a term leaves the term unchanged; applying the failure operator signals a failure:

$$\Gamma, \varepsilon \vdash \langle \mathtt{id} \rangle t \Rightarrow t(\Gamma, \varepsilon) \qquad\qquad \Gamma, \varepsilon \vdash \langle \mathtt{fail} \rangle t \Rightarrow\uparrow (\Gamma, \varepsilon)$$

The operators, such as `id` and `fail`, are combined into expressions using *strategy combinators*. The purpose of the combinators is to describe control flow. Strategy expressions are built from primitive operators and combinators. The combinators are used to express application – evaluation – strategies of transformations in terms of how strategy application failures are handled. Any System S operator (except identity) may fail. Strategy combinators are used to specify what should happen when failures occur.

Sequential Composition The sequential application of two strategies s_1 and s_2 is expressed using the sequential composition combinator, $s_1; s_2$.

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow t'(\Gamma', \varepsilon') \quad \Gamma', \varepsilon' \vdash \langle s_2 \rangle t' \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}{\Gamma, \varepsilon \vdash \langle s_1; s_2 \rangle t \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}$$

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow\uparrow (\Gamma', \varepsilon')}{\Gamma, \varepsilon \vdash \langle s_1; s_2 \rangle t \Rightarrow\uparrow (\Gamma', \varepsilon')}$$

The assertions describe that strategy s_1 is first applied to the current term t. If it succeeds, s_2 is applied to its result; the result of the combination is the result of s_2. If s_1 fails, the combination fails. The following equations are consequences of the definitions above. They show that the `id` operator is a unit for sequential composition and that `fail` is a left zero.

$$\texttt{id}; s \equiv s \quad s; \texttt{id} \equiv s \quad \texttt{fail}; s \equiv \texttt{fail}$$

Not that in the general case, $\exists s : s; \texttt{fail} \not\equiv \texttt{fail}$. This follows from the way the state and the environment propagates over s: any environment ε before s will in general be ε' after s, whereas `fail` preserves the environment. Because of this, `fail` is not a right zero for sequential composition.

Guarded Choice The *guarded choice* (sometimes referred to as just the choice combinator) $s_1 < s_2 + s_3$ resembles an if-then-else expression, e.g.:

$$\texttt{id} < s_2 + s_3 \equiv s_2 \quad \texttt{fail} < s_2 + s_3 \equiv s_3$$

First, s_1 is applied. If s_1 succeeds, s_2 is applied and the result of s_2 is the result of the combined expression; if s_2 fails, the combination fails. Should s_1 fail, s_3 is applied and the result of s_3 is the result of the combination; if s_3 fails, the combination fails.

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow t'(\Gamma', \varepsilon') \quad \Gamma', \varepsilon' \vdash \langle s_2 \rangle t' \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}{\Gamma, \varepsilon \vdash \langle s_1 < s_2 + s_3 \rangle t \Rightarrow \overline{t''}(\Gamma'', \varepsilon'')}$$

$$\frac{\Gamma, \varepsilon \vdash \langle s_1 \rangle t \Rightarrow \uparrow (\Gamma', \varepsilon') \quad \Gamma', \varepsilon \vdash \langle s_3 \rangle t \Rightarrow \overline{t'}(\Gamma'', \varepsilon'')}{\Gamma, \varepsilon \vdash \langle s_1 < s_2 + s_3 \rangle t \Rightarrow \overline{t'}(\Gamma'', \varepsilon'')}$$

An important feature of the guarded choice is that if s_1 fails, both the effects due to s_1 on the term t are *and* to the environment (but not the state Γ) are undone. This means that the choice combinator implements a notion of (local) backtracking.

Negation, Left and Right Choices For notational convenience, the operators *not*, *left choice*, and *right choice* may be defined using guarded choice:

$$\begin{array}{rrl}
\textit{left choice} & s_0 \texttt{<+} s_1 \equiv & s_0 < \texttt{id} + s_1 \\
\textit{right choice} & s_0 \texttt{+>} s_1 \equiv & s_1 < \texttt{id} + s_0 \\
\textit{not} & \texttt{not}(s) \equiv & s < \texttt{fail} + \texttt{id} \\
\textit{try} & \texttt{try} s \equiv & s \texttt{ <+ id}
\end{array}$$

3.2.2 Primitive Traversal Strategies

The combinators in the previous section addressed the first of the two concerns of rule application: how rule application failure may be handled. The second concern – where in a term rules should be applied – is addressed by *primitive traversal strategies*. There are two primitive traversal strategies: `one` and `all`. They enable term traversal by local navigation into subterms.

`topdown(s) =`	`s; all(topdown(s))`	*top-down traversal*
`bottomup(s) =`	`all(bottomup(s)) ; s`	*bottom-up traversal*
`repeat(s) =`	`try(s; repeat(s))`	*apply s until it fails*
`oncetd(s) =`	`s <+ all(oncetd(s))`	*apply* `s` *once, start at the top*
`oncebu(s) =`	`all(oncebu(s)) <+ s`	*apply* `s` *once, start at the bottom*
`innermost(s) =`	`bottomup(try(s; innermost(s)))`	*innermost traversal*
`outermost(s) =`	`repeat(oncetd(s))`	*outermost traversal*

Table 3.1: A selection of frequently used traversal and application strategies.

All Subterms The `all(s)` strategy applies the strategy expression s to each subterm of the current term, potentially rewriting each. `all(s)` succeeds if and only if s succeeds for all subterms.

$$\frac{\Gamma_0, \varepsilon_0 \vdash \langle s\rangle\, t_1 \Rightarrow t_1'(\Gamma_1, \varepsilon_1) \quad \ldots \quad \Gamma_{n-1}, \varepsilon_{n-1} \vdash \langle s\rangle\, t_n \Rightarrow t_n'(\Gamma_n, \varepsilon_n)}{\Gamma_0, \varepsilon_0 \vdash \langle \mathtt{all}(s)\rangle\, c(t_1, \ldots, t_n) \Rightarrow c(t_1', \ldots, t_n')(\Gamma_n, \varepsilon_n)}$$

$$\frac{\Gamma_0, \varepsilon_0 \vdash \langle s\rangle\, t_1 \Rightarrow t_1'(\Gamma_1, \varepsilon_1') \quad \ldots \quad \Gamma_{i-1}, \varepsilon_{i-1} \vdash \langle s\rangle\, t_n \Rightarrow\uparrow (\Gamma_i, \varepsilon_i)}{\Gamma_0, \varepsilon_0 \vdash \langle \mathtt{all}(s)\rangle\, c(t_1, \ldots, t_n) \Rightarrow\uparrow (\Gamma_i, \varepsilon_i)}$$

The strategy `all(s)` behaves as follows with respect to failure, identity and constant terms:

$$\mathtt{all(id)} \equiv \mathtt{id} \quad \mathtt{<all}(s)\mathtt{>}c() \equiv c() \quad \mathtt{<all(fail)>}c(t_1, \ldots, t_n) \equiv \mathtt{fail}\ (\text{if } n > 0)$$

One Subterm The traversal strategy `one(s)` is similar to `all`, but applies s to exactly one subterm. It fails if s does not succeed for any of the subterms.

$$\frac{\Gamma, \varepsilon \vdash \langle s\rangle t_1 \Rightarrow\uparrow (\Gamma_1) \ \ldots\ \Gamma_{i-2}, \varepsilon \vdash \langle s\rangle t_{i-1} \Rightarrow\uparrow (\Gamma_{i-1})\ \Gamma_{i-1}, \varepsilon \vdash \langle s\rangle t_i \Rightarrow t_i'(\Gamma_i, \varepsilon_i)}{\Gamma, \varepsilon \vdash \langle \mathtt{one}(s)\rangle c(t_1, \ldots, t_n) \Rightarrow c(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n)(\Gamma_i, \varepsilon_i)}$$

$$\frac{\Gamma, \varepsilon \vdash \langle s\rangle t_1 \Rightarrow\uparrow (\Gamma_1, \varepsilon_1) \quad \ldots \quad \Gamma_{n-1}, \varepsilon \vdash \langle s\rangle t_n \Rightarrow\uparrow (\Gamma_n, \varepsilon_n)}{\Gamma, \varepsilon \vdash \langle \mathtt{one}(s)\rangle c(t_1, \ldots, t_n) \Rightarrow\uparrow (\Gamma_n, \varepsilon_n)}$$

The `one(s)` strategy backtracks (undoes) all modifications to the variable environment made by failing applications of s, but changes to the system state are kept.

Generic Traversal Strategies

An important feature of System S (and Stratego) is its ability to define signature-independent (and thereby language-independent) traversal strategies. This support is the result of mixing primitive traversal operators and strategy combinators. The mix yields the notion of *generic traversal strategies*. Examples of generic traversal strategies are given in Table 3.1.

Each generic traversal strategy $s_t(s)$ is parametrised with a strategy s that is applied throughout a term according the traversal scheme specified by s_t. The argument strategy s is used to insert language-specific rewriting logic, thereby instantiating the generic strategy for a specific subject language and signature.

3.2.3 Building and Matching Terms

System S supports two complementary operations for applying patterns to terms: match and build. Patterns are matched against terms using the match operator (`?`). Variables in the pattern are bound to their respective subterms. Terms are instantiated from patterns using the build operator (`!`). Variables are replaced by the terms they have previously been bound to.

Term Matching The assertions for term matching are given below:

$$\frac{\exists \varepsilon' : \varepsilon' \sqsupseteq_p \varepsilon \wedge \varepsilon'(p) \equiv t}{\Gamma, \varepsilon \vdash \langle ?p \rangle\, t \Rightarrow t(\Gamma, \varepsilon')} \qquad \frac{\nexists \varepsilon' : (\varepsilon' \sqsupseteq_p \varepsilon \wedge \varepsilon'(p) \equiv t)}{\Gamma, \varepsilon \vdash \langle ?p \rangle \Rightarrow\uparrow (\Gamma, \varepsilon)}$$

The semantics is compatible with the previously defined notion of match with variable substitution σ, with one exception: variables in p may already be bound. These variables are not rebound, but the corresponding subterms of t must match the terms bound by the variables of p. For example, a match of the pattern `Plus(x,y)` against the term `Plus(Int("0"),Int("1"))` (attempts to) bind the variable `x` to the term `Int("0")`. The match fails if the variable `x` is already bound to a term that is not `Int("x")`.

Term Building Term building is, in a sense, the inverse of matching. The build semantics is defined as:

$$\Gamma, \varepsilon \vdash \langle !p \rangle t \Rightarrow \varepsilon(p)(\Gamma, \varepsilon)$$

With the environment $\varepsilon = [x \mapsto$ `Int("0")`$, y \mapsto$ `Int("1")`$]$, the expression `!Plus(x,y)` will result in the the term `Plus(Int("0"),Int("1"))`.

3.2.4 Variable Scoping

The static scoping of term variables $x_1, \ldots, x_n$ can be controlled using the scope operator $\{x_1, \ldots, x_n : s\}$. Given $\varepsilon_0 = [y_1 \mapsto\uparrow, \ldots, y_n \mapsto\uparrow]$ and $\varepsilon_1 = [y_1 \mapsto \overline{t_1}, \ldots y_n \mapsto \overline{t_n}]$:

$$\frac{\Gamma, \varepsilon_0\varepsilon \vdash \langle [y_1/x_1, \ldots, y_n/x_n]s \rangle\ t \Rightarrow \overline{t'}(\Gamma', \varepsilon_1\varepsilon')}{\Gamma, \varepsilon \vdash \langle \{x_1, \ldots, x_n : s\}\ t \Rightarrow \overline{t'}(\Gamma', \varepsilon')}(y_1, \ldots, y_n \text{ fresh})$$

The operator introduces a new scope in which the strategy s is evaluated where the variables $x_1, \ldots, x_n$ have been replaced by fresh copies. This results in the usual notion of variable scoping: After s finishes, any binding for x_i, $1 \leq i \leq n$ introduced by s is removed from the environment. The scope operator succeeds if s succeeds and fails if s fails.

A useful syntactical abstraction over the scope operator is the `where` clause, later used for defining conditional rewrite rules. A `where(s)`-clause temporarily saves the current term, applies s to it, then restores the current term:

$$\texttt{where}(s) \equiv \{x : ?x; s; !x\}$$

It follows from the previous definitions that all variable bindings due to s are kept if s succeeds, and that `where(s)` fails iff s fails.

3.2.5 Rewrite Rules

The System S calculus can express rewrite rules with or without conditions, R_c and R_u, respectively:

$$R_u : p_l \rightarrow p_r \equiv ?p_l; !p_r$$
$$R_c : p_l \rightarrow p_r \text{ where } s \equiv ?p_l; \text{ where}(s); !p_r$$

The following is an example of a rewrite rule, named `Simplify`, defined in Stratego:

```
Simplify:
  Add(Int(x), Int(y)) → Int(z)
where <addS> (x,y) ⇒ z
```

The condition of this rule consists of the application of the strategy `addS` to the tuple `(x,y)`. (This tuple is the application of a nameless constructor with numeric arity two.) The result is "assigned" to the variable `z` using another syntactic abstraction, the $\Rightarrow$ operator, defined as follows:

$$s; ?p \equiv s \Rightarrow p$$

3.2.6 Additional Constructs

This section defined the core constructs of the System S calculus which are necessary for describing the language extensions proposed later in this dissertation. System

Strategy Expression	Meaning — *(basic constructs)*
$!p$	*(build)* Instantiate the term pattern p and make it the current term
$?p$	*(match)* Match the term pattern p against the current term
s_0 < s_1 + s_2	*(left choice)* Apply s_0. If s_0 fails, apply s_1. Else, roll back, then apply s_2.
s_0 ; s_1	*(composition)* Apply s_0, then apply s_1. Fail if either s_0 or s_1 fails
`id, fail`	*(identity, failure)* Always succeeds/fails. Current term is not modified
`one(`s`)`	Apply s to one direct subterm of the current term
`all(`s`)`	Apply s to all direct subterms of the current subterm

Figure 3.3: Basic language constructs.

S has several additional language constructs. These are presented informally using examples in the next section. Each of the explained constructs is used in some of the examples containing Stratego code throughout the following chapters, but understanding their precise and detailed semantics is not required. For a complete introduction to all of Stratego, refer to the Stratego/XT manual [BKVV05]. Specific caveats and considerations are noted along with the examples where pertinent.

3.3 Stratego

Stratego is a domain-specific language for writing program transformation libraries and components. It is based on the System S rewriting calculus. The language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction. The program object model used for representing subject programs are terms.

In the next sections, the parts of Stratego which are relevant for comprehending the remainder of this dissertation are explained in detail. A short description is given in Figure 3.3 and Figure 3.4 of the core Stratego language constructs offered to the programmer. The following sections informally describe additional features of Stratego.

3.3.1 Signatures, Patterns and Terms

Stratego supports the declaration of signatures for describing the abstract (or concrete) syntax of subject languages. Stratego signatures are very close to the concept of language (as opposed to algebraic) signatures described previously. Consider the following example:

Strategy Expression	Meaning — *(syntacic sugar)*
`\`p_l `->` p_r`\`	Anonymous rewrite rule from term pattern p_l to p_r
`?`x`@`p	Equivalent to `?`x `;` `?`p; bind current term to x then match p
`<`s`>` p	Equivalent to `!`p `;` s; build p then apply s
s `=>` p	Equivalent to s `;` `?`p; match p on result of s

Figure 3.4: Syntactic sugar.

```
1 signature
2 sorts Exp Stmt
3 constructors
4   Var : String → Var
5       : Var → Exp
6   Int : String → Exp
7   Float : String → Exp
8   Plus : Exp × Exp → Exp
9   Assign : Var × Exp → Stmt
```

This example illustrates the following differences between Stratego and algebraic signatures:

- Stratego signatures are not named. A program written in Stratego may have several signature declarations. The sorts and constructors from all of these declarations will be combined into one implicit "super signature".
- Only the arity of constructors is guaranteed by the Stratego language, i.e. it is a one-sorted system which allows synonym names for its sort. Given the signature above, the constructor `Plus` may be applied to any two subterms. Their sorts are never checked. Additionally, sorts need not be declared before they are used in constructor definitions, e.g. lines 7–8 above, where the sort `Var` is undeclared. It is considered good form to declare all sorts, however. A separate tool, called `format-check`, can be applied to a term to check if it is valid with respect to a given signature.
- Stratego has builtin (primitive) sorts and special term syntax for strings (`String`), lists (`List(x)`), tuples (`Tuple(x)`), integer (`Int`) and real (`Real`) numbers. The sort `Term` is used (by convention) to indicate an "any" sort. That is, any term may be inserted where a `Term` is expected.
- Nameless constructors of arity one are allowed, and these are called injections. Injections declare the terms of the argument sort may be placed wherever the target sort is allowed. In effect, injections declare their argument sort to be a subsort of the target sort, and are used by the `format-check` tool.

Strategy	*Meaning*
`rules(`rd_1 ... rd_n`)`	define rules $rd_1, \ldots, rd_n$
`{|`r_1, ..., r_n: `s|}`	start new scope for rule names $r_1, \ldots, r_n$
s_1 `/`$r_1, \ldots, r_n$`\` s_2	fork rule sets $r_1, \ldots, r_n$, apply s_1 then s_2, intersect rule sets
`/`$r_1, \ldots, r_n$`\*` s	apply s until rule sets $r_1, \ldots, r_n$ reach fixpoint
Rule definition (rd)	*Meaning*
`R :` $p_1 \rightarrow p_2$ where s	introduce rule R
`R :+` $p_1 \rightarrow p_2$ where s	extend rule R with another left-hand side p_1 (and r.h.s. p_2)
`R :-` p	undefine rules R with left-hand side p

Table 3.2: Essential basics of dynamic rules.

3.3.2 Congruences

A feature of System S (but not unique to it) is the combination of term traversals and rewriting into one compact construct, called congruences. Consider the following constructor:

```
C : S1 × ... × Sn → S
```

A congruence for this constructor is defined as the following rewrite rule with higher-order parameters $s_1, \ldots, s_n$:

$$c(s1, \ldots, s_n) : c(x_1, \ldots, x_n) \rightarrow c(y_1, \ldots, y_n) \texttt{ where } \langle s_1 \rangle x_1 \Rightarrow y1; \ldots; \langle s_n \rangle x_n \Rightarrow y_n$$

Given the above definition of a congruence and the previous definition of a rewrite rule, the expression

```
Plus(s0, s1)
```

syntactically expands to the following:

```
?Plus(x0,x1) ; where(<s0> x0 => x0' ; <s1> x1 => x1') ; !Plus(x0',x1')
```

While congruences are syntactically succinct, they mix data traversal strategies and term rewriting logic. This ties rewriting programs to very specific signatures and impairs reuse across subject languages.

3.3.3 Scoped, Dynamic Rules

Stratego supports the notion of dynamic rewrite rules that may be introduced and removed dynamically at runtime. These rules are used to capture and propagate context through the rewriting strategies. Figure 3.2 gives a brief summary of the dynamic rule basics.

The expression `rules(`R`: t -> r)` creates a new rule in the rule set for R. The scope operator `{| R : s |}` introduces a new scope for the rule set R around the strategy s. Dynamic rule scopes are dynamic – they follow the flow of the program. Variable scopes, on the other hand, are static – they follow the grammatical structure of the program text. Changes (additions, removals) to the rule set R done by the strategy s are undone after s finishes (both in case of failure and success of s). Sometimes, multiple rules in a rule set R may match. For example, the rule extension `rules(`R` :+ t -> r)` may be used several times with overlapping left hand sides. To get the results of all matching rules in R, one may use `bagof-`R. The additional operations relating to dynamic rewrite rules will be explained in the context of constant propagation, in Chapter 5.

The following example illustrates an application of dynamic rules to the problem of propagating variable constants. This example will be expanded upon in later chapters. The rule `PropConstAssign` must be applied to terms representing variable assignments in the subject language. If the right hand side of the assignment is a constant, the dynamic rule `PropConst` is added. This dynamic rule maps a given subject language variable to its known constant.

```
PropConstAssign:
Assign(Var(x), e) → Assign(Var(x), e')
where
    prop-const> e ⇒ e'
  ; if <is-value> e' then rules( PropConst : Var(x) → e' )
                      else rules( PropConst :- Var(x) ) end
```

If the constant is not known, i.e. the term `e` is not a value, any previous mappings for this subject language variable is removed.

Concrete Syntax Patterns

Concrete syntax patterns supplement term patterns and may sometimes result in more succinct transformation programs. Syntax patterns are by convention enclosed in "semantic brackets" (`|[ ]|`). They will be expanded in-place by the Stratego compiler to their equivalent AST term patterns.

```
?|[  e0 :=  e1 +  e2 ]| ≡ ?Assign(e0, Plus(e0), Plus(e2))
```

The grammar used to parse the concrete syntax must be specified to the compiler. The grammar is defined using a parser from the XT collection of transformation components described below.

3.3.4 Overlays

Overlays may be thought of as "term macros" and are used to abstract pattern matching over terms. Consider the following overlay declaration:

```
PlusOne(x) = Plus(x, Int("1"))
```

When compiling a program where this overlay is defined, the Stratego compiler will substitute every occurrence of the term `PlusOne(`x`)` with the term `Plus(`x`,Int("1"))`, for example:

$$\texttt{?PlusOne(Int("42"))} \xrightarrow{\textit{overlay expansion}} \texttt{?Plus(Int("42"),Int("1"))}$$

The x in this case is *not* a Stratego variable. Overlay substitution may be considered a "meta-rewriting" pre-processor step where all constant terms and patterns in a given Stratego program are expanded. After this pre-processing is finished, "normal" compilation resumes.

3.3.5 Modules

Stratego programs are organised into modules. Each module corresponds to a file, and is divided into typed sections. A module may import any number of other modules. A module import is (almost) equivalent to textual inclusion of the imported module[1]. Circular dependencies are allowed. Each section type, e.g. `strategies`, `overlays` and `rules`, specifies which declarations are allowed within that section. One exception exists: both strategies and rules may be declared freely within both `rules` and `strategies` sections.

3.3.6 Stratego/XT

A short note on the name "Stratego/XT" is necessary. The Stratego language was designed to support the development of transformation components at a high level of abstraction. It is distributed together with XT, a collection of flexible, reusable transformation components and declarative languages for deriving new components. Complete software transformation systems are composed from these components. The composition of Stratego and XT is named Stratego/XT.

The traditional usage pattern of Stratego/XT is illustrated in Figure 3.5. The developer starts by constructing or reusing a syntax definition for the subject language L. This definition is used to automatically derive a language infrastructure, such as a parser, pretty printer and a signature declaration for the abstract syntax of L. The developer may then write transformations using the derived infrastructure against the language L. The robustness and quality of the infrastructure is to a large extent

[1] The module name and the import declarations are removed.

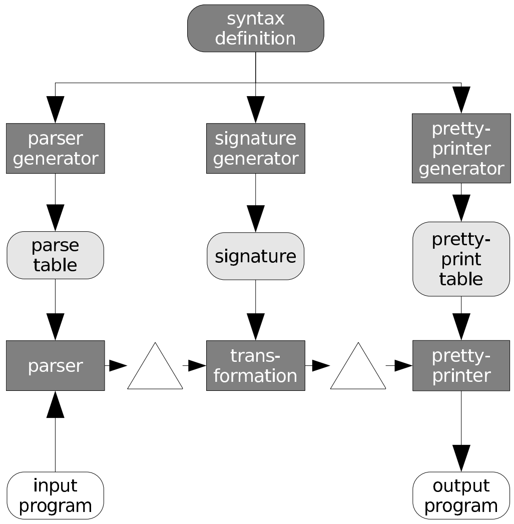

Figure 3.5: Derivation of language infrastructures from syntax definitions (grammars).

determined by the accuracy and quality of the grammar. For many mainstream languages, constructing a solid grammar is highly non-trivial. Consequently, robust and practical mechanisms for easily reusing existing language infrastructures is therefore desirable.

3.4 Summary

This chapter discussed the strategic programming methodology, a programming approach where data traversal patterns are separated from the data processing logic. It described (a subset of) the System S core calculus which applies the principles of strategic programming to term rewriting. The result is a clear separation between rewrite rules, which perform data processing, and generic traversals with combinators, which are used to encode data traversals. In the context of program transformations, the separation enables independent reuse of language specific rewrite rules and rule application strategies. This promotes language independence by allowing generic strategies to be reused across language specific rule sets. Basic elements of term rewriting theory were also introduced, together with their relation to universal algebra.