Design Patterns for Functional Strategic Programming

Ralf Lämmel
Vrije Universiteit
De Boelelaan 1081a
NL-1081 HV Amsterdam
The Netherlands
ralf@cs.vu.nl

Joost Visser
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands
Joost.Visser@cwi.nl

Abstract

We believe that design patterns can be an effective means of consolidating and communicating program construction expertise for functional programming, just as they have proven to be in object-oriented programming. The emergence of combinator libraries that develop a specific domain or programming idiom has intensified, rather than reduced, the need for design patterns.

In previous work, we introduced the fundamentals and a supporting combinator library for *functional strategic programming*. This is an idiom for (general purpose) generic programming based on the notion of a *functional strategy*: a first-class generic function that can not only be applied to terms of any type, but which also allows generic traversal into subterms and can be customised with type-specific behaviour.

This paper seeks to provide practising functional programmers with pragmatic guidance in crafting their own generic programs using functional strategies. To this end, we propose a novel description format for functional design patterns, and we initiate a catalogue of *strategy design patterns*. These design patterns aim at consolidating strategic programming expertise in accessible form.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Design; D.2.13 [**Software Engineering**]: Reusable Software; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.4 [**Programming Languages**]: Processors

General Terms

Design, Documentation, Languages

Keywords

Design patterns, Combinator libraries, Generic programming, Program analysis, Program transformation, Functional programming, Strafunski

Third ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02) Pittsburgh, PA, October 5, 2002

1 Introduction

The notion of a *design pattern* is well-established in object-oriented programming. A pattern systematically names, motivates, and explains a common design structure that addresses a certain group of recurring program construction problems. Essential ingredients in a pattern are its applicability conditions, the trade-offs of applying it, and sample code that demonstrates its application.

We contend that design patterns can be an effective means of consolidating and communicating program construction expertise for functional programming just as they have proven to be in object-oriented programming. One might suppose that the powerful abstraction mechanisms of functional programming obviate the need for design patterns, since many pieces of program construction knowledge can be captured by reusable functions. This confusion of design patterns with reusable components misses the point: patterns start where components end, in the sense that the former describe how the latter can be constructed and used. Thus, additional abstraction *creates* the need for new design patterns.

In this paper we present a set of design patterns for the functional incarnation of *strategic programming*. Strategic programming is a novel generic programming idiom in which the notion of a *strategy* plays a crucial role [20, 11]. This idiom has been worked out for the object-oriented programming paradigm, for term rewriting, and for functional programming (see below). It was developed for application areas that involve term traversal over large abstract syntaxes. Its expressiveness has proven to pay off in application areas such as program transformation and analysis [2], and reverse engineering [3]. In fact, strategic programming is a general-purpose generic programming idiom because the mere ability to traverse into terms is beneficial in almost every non-trivial software application.

In [12], we presented a realization of the strategic programming idiom in the functional programming paradigm, in particular in the Haskell programming language. We introduced the notion of a *functional strategy*, and we presented a *library of strategy combinators*. Functional strategies are functions with the following defining characteristics. They are functions that (i) are generic, i.e., they work on term arguments of any type, (ii) can display type-specific behaviour, (iii) can traverse into terms, and (iv) are first-class citizens. The expressiveness of functional strategies goes beyond that of ordinary parametrically and ad-hoc polymorphic functions.

The tiny example in Figure 1 has been constructed to illustrate the characteristics (i)–(iv) of strategies. The function *increment* is a functional strategy that increments all integers in a data structure by 1 regardless of the type of the data structure and of the fact where

```
increment    = topdown (adhoc id inc)
                 where inc :: Int → Int
                       inc i = i + 1
increment [(True, 1)]              ⟹ [(True, 2)]
increment (Just (1, ([1], 1)))     ⟹ Just (2, ([2], 2))
```

Figure 1. Example of a functional strategy.

the integers occur. The function is demonstrated by its application to terms of the types $[(Bool, Int)]$ and $Maybe\ (Int, ([Int], Int))$. The definition of *increment* makes use of a library traversal scheme *topdown*, and a basic combinator *adhoc* for updating the identity function by a type-specific case *inc* for *Int*egers. To summarise, the *increment* strategy *traverses* its argument in *generic* fashion, it displays behaviour *specific* to integers, and it is implemented as a *first-class* function using function combinators.

An extensive library of reusable strategy combinators is distributed as part of *Strafunski*: a Haskell-based bundle that supports generic programming with functional strategies.[1] The combinators in the library address a wide range of themes, such as traversal, fixpoints, control and data flow, and effect management. The library combinators can be composed, customised, and applied to construct application programs.

The novelty of the strategic programming idiom implies that few are experienced and well-versed in it. Though *Strafunski*'s library provides an extensive array of predefined strategy combinators, actual deployment of these combinators for program construction is an acquired skill, as is any (functional) programming style. In this paper, we attempt to convey our (limited) deployment expertise in a set of *design patterns*, such that others can likewise make effective use of the power that the idiom of strategic programming offers.

The paper is structured as follows. In Section 2, we explain the notion of a functional design pattern, and we fix a format for design pattern description. In Section 3, we review the fundamentals of strategies and we outline *Strafunski*'s support for development of and with strategies. In Section 4, we present a catalogue of strategy design patterns. Each pattern is illustrated with code samples. The originality of the patterns and the concise and accessible style should appeal to the linear reader. The non-linearity of the catalogue turns it into a valuable tool for Strafunski users.

2 Functional design patterns

The notion of a *design pattern* is well-established in object-oriented programming. In the first pattern catalogue, design patterns are defined as *"descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context"* [5, page 3]. Each pattern systematically names, motivates, and explains a common design structure that addresses a certain group of recurring program construction problems. Essential ingredients in a pattern are its applicability conditions, and the results and trade-offs of applying it. After the initial 23 patterns of the first catalogue, numerous further patterns have been described in roughly the same style and format. Design patterns have proven to be an effective means of consolidating and communicating object-oriented program construction expertise.

[1] *Strafunski* home page: `http://www.cs.vu.nl/Strafunski`
`Stra` refers to strategies, `fun` refers to functional programming, and their harmonious composition is a homage to the music of Igor Stravinsky.

NAME — CATEGORY

Aka*
Alternative names.

Intent
Short statement of the pattern's purpose.

Motivation
Description of a particular design problem and a brief indication of how the pattern can be used to solve it.

Applicability*
Preconditions for using the pattern.

Schema
A schematic code fragment that indicates the participants in the pattern and their relationships.

Description
Explanation of the schema that details the responsibilities of all participants and describes how they collaborate to carry them out.

Sample code
Actual working Haskell code in which the pattern is used.

Consequences*
Description of the results and trade-offs of applying the pattern.

Related patterns*
Similarities, differences, and connections to other patterns.

Figure 2. Format of each design pattern description.

Beyond combinator libraries

Can design patterns be useful in functional programming as well? Are design patterns needed anyway given the powerful abstraction mechanisms of functional programming? One might argue that there is no need for design patterns, since many pieces of program construction knowledge can be captured in reusable functions. We rather claim that design patterns and reusable components are complementary. Design patterns are an effective means to describe how reusable components can be constructed and used. The abstraction mechanisms of functional programming have enabled the emergence of a range of combinator libraries for various specific programming domains. These include libraries for parsing [7], pretty-printing [6], monad transformers [13], graphical user interfaces [18], and polytypic programming [8]. The deployment of the combinators in each of these libraries is an acquired skill, that needs to be communicated to and among their users. Design patterns would be an effective vehicle for such communication. To summarise, combinator libraries *create* rather than obviate the need for design patterns. Apart from combinators libraries, numerous advanced idioms in functional programming can be identified that could benefit from a design pattern approach of explanation. Examples are strictification, first-class polymorphism, monad and list comprehensions, parallel and distributed programming, existential types and generalised folds. Each of these idioms involves specific expertise that could be consolidated concisely in an appropriate set of design patterns.

Pattern description format

We have chosen a format for functional design pattern description that is very similar to the one known from object-oriented literature. This format is shown in Figure 2. Of course, a few modifications to the object-oriented precursors are in order to accommodate the characteristics of functional programming and the space limitations of this publication. We have chosen to make particular items op-

```
Strategy types                                                          Description
  data TP m      = abstract                                               -- type-preserving, roughly: ∀t. t → m t
  data TU u m    = abstract                                               -- type-unifying, roughly: ∀t. t → m u

Basic combinators
  applyTP    :: (Monad m, Term t) ⇒ TP m → t → m t                       -- strategy application
  applyTU    :: (Monad m, Term t) ⇒ TU u m → t → m u                      --    overloaded for a term type t

  adhocTP    :: (Monad m, Term t) ⇒ TP m → (t → m t) → TP m               -- strategy update
  adhocTU    :: (Monad m, Term t) ⇒ TU a m → (t → m u) → TU u m           --    to add type-specific behaviour

  idTP       :: Monad m ⇒ TP m                                           -- generic monadic identity function
  constTU    :: Monad m ⇒ u → TU u m                                     -- generic monadic constant function

  failTP     :: MonadPlus m ⇒ TP m                                       -- always fail
  failTU     :: MonadPlus m ⇒ TU u m                                     --    using a monad for partiality

  seqTP      :: Monad m ⇒ TP m → TP m → TP m                             -- perform strategies in sequence
  seqTU      :: Monad m ⇒ TP m → TU u m → TU u m                          --    where the first strategy is type-preserving

  passTP     :: Monad m ⇒ TU u m → (u → TP m) → TP m                      -- value passing with shared term argument
  passTU     :: Monad m ⇒ TU u m → (u → TU u' m) → TU u' m                --    where the first strategy is type-unifying

  choiceTP   :: MonadPlus m ⇒ TP m → TP m → TP m                         -- attempt alternative strategies
  choiceTU   :: MonadPlus m ⇒ TU u m → TU u m → TU u m                    --    using the "+" of an extended monad, e.g., Maybe

  allTP      :: Monad m ⇒ TP m → TP m                                    -- apply argument to all immediate subterms
  allTU      :: (Monad m, Monoid u) ⇒ TU u m → TU u m                    --    for TU, results are reduced with a monoid's "+"

  oneTP      :: MonadPlus m ⇒ TP m → TP m                                -- apply argument to one kid
  oneTU      :: MonadPlus m ⇒ TU u m → TU u m                            --    try kids from left to right, fail if none succeeds

  msubstTP   :: (Monad m, Monad m') ⇒ (∀ t . m t → m' t) → TP m → TP m'  -- substitute one monad by another
  msubstTU   :: (Monad m, Monad m') ⇒ (m u → m' u) → TU u m → TU u m'    --    using a helper function
```

Figure 3. Strategy types and basic strategy combinators.

tional (indicated by an asterisk). More importantly, the class diagrams of object-oriented programming seem to have no obvious functional counterparts since such a focus on structure, inheritance and aggregation relationships is alien to our setting. For the kind of abstractions we are dealing with, we have no knowledge of suitable diagrammatic functional program representations that would be suitable in our documentation context. So instead of diagrams we provide *schematic* Haskell code fragments.

3 Strategic programming

Before we present the core contribution of the paper, that is, the pattern catalogue for functional strategic programming, we review this style of programming from the user's perspective. The fundamentals of functional strategies were introduced in [12], and further background material is available (see the *Strafunski* web site).

3.1 Functional strategies

In the introduction, functional strategies were defined by enumerating their defining characteristics:

i They can work on term arguments of *any* type. (Genericity)
ii They can traverse into subterms. (Traversal)
iii They can display type-specific behaviour. (Customisation)
iv They can be named, passed as argument, etc. (First-class)

This abstract definition can be made more concrete by establishing a minimal set of basic strategy combinators that realizes these characteristics. Figure 3 shows a corresponding combinator suite including combinators for traversal and type-specific customisation.

It is clear that strategies go beyond ordinary parametrically polymorphic functions because of their ability to traverse into terms and to customise uniform behaviour by type-specific cases. This is illustrated by the *increment* example from the introduction. This function is basically of type $\forall\alpha.\ \alpha \to \alpha$. The only parametrically polymorphic inhabitant of this type would be the identity function, to which *increment* is clearly not equivalent. Strategies also go beyond ad-hoc polymorphism in the sense of overloaded function declarations. This is because strategy customisation relies on a combinator as opposed to a syntactical declaration form. This is also illustrated by the introductory example *increment* because the node processor that is passed to the *topdown* traversal scheme is constructed via the *adhoc* combinator.

We will now explain the combinator suite in Figure 3 in detail.

Strategy types

Two strategy types are distinguished:

TP for type-preserving strategies (output and input type coincide),
TU for type-unifying strategies (output is always of a given type *u* regardless of the type of the input term).

Because of these two types, the combinators in Figure 3 come in pairs as pointed out by the respective postfixes. Both types are parameterised with a monad *m*. This parameterisation allows functional strategies to involve standard monadic effects such as state, environment, I/O, failure, and others. We will consider the types *TP* and *TU* as *abstract* in the present paper. This is because there are actually several possible models of strategies in Haskell (see the *Strafunski* web site for background material). These models rely on either dynamic typing, or first-class polymorphism, or rank-2 types as for the mere types of strategies. Furthermore, the several models employ an interface for the terms that are built via user-supplied algebraic datatypes (cf. class constraint *Term t* ⇒ ... in Figure 3). This interface supports the strategic expressiveness for traversal and customisation.

Strategy application and update

The *apply* combinators justify our claim that strategies are generic functions (recall *i*). Their result types model the type schemes underlying type preservation and unification (cf. $t \to m\,t$ and $t \to m\,u$). The *adhoc* combinators support type-specific customisation of a strategy (recall *iii*). If s is a strategy, and f is a function on some type t, then the strategy $adhoc\,s\,f$ will behave like f when applied to a term of type t and like s on terms of any other type.[2]

Nullary combinators

Our combinator suite contains the following constant strategies. The *idTP* combinator is a generic version of the monad member *return* (which in turn is just the monadic identity function), i.e., *idTP* returns the input term. The strategy *constTU x* ignores the input term and always returns the argument *x*. It is the generic counterpart of the *const* function. The combinators *failTP* and *failTU* denote the always failing strategy in the sense of a *MonadPlus* with a member *mzero* for failure. Instantiation of the monad parameter *m* of *TP* and *TU* with such a monad allows us to deal with partiality, recovery from failure, backtracking, and non-determinism.

Binary composition combinators

The *seq* and *pass* combinators perform their two argument strategies in sequence. For the *seq* combinators, the first argument strategy is always type-preserving, and its output is given to the second argument strategy as input. For the *pass* combinators, the first argument strategy is always type-unifying for some type *u*, and the second argument strategy is accordingly parameterised with a value of type *u* while otherwise both strategies are applied to the same input term. The *choice* combinators support (left-biased) composition of strategies with alternative branches relying on the *MonadPlus* class, more precisely on its member *mplus*. Using the *Maybe* monad, strategies can recover from failure because the right argument strategy is applied in case that the attempt to apply the left strategy failed. Using other monads, we can deal with non-deterministic or backtracking strategies as well.

Unary traversal combinators

The *all* and *one* combinators are *one-step* traversal combinators (recall *ii*). In a sense, they push their argument strategy one level down into their input term. They do not yet anticipate any scheme of recursive traversal. The *all* combinators apply their argument strategy to *all* immediate subterms, while the *one* combinators try it left-to-right on each of the immediate subterms and stop after the first succeeds. To this end, the types of the potentially failing strategy passed to the *one* combinators involves *MonadPlus*. The type-preserving variants *allTP* and *oneTP* preserve the outermost constructor of the input term. The type-unifying *allTU* relies on the binary operator *mappend* of a *Monoid* to reduce the results of processing the subterms to a single result.

[2] We omitted the *applyTP* combinator in the definition of the introductory strategy *increment*. If we assume a rank-2 model of functional strategies, ordinary function application is sufficient. We also did not qualify *adhoc* by "*...TP*" because we assume that *adhoc* is overloaded for *TP* and *TU*.

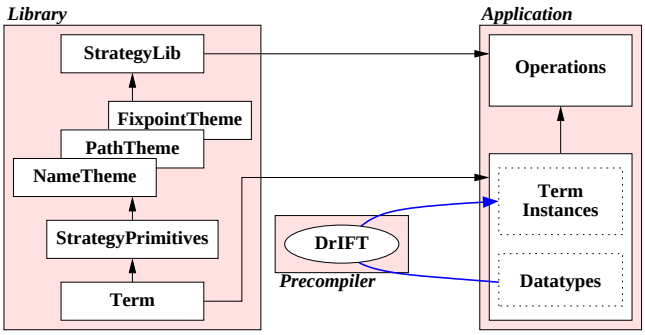

Figure 4. *Strafunski = Library + Precompiler*

Monad transformation

The *msubst* combinators can be used to migrate from one monad to another. This is useful, for example, if we want to hide the fact that a certain strategy has the potential to fail while the overall strategy cannot due to recovery of failure. In this case, we would migrate from the *Maybe* to the *Identity* monad.

3.2 Strafunski

Strafunski is a Haskell-based bundle that supports generic programming with functional strategies. Figure 4 provides an overview of the elements of *Strafunski*, and their relation to an application constructed with it. *Strafunski* consists of two components: a *library* and a *precompiler*, which we will discuss in turn.

Library

The library of *Strafunski* consists of a number of Haskell modules that address various aspects of strategic programming.

StrategyLib This is the top-level module of the library, provided for convenience. It allows the user to import the entire library with a single import statement.

Themes A series of modules is provided that covers a range of generic programming themes. For example, the *FixpointTheme* deals with iterative term transformation which terminates when some kind of fixpoint is found. The *TraversalTheme* defines various traversal schemes. The *NameTheme* provides abstract algorithms for different kinds of name analysis, useful in language processing applications. In the *OverloadingTheme*, the basic strategy combinators of Figure 3 are overloaded for *TP* and *TU* to implement the intuition that the combinators come in pairs. Overloading allows one to postpone commitment to a particular strategy type when defining a new strategy. With each version of *Strafunski*, the theme library is elaborated. Excerpts of the current state of affairs are shown in Figure 5.

StrategyPrimitives This module provides basic strategy types, and a basic set of strategy combinators as it was shown earlier in Figure 3. Together they form an *abstract* datatype, whose internals are not exposed beyond the module. In fact, we have experimented with several implementations of the datatype that each have different characteristics with respect to performance, extensibility, and use of type features. The *Strafunski* bundle can be configured to use one out of several alternative models.

Term In general, this module provides a generic term interface as a type class *Term*. The further internals are model-dependent. The current default model of *Strafunski* employs a universal term

```
Fixpoint theme                                                          Description
  repeat s      = try (seqTP s (repeat s))                                -- keep applying s until it fails
  outermost s   = repeat (oncetd s)                                       -- outermost evaluation strategy
  innermost s   = repeat (oncebu s)                                       -- innermost evaluation strategy

Traversal theme
  topdown s     =  s `seqTP` (allTP (topdown s))                          -- apply s in top-down fashion to all nodes
  bottomup s    =  (allTP (bottomup s)) `seqTP` s                         -- idem, in bottom-up fashion
  stoptd s      =  s `choiceTP` (allTP (stoptd s))                        -- cutoff traversal below nodes where s succeeds
  oncetd s      =  s `choiceTP` (oneTP (oncetd s))                        -- terminate traversal at first node where s succeeds
  oncebu s      =  (oneTP (oncebu s)) `choiceTP` s                        -- idem, bottom-up
  crush s       =  liftop2 mappend s (allTU (crush s))                    -- combine results of applying s to all nodes with a monoid's "+"
  select s      =  s `choiceTU` (oneTU (select s))                        -- return result of first succesful application of s (top-down)

Control & data-flow theme
  try s         =  s `choiceTP` idTP                                      -- recover from failure
  liftop2 o s s' =  s `passTU` λa → s' `passTU` λb → constTU (o a b)      -- combine the result of two strategies with binary operator o
  before s f    =  s `passTU` (constTU ∘ f)                               -- apply type-unifying strategy s and then f to its result
```

Figure 5. Excerpts from Strafunski's theme modules.

representation in combination with dynamic typing in order to support the combinators for type-specific customisation and one-step traversal. This module remains hidden for the rest of the library and the user code.

Precompiler

To use the *Strafunski* library in an application, instances of the *Term* class must be provided for the datatypes of the application. This can be done manually, but *Strafunski* provides a precompiler to automate the process. This is possible because these *Term* instances follow a very simple scheme for all algebraic datatypes. Currently, the precompiler is implemented as an extension of the DrIFT tool (formerly known as Derive [22]). For the upcoming sample code that illustrates the strategic design patterns, we precompiled a set of datatypes representing the abstract syntax of Haskell 98. The resulting *Term* instances allow us to construct strategies that traverse over Haskell parse trees.

Thus, functional strategic programming with *Strafunski* proceeds along the following steps:

1. Apply the precompiler to the system of datatypes that represent the terms on which to operate.
2. Import the precompiled datatypes and *StrategyLib* into an application module.
3. Select, combine, and specialise appropriate strategy combinators from *Strafunski*'s library, and apply the resulting strategies to the terms that need to be processed.

Clearly, the last of these steps deserves elaboration, which we will provide in the form of a catalogue of design patterns.

4 Pattern catalogue

In this section we initiate the catalogue of design patterns for functional strategic programming. Figure 6 gives an overview. Before we turn to the patterns themselves, a few remarks are in order on the application experience from which the patterns originate, and on the organisation of the catalogue.

Application experience

The design patterns that we will present below are based on our experiences in applying functional strategic programming to applications that deal with program analyses and transformations for various languages such as Cobol, Haskell, and Java. *Strafunski* is suited for all kinds of language processors such as those needed in language implementation, software re-engineering, reverse engineering, renovation, and re-documentation. Various papers with case studies can be found on the *Strafunski* web site. Some applications are also included in the *Strafunski* distribution. The sample code of the various patterns has been extracted from actual applications and adapted for presentation. Rather than switching between various datatype systems, we have chosen to stick throughout the catalogue to the Haskell abstract syntax. Note that the mere size of this syntax (about 30 types and 100 data constructors) clearly demonstrates the benefits of *robustness* and *conciseness* of strategic programming. While our sample code includes non-trivial functionality for the entire Haskell syntax, only a handful of the types and data constructors need to be mentioned explicitly.

Organising the catalogue

Each design pattern is aimed at solving only a single design problem. Clearly, in actual applications design problems never come alone, and combinations of patterns must be used. The selection of appropriate design patterns for a given set of design problems can be guided by categorising the patterns according to various criteria. We briefly discuss three such criteria.

1. We have divided our design patterns into four main categories, as indicated in the upper right-hand corner of the description format:

Rewriting Patterns in this group address design problems encountered by any strategic programmer. They explain how to model rewriting systems with functional strategies.

Interfacing Patterns in this group demonstrate how powerful strategic functionality can be hidden behind simpler interfaces.

Logic Patterns in this group describe how flows of data and control can be managed within strategies.

Parameterisation Patterns in this group show how additional levels of abstraction can be achieved.

Pattern	Intent	Concerns
REWRITE STEP	Capture a single type-specific computation step	non-generic computation step
GENERIC REWRITE STEP	Lift type-specific rewrite steps to the strategy level	generic computation step
TRAVERSAL	Instantiate a traversal scheme with (generic) rewrite steps	traversal behaviour *vs.* computation step
KEYHOLE OPERATION	Do not expose strategy types to the top level	strategic behaviour *vs.* simply-typed interface
SUCCESS BY FAILURE	Use a potentially failing computation to control traversal	traversal control
CIRCUITRY	Use composition and recursion to assemble strategies	control- and data-flow *vs.* computation steps
ROLE PLAY	Define a transformation as a pipeline of steps with designated roles	analyses *vs.* guards *vs.* side-effects *vs.* transformations
TRAVERSAL SCHEME	Capture traversal control in a fully generic, reusable combinator	purely generic traversal behaviour
LOCAL EFFECT	Do not expose monadic effects beyond where they are needed	effectful behaviour *vs.* effect-free interface
PROPAGATION	Propagate data downwards into the traversed tree	environment passing
GENERIC CONTAINER	Use a strategy as a generic data container	heterogeneous datatypes
TYPE ARGUMENT	Parameterise behaviour by a type argument	type-specific behaviour *vs.* specifically typed values
META SCHEME	Parameterise an algorithm by higher-order strategies	skeleton of a traversal scheme

Figure 6. Pattern intents; concerns that each pattern isolates or separates.

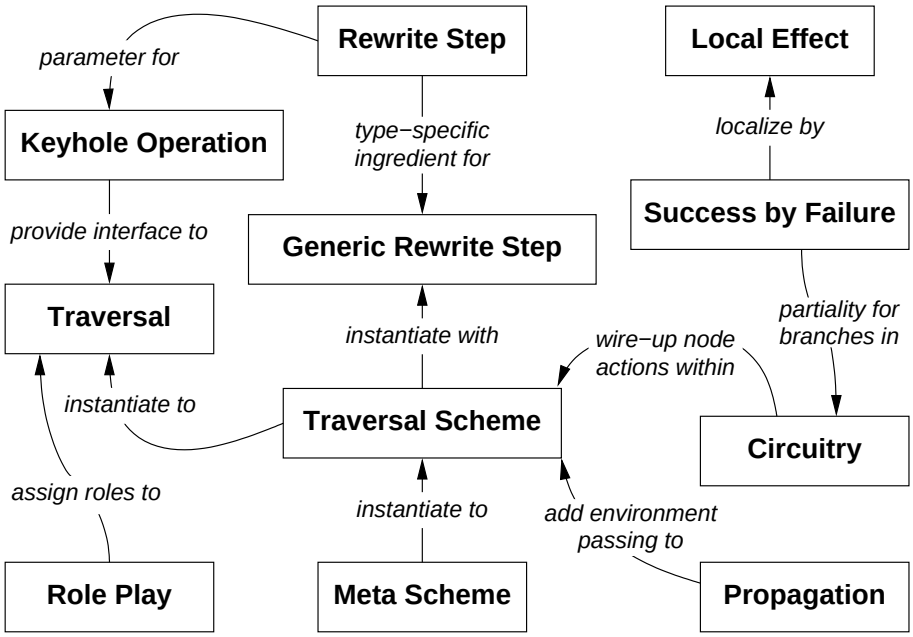

Figure 7. Some design pattern relationships.

The patterns in the REWRITING group are required reading for any strategic programmer. The INTERFACING and LOGIC groups are more sophisticated. These will bring the skills of a strategic programmer to a higher level. Finally, the PARAMETERISATION group deals with less frequent design problems. They are of interest for those who want to build, revise, or extend strategy libraries.

2. Each design pattern can be characterised by the kind of isolation and separation of concerns that can be accomplished with them. The table in Figure 6 contains a dedicated column. To give an example, the design pattern TRAVERSAL SCHEME isolates the concern of purely generic traversal behaviour in the sense that all type-specific behaviour will be supplied by the instantiation of a traversal scheme.

3. Another way to categorise the patterns is to focus on their interrelationships. Figure 7 gives a schematic overview (without claiming completeness). For instance, the PROPAGATION pattern describes how to add environment propagation to a TRAVERSAL SCHEME, which can be instantiated to a concrete TRAVERSAL.

REWRITE STEP — REWRITING

Intent

Capture a single type-specific computation step.

Motivation

Generic programming involves type-specific and generic functionality. By capturing type-specific computations and assigning a name to them, they can easily be reused in different contexts. A rewrite step is such a reusable piece of type-specific functionality.

Applicability

REWRITE STEP captures a computation that deals with data of a single type. To capture computations that deal with several types, use GENERIC REWRITE STEP instead.

Schema

$$\begin{array}{lcl} \mathit{step} & :: & T \to T' \\ \mathit{step\ pat} & = & \mathit{rhs} \\ \mathit{step\ v} & = & \ldots \end{array}$$

Description

Model a rewrite step with a unary function *step* on a specific term type T. The result type T' may or may not coincide with T, and may or may not be monadic. Define *step* with equations that pattern-match on the argument. If these equations are not exhaustive, then the function has to be complemented by a catch-all case. You can use the *Maybe* type constructor to indicate when *step* fails to fire. Alternatively, the catch-all case can return the input term itself, or a distinguished value, such as the empty list.

Sample code

Return the type constructor name from a type expression

$$\begin{array}{l} \mathit{refTypes} :: \mathit{HsType} \to [\mathit{HsName}] \\ \mathit{refTypes}\ (\mathit{HsTyCon}\ (\mathit{UnQual}\ n)) = [n] \\ \mathit{refTypes}\ \_ = [\,] \end{array}$$

Return the type constructor name from a type declaration

$$\begin{array}{l} \mathit{decTypes} :: \mathit{HsDecl} \to [\mathit{HsName}] \\ \mathit{decTypes}\ (\mathit{HsTypeDecl}\ \_\ n\ \_\ \_) = [n] \\ \mathit{decTypes}\ (\mathit{HsDataDecl}\ \_\ \_\ n\ \_\ \_\ \_) = [n] \\ \mathit{decTypes}\ (\mathit{HsNewTypeDecl}\ \_\ \_\ n\ \_\ \_\ \_) = [n] \\ \mathit{decTypes}\ \_ = [\,] \end{array}$$

The two above rewrite steps deal with name analysis for Haskell programs. They work on the abstract syntax of type expressions and declarations, respectively. The first step retrieves the name of a type constructor referred to by the type expression, if any. The second step retrieves the name of a type constructor declared in the given declaration, if any. We use lists of names as result type so that we are able to deal with cases where there is one type name, no type name, and potentially even several type names. In both cases, we need a catch-all case because obviously not all syntactical patterns are covered by the pattern-match cases. The catch-all cases simply return the empty list.

Consequences

By capturing relatively small pieces of type-specific behaviour in separate rewrite rules, reusable building blocks for larger strategic programs are identified.

Related Patterns

The construction of rewrite steps is a prerequisite for creating GENERIC REWRITE STEPS, but they can also be passed as actual parameters to KEYHOLE OPERATIONS.

GENERIC REWRITE STEP — REWRITING

Intent

Lift type-specific rewrite steps to the strategy level, making them applicable to terms of all types.

Motivation

Each individual rewrite step captures a computation that deals with data of a single type. At some point in the synthesis of generic programs, type-specific rewrite steps need to be made generic. This involves the composition of possibly several type-specific rewrite steps (for different types) and the indication of a generic default for all the types that are not covered by the type-specific computations. This entire composition is called a generic rewrite step.

Applicability

The non-generic rewrite steps composed into a generic one must be specific for *different* types. To compose rewrite steps that are specific for the same type, use the *choice* combinators, following SUCCESS BY FAILURE.

Schema

$$\mathit{poly} = \mathit{def}\ \text{`}\mathit{adhoc}\text{`}\ (\ldots\ s_1\ \ldots)\ \ldots\ \text{`}\mathit{adhoc}\text{`}\ (\ldots\ s_n\ \ldots)$$

Description

To compose rewrite steps $s_1, \ldots, s_n$ for different types into a strategy for *any* type, use (repeated application of) the *adhoc* combinators. Start from a default strategy *def* to deal with all types not covered by $s_1, \ldots, s_n$. Typical defaults are the strategies *fail*, *idTP*, or *constTU x*. Since strategies are preferably monadic entities in Strafunski, you have to make the specific rewrite steps monadic, if they are not yet by themselves. The *Identity* monad is the most basic choice.

Sample code

Return constructor names that are declared or referred to

$$\begin{array}{l} \mathit{anyTypes} :: \mathit{TU}\ [\mathit{HsName}]\ \mathit{Identity} \\ \mathit{anyTypes} = \mathit{constTU}\ [\,] \\ \qquad\qquad \text{`}\mathit{adhocTU}\text{`}\ (\mathit{return} \circ \mathit{decTypes}) \\ \qquad\qquad \text{`}\mathit{adhocTU}\text{`}\ (\mathit{return} \circ \mathit{refTypes}) \end{array}$$

In this sample we reuse the type-specific rewrite steps *decTypes* and *refTypes* that illustrate the REWRITE STEP pattern. These steps are specific to the types *HsDecl* and *HsType*. We combine them into a single generic rewrite step with *constTU* $[\,]$ as default. The composed function identifies type constructor names in a given term, both in declaration and use sites. The chosen generic default specifies that the empty list should be returned when faced with terms of any other type than *HsDecl* or *HsType*. The non-generic rewrite steps are made monadic by composing them with *return*. We have opted for the trivial *Identity* monad.

Consequences

By making a rewrite step generic, it can be applied to terms of any type, and it becomes amenable to specialisation with further type-specific behaviour.

Related Patterns

Lifting rewrite steps to the strategy level, i.e., turning them into generic rewrite steps is a prerequisite for passing them as arguments to a TRAVERSAL SCHEME, and for using them as building blocks in CIRCUITRY.

TRAVERSAL — REWRITING

Intent

Instantiate a traversal scheme with generic rewrite steps.

Motivation

Traversal is at the heart of strategic programming. Many schemes of traversal are readily available in Strafunski's combinator library. You can construct a traversal by passing your own rewrite steps to an appropriate predefined traversal scheme.

Schema

$$\begin{aligned} &\mathit{instantiation} = \mathit{scheme}\ \mathit{ap}_1\ \ldots\ \mathit{ap}_n \\ &\quad \textbf{where}\ \mathit{ap}_1 = \ldots\ \text{`}\mathit{adhoc}\text{`}\ \ldots \\ &\qquad\qquad \ldots \\ &\qquad\qquad \mathit{ap}_n = \ldots\ \text{`}\mathit{adhoc}\text{`}\ \ldots \end{aligned}$$

Description

To select an appropriate traversal scheme from the library you have to decide whether you need a type-unifying one (for analysis) or a type-preserving one (for transformation). Further, you have to decide on the desired order of traversal (e.g., top-down or bottom-up), whether the traversal should be cut-off below certain nodes (stop conditions), how to combine intermediate results, and more. These decision will usually lead to the identification of a library scheme. Furthermore, you have to identify the rewrite steps required to solve the problem. These rewrite steps are usually generic, and they serve as the actual parameters $\mathit{ap}_1, \ldots, \mathit{ap}_n$ that *instantiate* the traversal *scheme*.

Sample code

Collect all type constructor names from a given term

$$\begin{aligned} &\mathit{allTypes} :: \mathit{TU}\ [\mathit{HsName}]\ \mathit{Identity} \\ &\mathit{allTypes} = \mathit{crush}\ \mathit{anyTypes} \end{aligned}$$

Using the predefined combinator *crush* with type:

$$\mathit{crush} :: (\mathit{Monad}\ m, \mathit{Monoid}\ u) \Rightarrow \mathit{TU}\ u\ m \rightarrow \mathit{TU}\ u\ m$$

The strategy *allTypes* uses the generic rewrite step *anyTypes*, that illustrates the GENERIC REWRITE STEP pattern, to collect all possible declaring and referring occurrences of type constructor names. For this purpose we selected the traversal scheme *crush*, which performs deep reduction in top-down order over the entire input term (no cut-off). The intermediate results are combined via a *Monoid*'s binary operator. Since we use lists as a monoid, this binary operator will be resolved to the append operator "++". The resulting traversal can be applied to terms of any type.

The introductory example in Figure 1 provides another example. The chosen traversal scheme is *topdown*. The basic rewrite step *inc* is nothing more than increment for integers.

Related Patterns

If you discover that the particular traversal scheme you need is not present in Strafunski's library, then you may consult TRAVERSAL SCHEME to find out how to roll your own. Traversals can be provided via a simple, non-strategic interface by a KEYHOLE OPERATION, and they can be used to fulfil various roles in a ROLE PLAY.

KEYHOLE OPERATION — INTERFACING

Aka

Wrapper Worker

Intent

Do not expose strategy types to the top level.

Motivation

When strategies serve directly as argument or result, this can be observed via the types *TP* and *TU*. The resulting functions are subject to function application via *applyTP* and *applyTU*. If you want to use strategies without exposing them to the outside, you can use a keyhole operation. On the inside, you can work with the full power of strategies, while on the outside, all you see is a plain function without any trace of *TP* or *TU*.

Schema

$$\begin{aligned} &\mathit{wrapper}\ \mathit{fp}_1\ \ldots\ \mathit{fp}_n = \ldots\ \mathit{apply}\ \mathit{worker}\ \ldots \\ &\quad \textbf{where}\ \mathit{worker} = \ldots\ (\ldots\ \text{`}\mathit{adhoc}\text{`}\ \mathit{fp}_1)\ \ldots \\ &\qquad\qquad\qquad\qquad \ldots \\ &\qquad\qquad\qquad\qquad \ldots\ (\ldots\ \text{`}\mathit{adhoc}\text{`}\ \mathit{fp}_n)\ \ldots \end{aligned}$$

Description

Divide the functionality of your algorithm over a top-level *wrapper* function which directly operates on terms, and a nested *worker* strategy. Use *adhoc* when specific argument strategies are used in the definition of the worker. Use *apply* to define the *wrapper* in terms of the generic *worker*.

Sample code

Check whether a Haskell type constructor is fresh

$$\begin{aligned} &\mathit{isFreshType} :: \mathit{HsName} \rightarrow \mathit{HsModule} \rightarrow \mathit{Bool} \\ &\mathit{isFreshType}\ n = \mathit{runIdentity} \circ \mathit{applyTU}\ \mathit{worker} \\ &\quad \textbf{where} \\ &\qquad \mathit{worker} = \mathit{allTypes}\ \text{`}\mathit{before}\text{`}\ \mathit{isNotElem} \\ &\qquad \mathit{isNotElem} = (\neg) \circ \mathit{elem}\ n \end{aligned}$$

General focus selection

$$\begin{aligned} &\mathit{selectFocus} :: (\mathit{MonadPlus}\ m, \mathit{Term}\ f, \mathit{Term}\ t) \\ &\qquad\qquad \Rightarrow (f \rightarrow m\ f) \rightarrow t \rightarrow m\ f \\ &\mathit{selectFocus}\ \mathit{getFocus} = \mathit{applyTU}\ \mathit{worker} \\ &\quad \textbf{where} \\ &\qquad \mathit{worker} = \mathit{select}\ (\mathit{adhocTU}\ \mathit{failTU}\ \mathit{getFocus}) \end{aligned}$$

The operation *isFreshType* implements a predicate to test if a certain type constructor name *n* is not yet used in a given Haskell module. Note that this is a completely monomorphic function. The wrapped worker is basically the traversal *allTypes* that illustrates the TRAVERSAL pattern but postfixed by a membership test *isNotElem*. After wrapping the worker we further post-process the result with $\mathit{runIdentity} :: \mathit{Identity}\ a \rightarrow a$ to get out of the trivial *Identity* monad.

The second example above deals with selection of terms from a focus where the helper *getFocus* for focus detection is passed to a keyhole operation. Internally, the traversal scheme *select* is used.

Consequences

With a keyhole operation you fit a non-strategic interface on strategic functionality. This means you can apply the wrapper with ordinary function application instead of using *apply*, and you can pass monomorphic function arguments to the wrapper. On the other hand, if you want to pass the generic functionality to a traversal scheme, or update it with *adhoc*, you will have to go around the wrapper and use the worker directly.

SUCCESS BY FAILURE — LOGIC

Intent

Use a potentially failing computation to control traversal.

Motivation

To correctly implement certain traversals, their constituent rewrite steps should only be performed under certain conditions. For instance, a particular rewrite step should trigger only if another does not, or vice versa. To model success and failure of rewrite steps and strategies, you can use the *Maybe* monad or a backtracking monad. Generic failure is captured by the *fail* combinators, and the *choice* combinators allow you to recover from failure.

Schema

```
partial = gstep `choice` ...
  where
    gstep    =  fail `adhoc` step
    step pat =  return ...
    step v   =  mzero
```

Description

A *partial* strategy, i.e., one that potentially fails, is typically constructed from type-specific rewrite *step*s that use the *mzero* of a *MonadPlus* to encode failure. When lifting such a partial *step* to the strategy level with *adhoc*, the generic *fail* combinator is used as default strategy. Finally, the *choice* combinator is used to combine potentially failing generic rewrite steps.

Sample code

The library scheme for selection

```
select :: MonadPlus m ⇒ TU a m → TU a m
select s = s `choiceTU` (oneTU (select s))
```

Identify different kinds of type constructor names

```
decCon = (adhocTU failTU typeCon)
           `choiceTU`
           (adhocTU failTU dataCon)
  where
    typeCon (HsTypeDecl _ n _ _) = return n
    typeCon _ = mzero
    dataCon (HsDataDecl _ _ n _ _ _) = return n
    dataCon _ = mzero
```

A prime example of a partial strategy combinator is *select*. Its argument strategy is meant for the identification of selectable entities. This process must be necessarily partial. A *choice* is used in the definition of *select* because selection can recover from failure of identification for a given node by recursing into the children. If the identification strategy fails at all levels, selection will altogether fail.

The second sample illustrates the use of potentially failing strategies to merge rewrite steps that are specific for the same type. The partial rewrite steps *typeCon* and *dataCon* are both specific for type *HsDecl*. The *decCon* strategy reverts to the second if the first fails.

Related Patterns

If you want to prevent the monadic effects of partiality or nondeterminism to invade parts of your code that do not rely on them, you may want to use LOCAL EFFECT.

CIRCUITRY — LOGIC

Intent

Use composition and recursion to assemble strategies into a composite traversal with appropriate control and data flow.

Motivation

When composing traversals, one should take care to sequence the ingredient steps in the right order, to pass data to the steps that need them, and to traverse the appropriate parts of the input term. By connecting your steps with appropriate combinators and recursive calls you can wire up the control and data flow between them.

Schema

```
circuit = comp_0 `co_1` comp_1 ... `co_n` comp_n
```

Description

There are two kinds of circuitry. Depending on whether you need the circuit itself to be a strategy or not, you may either compose strategies, or keyhole operations and other monadic functions. When composing strategies, the composition operators co_i are typically *pass*, *seq*, and *choice* combinators. The *seq* operators are used to prefix a strategy with a type-preserving strategy. The *pass* operators are used to compute a value via a type-unifying strategy and to pass it on. The *choice* operators are meant for branching control-flow. When composing keyhole operations, the monadic bind operator "$\gg=$", or do-notation are used as composition operators. All the composed components potentially include recursive references to *circuit*.

Sample code

Compute free variables in a given Haskell fragment

```
freeHsVars :: TU [HsName] Identity
freeHsVars = refHsVars `passTU` λrefs →
               decHsVars `passTU` λdecs →
               allTU freeHsVars `passTU` λfrees →
               constTU (union frees refs \\ decs)
  where
    refHsVars, decHsVars :: TU [HsName] Identity
    refHsVars = adhocTU (constTU []) (return ∘ step)
      where
        step (HsVar (UnQual n)) = [n]
        step _ = []
    decHsVars = ...
```

The above strategy performs free variable analysis on arbitrary Haskell program fragments. Free variables are obtained by subtracting (cf. "\\") the locally declared variables *decs* from the *union* of the locally referenced variables *refs* and the free variables *frees* from the subterms. We use two generic rewrite steps *refHsVars* and *decHsVars* for the identification of declaring and referring occurrences of Haskell variables. The strategy is recursively defined to descend into terms via *allTU*. The *passTU* combinator is used to connect all the type-unifying computations.

Related Patterns

The SUCCESS BY FAILURE pattern shows how partiality of strategies can be modelled, and how it can be used to realize branches in the control and data-flow between your strategic components. The ROLE PLAY pattern suggests how to distribute responsibilities among the strategic components of your circuitry.

ROLE PLAY — LOGIC

Intent

Define a transformation as a pipeline of steps with designated roles.

Motivation

A transformation can usually be decomposed into separate steps which play sharply delimited roles. Typical steps are analyses, guards, side effects, and atomic transformations. When each step has its own limited responsibilities, it becomes easier to construct, understand, and modify the transformation. The individual steps can be formed into a complete transformation pipeline with (sequential) composition.

Schema

$transformation = role_0\ `co_1`\ role_1\ \ldots\ `co_n`\ role_n$

Description

Decompose the transformation task you need to implement into basic roles. An *analysis* is type-unifying, and does not modify its input term. A *guard* checks whether a particular condition is satisfied by its input term. It is typically implemented as a Boolean expression wrapped by the *guard* function, or a strategy of type $TU\ ()\ m$, where the monad m supports partiality. Side effects are realized by access to an extended monad interface for a state. Atomic *transformation steps* are type-preserving. Depending on whether you need the transformation pipeline itself to be a strategy, you may wire up the steps of the transformation by strategic or simply monadic CIRCUITRY.

Sample code

Replace a focused type expression by a type synonym

```
toAlias :: HsName → HsModule → Maybe HsModule
toAlias n m =
  do t ← selectTypeFocus m      -- analysis
     t' ← getAlias n m          -- analysis
     guard (t ≡ t')             -- guard
     replaceTypeFocus n m       -- atomic transformation
```

The *toAlias* pipeline implements a simple refactoring for Haskell datatypes. Assuming that a focus has been placed on some type expression t, we want to replace t by a type synonym (or alias) named n. A precondition for this replacement is that n is defined as t in the given Haskell module m. The transformation's circuitry is implemented as a sequence of KEYHOLE OPERATIONS and a simple guard. For brevity, we do not show the definitions of the keyhole operations. Firstly, we look up the type expression t from the focus via *selectTypeFocus*. This is an analysis. Secondly, we look up the right-hand side expression t' from the declaration for n via *getAlias*. This is again an analysis. Then, we place a guard to enforce that the focused type expression t actually coincides with t'. Finally, we perform the actual transformation that replaces the focused type expression by a reference to n via *replaceTypeFocus*.

Related Patterns

The CIRCUITRY pattern explains how to wire the data and control flow between the individual steps of a pipeline.

TRAVERSAL SCHEME — PARAMETERISATION

Aka

Abstract Algorithm, Recursion Scheme.

Intent

Capture traversal control in a fully generic, reusable strategy combinator, which abstracts over any type-specific operations.

Motivation

The traversal behaviour of many traversals can be captured in a reusable traversal *scheme*. To ensure its reusability, type-specific computations should not be hard-wired into it, but should rather be supplied via appropriate parameters. Thus, a traversal scheme captures generic traversal behaviour in an abstract algorithm.

Schema

$scheme\ fp_1\ \ldots\ fp_n =$
$\quad \ldots\ fp_1\ \ldots\ fp_n\ \ldots\ (scheme\ fp_1\ \ldots\ fp_n)\ \ldots$

Description

Divide your algorithm into a fully generic *scheme* with formal parameters for type-specific computations. These parameters are either of strategy types, or they are of monomorphic function types subject to type-specific customisation via *adhoc* inside the traversal scheme. The traversal scheme by itself does not involve any type-specific functionality.

Sample code

Generic free name analysis

```
freeNames :: Eq n
          ⇒ TU [n] Identity
          → TU [n] Identity
          → TU [n] Identity

freeNames refNames decNames = fnames
  where
    fnames = refNames `passTU` λrefs →
             decNames `passTU` λdecs →
             allTU fnames `passTU` λfrees →
             constTU (union frees refs \\ decs)
```

Instantiations

```
freeHsVars = freeNames refHsVars decHsVars
freeHsTVars = freeNames refHsTVars decHsTVars
freeJaVars = freeNames refJaVars decJaVars
```

The combinator *freeNames* captures that free names (e.g., variables) in a given program fragment can be collected by a a strategy which looks up the names from all the relevant patterns dealing with names in the given language. While *freeNames* is language-independent, the sample code for the CIRCUITRY pattern defines a Haskell-specific free variable analysis. The combinator *freeNames* is parameterised by type-specific (say, language- or syntax-dependent) ingredients of the traversal. By supplying appropriate actual parameters for recognition of referred and declared names, we can obtain different concrete name analysis algorithms, e.g., for free Haskell variables, free Haskell *type* variables, or free Java variables.

Examples of simpler traversal schemes are the predefined combinators of Strafunski's *TraversalTheme* (see Figure 5).

Related Patterns

Instantiation of a traversal scheme to synthesise an actual traversal is described in the TRAVERSAL pattern.

LOCAL EFFECT — INTERFACING

Intent

Do not expose monadic effects beyond where they are needed.

Motivation

Effects such as partiality, non-determinism, and state can be used in strategic programming by employing appropriate (stacked) monads. Often, such effects are only needed locally. With a local effect you can prevent locally needed monads to pollute the rest of your program.

Schema

```
effectful :: TP MEffect
effectful = ... effect ...

effectless :: TP M
effectless = msubstTP m2m effectful
  where m2m :: MEffect a → M a
        m2m = return ∘ runME
```

Description

Implement the functionality that requires an effect in a combinator that exposes the corresponding monad *MEffect*. Call this *effectful* combinator from a second combinator that exposes a different monad *M*, without the effect. Use the *msubst* combinator to substitute one monad by the other, using a function *m2m* that runs the effectful computation and returns its value inside the monad without effect. Instead of using unrelated monads, you can construct *MEffect* by applying a monad transformer to *M*. The function *m2m* should then 'unlift' the transformed monad to recover the original monad.

Sample code

Localize a state transformer

```
localStT :: Monad m ⇒ s → TP (StateT s m) → TP m
localStT s
  = msubstTP unlift
    where unlift tm = evalStateT tm s
```

Replace all strings with a deBruijn index

```
deBruijn :: Monad m ⇒ TP m
deBruijn
  = localStT "1" (topdown poly)
    where poly = adhocTP idTP mono
          mono = λ_ → do n ← get
                         put (n ++ "'")
                         return n
```

The *deBruijn* strategy replaces all Strings in a given input term by unique identifiers, starting with `"1"`, and then adding a prime at each step. Internally, a state monad transformer is used to keep track of the most recently generated identifier. Externally, i.e., looking at the type of *deBruijn*, there is no trace of this state monad. This is accomplished with the *localStT* combinator, which converts a strategy that employs a monad with state transformer into a strategy on the same monad, without the transformer. The parameter *s* represents the initial state. The conversion is accomplished by calling *msubstTP* with a function *unlift* that evaluates the state transformer initialised with *s*.

Consequences

Localising a monadic effect can improve not only readability of your code. The performance of your program may benefit as well.

PROPAGATION — PARAMETERISATION

Aka

Hand Me Down, Transportation

Intent

Propagate data downwards into the traversed tree.

Motivation

What you do with lower nodes in the tree might be dependent on information collected or constructed at higher nodes. With PROPAGATION, such information is handed down via a parameter of the recursive call of a traversal.

Schema

```
traversal :: e                    -- initial data
          → (e → TU e m)          -- data modifier
          → (e → S)               -- node action
          → S                     -- traversal

traversal e modify action
  = ...
    action e
    ...
    modify e `pass` λe' →
    traversal e' action modify
    ...
```

Description

To add data propagation behaviour to a traversal, you should first parameterise the node action(s) of your traversal with the type *e* of this data. Furthermore, you should add two parameters to your *traversal*: one parameter for the initial data to start the traversal with, and another parameter for the function to modify the data at each step downward during traversal. The latter function takes the current data and the current node as input, and it computes new data. At each node, three things happen. Firstly, the node *action* is applied, using the current data *e*. Secondly, the current data is modified. Thirdly, the new data is used in a recursive call of the complete *traversal*.

Sample code

A propagating version of the traversal scheme *select*

```
selectenv :: MonadPlus m
          ⇒ e → (e → TU e m) → (e → TU a m)
          → TU a m

selectenv e s' s = (s e)
                   `choiceTU`
                   (s' e `passTU` λe' →
                    oneTU (selectenv e' s' s))
```

The shown strategy combinator unites propagation with selection. It is an elaboration of the simpler traversal scheme *select*. It uses *choiceTU* and *oneTU* in the same manner as *select* (see Figure 5). All the additional behaviour directly implements the PROPAGATION pattern. The *selectenv* combinator is used in program analyses when a type-unifying node processor relies on environment propagation, e.g., to maintain bound variables along the way down to a focused fragment. For other predefined traversal schemes, a propagating version can be given in a similar way.

Related Patterns

Instead of using Propagation, you might use a reader monad to propagate information down the tree. The LOCAL EFFECT pattern explains how to keep the monadic propagation effect local.

GENERIC CONTAINER — INTERFACING

Intent

Use a strategy as a generic data container.

Motivation

Sometimes terms of different types need to be stored in the same container. Such a generic container can be modelled with strategies.

Schema

```
type GC = ... S ...
emptyGC :: GC
emptyGC = ...
addGC :: Term t ⇒ t → GC → GC
addGC t c = ... modify c t ...
elemGC :: Term t ⇒ t → GC → Bool
elemGC t c = ... apply c t ...
```

Description

Just as monomorphic functions can be used to represent homogeneous data structures such as maps and sets, strategies can be used as heterogeneous data structures. Define your generic container type as a data structure that involves a strategy type *S*. Define the operations on your container in terms of strategy combinators. Operations modifying a container involve function modification lifted to the strategy level. Looking up data from a container involves strategy application.

Sample code

A generic container for assigning integer codes to terms.

```
type Coder = (Int, TU Int Maybe)
noCode :: Coder
noCode = (0, failTU)
getCode :: Term x ⇒ Coder → x → Maybe Int
getCode (_, s) = applyTU s
setCode :: (Term x, Eq x) ⇒ Coder → x → Int → Coder
setCode (i, s) x i' = (i, modifyTU s x (return i'))
nextCode :: Coder → (Int, Coder)
nextCode (i, s) = (i, (i + 1, s))
enCode :: (Term x, Eq x) ⇒ Coder → x → Coder
enCode c x = maybe gen found (getCode c x)
  where
    gen = let (i, c') = nextCode c in setCode c' x i
    found = const c
```

The type *Coder* assigns unique integers to terms of arbitrary types. It contains a counter as first component that records the highest code issued so far. The second component is a type-unifying strategy which represents the mapping from terms to codes assigned so far. We can provide an initial coder with *no* codes assigned, *get* a code of a term, *set* the code for a term, and generate the *next* code. When a given term is *enCode*d, the *Coder* is only modified if no code was previously assigned to the term.

Consequences

Generic containers can be made observable only per type, i.e., element retrieval or enumeration can only be done if the types of the elements are provided as input.

Related Patterns

Container operations are usually KEYHOLE OPERATIONS.

TYPE ARGUMENT — PARAMETERISATION

Intent

Parameterise behaviour by a type argument.

Motivation

Sometimes, you want your strategy to display type-specific behaviour even though it does not directly consume or produce any values that involve this specific type. By adding a type argument you can specify your type of choice.

Schema

```
type TypeArg a = ...
typeArg = ...
foo :: TypeArg a → S
foo ta = ... ta ...
fooT :: S
fooT = foo (typeArg :: TypeArg T)
```

Description

Type arguments have to be modelled as value arguments. The challenge is to prevent having to supply a value of the intended type. Type arguments can be modelled in various ways. In general, you need a dedicated type constructor *TypeArg*, and you need an actual representation of the type argument, say, *typeArg*. A strategy *foo* which is controlled by a type argument then takes an argument of type *TypeArg a*. The strategy employs the type argument *ta* to internally disambiguate unresolved polymorphism. An actual instance *fooT* of *foo* will simply construct the appropriate type argument by type annotation.

Sample code

A tick combinator for counting with a type argument

```
type TypeGuard a = a → ()
typeGuard = const ()
typeTick g
  = adhocTU (constTU 0) ((λ() → return 1) ∘ g)
```

Count subterms of type *HsDecl*

```
countHsDecls :: HsModule → Int
countHsDecls = runIdentity ∘ applyTU worker
  where
    worker = crush poly
    poly = typeTick (typeGuard :: TypeGuard HsDecl)
```

Here we opt for type guards as type arguments, that is, functions with the intended type as domain, and *()* as co-domain. The *typeTick* combinator is controlled by a type guard which it uses internally to disambiguate the polymorphism of the non-generic argument of *adhocTU*. In fact, *typeTick* returns 1 if a term of the intended type is encountered, and 0 otherwise. The function *countHsDecls* counts all Haskell declarations within the given Haskell module. It is structured as a keyhole operation around an instantiation of *typeTick* that takes *HsDecl* as actual type argument. The type-unifying traversal scheme *crush* is used to traverse an input term with the instantiated type argument. Counting via *crush* relies here on the fact that all numeric types (class *Num*) instantiate the *Monoid* class.

Related Patterns

Type arguments can be used for strategies that perform per-type element retrieval or enumeration on GENERIC CONTAINERS.

META SCHEME PARAMETERISATION

Intent

Parameterise an algorithm by higher-order strategies.

Motivation

The most basic and common way in which strategy combinators are parameterised is by plain strategies, i.e., constant combinators. This kind of parameterisation is heavily used for traversal schemes to separate out type-specific behaviour for node processing. A more flexible algorithm can be obtained if a meta-scheme is established, i.e., when some parameters are non-constant combinators themselves. This allows you to vary, for instance the TRAVERSAL SCHEME employed by an algorithm or the composition operators of its CIRCUITRY.

Schema

$$\mathit{meta}\ b_1\ \ldots\ b_n\ u_1\ \ldots\ u_m\ s_1\ \ldots\ s_k =$$
$$\quad \ldots\ \mathit{meta}\ \ldots\ \ldots$$

Description

Parameterise your strategy definition by strategy combinators. There is potential for binary combinator arguments $b_1, \ldots, b_n$, unary combinator arguments $u_1, \ldots, u_m$, and plain strategy arguments $s_1, \ldots, s_k$. Combinators with more than two arguments are possible as well but note that all basic strategy combinators and most library schemes are unary or binary. You can introduce parameters for aspects such as traversal control, data-flow and control-flow. When you instantiate the higher-order parameters of a meta-scheme, you are turning it into a plain traversal scheme.

Sample code

A meta scheme for traversal

$$\mathit{traverse}\ o\ t\ s \quad = \quad s\ `o`\ t\ (\mathit{traverse}\ o\ t\ s)$$

Traversal schemes by instantiation

$$\begin{array}{ll} \mathit{topdown} & = \mathit{traverse}\ \mathit{seqTP}\ \mathit{allTP} \\ \mathit{bottomup} & = \mathit{traverse}\ (\mathit{flip}\ \mathit{seqTP})\ \mathit{allTP} \\ \mathit{crush} & = \mathit{traverse}\ (\mathit{liftop2}\ \mathit{mappend})\ \mathit{allTU} \\ \mathit{select} & = \mathit{traverse}\ \mathit{choiceTU}\ \mathit{oneTU} \end{array}$$

The combinator *traverse* is a highly parameterised traversal scheme. It is parameterised in a binary combinator *o* for the composition of node processing with recursive descent. It is further parameterised in a unary combinator *t* to control the traversal in the sense of how to descend into subterms. Finally, *traverse* carries a nullary strategy argument *s* for node processing. These three parameters provide a good example of the different kinds of parameters that are feasible in a meta-scheme. The argument *s* isolates type-specific behaviour whereas *o* and *t* parameterise the meta-scheme with respect to the combination of intermediate results (i.e., data-flow) and traversal behaviour.

The meta-scheme is illustrated with a number of instantiations that reconstruct traversal schemes *topdown*, ..., *select* from Strafunski's library (see Figure 5). As an aside, note that *traverse* is still completely neutral with respect to type-unification vs. type-preservation. We could even instantiate the parameters of *traverse* without commitment to *TP* or *TU* based on Strafunski's *OverloadingTheme*. Then, we use *all*, *one*, *choice*, ... instead of the postfixed counterparts.

Related Patterns

By appropriate instantiation of the combinator arguments of a meta-scheme, you can obtain various TRAVERSAL SCHEMES.

5 Conclusion

Contribution

We have advocated design patterns to capture and communicate functional programming expertise, and we have proposed a format for their description. We have identified basic and more advanced design patterns for generic programming with functional strategies, and we have described them in our format. These patterns capture expertise in deploying the strategy combinators of *Strafunski*'s combinator library. Each pattern is motivated, explained and illustrated with detailed sample code. We have organised the catalogue of strategic design patterns by indicating their interrelationships and articulating the concerns they help to address and separate.

Evaluation

To properly assess the value of our pattern catalogue and of functional design patterns in general, comparative empirical studies would have to be conducted. One would need to compare productivity of programming teams or communities that use design patterns and those that do not. Pending such studies, we can make the following (subjective) observations. We have found the proposed design pattern format a quite natural instrument for bringing together bits of design information that are otherwise scattered over language manual, library documentation and example applications. The catalogue can be consulted efficiently due to its conciseness and non-linearity, and due to the fixed items per pattern. The strategy design patterns themselves have helped to structure our conversations and thoughts about strategic programming, and they even provided guidance in structuring the combinator library.

Related work

Object-oriented design patterns We have taken our inspiration from the literature on object-oriented design patterns [5]. As indicated in Section 4, we have made some modifications to accommodate the characteristics of functional programming. A general comparison reveals further differences. The object-oriented design patterns are predominantly concerned with code organisation, distribution of responsibilities over classes and objects, tuning dependencies to maximise variability and maintainability. The functional design patterns are more concerned with issues of parameterisation, reusability, composability, and behaviour. We conjecture that these differences are (partly) due to the available abstraction mechanisms in both paradigms. As an aside, functional programming idioms have served as a source of inspiration for the formulation of some object-oriented design patterns [10].

Generic programming In previous work [12, 11, 21], we have discussed other approaches to generic programming, including polytypic programming [9] and adaptive programming [15]. To summarise, none of these approaches simultaneously realizes all defining characteristics of strategies. As a result, not all benefits of strategic programming are attained. Also, our approach to genericity is more lightweight, and it is intended as a general-purpose programming technique. The use of design patterns has been proposed for adaptive programming, and sketches of a handful of patterns can be found at [14]. As for polytypic programming, patterns might prove equally appropriate, although none have been proposed yet.

Strategic programming Several of our patterns are applicable to strategic term rewriting à la Stratego [20] since we derived functional strategies from this idiom. On the other hand, several

of our patterns involve technical aspects of the *Strafunski* style such as type case, monads, or more generally higher-order typed functional programming. This implies that some of our patterns would have to be re-interpreted in the context of strategic term rewriting. The dependence of design patterns on a particular programming language and on certain favoured encoding techniques is not unusual. Some of the folklore "Gang-of-Four" patterns also need to be re-interpreted for object-oriented programming languages which favour multiple dispatch, an object-based as opposed to a class-based approach, first-class methods, and others.

Future work

Further functional design patterns Our pattern catalogue for functional strategic programming, as supported by *Strafunski* and its combinator library is by no means complete. In fact, we use further patterns in our strategic programs, and so we plan to extend our catalogue accordingly. Similar pattern catalogues could be developed for other combinator libraries. Design patterns would also be appropriate to capture expertise regarding sophisticated corners of functional programming, such as strictification, existential types, and parallelism. More generally, even general purpose functional programming could benefit from catalogues of design patterns, in the same way as the folklore "Gang-of-Four" patterns address general purpose object-oriented programming rather than any specific application domain.

Functional program refactoring Further inspiration can be taken from the object-oriented literature. The notion of *refactoring* [16, 4] seems particularly helpful. This is also proposed in [19]. The style and vocabulary employed in our functional pattern catalogue can serve as a starting point for the elaboration of a catalogue of refactorings for functional programs. In some pattern descriptions we have already hinted at how one design can be transformed into another, e.g., to extract a TRAVERSAL SCHEME from an application-specific TRAVERSAL. The formal foundations of functional program transformation are reasonably well-understood [1, 17] but a proper catalogue of refactorings that is convenient for the adaptation of functional programs is a challenging subject for future work.

6 References

[1] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

[2] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Language Descriptions, Tools and Applications (LDTA 2001).

[3] A. v. Deursen and J. Visser. Building program understanding tools using visitor combinators. In *10th International Workshop on Program Comprehension (IWPC 2002)*. To Appear, 2002.

[4] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[7] G. Hutton and E. Meijer. Functional pearl: Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[8] P. Jansson and J. Jeuring. PolyLib—A library of polytypic functions. In R. Backhouse and T. Sheard, editors, *Proc. of WGP'98, Marstrand, Sweden, 18 June 1998*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

[9] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, Berlin, 1996.

[10] T. Kühne. *A Functional Pattern System for Object-Oriented Design*. Verlag Dr. Kovač, Hamburg, Germany, 1999. PhD thesis, Darmstadt University of Technology.

[11] R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming – An inquiry into trans-paradigmatic genericity. Draft; Available from the authors' web site, 2002.

[12] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C. Ramakrishnan, editors, *Proc. of PADL 2002, Portland, OR, USA*, volume 2257 of *LNCS*. Springer-Verlag, Jan. 2002.

[13] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343. ACM Press, 1995.

[14] K. Lieberherr. Patterns for Adaptive Programming, 1997–2001. `http://www.ccs.neu.edu/research/demeter/adaptive-patterns/AOP/`.

[15] K. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, July 1997.

[16] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[17] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[18] M. Sage. FranTk: a declarative GUI language for Haskell. In *Proceedings of the International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 106–117. ACM Press, Sept. 18–21 2000.

[19] S. Thompson and C. Reinke. Refactoring Functional Programs. Technical Report 16-01, Computing Laboratory, University of Kent at Canterbury, October 2001.

[20] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98)*, volume 34.1 of *ACM Sigplan Notices*, pages 13–26. ACM Press, Sept. 1998.

[21] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices, OOPSLA 2001 Conference Proceedings*, 36(11):270–282, Nov. 2001.

[22] N. Winstanley. Derive User Guide, version 1.0. Available at `http://www.dcs.gla.ac.uk/~nww/Derive/`, 1997.