Programmable Rewriting Strategies in Haskell
— White Paper —

Ralf Lämmel\textsuperscript{1,2}

\textsuperscript{1} Vrije Universiteit, Amsterdam
\textsuperscript{2} Centrum voor Wiskunde en Informatica, Amsterdam

Abstract

Programmable rewriting strategies provide a valuable tool for implementing traversal functionality in grammar-driven (or schema-driven) tools. The working Haskell programmer has access to programmable rewriting strategies via two similar options: (i) the \textit{Strafunski} bundle for generic functional programming and language processing, and (ii) the “Scrap Your Boilerplate!” approach to generic functional programming. Basic rewrite steps are encoded as monomorphic functions on datatypes. Rewriting strategies are polymorphic functions composed from appropriate basic strategy combinators.

We will briefly review programmable rewriting strategies in Haskell. We will address the following questions:

- What are the merits of Haskellish strategies?
- What is the relation between strategic programming and generic programming?
- What are the challenges for future work on functional strategies?

\textit{Keywords:} Rewrite strategies, programming languages, Haskell, functional programming

\section{1 Strategic programming}

Our use of the term ‘strategy’ originates from the work on programmable rewriting strategies for term rewriting à la Stratego \cite{30,40,38}. Strategic pro-
grammers can separate basic rewrite steps from the overall scheme of traversal and evaluation. These schemes are programmable by themselves! There are one-layer traversal primitives that facilitate the definition of whatever recursion pattern for traversal. There are further, perhaps less surprising, basic combinators for controlling the evaluation in terms of the order of steps, the choices to be made, the fixpoints to be computed, and others. An extended exposition of what we call ‘strategic programming’ can be found in [24].

Related forms of programmable strategies permeate computer science. For instance, evaluation strategies without any traversal control are useful on their own in rewriting [7,4]. In theorem proving, one uses a sort of strategies as proof tactics and tacticals [33]. In parallel functional programming, one uses a sort of strategies to synthesise parallel programs [36].

2 Functional strategies in Strafunski

The Strafunski project [1,27,19,25,26,28] incarnated programmable rewriting strategies for functional programming, namely for Haskell. Strategies are essentially polymorphic functions on datatypes (or ‘term types’). The basic rewrite steps are readily specified as monomorphic functions on datatypes. For instance, the following rewrite step encodes some sort of constant elimination for arithmetic expressions:

\[
\text{const_elim} :: \text{Expr} \to \text{Maybe Expr} \\
\text{const_elim} ((\text{Const } 0) \ '\text{Plus}' \ x) = \text{Just } x \\
\text{const_elim} _ = \text{Nothing}
\]

In concrete syntax, and without Haskellish noise, this reads as “0 + x → x”. In the example, we wrap the result of the rewrite step in the Maybe monad, which allows us to observe success vs. failure of a rewrite step. We can use stacked monads (rather than just Maybe) in rewrite steps and strategies. This allows us to deal with state, environment, nondeterminism, and backtracking.

In Strafunski, there are two (monadic) types of strategies:

- **TP** — type-preserving strategies: domain and co-domain coincide.
- **TU** — type-unifying strategies: all datatypes are mapped to one result type.

Strafunski’s strategy library is based on primitive strategy combinators:

- **idTP** — the identity function.
- **failTP** — the always failing strategy.
- **adhocTP** — update strategy in one type.
- **seqTP** — sequential composition.
- **choiceTP** — left-biased choice.
- allTP — apply a strategy to all immediate subterms.
- oneTP — apply a strategy to one immediate subterm.
- Similar operators are offered for TU.

Rewrite steps can be turned into functional strategies using the adhocTP combinator. The strategy (idTP ‘adhocTP` const_elim) will succeed for all types other than Expr. The strategy (failTP ‘adhocTP` const_elim) will fail for all types other than Expr.

We can now define all kinds of reusable evaluation and traversal schemes, e.g.:

```haskell
-- Exhaustive application of a strategy
repeatTP :: MonadPlus m => TP m -> TP m
repeatTP s = (s `seqTP` (repeatTP s)) `choiceTP` idTP

-- Full type-preserving traversal in top-down order
full_tdTP :: Monad m => TP m -> TP m
full_tdTP s = s `seqTP` (allTP (full_tdTP s))

-- Type-preserving traversal stopping at successful branches
stop_tdTP :: MonadPlus m => TP m -> TP m
stop_tdTP s = s `choiceTP` (allTP (stop_tdTP s))

-- One-hit type-preserving traversal in bottom-up order
once_buTP :: MonadPlus m => TP m -> TP m
once_buTP s = (oneTP (once_buTP s)) `choiceTP` s
```

The essence of “The essence of strategic programming” [24]

As an illustrative use case, the strategy

once_buTP (failTP `adhocTP` const_elim)

attempts a single constant elimination when given a term. Applying this strategy exhaustively (cf. the evaluation strategy repeatTP), amounts to (naive) innermost normalisation. This use case demonstrates the overall tenor of strategic programming:

Separate problem-specific rewrite steps (i.e., const_elim) from the overall, possibly reusable scheme for traversal and evaluation (i.e., once_buTP). Both parts are put together by mere parameter passing, or by function composition. The schemes for traversal and evaluation are fully programmable by the virtue of one-layer traversal primitives (i.e., allTP and oneTP).

3 What are the merits of Haskellish strategies?

Applied setup

Haskellish strategies were born in an applied programming context. That is, we have designed them in an attempt to make functional programming fit for the implementation of program analyses and transformations — as relevant
in the context of language implementation, software reverse engineering, re-engineering, and others. For instance, Strafunski’s functional strategies readily deal with huge systems of algebraic datatypes as opposed to making assumptions such as use of single datatypes [15] or functorial encodings [35]. Also, functional strategies are versatile in terms of the recursion schemes that can be accommodated — when compared to programming with merely generalised folds [31]. Furthermore, functional strategies are conveniently customisable, whereas customisation is considered as a subordinated issue in other setups, which offer fully generic functions such as generic maps [14,13]. Customisation is crucial for strategic programming because traversal strategies involve type-specific cases on a regular basis.

Functional strategies have been used in various ways, e.g.:

- State-of-the-art Haskell refactoring tools [29].
- Language extension for Fortran [9].
- Java refactoring [27] (a subset of Java to be precise).
- Simple software metrics for Java [28].
- Reverse engineering for Cobol [28] (call-graph extraction).
- A framework for language-parametric refactoring [20].

**Language economy**

Functional strategies are easily supported in Haskell. There are different implementational models [27,19,26]. No proper language extension is needed. For some bits, code needs to be derived per user-defined datatype, which is however automated for the convenience of the programmer. Most strategic idioms are readily provided by Haskell. Most notably, rewrite steps are just functions defined by pattern matching. Also, monads [41] fit nicely with the effects that one encounters during strategic programming. The `Maybe` monad models the potential of failure. The list monad (and friends) is used to deal with nondeterminism and backtracking, alike for the state and the environment monad. Haskell has a strong record in implementing combinator libraries for programming domains, e.g., for parsing, pretty printing, XML processing, graphical user interfaces, and data structures. Strafunski’s strategies come just as another combinator library. Strategic programming in Haskell means that debugging, compilation, type checking, type inference, etc. come for free.

**Strongly typed, first-class strategies**

Strategy combinators are higher-order functions, which carry interesting types. So Haskell, again, is the right choice. Firstly, the type of a strategy combina-
tor clarifies if it is type-preserving (“TP”) or type-unifying (“TU”). Secondly, the chosen Monad instance in the type points out effects including potential of failure. Thirdly, the type indicates possible arguments that need to be passed in addition to the term, on which traversal is performed. While the influential system Stratego is largely untyped (but it could be typed [21]), Haskellish strategies are typed in all beauty of polymorphism and higher-order functions. This is taken to a limit in “The Sketch of a Polymorphic Symphony” [19], where we define ‘the mother of traversal’, which is a highly parametric traversal scheme.

We adopt an example from [20] to illustrates the virtue of typed, higher-order strategies. The following function signature types a strategy extract for a language-parametric program transformation. That is, the strategy models the extraction refactoring for whatever abstraction form — be it a method declaration, a function declaration, or others:

\[
\text{extract} :: \text{Abstraction abstr name tpe apply} \\
\Rightarrow \text{TU \{\text{name,tpe}\} Identity} \quad \text{-- Recognise declarations} \\
\Rightarrow \text{TU \{\text{name}\} Identity} \quad \text{-- Recognise using references} \\
\Rightarrow (\text{apply} \to \text{Maybe apply}) \quad \text{-- Recognise focused fragment} \\
\Rightarrow ([\text{abstr} \to \text{[abstr]}]) \quad \text{-- Mark host for new abstraction} \\
\Rightarrow ([\text{abstr} \to \text{Maybe [abstr]}]) \quad \text{-- Remove marking for host} \\
\Rightarrow ([\text{name,tpe}] \to \text{apply} \to \text{Bool}) \quad \text{-- Side conditions on fragment} \\
\Rightarrow \text{name} \quad \text{-- Name for new abstraction} \\
\Rightarrow \text{prog} \quad \text{-- Input program} \\
\Rightarrow \text{Maybe prog} \quad \text{-- Output program}
\]

The above Haskell type clearly identifies 4 type parameters for syntactical categories prog (programs), abstr (abstraction form), name (name of parameters and abstractions), and tpe (type of parameters) with a relationship Abstraction on them for the sake of making the function extract parametric with regard to the relevant abstraction form.

Using an untyped extract is beyond a Haskell programmer’s imagination. How would one possibly understand and correctly use an untyped function with 8 value arguments; 6 of the 8 of a higher-order type; 2 out of the 6 of a strategically polymorphic type?

4 Aren’t strategies just about generic programming?

In Strafunski, strategy types are opaque. Strafunski’s strategy library really provides an abstract datatype for strategies. This allows for different models of strategies. Some models have been described in the literature [27,19,26]. Some strategic improvements could be accommodated by new models without changing Strafunski’s API. The opaque status also encourages a point-free style (or combinator style) of strategic programming. We can clearly see that
Strafunski’s strategy types are opaque because there are even basic combina-
tors for strategy application, which resemble function application:

\[
\text{applyTP} :: (\text{Monad } m, \text{ Term } t) \Rightarrow \text{TP } m \rightarrow t \rightarrow m \ t
\]
\[
\text{applyTP } s \ t = \ldots -- \text{opaque implementation omitted}
\]

However, strategy types are not inherently opaque, and in the “Scrap Your
Boilerplate” approach to generic programming [2,22,23] they indeed aren’t.
In this approach, generic traversal schemes and all that are just straight poly-
morphic functions, possibly of a rank-2 type (as supported by the GHC im-
plementation of Haskell, but also elsewhere). The “Scrap Your Boilerplate”
approach is based on two Haskell classes Typeable and Data (the former be-
ing a superclass of the latter) for a handful of generic function combinators.
(The GHC implementation of Haskell automatically derives instances of these
classes per user-defined datatype.) Strafunski’s strategy library can be recon-
structed in this framework [26] by basically using just two of its combinators:
cast for type-safe cast and gfoldl for one-layer traversal.

Then, strategy types become non-opaque, concise and versatile:

\[
\text{type GenericM } m = \forall a. \text{Data } a \Rightarrow a \rightarrow m \ a \quad -- \text{corresponds to TP } m
\]
\[
\text{type GenericT} = \forall a. \text{Data } a \Rightarrow a \rightarrow a \quad -- \text{transformations}
\]
\[
\text{type GenericQ } r = \forall a. \text{Data } a \Rightarrow a \rightarrow r \quad -- \text{queries}
\]

In Strafunski, we did not favour variations like GenericT because this would
have implied a proliferation of combinators for the various opaque types. To
illustrate the use of these \forall types, we reconstruct the traversal scheme
stop_tdTP:

\[
\text{stop_tdTP} :: \text{GenericM Maybe } \rightarrow \text{GenericT}
\]
\[
\text{stop_tdTP } s \ x = \text{case } s \ x \ of
\text{Nothing } \rightarrow \text{gmapT (stop_tdTP } s) \ x
\text{Just } x' \rightarrow x'
\]

We used the combinator gmapT :: GenericT \rightarrow GenericT, which is the non-
monadic variation on allTP [22]. The type of stop_tdTP says that this com-
binder takes a polymorphic function and returns one. We use the type aliases
for readability; we could as well inline the \forall types. As an exercise in ver-
satility, we have reconstructed a more specifically typed scheme stop_tdTP.
The original scheme involved the opaque type TP m, where m could be instan-
tiated later to any instance of MonadPlus. The reconstructed scheme fixes the
monad for the argument type to Maybe, which allows us to guarantee success
of the composed strategy (cf. the non-monadic result type GenericT).

Once we get used to forming generic function types, we will not limit
ourselves to strategy types. That is, while strategies are unary polymorphic
functions on datatypes, there are other polymorphic type schemes of inter-
est. Generic functions do not need to be unary, neither do they need to be
polymorphic in the argument position. For instance:

type GenericEq = forall a b. (Data a, Data b) => a -> b -> Bool -- generic equality
type GenericB = forall a. Data a => a -- build a term; no traversal!
type GenericR m = forall a. Data a => m a -- read a term using a monad

So while the Strafunski approach emphasised unary term traversal, the "Scrap Your Boilerplate" approach to generic functional programming allows us to abstract over more than just unary term traversal. We can abstract over multi-parameter traversal, over term generation, serialisation, and de-serialisation, zipping, and others [23]. Especially the correspondence between term traversal and term building is a duality that was uncovered some time ago by squiggolists: given a regular datatype (such as lists), or perhaps even any datatype, one can fold a datum of the type ("traverse it"), and unfold it ("build it") [31,3]. Other generic programming approaches also serve this generality. For instance, generic programming extensions like PolyP [14] or Generic Haskell [12,6] employ powerful techniques for structural induction on the type structure of data to be consumed or produced. In this context, the "Scrap Your Boilerplate" approach is characterised as follows:

• The approach blends well with normal Haskell programming.
• The approach is lightweight. It is based on two simple Haskell type classes.
• The approach does not require any compile-time code specialisation.
• Generic functions operate immediately on Haskell datatypes.
• Generic functions are first-class citizens: traversal schemes are higher-order.
• Generic functions are easily customised by (nominal) type case.

5 Where to go from here?

Strategic programming is a young research field. Several challenges are readily waiting. The following list is biased towards functional strategies, and relates to the current Strafunski and "Scrap Your Boilerplate" implementations, but most challenges are relevant for programmable rewriting strategies in general.

Analysis opportunities

The functional strategist might want to take advantage of analyses that improve static guarantees or run-time performance of his or her strategies. Some typical examples follow:

Termination Strategic traversal schemes are like recursion schemes: they are meant as disciplined replacement for free-wheeling recursive program-
ming. Nevertheless, the versatility of strategies makes it still quite easy to encode diverging strategies. For instance, \( \text{repeatTP idTP} \) will diverge. The implied usage pattern for fixpoint iteration with \( \text{repeatTP} \) is that the argument strategy should eventually fail.

**Stupidity** Just as there are ‘stupid casts’ in object-oriented programming (i.e., type casts that cannot possibly succeed), so there are ‘stupid strategies’ in strategic programming. For instance, the strategy \( \text{full_tdTP (failTP ‘adhocTP’ f)} \) is stupid because a full traversal is meant to have a chance of succeeding for whatever type, but the given composition will undoubtedly fail for all types except for the domain of \( f \).

**Shortcutting** On the basis of the type-specific cases of a strategy it would be often feasible to shortcut traversal leading to a more efficient traversal. For instance, the strategy \( \text{full_tdTP (idTP ‘adhocTP’ f)} \) does not need to be pushed into a term any further if it is clear that subterms of \( f \)’s domain are out of reach — on the basis of static type information. For such hopeless branches, the strategy can be shortcut to \( \text{idTP} \).

**Composability** Chains of strategies need to cooperate in the sense that a given strategy in the chain should be enabled, or at least not disabled by earlier elements in the chain. (One could call this an advanced form of stupidity perhaps, so it is not stupid!) Enabling and disabling can be understood in terms of pre- and post-conditions for strategies, in which case related work on program transformation might turn out to be of use \([18,34]\).

*Expressiveness opportunities*

The functional strategist might even ask for extra expressiveness, which, in an extreme case, requires Haskell extensions. Alternatively, the extra-strategic expressiveness can also be accommodated by the virtue of a more open Haskell system, or by preprocessing, or perhaps by appropriate combinator libraries. Some prime examples follow:

**Sexy types** There is a potential need for designated types to declare, check, and infer success behaviour, determinism, and some forms of pre- and post-conditions. Also, the effects involved in strategies (such as failure, state, environment) were more conveniently used with an effect type system perhaps \([10]\) — as opposed to explicit monad transformers. A Haskell 20XX with a very open type system would be of use here.

**Object syntax** The prime application domain of strategic programming is program analysis and transformation. Encoding rewrite steps in terms of abstract syntax is relatively inconvenient for real-world programming lan-
languages. Haskell could support concrete syntax, just as rewriting technology like ASF+SDF [17,5] does already for a long time. Stratego was already equipped with concrete object syntax [39]; Haskell has to catch up.

**Graphs** Many program analyses and transformations favour graph-based intermediate representations. Haskell’s laziness allows for cyclic data structures. Node identities have to be ‘managed’ carefully. Constructing and transforming many-sorted graphs is difficult in Haskell. We are in need of a typeful approach that retains the convenience of pattern matching and building, and that provides us with the illusion of destructive update.

**Attribute grammars** Strategies and attribute grammars are complementary in that the former are more operational, whereas the latter are more declarative. Also, the former emphasise traversal, whereas the latter emphasise attribute dependencies. Research on a possible marriage of strategies and attribute grammars promises interesting insights. Alike strategies, attribute grammars are conveniently embedded into Haskell [8].

**Constraint programming** Another unexplored combination of worlds is the integration of programmable strategies and constraint programming, or residuation and narrowing — as available in a hybrid language like Curry [11]. Constraints could provide a versatile means to make strategies less operational, more declarative. Constraints could also provide means to narrow down the search space for strategies.

**XML & XPath** Next to language processing on the basis of syntaxes, strategies are thought to be useful for XML document processing. Functional combinator libraries for XML processing do exist [42], but they lack the typing strength of functional strategies. It should be possible to use strategies as a means to provide the illusion of an XPath-like language for controlling fully typed XML transformations.

*Strategy mining & refactoring to strategies*

The modularity and conciseness of legacy Haskell programs could benefit from the strategic style of programming. This calls for ‘strategy mining’. There exists related work on recovering recursion schemes like folds in legacy code [37]. When developing and enhancing existing Haskell programs, strategic style has to be installed or improved by means of refactoring. In fact, this is a form of ‘refactoring to patterns’ [16] because the strategic style of programming can be viewed as a collection of design patterns for traversal functionality [25]. In both cases, entangled traversal code is turned into strategically organised traversal code.
6 Concluding remark

We have briefly reviewed Haskell-based support for programmable rewriting strategies. We have also briefly discussed the link between rewriting strategies and generic programming. Finally, we have listed challenges for future work on Haskellish rewriting strategies.

Please, stay tuned at [1,2].

Acknowledgement

The Strafunski project [1,27,19,25,26,28] is joint work with Joost Visser. The “Scrap Your Boilerplate” approach [2,22,23] is joint work with Simon Peyton Jones.

References


