Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 135–153
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Programming Errors in Traversal Programs Over Structured Data

Ralf Lämmel

University of Koblenz-Landau

Simon Thompson

University of Kent

Markus Kaiser

University of Koblenz-Landau

Abstract

Traversal strategies provide an established means of describing automated queries, analyses, transformations, and other non-trivial computations on deeply structured data (including, most notably, data representations of software artifacts such as programs). The resulting traversal programs are prone to programming errors. We are specifically concerned with errors that go beyond classic type errors, in particular: (i) divergence of traversal, (ii) unintentional extent of traversal into data, (iii) trivial traversal results, (iv) inapplicability of the constituents of a traversal program along traversal. We deliver a taxonomy of programming errors, and start attacking some of them by refinements of traversal programming.

Keywords: Traversal strategies, Traversal programming, Term rewriting, Stratego, Strafunski, Generic programming, Scrap Your Boilerplate, Type systems, Static program analysis.

1 Introduction

Consider the general problem domain of extracting data from program or data representations (such as ASTs) as well as transforming such representations in a systematic fashion. Over the last 10 years, this problem domain has triggered advances in term rewriting and general-purpose programming

doi:10.1016/j.entcs.2009.09.045

with rewriting-like capabilities [22,23,13,24,21] so that traversals (perhaps even highly reusable traversal strategies) are *programmable*.

Despite these advances, the use and the definition of programmable traversal strategies has remained the domain of the expert, rather than gaining wider usage. This could in part be due to necessary language, library, and tool support, but we contend that the principal obstacle to wider adoption is the severity of some possible pitfalls, which make it difficult to use strategies in practice. Some of the programming errors that arise are familiar, e.g., type errors, but other errors are of a novel nature. Their appearance can be off-putting to the newcomer to the field, and it can limit the productivity even of experienced strategists.

This paper is a first step in a programme which aims to make strategic programming more accessible and approachable through providing an introduction to — and indeed a taxonomy of — some of the common pitfalls of strategic programming. We also begin refining strategic programming so that the next generation of strategic programming may be considerably easier to use.

A running example. To use a purposely simple example, consider the transformation problem of "incrementing all numbers in a term". (Clearly, programming errors become more severe with increasing the problem size.) Suppose ℓ is the rewrite rule that maps any given number n to $n+1$. It remains to compose a strategy that can essentially iterate ℓ over any term. Here is an indication of some of the things that may go wrong with the application of the composed strategy:

- It fails to terminate.
- It fails to find numbers in the input term.
- It fails, i.e., it returns a trivial failure term.
- It increments some numbers in the input term more than once.

Structure of the paper. § 2 quickly introduces a suitable model of (Haskell-based) traversal strategies.[1] § 3 takes an inventory of programming errors in traversal programming. § 4 attacks the errors by proposing some refinements of traversal programming. § 5 briefly applies the notion of static analysis to the problem of determining properties of traversal programs as

[1] Haskell in all its glory has infinite and partial data structures, such as trees with undefined leaves, or indeed undefined subtrees. In the presence of infinite and partial structures, the discussion of strategy semantics and properties (most notably, termination) becomes more subtle. We are currently limiting our discussion to finite, fully defined data. (The subject of coinductive strategies over coinductive types may be an interesting topic for future work.)

well as detecting errors therein. § 6 discusses related work. § 7 concludes the paper.

2 Strategic programming

We assume basic strategy combinators as they were pioneered by the Stratego language [22]: "*id*" — the always succeeding strategy returning the input term as is; "*fail*" — the always failing strategy; "*sequ* s s'" — sequential composition of s and s'; "*choice* s s'" — try s first, and try s' second, if s failed; "*all* s" — apply s to all immediate subterms of a given term, and fail if there is any subterm for which s fails; "*one* s" — apply s to the leftmost immediate subterm of a given term such that the application does not fail, and fail if there is no such immediate subterm. The combinators *all* and *one*, when used recursively, enable the key capability of strategic programming: traversal arbitrarily deeply into terms.

Let us define some recursive strategy combinators that model traversal schemes as they appear in the literature [22,8,19]. The following folklore schemes are actually written in Haskell syntax, subject to an embedding of the aforementioned strategy combinators into Haskell. [2]

```
-- Rewrite root first, then recurse into all immediate subterms of intermediate result
full_td s = sequ s (all (full_td s))
-- Rewrite all subterms in bottom-up manner
full_bu s = sequ (all (full_bu s)) s
-- Try to rewrite root; upon success: cease; upon failure: recurse into all immediate subterms
stop_td s = choice s (all (stop_td s))
-- Find a subterm to be rewritten in bottom-up manner; rewriting ceases upon success
once_bu s = choice (one (once_bu s)) s
-- Repeat once-bottom-up traversal until it fails
innermost s = repeat (once_bu s)
 where
  repeat s = try (sequ s (repeat s))
  try s = choice s id
```

Without loss of generality, we use a Haskell model based on the SYB approach to generic programming [11]. Hence, first-order strategies are revealed as polymorphic functions on "term types":

```
type Strategy = forall x. Data x => x -> Maybe x
```

The use of *Maybe* enables failing strategies:

```
data Maybe x = Nothing | Just x
```

[2] The paper's website (http://www.uni-koblenz.de/~laemmel/syb42) provides access to a source distribution from which the source portions in the paper have been extracted.

That is, a strategy returns *Nothing* to signal failure, while a successful computation returns a value of the form *Just* x. The *Data* constraint in the definition of *Strategy* enables the non-parametrically polymorphic traversal capability of *all* and *one*. For our discussion, the further details of the SYB approach are not important. The above traversal schemes are all of the following type:

```
full_td, ..., innermost :: Strategy -> Strategy
```

There is yet another combinator, *adhoc*, which models update of a polymorphic strategy in a point (i.e., a type). In "*adhoc* g s", the argument g is the polymorphic default strategy to be applied whenever the monomorphic function s cannot be applied, as far as its specific type is concerned. (In the non-strongly typed setting of Stratego, a rewrite rule is immediately a "polymorphic" strategy, as if it were combined with *fail* as default.)

For instance, assume that *increment* is a function on numbers that simply increments them. Using *adhoc*, we can make the *increment* function generic so that it can be passed to a traversal scheme, which can be finally applied to a term. Thus:

```
full_td (adhoc id increment) myTerm
```

3 Inventory of programming errors

Let us consider again the simple scenario proposed in the introduction: increment all numbers in a term. For concreteness' sake, we choose the terms to be "trees" and the numbers to be "naturals". Further, we assume a Peano-like definition of the data type for naturals; the Peano-induced recursion will be useful in showcasing a number of programming errors. Here are the data types for naturals and trees:

```
data Nat = Zero | Succ Nat
data Tree a = Node {rootLabel :: a, subForest :: [Tree a]}
```

Here are simple tree samples (that we will use throughout the paper):

```
tree1 = Node { rootLabel = Zero, subForest = [] } -- A tree of numbers
tree2 = Node { rootLabel = True, subForest = [] } -- A tree of Booleans
tree3 = Node { rootLabel = Succ Zero, subForest = [tree1,tree1] } -- Two subtrees
```

The rewrite rule for incrementing naturals is represented as follows: [3]

```
increment n = Just (Succ n)
```

[3] The use of the constructor *Just* implies that *increment* denotes a successful computation.

It remains to complete the rewrite rule into a traversal strategy that increments all naturals in an arbitrary term (such as in *tree1* and *tree3* — trees labeled with naturals). Given the options *full_td*, *full_bu*, *stop_td*, *once_bu*, and *innermost*, which traversal scheme is the correct one for the problem at hand? An experienced strategist may quickly exclude one or two options. For instance, it may be obvious that the scheme *once_bu* is not appropriate because we want to increment *all* naturals, while *once_bu* would only affect one natural. The following paragraphs attempt different schemes and vary other details, thereby showcasing various programming errors.

3.1 Unbounded recursion

Let us attempt a full top-down traversal. Alas, the following strategy diverges:

Haskell-prompt> `full_td (adhoc id increment) tree1`
... an infinite tree is printed ...

The intuitive reason for non-termination is that *full_td* applies the argument strategy *prior to descent*, which may be problematic in case the argument strategy increases the depth of the given term, which is exactly what *increment* does. If *full_td* is not appropriate for the problem at hand, let us try another scheme, be it *innermost*. Again, we witness non-termination:

Haskell-prompt> `innermost (adhoc fail increment) tree1`
... no output ever is printed ...

The combinator *innermost* repeats the traversal trategy *once_bu (adhoc fail increment)* until it fails, but it never does because the subtree position with the natural always fits. Hence, *tree1* is rewritten indefinitely.

3.2 Incorrect quantification

Let us try yet another scheme, *full_bu*:

Haskell-prompt> `full_bu (adhoc id increment) tree1`
`Just (Node {rootLabel = Succ Zero, subForest = []})`

(That is, the root label was indeed incremented.) This particular test case looks fine, but *if we were testing* the same strategy with trees that contain non-zero naturals, then we would learn that the composed strategy replaces each natural n by $2n+1$ as opposed to $n+1$. To see this, one should notice that a natural n is represented as a term of depth n, and the choice of the scheme *full_bu* implies that *increment* applies to each "sub-natural". The scheme *full_bu* performs a full sweep over the input, which is not appropriate here because

we do not want to descend into naturals.

More generally, strategies need to "quantify" terms of interest:

- The type of the terms of interest.
- The number of redexes to be affected (e.g., one or any number found).
- The traversal order in which terms of interest are to be found.
- The degree of recursive descent into subterms.

The programmer is supposed to express quantification (say, "to control traversal") by choosing the appropriate traversal scheme. The choice may go wrong, when the variation points of the schemes are not understood, or accidentally considered irrelevant for the problem at hand.

3.3 Incorrect polymorphic default

Finally, let us try *stop_td*. Alas, no incrementing seems to happen:

```
Haskell-prompt> stop_td (adhoc id increment) tree1
Just (Node {rootLabel = Zero, subForest = []})
```

(That is, the result equals *Just tree1*.) The problem is that the strategy should continue to descend as long as no natural was hit, but the polymorphic default *id* makes the strategy stop for any subterm that is not a natural. If we replace *id* by *fail*, then we finally arrive at a proper solution for the original problem statement:

```
Haskell-prompt> stop_td (adhoc fail increment) tree1
Just (Node {rootLabel = Succ Zero, subForest = []})
```

fail is the archetypal polymorphic default for certain schemes, while it is patently inappropriate for others. To see this, suppose, we indeed want to replace each natural n by $2n + 1$, as we accidentally ended up doing in § 3.2. Back then, the use of *full_bu* with *id* as polymorphic default worked fine, and indeed *fail* would not:

```
Haskell-prompt> full_bu (adhoc fail increment) tree1
Nothing
```

3.4 Incorrect monomorphic default

To illustrate another programming error, let us consider a refined problem statement. That is, let us increment even numbers only. In the terminology

of rewriting, this statement seems to call for a conditional rewrite rule: [4]

```
-- Pseudo code for a conditional rewrite rule
increment_even : n -> Succ(n) where even(n)
```

In Haskell notation:

```
increment_even n = do guard (even n); increment n
```

Other than that, we keep using the traversal scheme that we found earlier:

```
Haskell-prompt> stop_td (adhoc fail increment_even) tree1
Just (Node {rootLabel = Succ Zero, subForest = []})
```

This particular test case looks fine, but *if we were testing* the same strategy with trees that contain odd naturals, then we would learn that the composed strategy in fact also increments those. The problem is that the failure of the precondition for *increment* propagates to the traversal scheme which takes failure to mean "continue descent". However, once we descend into odd naturals, we will hit an even sub-natural in the next step, which is hence incremented. So we need to make sure that recursion ceases for *all* naturals. Thus:

```
increment_even n
  | even n = Just (Succ n)
  | otherwise = Just n
```

3.5 Unreachable constituents

Consider the following patterns of strategy expressions:

- *adhoc* (*adhoc* g s_1) s_2
- *choice* f_1 f_2
- *sequ* f_1 f_2

In the first pattern, if the constituents s_1 and s_2 are of the same type (or more generally, the type of s_2 can be specialized to the type of s_1), then s_1 has no chance of being applied. Likewise, in the second pattern, if f_1 never possibly fails, then f_2 has no chance of being applied. Finally, in the third pattern, if f_1 never possibly succeeds, which is likely to be the symptom of a programming error by itself, then, additionally, f_2 has no chance of being applied.

Let us illustrate the first kind of programming error: multiple branches of the same type in a given adhoc-composed type case. Let us consider a refined

[4] Both the original *increment* function and the new "conditional" *increment_even* function go arguably beyond the basic notion of a rewrite rule that requires a non-variable pattern on the left-hand side. We could easily recover classic style by using two rewrite rules — one for each form of a natural.

problem statement such that incrementing of naturals is to be replaced by (i) increment *by one* for all odd numbers, (ii) increment *by two* for all even numbers. Here are the constituents that we need:

```
increase_odd n
  | odd n = Just (Succ n)
  | otherwise = Nothing

increase_even n
  | even n = Just (Succ (Succ n))
  | otherwise = Nothing
```

(We leave it as an exercise to the reader to argue whether or not the monomorphic default *Nothing* is appropriate for the given problem; cf. § 3.4.) Intuitively, we wish to chain together these type-specific cases so that they both are tried. It is not uncommon that strategic programmers (say, in Strafunski) attempt a composition like the following; alas no incrementing seems to happen:

```
Haskell-prompt> stop_td (adhoc (adhoc fail increase_even) increase_odd) tree1
Just (Node {rootLabel = Zero, subForest = []})
```

In the sample tree, the natural number, *Zero*, is even but the dominating type-specific case applies to odd numbers; hence no incrementing happens. The two rewrite rules must be composed differently:

```
Haskell-prompt> stop_td (adhoc fail (mchoice increase_even increase_odd)) tree1
Just (Node {rootLabel = Succ (Succ Zero), subForest = []})
```

Here, we rely on a rank-1 choice combinator: [5]

```
mchoice :: MonadPlus m => (x -> m x) -> (x -> m x) -> x -> m x
mchoice f g x = mplus (f x) (g x)
```

3.6 Unreachable types

We face a more conditional, more subtle form of an unreachable (monomorphic) constituent when the constituent's applicability depends on the fact whether its *type* can be encountered at all — along the execution of the encompassing traversal strategy. Consider the following strategy application that traverses the sample tree *tree2* — a tree labeled with Boolean literals (as opposed to naturals):

```
Haskell-prompt> stop_td (adhoc fail increment) tree2
```

[5] *mplus* is addition on values of type *m x* for any *MonadPlus m*. In the case of the *Maybe* monad, *mplus* is the operation that returns its left operands, if it is not *Nothing*, and it returns its right operand, otherwise.

```
Just (Node {rootLabel = True, subForest = []})
```

(That is, the result equals *Just tree2*.) In fact, one can see that the strategy will preserve *any* term of type *Tree Boolean*. Terms of interest, i.e., naturals, cannot possibly be found below any root of type *Tree Boolean*. It seems plausible that the function shown manifests a programming error: we either meant to traverse a different term (i.e., one that contains naturals), or we meant to invoke a different strategy (i.e., one that affects Boolean literals or polymorphic trees).

3.7 Incorrect success/failure handling

The earlier problems with (polymorphic and monomorphic) defaults feed into a more general kind of problem: misunderstood success/failure behavior of traversal schemes and their strategy parameters. (We should generally note that the various kinds of programming errors discussed are not fully orthogonal.) Here is a simple example of misunderstanding.

```
main = do
         (tree::Tree Nat) <- readLn
         tree' <- stop_td (adhoc fail increment) tree
         putStrLn "1 or more naturals incremented successfully"
```

The program invokes a traversal strategy for incrementing naturals in a tree that is constructed from input. The output statement, which follows the traversal, documents the programmer's (incorrect) thinking that the successful completion of the *stop_td* scheme implies at least one application of the argument strategy, and hence, *tree* $\neq$ *tree'*.

In the above (contrived) example, misunderstood success/failure only leads to incorrect text output. In general, programmers may compose traversal programs in ways that their control pattern depends on assumptions as wrong as the one above. Even when misunderstood success/failure behavior does not affect correctness, it may instead lead to defensive and convoluted code. For instance, in the following strategy expression, the application of *try* (defined in § 2) is superfluous because the traversal to which it is applied cannot possibly fail.

```
try (full_td (adhoc id increment))
```

We should mention that part of the confusion regarding success/failure behavior stems from the overloaded interpretation of success/failure — to relate to either strategy control or pre-/post-condition checking. A new language design may favor to separate these aspects — possibly inspired by forms of exception handling known from the general programming field [20].

3.8 Incorrect plan

A strategic programming (sub-) problem is normally centered around some problem-specific constituents ("rewrite rules") that have to be organized in a more or less complex strategy. Organizing this strategy involves the following decisions:

(i) Which traversal scheme is to be used?

(ii) What polymorphic and monomorphic defaults are to be used?

(iii) What is the level of composition?
- The polymorphic level of strategy arguments.
- The top-level at which possibly multiple traversals can be combined.
- The monomorphic level, i.e., before becoming polymorphic by means of *adhoc*.

(iv) What is the composition operator? Is it *adhoc*, *choice*, or *sequ*?

We return to the example from § 3.5, which incremented odd and even numbers differently. Let us assume that we have resolved decisions (i) and (ii) by choosing the scheme *stop_td* and the default *fail*; we still have to consider a number of options due to (iii) and (iv). The following list is not even complete because it omits order variations for composition operators.

Note: *mchoice* and *msequ* are rank-1 variations on *choice* and *sequ*.[6]

1. *stop_td (adhoc (adhoc fail increase_even) increase_odd)*
2. *stop_td (adhoc fail (mchoice increase_even increase_odd))*
3. *stop_td (adhoc fail (msequ increase_even increase_odd))*
4. *stop_td (choice (adhoc fail increase_even) (adhoc fail increase_odd))*
5. *stop_td (sequ (adhoc fail increase_even) (adhoc fail increase_odd))*
6. *choice (stop_td (adhoc fail increase_even)) (stop_td (adhoc fail increase_odd))*
7. *sequ (stop_td (adhoc fail increase_even)) (stop_td (adhoc fail increase_odd))*

Option (1.) had been dismissed already because the two branches involved are of the same type. Option (2.) had been approved as a correct solution. Option (4.) turns out to be equivalent to option (2.). (This equivalence is implied by basic properties of defaults and composition operators.) The strategies of the other options do not implement the intended operation, even though it may be difficult to understand exactly how they differ.

[6] We had defined *mchoice* earlier; *msequ* is function composition lifted to *Maybe* values.

4 Refinements of traversal programming

It seems plausible to ask whether we can attack some or all of the identified kinds of programming errors by devising refinements of traversal programming. For brevity, we do not discuss some of the more pragmatic approaches such as debugging. Instead we want to focus on techniques that give (more) "correctness by design or by static checks". The following discussion comes without any claim of completeness. Also, the reported experiments are rendered (mostly) in a Haskell-biased manner. However, we do tend to summarize each experiment by clarifying the language-independent refinement idea behind the experiment.

4.1 Less generic strategies

The prevention of some of the aforementioned programming errors may benefit from variations on the strategy library that are "less problematic". One method is to reduce the genericity of the traversal schemes to rank 1, i.e., the problem-specific arguments become monomorphic. The following definitions take a monomorphic argument *s*, which is then generalized *within the scheme* by means of the appropriate polymorphic default, *id* or *fail*:

```
full_td s = sequ (adhoc id s) (all (full_td s))
full_bu s = sequ (all (full_bu s)) (adhoc id s)
stop_td s = choice (adhoc fail s) (all (stop_td s))
once_bu s = choice (one (once_bu s)) (adhoc fail s)
innermost s = repeat (once_bu s)
```

The rank-1 schemes reduce programming errors as follows. Most obviously, polymorphic defaults are correct by design because they are hard-wired into the schemes. Also, the *adhoc* idiom has no purpose anymore, and hence, no problem with overlapping type-specific cases occurs. No other programming errors are directly addressed, but one can say that "incorrect plans" are less likely simply because there are fewer feasible options for plans.

However, there are scenarios that call for *polymorphic*, problem-specific constituents of traversals; cf. [22,13,14] for some concrete samples. Hence, the original (generic) schemes must be retained. Some cases of strategies with multiple type cases can be decomposed into multiple traversals, but even when it is possible, it may still be burdensome and negatively affect performance.

An improved, strategic programming language could assume an implicit coercion scheme such that the appropriate polymorphic default is applied whenever a single monomorphic case is passed to the scheme. In this manner, the API surface is not increased, and the omission of (potentially ill-specified) polymorphic defaults is encouraged.

4.2 Fallible and infallible strategies

Let us investigate another variation on (part of) the strategy library that is "less problematic". The proposed method is to provide more guidance regarding the success/failure behavior of traversal schemes and their arguments. Let us recall the types of the sample schemes:

full_td, ..., innermost :: Strategy −> Strategy
where ***type*** *Strategy* = ***forall*** *x. Data x => x −> Maybe x*

In fact, this is a specialized type that we had chosen for simplicity of the initial presentation. In general, the type is parametrized by a monad:

full_td, ..., innermost :: GenericM m −> GenericM
where ***type*** *GenericM* = ***forall*** *x. Data x => x −> m x*, and *m* is a monad.

The monad parameter may be used for different purposes, in particular for modeling success and failure based on the *Maybe* monad or any other monad with a "zero" (i.e., failure). The general types of the schemes hide some intentions regarding the common success/failure behavior of arguments and composed strategies. In particular, it would be valuable for the programmer to know when *success is guaranteed*.

We say that a strategy is *infallible* if it does not possibly fail, i.e., if it will succeed (or diverge). It is relatively easy to confirm the following claims about infallibility. Given is a strategy s. If s is infallible, then *full_td* s and *full_bu* s are infallible. No matter the argument s, the strategies *stop_td* s and *innermost* s are infallible. No infallibility claim about *once_bu* can be stated; this scheme is intrinsically fallible.

We can easily provide a (non-classic) proof of the above claims, where we use these claims as new types of the traversal schemes. That is, a type is made infallible by stripping off the monad wrapper from the type. The definitions of the infallible schemes remain unchanged, except that some of the basic strategy combinators also need to be trivially complemented by variations with infallible types.

```
-- Generic transformations
type GenericT = forall a . Data a => a -> a
-- Generic monadic transformations
type GenericM m = forall a . Data a => a -> m a

full_td :: GenericT -> GenericT
full_bu :: GenericT -> GenericT
stop_td :: GenericM Maybe -> GenericT
innermost :: GenericM Maybe -> GenericT
```

The idea is now that a programmer favors the infallible schemes, whenever

possible, and falls back to the original (fallible) ones, whenever necessary. Also, if the type of an infallible strategy combinator points out a potentially failing argument, then this status signals to the programmer that failure of the argument strategy is indeed usefully anticipated by the combinator.

An improved, strategic programming language may leverage (in)fallibility rules systematically: (i) it may infer precise types for strategies (as far as (in)fallibility is concerned); (ii) it may allow for programmer annotations that capture expectations with regard to (in)fallibility and verify those; (iii) it may emit warnings when supposedly fallible arguments are infallible.

4.3 Checked adhoc chains

The problem of unreachable cases in an adhoc chain can be avoided by a type check that specifically establishes that all cases are distinct in terms of the covered types. In addition, such a regime allows us to factor out the polymorphic default to reside in the traversal scheme, thereby eliminating another source of error.

In the following, we use an advanced Haskell library, HList [7], to describe the constituents of a traversal scheme as a *family of monomorphic cases*, in fact, as an appropriately constrained heterogeneous list of functions. Consider the pattern that we used so far: *adhoc* (*adhoc* g s_1) s_2. Two type-specific cases, s_1 and s_2, are involved, which are used to point-wisely override the generic default g. The type-specific cases can be represented as the heterogeneous list *HCons* s_1 (*HCons* s_2 *HNil*).[7] Such a list may be converted to a plain adhoc chain by a function, *familyM*, which takes a polymorphic default as an additional argument. This function also checks that type-specific cases do not overlap. Here are the schemes that are parametrized in families of cases; the new schemes leverage the original schemes.[8]

```
full_td s = StrategyLib.Schemes.full_td (familyM id s)
full_bu s = StrategyLib.Schemes.full_bu (familyM id s)
stop_td s = StrategyLib.Schemes.stop_td (familyM fail s)
once_bu s = StrategyLib.Schemes.once_bu (familyM fail s)
innermost s = StrategyLib.Schemes.innermost (familyM fail s)
```

The function *familyM* is defined by induction on the heterogeneous list structure:

```
class (Monad m, HTypeIndexed f) => FamilyM f m
  where
    familyM :: GenericM m -> f -> GenericM m
```

[7] The empty heterogeneous list is represented as *HNil*, whereas the non-empty heterogeneous list with head h and tail t is represented as *HCons* h t.

[8] We refer to the original schemes of § 2 by using the prefix *StrategyLib.Schemes.*....

```
instance Monad m => FamilyM HNil m
  where
    familyM g _ = g

instance (Monad m, FamilyM t m, Typeable x, HOccursNot (x -> m x) t)
      => FamilyM (HCons (x -> m x) t) m
  where
    familyM g (HCons h t) = adhoc (familyM g t) h
```

Most notably, the constraint *HTypeIndexed* (provided by the HList library) establishes that the cases are distinct in terms of the covered type. As a proof obligation, the instance for non-empty lists (cf. *HCons* ...) must establish that the head's type does not occur again in the tail of the family; cf. the constraint *HOccursNot* ... (provided by the HList library).

An improved, strategic programming language may indeed assume a suitable notion of "type case", subject to the kind of checking and mapping to regular strategies, as described above. Essentially, the presented idea generalizes the simple idea of rank-1 schemes; cf. § 4.1.

4.4 Reachable type constraints

Let us consider again the particularly subtle form of unreachable (monomorphic) cases, where they turn out to be unreachable just because their types cannot be expected "below" the possible root types. A basic remedy is to constrain (at a type level) the applicability of a given traversal strategy such that the type of the argument cases must be in a "reachability" relationship with the type of the root. Inspired by [10], we can illustrate this idea (without loss of generality) for rank-1 schemes. Consider the following constrained variation on the *full_td* scheme:

```
full_td :: (Monad m, Typeable x, Data y, HBelowEq y x) => (x -> m x) -> y -> m y
full_td s = StrategyLib.Schemes.full_td (adhoc id s)
```

All the constraints in the type are readily implied by the original version (modulo skolemization and simplification), except *HBelowEq*, which models the relationship between types such that *HBelowEq x y* holds whenever *x* is the type *y*, or the type of an immediate or non-immediate subterm.

```
class HBelowEq x y
instance HBelowEq x x -- reflexivity
-- Other instances are derived from data types of interest.
```

For instance, the data type for polymorphic trees and the leveraged data type for polymorphic lists imply the following contributions to the relation *HBelowEq*:

```
instance HBelowEq a [a]
instance HBelowEq a (Tree a)
instance HBelowEq [Tree a] (Tree a)
```

Again, the Haskell experiment shown merely serves for illustration. In an improved, strategic programming language, all traversal schemes may be annotated by constraints for reachability, or these constraints may even be inferred automatically. All reachability constraints would be statically checked.

5 Static strategy analysis

All refinements of the previous section required variations of familiar strategy combinators and traversal schemes. In particular, we used less parametric and more type-constrained variations. In contrast, we will now demonstrate the potential utility of strategy analysis in performing static checks on otherwise classic traversal programs. (There could be additional programmer annotations, say "contracts", to be observed by a static analysis. We do not further discuss this elaboration here.)

It should be clear that an analysis is likely to be useful in determining the status of a strategy (i) to involve constituents that have no chance of being applied; (ii) to always fail (or diverge); (iii) to always succeed without any rewriting done (or diverge); (iv) to definitely terminate.

In the following, we develop a simple analysis for the basic property of an infallible strategy. We denote this analysis as "cf — can fail". We model the analysis in Haskell. The following data type models strategy expressions for transformations, i.e., *type-preserving* strategies; cf. "TP":

```
data TP x = Id | Fail | Seq (TP x) (TP x) | Choice (TP x) (TP x)
          | All (TP x) | One (TP x)
          | Rec (x -> TP x) | Const x
```

Clearly, the data type covers the known strategy combinators; cf. *Id*, *Fail*, etc. The type parameter of the data type caters for analysis defined by fixpoint computation. The constructors *Rec* and *Const* model fixpoint combinator and recursive reference, respectively. We can model familiar traversal schemes as "TP" terms:

```
full_td s = Rec (\x -> Seq s (All (Const x)))
once_bu s = Rec (\x -> Choice (One (Const x)) s)
stop_td s = Rec (\x -> Choice s (All (Const x)))
```

The analysis "can fail" is described as follows:

```
cf Id = False                          cf (All s) = cf s
cf Fail = True                         cf (One s) = True
cf (Seq s s') = cf s || cf s'          cf (Const x) = x
cf (Choice s s') = cf s && cf s'       cf (Rec f) = cf (f False)
```

The equation for *Rec* models a degenerated fixpoint calculation. That is, calculation starts from *False* (denoting "will succeed or diverge"), and just one iteration suffices because the complete partial order only contains two elements.

The following table presents some analysis results such that the "can fail" property of the traversal scheme is computed from the "can fail" property of the argument strategy. (We reconfirm the infallibility claims from § 4.2.) For instance, consider the last row of the table: no matter whether the argument of *stop_td* can fail or not, the traversal will succeed (or diverge).

Scheme s	$cf(s$ (Const False))	$cf(s$ (Const True))
full_td	False	True
once_bu	False	True
stop_td	False	False

Clearly, more advanced analyses (such as a termination analysis) require more interesting abstract domains than *Bool*. Also, some analyses (such as a reachability analysis) need to relate to meta-data (signatures) of the traversed data.

6 Related work

Mentions of simple "laws" for strategies as well as strategy properties can be found scattered over the strategic programming literature [22,6,24,21,9,3]. The present paper provides the first substantial attempt of a systematic discussion of programming errors and their relationship to strategy properties.

The technique of § 4.1 to use less generic traversal schemes has also been explored in [16] in the context of devising simpler types for traversal programs and more efficient implementations. The technique of § 4.4 to statically check for reachable types is inspired by adaptive programming [18,15,12] that subjects its traversal specifications to a similar check.

Constrained forms of traversal programming may be less prone to the errors that we discussed. For instance, one can limit the programmability of traversal (e.g., in ASF+SDF with traversal functions [21]), or impose more structure on traversal programs (e.g., in adaptive programming, where traversal specifications and computations or actions are separated). It is our goal to admit full programmability, but ban programming errors by static analysis

or type checking.

Automated program calculations based on algebraic laws were devised for specializing (optimizing) strategic programs [3]. For instance, there are laws whose systematic application reveals the "uselessness" of certain subterm traversals in a complete traversal. (Here, it is assumed that the types of type-specific cases are known as well as meta-data (data-type declarations) for the traversed data.) It should be possible to use similar calculations to set up strategy analyses.

Some of the discussed strategy properties and the corresponding analyses naturally call for a more general treatment. For instance, dead-code elimination, strictness analysis [17] or termination checking [1,2] are known procedures for functional programs, perhaps even generic functional programs. We hope to exploit this body of knowledge in the future. We assume that there is enduring value in studying properties right at the level of strategies because domain-specific languages are generally meant to provide domain-specific checks and optimizations, while feedback should relate to domain concepts, too.

7 Concluding remarks

The ultimate motivation for the presented work is to provide input for the next generation of strategic programming. Here we assume that domain-specific support (as in the case of ASF+SDF and Stratego) is mandatory. However, we also feel constrained by the relatively small market for strategic programming languages. Hence, we hope to operate on the grounds of a general purpose programming-language framework — one that must be sufficiently extensible to provide designated support for the traversal domain.

We seek a form of traversal programming such that programs are subject to well-defined properties that support a discipline of traversal programming. Some properties may be implicitly assumed (e.g., termination); others may need to be explicitly stated by the programmer (e.g., expectations regarding the success/failure behavior). The validity of desirable properties and the absence of undesirable properties have to be checked statically. A facility for explicitly stating properties may be viewed as a means to adopt "design by contract" to traversal programming. Here, we are inspired, for example, by functional programming contracts [4,5]. The provision of strategy-biased and statically checked contracts would require a form of dependent types, an extensible type system, or, in fact, an extensible language framework that admits pluggable static analysis.

A related challenge for the next generation of strategic programming is performance. (In fact, disappointing performance may count as another kind

of programming error.) We hope to eventually gather enough analytical power and strategy properties so that the declarative style of strategic programming can be mapped to highly optimized code. Here, we are inspired by previous work on fusion-like techniques for traversal strategies [6], and calculational techniques for the transformation of traversal strategies [3].

Acknowledgement

Simon Thompson has received support by the Vrije Universiteit, Amsterdam for a related research visit in 2004. The authors received helpful feedback from the LDTA reviewers.

References

[1] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[2] Andreas Abel. Type-based termination of generic programs. *Science of Computer Programming*, 2007. MPC'06 special issue. Submitted.

[3] Alcino Cunha and Joost Visser. Transformation of structure-shy programs: applied to XPath queries and strategic functions. In *PEPM'07: Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 11–20. ACM Press, 2007.

[4] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP'02: Proceedings of the 7th ACM SIGPLAN international conference on Functional programming*, pages 48–59. ACM Press, 2002.

[5] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed Contracts for Functional Programming. In *FLOPS'06: Functional and Logic Programming, 8th International Symposium, Proceedings*, volume 3945 of *LNCS*, pages 208–225. Springer, 2006.

[6] Patricia Johann and Eelco Visser. Strategies for Fusing Logic and Control via Local, Application-Specific Transformations. Technical Report UU-CS-2003-050, Department of Information and Computing Sciences, Utrecht University, 2003.

[7] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell'04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[8] Ralf Lämmel. The Sketch of a Polymorphic Symphony. In *WRS'02: Proceedings of International Workshop on Reduction Strategies in Rewriting and Programming*, volume 70 of *ENTCS*. Elsevier Science, 2002. 21 pages.

[9] Ralf Lämmel. Typed generic traversal with term rewriting strategies. *Journal Logic and Algebraic Programming*, 54(1-2):1–64, 2003.

[10] Ralf Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL'07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 137–142. ACM Press, 2007.

[11] Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI'03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37. ACM Press, 2003.

[12] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic Programming Meets Adaptive Programming. In *AOSD'03: Conference proceedings of Aspect-Oriented Software Development*, pages 168–177. ACM Press, 2003.

[13] Ralf Lämmel and Joost Visser. Typed Combinators for Generic Traversal. In *PADL'02: Proceedings of Practical Aspects of Declarative Programming*, volume 2257 of *LNCS*, pages 137–154. Springer, January 2002.

[14] Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *PADL'03: Proceedings of Practical Aspects of Declarative Programming*, volume 2562 of *LNCS*, pages 357–375. Springer, January 2003.

[15] Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and Efficient Implementation. *ACM Transactions on Programming Languages and Systems*, 26(2):370–412, 2004.

[16] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 49–60. ACM, 2007.

[17] Alan Mycroft. The Theory and Practice of Transforming Call-by-need into Call-by-value. In *International Symposium on Programming, Proceedings of the 4th 'Colloque International sur la Programmation'*, volume 83 of *LNCS*, pages 269–281. Springer, 1980.

[18] Jens Palsberg, Boaz Patt-Shamir, and Karl J. Lieberherr. A New Approach to Compiling Adaptive Programs. *Science of Computer Programming*, 29(3):303–326, 1997.

[19] Deling Ren and Martin Erwig. A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In *Haskell'06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 13–24. ACM Press, 2006.

[20] Barbara G. Ryder and Mary Lou Soffa. Influences on the design of exception handling: ACM SIGSOFT project on the impact of software engineering research on programming language design. *SIGPLAN Notices*, 38(6):16–22, 2003.

[21] Mark van den Brand, Paul Klint, and Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Transactions Software Engineering Methodology*, 12(2):152–190, 2003.

[22] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP'98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM Press, 1998.

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

[24] Joost Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, February 2003.