Programming errors in traversal programs over structured data

Ralf Lämmel[1] and Simon Thompson[2] and Markus Kaiser[1]

[1] *University of Koblenz-Landau, Germany*

[2] *University of Kent, UK*

Abstract

Traversal strategies à la Stratego (also à la Strafunski and "Scrap Your Boilerplate") provide an exceptionally versatile and uniform means of querying and transforming deeply nested and heterogeneously structured data including terms in functional programming and rewriting, objects in OO programming, and XML documents in XML programming.

However, the resulting traversal programs are prone to programming errors. We are specifically concerned with errors that go beyond conservative type errors; examples we examine include divergent traversal, prematurely terminated traversal, and traversal with dead code.

Based on an inventory of typical programming errors and an understanding of idiomatic capabilities or challenges of traversal strategies we explore options of type checking and static analysis so that some categories of errors can be avoided. This development uses Haskell for illustrations and for specifying algorithms, but the main ideas are largely language-agnostic and should still be comprehensible nonetheless.

Keywords: Traversal Strategies, Traversal Programming, Term Rewriting, Stratego, Strafunski, Generic Programming, Scrap Your Boilerplate, Type systems, Static Program Analysis, Functional Programming, XSLT, Haskell.

* A short version of this paper appeared in the proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008), published by ENTCS. The full version is generally more detailed and updated, but its distinctive contribution is research on static program analysis for traversal strategies; c.f., §5: only one of the four analyses was sketched briefly in the short version. Both versions of the paper and download instructions for all related source code are hosted at:
- `http://userpages.uni-koblenz.de/~laemmel/syb42`
- `http://code.google.com/p/strafunski/`

***AST example*: Abstract syntax of a simple functional language**

```
type Block     = [Function]
data Function  = Function Name [Name] Expr Block
data Expr      = Literal  Int
               | Var Name
               | Lambda Name Expr
               | Binary Ops Expr Expr
               | IfThenElse Expr Expr Expr
               | Apply Name [Expr]
data Ops       = Equal | Plus | Minus
type Name      = String
```

***Company example*: Organizational structure of a company**

```
data  Company     = Company [Department]
data  Department  = Department Name Manager [Unit]
data  Manager     = Manager Employee
data  Unit        = EmployeeUnit Employee
                  | DepartmentUnit Department
data  Employee    = Employee Name Salary
type  Name        = String    -- names of employees and departments
type  Salary      = Float     -- salaries of employees
```

Fig. 1. **Illustrative data models in Haskell's notation for algebraic data types. Queries and transformation on such data may require traversal programming.** The data models involve multiple types, nested types, and types with multiple alternatives. As a result, traversal programmers need to carefully control queries and transformation—leaving room for *programming errors.*

1 Introduction

Traversal programming

In the context of data programming with XML trees, object graphs, and terms in rewriting or functional programming, consider the general scenarios of *querying and transforming deeply nested and heterogeneously structured data.* Because of deep nesting and heterogeneity as well as plain structural complexity, data programming may benefit from designated idioms and concepts. In this paper, we focus on the notion of *traversal programming* where functionality is described in terms of so-called *traversal strategies*: we are specifically concerned with *programming errors* in this context.

As an indication of some of the application domains for such traversal programming, consider Fig. 1; the figure shows two kinds of (Haskell-based) data

Illustrative queries

The AST example

(1) Determine all recursively defined functions.
(2) Determine the nesting depth of function definition.
(3) Collect all the free variables in a given code fragment.

The company example

(1) Total the salaries of all employees.
(2) Total the salaries of all employees who are not managers.
(3) Calculate the manager / employee ratio in the leaf departments.

Illustrative transformations

The AST example

(1) Inject logging code around function applications.
(2) Perform partial evaluation by constant propagation.
(3) Perform unfolding or inlining for a specific function.

The company example

(1) Increase the salaries of all employees.
(2) Decrease the salaries of all non-top-level managers.
(3) Integrate a specific department into the hosting department.

Fig. 2. Illustrative scenarios for traversal programming.

models: one is an abstract syntax of a programming-language fragment; the other one is a fragment of the organizational structure of a company, as it is used in an information system.

Now also consider the illustrative, specific scenarios for querying and transforming data as shown in Fig. 2. These are good representatives for scenarios that benefit from designated support for traversal programming. We refer the reader to [63,61,34,44,45,71] for some published accounts of actual applications of traversal programming with traversal strategies.

Traversal programming with *traversal strategies*

In this paper, we are concerned with one particular approach to traversal programming—the approach of *traversal strategies*, which relies on so-called one-layer traversal combinators as well as other, less original combinators for controlling and composing recursive traversal. Traversal strategies support an exceptionally versatile and uniform means of traversal. The term *strategic programming* is also in use for this approach to traversal programming [41].

The notion of traversal strategies was pioneered by Eelco Visser and collaborators [47,64,63] in the broader context of term rewriting. This seminal work also led to Visser et al.'s Stratego/XT [62,9]—a domain-specific language, in fact, an infrastructure for software transformation. Other researchers in the term rewriting and software transformation communities have also developed related forms of traversal strategies; see, e.g., [8,7,60,72].

The Stratego approach inspired strategic programming approaches for other programming paradigms [43,41,65,3,4]. The functional approach (also known as "Strafunski"), in turn, inspired the "Scrap your boilerplate" form of generic functional programming [38–40,55,26,27], which again inspired cross-paradigm variations, e.g., "Scrap your boilerplate in C++" [50]. Meanwhile there is a good understanding of different approaches to traversal programming, with traversal strategies à la Stratego as one option among several other options such as TXL's strategies [10], adaptive traversal specifications [46,42,1,1], Cω's data access [5], XPath-like queries and XSLT transformations [36,14], HATS strategies [72,69,70], Stratego-like strategies amalgamated with attribute grammars [31], or diverse forms of generic functional programming [28,23,56].

Research topic: *programming errors* in traversal programming

Despite the advances on foundations, programming support and applications of traversal programming with strategies, the use and the definition of programmable traversal strategies has remained the domain of the expert, rather than gaining wider usage. 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.

It does not require detailed knowledge of traversal strategies to acknowledge the potential of programming errors. To give an idea, let us consider the traversal scenarios of Fig. 2 again, and let us anticipate potential programming errors. Here are two examples:

A programming error implying an incorrect result In the company example, as we are *totaling the salaries of all employees who are not managers*, suppose we special-case the type of managers so that they are not counted, but we accidentally forget to cut off traversal below managers, in which case, they would still be counted as regular employees.

A programming error implying divergence In the AST example, as we are *unfolding a specific function*, we may accidentally organize the traversal for transformation in such a way that we perform unfolding for a recursive definition indefinitely. For instance, we may accidentally use top-down traversal (i.e., "inline before descent").

Our research aims to make strategic programming more accessible and more predictable through better understanding of idioms, traversal schemes, and their properties; also through developing techniques that give (more) "correctness by design or by static checks". In this paper, we provide i) an introduction to some of the common pitfalls of strategic programming, ii) a discussion of some programming techniques which can alleviate some of the problems, and also iii) a set of program analyses to address other pitfalls. In this manner, we begin refining strategic programming towards a next generation that may be easier to use. We envisage that a future strategic programming environment may help users to avoid programming errors by detecting favorable or unfavorable behaviors statically: that is, without running the program.

Contributions of the paper

(1) We provide a fine-grained inventory of programming errors in traversal programming with traversal strategies; see §3. This inventory aims to link programming errors to mistaken use of idioms for traversal programming so that we can start thinking of constrained idioms, or extra checks.

(2) We explore options for constrained traversal schemes, subject to refined types for the schemes. Thereby, some categories of programming errors are avoided; see §4.

(3) We explore the utility of *static program analysis* in avoiding programming errors; see §5. This is an established technique for dealing with correctness and performance properties of software. Our work is the first to leverage static analysis for traversal strategies. We study success and failure behavior, a form of dead code, and divergent traversal. We experiment with abstract interpretation and deductive techniques (i.e., type systems).

A related subject: ***traversal properties***

In previous work [30], we have worked towards improving the understanding of traversal strategies by means of theorem proving and with the objective of collecting and mechanically proving formal properties of traversal schemes and strategy primitives: algebraic laws, success and failure behavior, termination properties. Here, our assumption is that knowledge of such properties and laws helps mastering traversal programming.

Despite major efforts at the end of mechanized modelling, we have not succeeded with mechanizing a general enough strategy calculus; instead, we have only succeeded proving selected success and failure behavior and termination conditions for a few specific traversal schemes, also subject to substantial encoding. We view the present work as a complementary effort to exploit language embedding techniques and static program analysis for the sake of constraining and checking actual traversal programs.

Haskell *en route*

In this paper, we use Haskell for illustrations and for specifying algorithms, but the main ideas are largely language-agnostic, and it is completely possible to follow the main ideas presented here with a "reading knowledge" of the Haskell language. In a few positions of the paper, we make use of monads, monoids, type-level programming, and higher-order style, but the main ideas of these sections should still be comprehensible nonetheless.

Scope of this work

Our work ties into Stratego- and Strafunski-like traversal strategies in that it intimately commits to idiomatic and conceptual details of that approach: the use of one-layer traversal combinators, the style of mixing generic and problem-specific behavior, the style of controlling traversal through success and failure, and a preference for certain, important traversal schemes.

Nevertheless, our research approach and the results may be applicable to an even broader class of traversal programming approaches, e.g., adaptive programming, more general, generic functional programming, XML queries (such as with XPath or XQuery), XML transformations (such as with XSLT). However, we do not explore or discuss such potential in detail. To give an example of potential relevance, let us briefly relate to XSLT [68], which is a particularly prominent declarative processing model for XML that can be used to implement transformations between XML-based documents.

An XSLT transformation consists of a collection of template rules, each of which describes how to handle the nodes which match a particular pattern. Processing begins at the root node, and proceeds by applying the best fitting template pattern, and recursively processing according to the template. So, the traversal is implicit in the XSLT processing model, but controlled by the particular transformation rules. For this reason, similar problems to those we study here—such as unreachable code or divergence—can similarly occur in XSLT, and require analyses of the form presented here. We return to this connection in the related work section.

Road-map of the paper

- §2 recalls strategic programming basics and the Haskell incarnation.
- §3 takes an inventory of programming errors in strategic programming.
- §4 explores options for constrained types of traversal schemes.
- §5 exercises static program analysis to check traversal programs.
- §6 discusses related work.
- §7 concludes the paper.

$s ::= t \to t$	(Rewrite rules as basic building blocks.)
$\mid\ id$	(Identity strategy; succeeds and returns input.)
$\mid\ fail$	(Failure strategy; fails and returns failure "$\uparrow$".)
$\mid\ s; s$	(Left-to-right sequential composition.)
$\mid\ s \mathbin{\leftarrow\!\!\!+} s$	(Left-biased choice; try first argument first.)
$\mid\ \Box(s)$	(Transform all immediate subterms by s; maintain constructor.)
$\mid\ \Diamond(s)$	(Transform one immediate subterm by s; maintain constructor.)
$\mid\ v$	(A variable strategy subject to binding or substitution.)
$\mid\ \mu v.s$	(Recursive closure of s referring to itself as v.)

Fig. 3. **Syntax of traversal primitives for type-preserving strategies.**

2 Background on strategic programming

This section re-introduces the notion of traversal strategy and the style of strategic programming. We integrate basic material that is otherwise scattered over many different publications.

Readers with knowledge of strategic programming are still advised to scan through the section since it also prepares the discussion of programming errors to some extent.

The section presents two alternative models of traversal strategies: "POPL style" (grammar-based syntax definition and natural semantics) vs. "Haskell style" (data-type-based syntax and interpreter-based semantics). Familiarity with one of the two styles should be fairly sufficient.

2.1 Essential syntax

Fig. 3 shows the syntax of a core calculus for traversal strategies; it covers transformations (also known as type-preserving strategies) in particular; a few additional primitives would be needed for queries (also known as type-unifying strategies). This core calculus follows closely Visser et al.'s seminal work [63].

There are the following forms of strategies. "*id*" denotes the always succeeding strategy returning the input term as is; it is essentially the polymorphic identity function. "*fail*" is the always failing strategy. Here, it is important to notice that strategies either fail, or succeed, or diverge. There is a special failure result, denoted by "$\uparrow$" in the upcoming semantics. "*sequ s s′*" denotes the sequential composition of s and s'. "*choice s s′*" denotes left-biased choice:

try s first, and try s' second, if s failed.

The interesting combinators are "$\Box$" and "$\Diamond$" (also known as "all" and "one"). These combinators model so-called one-layer traversal. "$\Box(s)$" applies s to all immediate subterms of a given term, and fails if there is any subterm for which s fails. "$\Diamond(s)$" applies s to the leftmost immediate subterm of a given term such that the application does not fail, and fails if there is no such immediate subterm. The one-layer traversal combinators, when used within a recursive closure, enable the key capability of strategic programming: to traverse arbitrarily deeply into heterogeneously typed terms.

We have omitted some combinators that would be needed for a complete programming language. For instance, Stratego also provides strategy forms for congruences (i.e., the application of strategies to the immediate subterms of specific constructors), tests (i.e., a strategy is tested for success, but its result is not propagated), negation (i.e., a strategy to invert the success and failure behavior of a given strategy) [63]. We have also omitted combinators here, but we will discuss queries briefly when we present the Haskell model. Our formal development of §5 is also restricted to transformations, but we are confident that the approach can be easily extended to queries.

2.2 Natural semantics

Following again [63] and subsequent work on the formalization of traversal strategies [35,30], we give the formal semantics of type-preserving strategies as a big-step operational semantics using a success/failure model; c.f., Fig. 4–Fig. 5. The shown version has been extracted from a mechanized model of traversal strategies, based on the theorem prover Isabelle/HOL; c.f., [30]. The judgement $s \,@\, t \leadsto r$ describes a relation between strategy expression s, input term t, and result r, which is either a proper term or failure—denoted as "$\uparrow$". Based on tradition, we separate positive rules (resulting in a proper term) vs. negative rules (resulting in "$\uparrow$"). Incidentally, this distinction helps already with understanding the success and failure behavior of strategies.

For instance, the positive rule [all^+] models that the strategy application $\Box(s) \,@\, c(t_1, \ldots, t_n)$ applies s to all the t_i such that new terms t'_i are obtained and reunited in the term $c(t'_1, \ldots, t'_n)$. The negative rule covers the case that at least one of the applications $s \,@\, t'_i$ resulted in failure.

The shown semantics uses variables only for the sake of recursive closures, and the latter construct's semantics is modelled by substitution. We could also furnish variables for the sake of parametrized strategy expressions, i.e., binding blocks of strategy definitions, as this is relevant for reusability in practice, but we do not furnish this elaboration of the semantics here for brevity's sake.

$$\frac{\exists \theta.\ (\theta(t_l) = t \wedge \theta(t_r) = t')}{t_l \to t_r \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{rule}^+]$$

$$id \mathbin{@} t \rightsquigarrow t \quad [\mathsf{id}^+]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow t' \ \wedge\ s_2 \mathbin{@} t' \rightsquigarrow t''}{s_1; s_2 \mathbin{@} t \rightsquigarrow t''} \quad [\mathsf{sequ}^+]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow t'}{s_1 \nleftarrow\!\!+ s_2 \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{choice}^+.1]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow \uparrow \ \wedge\ s_2 \mathbin{@} t \rightsquigarrow t'}{s_1 \nleftarrow\!\!+ s_2 \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{choice}^+.2]$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ s \mathbin{@} t_i \rightsquigarrow t'_i}{\Box(s) \mathbin{@} c(t_1, \ldots, t_n) \rightsquigarrow c(t'_1, \ldots, t'_n)} \quad [\mathsf{all}^+]$$

$$\frac{\begin{array}{l} \exists i \in \{1, \ldots, n\}. \\ \quad s \mathbin{@} t_i \rightsquigarrow t'_i \\ \ \wedge\ \forall i' \in \{1, \ldots, i-1\}.\ s \mathbin{@} t_{i'} \rightsquigarrow \uparrow \\ \ \wedge\ \forall i' \in \{1, \ldots, i-1, i+1, \ldots, n\}.\ t_{i'} = t'_{i'} \end{array}}{\Diamond(s) \mathbin{@} c(t_1, \ldots, t_n) \rightsquigarrow c(t'_1, \ldots, t'_n)} \quad [\mathsf{one}^+]$$

$$\frac{s[v \mapsto \mu v.s] \mathbin{@} t \rightsquigarrow t'}{\mu v.s \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{rec}^+]$$

Fig. 4. **Positive rules of natural semantics for type-preserving strategies.**

The shown semantics commits to the simple model of a deterministic semantics: each strategy application evaluates to one term deterministically, or it fails, or it diverges. Further, the semantics of the choice combinator is left-biased and no non-local backtracking is enabled. There is also work which studies a model of non-determinism via sets or lists of possible results where the empty set represents failure; c.f., related work on ELAN, for example, [6,7]. Non-determinism is also enabled naturally by a monadic style, functional programming embedding, as we will discuss below.

$$\frac{\nexists\theta.\ \theta(t_l) = t}{t_l \rightarrow t_r \,@\, t \rightsquigarrow \uparrow} \quad [\mathsf{rule}^-]$$

$$fail \,@\, t \rightsquigarrow \uparrow \quad [\mathsf{fail}^-]$$

$$\frac{s_1 \,@\, t \rightsquigarrow \uparrow}{s_1; s_2 \,@\, t \rightsquigarrow \uparrow} \quad [\mathsf{seq}^-.1]$$

$$\frac{s_1 \,@\, t \rightsquigarrow t' \ \wedge\ s_2 \,@\, t' \rightsquigarrow \uparrow}{s_1; s_2 \,@\, t \rightsquigarrow \uparrow} \quad [\mathsf{seq}^-.2]$$

$$\frac{s_1 \,@\, t \rightsquigarrow \uparrow \ \wedge\ s_2 \,@\, t \rightsquigarrow \uparrow}{s_1 \nleftarrow\!\!+ s_2 \,@\, t \rightsquigarrow \uparrow} \quad [\mathsf{choice}^-]$$

$$\frac{\exists i \in \{1,\ldots,n\}.\ s \,@\, t_i \rightsquigarrow \uparrow}{\Box(s) \,@\, c(t_1,\ldots,t_n) \rightsquigarrow \uparrow} \quad [\mathsf{all}^-]$$

$$\frac{\forall i \in \{1,\ldots,n\}.\ s \,@\, t_i \rightsquigarrow \uparrow}{\Diamond(s) \,@\, c(t_1,\ldots,t_n) \rightsquigarrow \uparrow} \quad [\mathsf{one}^-]$$

$$\frac{s[v \mapsto \mu v.s] \,@\, t \rightsquigarrow \uparrow}{\mu v.s \,@\, t \rightsquigarrow \uparrow} \quad [\mathsf{rec}^-]$$

Fig. 5. **Negative rules of natural semantics for type-preserving strategies.**

2.3 Traversal schemes

Familiar traversal schemes Assuming ad-hoc notation for parametrized strategy definition, i.e., think of macro expansion, some familiar traversal schemes and necessary helpers can be defined; c.f., Fig. 6. We refer to [33,55] for a more elaborated discussion of the design space for traversal schemes.

An illustrative programming error The traversal scheme *stop_bu*(s) is in fact bogus; the argument s will never be applied. Instead, any application of the scheme will simply perform a deep identity traversal. This property can be proven with relatively little effort by induction on the structure of terms, also using the auxiliary property that "$\Box(s)$" is the identity for all constant terms, i.e., terms without any subterms—no matter what s. It is relatively unlikely that someone may put such a traversal scheme into a library without

$\mathit{full\_td}(s)$	$= \mu v.s; \Box(v)$	-- Apply s to each subterm in top-down manner
$\mathit{full\_bu}(s)$	$= \mu v.\Box(v); s$	-- Apply s to each subterm in bottom-up manner
$\mathit{once\_td}(s)$	$= \mu v.s \leftarrow\!\!+ \Diamond(v)$	-- Find one subterm (top-down) for which s succeeds
$\mathit{once\_bu}(s)$	$= \mu v.\Diamond(v) \leftarrow\!\!+ s$	-- Find one subterm (bottom-up) for which s succeeds
$\mathit{stop\_td}(s)$	$= \mu v.s \leftarrow\!\!+ \Box(v)$	-- Variant of *full_td*(s) which stops upon success
$\mathit{stop\_bu}(s)$	$= \mu v.\Box(v) \leftarrow\!\!+ s$	-- An illustrative programming error; see the discussion.
$\mathit{innermost}(s)$	$= \mathit{repeat}(\mathit{once\_bu}(s))$	-- A form of innermost normalization à la term rewriting.
$\mathit{repeat}(s)$	$= \mu v.\mathit{try}(s; v)$	-- Fixed point iteration; apply s until it fails.
$\mathit{try}(s)$	$= s \leftarrow\!\!+ \mathit{id}$	-- Recovery from failure of s with catch-all *id*.

Fig. 6. **Familiar traversal schemes.**

noticing the problem. However, in our experience, it happens easily in practice that a problem-specific traversal may end up suffering from the same kind of problem. We suggest to label this category as "mis-understood success and failure behavior implying dead code of a kind".

2.4 Laws and properties

Fig. 7 lists algebraic laws obeyed by the strategy primitives. These laws should be helpful in understanding the primitives and their use in traversal schemes. We refer to [30] for a mechanized model of traversal strategies, which proves these laws and additional properties. These laws provide intuitions with regard to the success and failure behavior of strategies, and they also hint at potential sources of dead code.

For instance, the fusion law states that two subsequent "$\Box$" traversals can be composed into one. Such a simple law does not hold for "$\Diamond$", neither does it hold generally for traversal schemes. In fact, little is known about algebraic laws for traversal schemes, but see [29,54] for some related research.

Basic properties of success and failure behavior of the type-preserving schemes were studied in [30], and we summarize those properties here. We say that a strategy s is *infallible* if it does not possibly fail, i.e., for any given term, it either succeeds or diverges; otherwise s fallible. The following properties hold [30]:

- If s is infallible, then *full_td* s and *full_bu* s are infallible.
- No matter the argument s, *stop_td* s and *innermost* s are infallible.
- No matter the argument s, *once_td* s and *once_bu* s are fallible.

[unit of “;”]	$id; s$	$=$	s	$= s; id$
[zero of “;”]	$fail; s$	$=$	$fail$	$= s; fail$
[unit of “$\leftarrow\!+$”]	$fail \leftarrow\!+ s$	$=$	s	$= s \leftarrow\!+ fail$
[left zero of “$\leftarrow\!+$”]	$id \leftarrow\!+ s$	$=$	id	
[associativity of “;”]	$s_1; (s_2; s_3)$	$=$	$(s_1; s_2); s_3$	
[associativity of “$\leftarrow\!+$”]	$s_1 \leftarrow\!+ (s_2 \leftarrow\!+ s_3)$	$=$	$(s_1 \leftarrow\!+ s_2) \leftarrow\!+ s_3$	
[left distributivity]	$s_1; (s_2 \leftarrow\!+ s_3)$	$=$	$(s_1; s_2) \leftarrow\!+ (s_1; s_3)$	
[one-layer identity]	$\square(id)$	$=$	id	
[one-layer failure]	$\diamond(fail)$	$=$	$fail$	
[fusion law]	$\square(s_1); \square(s_2)$	$=$	$\square(s_1; s_2)$	
[“$\square$” with a constant]	$constant(t)$	$\Rightarrow$	$\square(s) @ t$	$= t$
[“$\diamond$” with a constant]	$constant(t)$	$\Rightarrow$	$\diamond(s) @ t$	$= \uparrow$
[“$\square$” with a non-constant]	$\neg constant(t)$	$\Rightarrow$	$\square(fail) @ t$	$= \uparrow$
[“$\diamond$” with a non-constant]	$\neg constant(t)$	$\Rightarrow$	$\diamond(id) @ t$	$= t$
[~~commutativity of “;”~~]	$s; s'$	$\neq$	$s'; s$	
[~~commutativity of “$\leftarrow\!+$”~~]	$s \leftarrow\!+ s'$	$\neq$	$s' \leftarrow\!+ s$	
[~~right distributivity~~]	$(s_1 \leftarrow\!+ s_2); s_3$	$\neq$	$(s_1; s_3) \leftarrow\!+ (s_2; s_3)$	

Fig. 7. **Algebraic laws and non-laws of strategy primitives.** In a few laws, in fact, implications, we use an auxiliary judgement *constant* that holds for all constant terms, i.e., terms with 0 subterms.

The discussion of termination behavior for traversal schemes is more complicated, but let us provide a few intuitions here. That is, it is easy to see that full bottom-up traversal converges as long as its argument strategy does not diverge because the scheme essentially performs structural recursion on the input term. In contrast, full top-down traversal may diverge rather easily, as we will illustrate in §3. We will study termination in some detail in §5.

2.5 Functional traversal strategies

Fig. 8 shows the strategy types and strategy primitives of a Strafunski-like, Haskell-based combinator library [43].[1] There is one combinator for each al-

[1] A note on Haskell's generality: 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

```
-- Type-preserving and -unifying strategies
type TP m = forall x. Data x  => x -> m x
type TU r = forall x. Data x  => x -> r

-- Basis combinators
idTP      :: Monad m       => TP m
failTP    :: MonadPlus m   => TP m
failTU    :: MonadPlus m   => TU (m r)
constTU   ::                  r -> TU r
sequTP    :: Monad m       => TP m -> TP m -> TP m
bothTU    ::                  TU u -> TU u' -> TU (u,u')
choiceTP  :: MonadPlus m   => TP m -> TP m -> TP m
choiceTU  :: MonadPlus m   => TU (m r) -> TU (m r) -> TU (m r)

-- One-layer traversal combinators
allTP     :: Monad m      => TP m -> TP m
oneTP     :: MonadPlus m  => TP m -> TP m
allTU     ::                 TU r -> TU [r]

-- Strategy extension
adhocTP :: (Typeable x, Monad m) => TP m -> (x -> m x) -> TP m
adhocTU :: Typeable x            => TU r -> (x -> r) -> TU r
```

Fig. 8. **Interface for a Strafunski-like combinator library.**

ternative in the grammar of Fig. 3. There is no strategy form for rewrite rules though because we replace the rewriting idiom of rewrite rules by the functional programming idiom of pattern-matching functions on algebraic data types. We show both type-preserving and type-unifying combinators; c.f., see the postfix *TP* vs. *TU*. There are additional combinators for *strategy extension*; c.f., *adhocTP* and *adhocTU*—to be discussed in a second.

The types of the combinators directly refer to the semantic domains of strategies in Haskell. The types *TP* and *TU* are polymorphic function types with a type-class constraint *Data* which enables one-layer traversal and strategy extension, as we will discuss in a second. The details of this type class are irrelevant for the purpose of this paper.

The formal semantics of Fig. 4 and Fig. 5 is essentially encoded in the definitions of the combinators—except that we use a more general monadic scheme as opposed to a restricted success/failure semantics, which arises as the special

notably, termination) becomes more subtle. In this paper, we are limiting our discussion to finite, fully defined data. (The subject of coinductive strategies over coinductive types may be an interesting topic for future work.) We also skip over the issues of laziness in most cases.

case of the *Maybe* monad. That is, a strategy returns *Nothing* to signal failure, while a successful computation returns a value of the form *Just x*. In case we do not need even any sort of failure in a traversal program, then we also use the *Id* monad.

```
data Maybe x = Nothing | Just x
newtype Id x = Id { getId :: x }
```

In fact, the *List* monad provides another option. We can use it to replace the *Maybe*-based success and failure model with non-determinism. Yet other design points are reachable through other monads and monad transformers.

The definitions of the *basic* combinators are trivial: [2]

```
idTP     = return
failTP   = const mzero
failTU   = const mzero
constTU  = const
sequTP   = \f g x -> f x >>= g
bothTU   = \f g x -> (f x, g x)
choiceTP = \f g x -> f x `mplus` g x
choiceTU = \f g x -> f x `mplus` g x
```

The definitions of the combinators for one-layer traversal and extension cannot be easily given because they require substantial underlying machinery. Also, there exist different models using, for example, a universal representation [43], type class-based overloading of the combinators [38], or GADT-based views on data types [27]. (As an aside, the code development of sections 2–4 is based on the basic SYB model [38].) Fortunately, the precise model is not important for the purposes of the present paper, and hence, we can "define" the combinators through pseudo-code—as if they were supported natively by the Haskell language. Here are the "definitions" of *allTP* and *allTU*:

```
allTP f (C t1 ... tn) = f t1 >>= \t1' ->
                        ...
                        f tn >>= \tn' ->
                        return (C t1' ... tn')

allTU f (C t1 ... tn) = [f t1, ..., f tn]
```

[2] A comment on Haskell usage: In the definition of *choiceTP*, the operator *mplus* is addition on values of type *m x* for any *MonadPlus m* and *mzero* is the unit of addition. In the case of the *Maybe* monad, *mplus* is the operation that prefers the first operand over the second in returning either of them, or *Nothing*, if they are both *Nothing*. In the definition of *sequTP*, the operator ">>=" is the monadic bind operator which generalizes function application. In the case of the *Maybe* monad, ">>=" applies the right operand to the value that is extractable from the first operand, if any, and it returns *Nothing* otherwise.

In this pseudo-code, we assume that we can match arbitrary constructor patterns. In the type-unifying case, we simply apply the argument strategy to all immediate subterms, and combine the results in a list. (As an aside, Haskell's laziness allows us to write traversals based on *allTU* so that they still do not actually visit all of the children. Hence, strictly speaking, a combinator *oneTU* is not needed for expressiveness reasons.)

In the type-preserving case, we follow closely the letter of the natural semantics. That is, again, we apply the argument strategy to all immediate subterms, but we construct a term from the results while reusing the outermost constructor of the input term. Because of monadic style, the sub-traversals may fail, and thereby make the complete one-layer traversal fail. For brevity, we omit the definition of *oneTP*.

It remains to discuss the combinators for strategy extension. These combinators embody *type-case* such that a polymorphic function can be "updated" (say, "extended") by a monomorphic function. That is, the resulting function applies the monomorphic operand when possible, and resorts to the polymorphic operand, otherwise. Thus, in pseudo-code:

```
adhocTP = adhoc
adhocTU = adhoc
adhoc g f x  | argumentTypeOf f == typeOf x  = f x
             | otherwise                     = g x
```

In the simple, untyped model of §2.1–§2.4, we can arbitrarily compose strategies. In particular, rewrite rules were no different from other strategies. In contrast, the Haskell model provides two levels of dispatch. At the upper level, we perform type-case through the ad-hoc combinators; at the lower level, we are committed to a specific type, and we perform regular pattern matching. (We mention in passing that there exists a proposal [15] for a language design that collapses the two levels of dispatch while staying otherwise in the realm of strong static polymorphic typing.) Type-case is implicit in the untyped model of §2.1–§2.4. These different designs come with different trade-offs, and we refer to [35] for an incomplete discussion of the matter. Our discussion of programming errors in §3 will again touch upon such dispatching subtleties.

2.6 Functional traversal schemes

Fig. 9 rephrases the traversal schemes of Fig. 6 within the bounds of the Haskell model, and adds three major schemes for type-unifying traversal. (The postfix "cl" hints at "collection".) Two of the collection schemes use a monoid type parameter to reduce intermediate results; c.f., the uses of *mempty* and *mconcat*.

The Haskell models of the type-preserving schemes are entirely straightfor-

Typical monadic, type-preserving traversal schemes

```
full_td , full_bu          :: Monad m      => TP m -> TP m
once_td, once_bu           :: MonadPlus m => TP m -> TP m
stop_td                    :: MonadPlus m => TP m -> TP m
innermost, repeat, try     :: MonadPlus m => TP m -> TP m

full_td  s      = s `sequTP` allTP (full_td  s)
full_bu  s      = allTP (full_bu  s) `sequTP` s
once_td s       = s `choiceTP` oneTP (once_td s)
once_bu s       = oneTP (once_bu s) `choiceTP` s
stop_td  s      = s `choiceTP` allTP (stop_td  s)
stop_bu  s      = allTP (stop_bu s)  `choiceTP` s
innermost s     = repeat  (once_bu s)
repeat  s       = try  (s  `sequTP` repeat  s)
try  s          = s `choiceTP` idTP
```

The most basic type-unifying traversal schemes

```
-- Query each node and collect all  results  in  a  list
full_cl  :: Monoid u => TU u -> TU u
full_cl  s = mconcat . uncurry (:)  .  bothTU s (allTU ( full_cl  s))

-- Collection with stop
stop_cl  :: Monoid u => TU (Maybe u) -> TU u
stop_cl  s = maybe mempty id
           . (s  `choiceTU` (Just  .  mconcat . allTU ( stop_cl  s)))

-- Find a node to query in top-down, left-to-right manner
once_cl  :: MonadPlus m => TU (m u) -> TU (m u)
once_cl  s = s `choiceTU` ( msum . allTU (once_cl s))
```

Fig. 9. **Traversal schemes in Haskell.** Compared to prior art, we have not hard-wired monadic style into the type-unifying schemes because we have come to realize that monadic style can be easily enough composed with monoidal style.

ward, but two details are worth noticing as they relate somewhat to our central topic of programming errors. First, the function definitions use general recursion, thereby implying the potential for divergence. Second, the types carefully express whether a basic monad is sufficient as opposed to a monad with "+" (and "0") is needed. Thereby the programmer receives a hint at the potential use of success and failure behavior for the control of a traversal scheme. For instance, *full_td*'s type uses *Monad* only; hence, it is perfectly acceptable for the argument strategy to be universally succeeding.

-- Increase the salaries of all employees.

```
increase_all_salaries  :: TP Maybe
increase_all_salaries  = full_td (idTP `adhocTP` i)
where
  i (Employee n s) = Just (Employee n (s+1))
```

-- Total the salaries of all employees.

```
total_all_salaries  :: TU Float
total_all_salaries  = getSum . full_cl (adhocTU (constTU mempty) f)
where
  f (Employee _ s) = Sum s
```

-- Total the salaries of all employees who are not managers.

```
total_all_non_managers :: TU Float
total_all_non_managers = getSum . stop_cl type_case
 where
   type_case :: TU (Maybe (Sum Float))
   type_case = constTU Nothing `adhocTU` employee `adhocTU` manager
   employee (Employee _ s) = Just (Sum s)
   manager (Manager _) = Just (Sum 0)
```

Fig. 10. **Samples of traversal programs.**

2.7 Functional traversals

We return to the illustrative traversal programming scenarios of Fig. 2. Three scenarios are implemented in Fig. 10. An implementation of the remaining scenarios is a good exercise that we recommend.

The implementation of *increase_all_salaries* is straightforward: we simply pick a scheme for *full* traversal such that we reach each node, and we extend the polymorphic identity function with a monomorphic function for employees so that we increase (in fact, increment) their salary components.

The implementation of *total_all_salaries* is straightforward, too, and one may expect that the implementation of *total_all_non_managers* easily follows. However, the basic collection scheme *full_cl* is inappropriate for *total_all_non_managers* because a full traversal would still reach and total managers, which are employees. Further, an always failing default is needed here—again, in contrast to the simpler case of *total_all_salaries*. Finally, the solution depends on details of data design. We use an extra type-specific case for managers to stop collection at the manager level. Without a type distinction for managers vs. employees, the traversal program would need to exploit the special *position* of managers within department terms as opposed to the manager *type*. All the remaining scenarios of Fig. 2 call for similar considerations, and hence, programming errors are quite conceivable.

3 Inventory of programming errors

This section presents a fine-grained inventory of programming errors in traversal programming with traversal strategies. We use a deceptively simple scenario as the running example because we want to focus on idiomatic issues as opposed to logical challenges. That is, we want to clarify that the mere idioms of traversal programming are insufficiently understood and too unconstrained in practice. Hence, programming errors are likely to occur.

3.1 The running example

To use a purposely simple example, consider the transformation problem of "incrementing all numbers in a term". 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.
- It does not terminate.
- It does not modify the input. (It does not increment the numbers.)
- It modifies the input, but numbers are changed more arbitrarily.

For concreteness' sake, we operate on n-ary "trees" over "naturals" as numbers. Further, we assume a Peano-like definition of the data type for naturals. The Peano-induced recursion implies a simple form of nesting. [3]

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:

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

[3] It goes without saying that the Peano-induced nesting form is contrived, but its inclusion allows us to cover nesting as such—any practical scenario of traversal programming involves nesting at the data-modelling level; c.f., nesting of departments in the company example, or nesting of expressions or function-definition blocks in the AST example of the introduction.

```
tree3 = Node { rootLabel = Succ Zero, subForest = [tree1, tree1] }  -- Two subtrees
```

The rewrite rule for incrementing naturals is represented as follows:

```
increment n = Succ n
```

In fact, let us use monadic style because the basic Strafunski-like library assumes monadic style for all combinators—in particular, for all arguments. Hence, we commit to the *Maybe* monad and its constructor *Just*:

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

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*, *once_td*, and *innermost*, which traversal scheme is the correct one for the problem at hand?**

An experienced strategist may quickly exclude a few 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 potential programming errors.

3.2 Unbounded recursion

Let us attempt a full top-down traversal. Alas, the first attempt diverges: [4]

```
> full_td (adhocTP idTP 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:

```
> innermost (adhocTP failTP increment) tree1
... no output ever is printed ...
```

[4] Throughout the section, we operate at the Haskell prompt; c.f., input past the ">" prompt sign, and we show the resulting output, if any, right below the input.

The combinator *innermost* repeats *once_bu* until it fails, but it never fails because there is always a redex to which to apply the *increment* rule. Hence, *tree1* is rewritten indefinitely.

3.3 Incorrect quantification

Let us try yet another scheme, *full_bu*:

```
> full_bu (adhocTP idTP 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.

The same kind of error could occur in the implementation of any other scenario as long as it involves nesting. In real-world scenarios, the nesting may actually arise also through mutual (data-type-level) recursion as opposed to the directly recursive definition of the natural numbers.

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.

3.4 Incorrect polymorphic default

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

```
> stop_td (adhocTP idTP 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 *idTP* makes the strategy stop for any subterm that is not a natural. Let us replace *idTP* by *failTP*. Finally, we arrive at a proper solution for the original problem statement:

```
> stop_td (adhocTP failTP increment) tree1
Just (Node {rootLabel = Succ Zero, subForest = []})
```

failTP 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 § 3.3. Back then, the polymorphic default *idTP* was appropriate for *full_bu*. In contrast, the default *failTP* is not appropriate:

```
> full_bu (adhocTP failTP increment) tree1
Nothing
```

The default challenge is also relevant for several of the illustrative scenarios from the introduction. We also refer back to the traversal scheme *once_cl* of Fig. 9, which we deployed in one specific traversal in Fig. 10: its relatively complicated type virtually calls for trouble.

3.5 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: [5]

```
-- 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:

```
> stop_td (adhocTP failTP increment_even) tree1
Just (Node {rootLabel = Succ Zero, subForest = []})
```

[5] 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.

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
```

3.6 Unreachable constituents

Consider the following patterns of strategy expressions:

- *adhocTP* (*adhocTP g* s_1) s_2
- *choiceTP* f_1 f_2
- *sequTP* 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 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:

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

atEven 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. § 3.5.) Intuitively, we wish to chain together these type-specific cases so that they both are tried. Let us attempt the following composition:

```
> stop_td (adhocTP (adhocTP failTP atEven) atOdd) tree1
Just (Node {rootLabel = Zero, subForest = []})
```

Alas, no incrementing seems to happen. The problem is that there are two type-specific cases for natural numbers, and the case for odd numbers dominates the one for even numbers. In the sample tree, the natural number, *Zero*, is even. Hence no incrementing happens. The two rewrite rules should be composed at the monomorphic level of the specific type of natural numbers—as opposed to the polymorphic level of strategy extension.

Here are rank-1 combinators for sequential composition and composition by choice. These are trivial function combinators on monadic functions, which we can use whenever we need to compose functions at the monomorphic level.

```
msequ :: Monad m => (x -> m x) -> (x -> m x) -> x -> m x
msequ s s' x = s x >>= s'

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

Using *mchoice*, we arrive at a correct composition:

```
> stop_td (adhocTP failTP (mchoice atEven atOdd)) tree1
Just (Node {rootLabel = Succ (Succ Zero), subForest = []})
```

3.7 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):

```
> stop_td (adhocTP failTP increment) tree2
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).

3.8 Incorrect success and failure handling

The earlier problems with (polymorphic and monomorphic) defaults feed into a more general kind of problem: misunderstood success and 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 (adhocTP failTP 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 and failure behavior only leads to incorrect text output, but in general, a more complex program may be incorrectly composed in ways that commit to misunderstood success and failure behavior, or incorrect assumptions about changed terms. In addition to correctness, there is also the potential for defensive and convoluted code. For instance, in the following strategy expression, the application of *try* is superfluous because the traversal to which it is applied cannot possibly fail, and hence we face a kind of a programming error:

```
try (full_td (adhocTP idTP increment))
```

We should mention that part of the confusion regarding success and failure behavior stems from the overloaded interpretation of success and failure—to relate to either strategy control or pre-/post-condition checking. A future language design for strategic programming may favor to separate these aspects—possibly inspired by forms of exception handling known from the general programming field [57,48].

3.9 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 applying *adhocTP*.

(iv) What is the composition operator?

- Sequential composition.
- Composition by choice.
- Type case (strategy extension).

We return to the example from § 3.6, 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 *failTP*; 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.

1. *stop_td (adhocTP (adhocTP failTP atEven) atOdd)*
2. *stop_td (adhocTP failTP (mchoice atEven atOdd))*
3. *stop_td (adhocTP failTP (msequ atEven atOdd))*
4. *stop_td (choiceTP (adhocTP failTP atEven) (adhocTP failTP atOdd))*
5. *stop_td (sequTP (adhocTP failTP atEven) (adhocTP failTP atOdd))*
6. *choice (stop_td (adhocTP failTP atEven)) (stop_td (adhocTP failTP atOdd))*
7. *sequ (stop_td (adhocTP failTP atEven)) (stop_td (adhocTP failTP atOdd))*

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.

4 Constrained traversal schemes

It seems plausible to ask whether we can attack some of the identified categories of programming errors by constraining the traversal schemes, subject to refined types. We will explore this question here—mainly within the bounds of the Strafunski-like Haskell incarnation of traversal strategies.

Despite such focus on Haskell, we expect the overall refinements to be generally insightful and potentially useful for other incarnations of strategic programming, say in Java, Scala, or Stratego, and future language designs with designated support for traversal strategies. Basic "reading knowledge" of Haskell should be sufficient.

Ultimately, this section shows that the embedding approach for traversal strategies is really limited. Most of the illustrated efforts cause encoding burden, and some of the categories of programming errors, e.g., divergent traversal, appear to be out of reach. Hence, we complement each refinement with a short indication of the "lessons learned" on a future language design (or type-system design) that is meant to better facilitate traversal programming.

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 such that the problem-specific arguments become monomorphic. There has been a proposal of a variation on "Scrap Your Boilerplate" that also points into this direction [49].

The following primed definitions take a monomorphic argument *s*, which is then generalized *within the definition* by means of the appropriate polymorphic default, *idTP* or *failTP*. We delegate to the more polymorphic schemes otherwise:

```
full_td' :: (Data x, Data y, Monad m) => (x -> m x) -> y -> m y
once_bu' :: (Data x, Data y, MonadPlus m) => (x -> m x) -> y -> m y
stop_td' :: (Data x, Data y, MonadPlus m) => (x -> m x) -> y -> m y
...
full_td' s = full_td (adhocTP idTP s)
once_bu' s = once_td (adhocTP failTP s)
stop_td' s = stop_td (adhocTP failTP s)
...
```

Note that the types of the primed schemes are simpler: there are no occurrences of "***forall***" anymore. These rank-1 schemes reduce programming errors as follows. Most obviously, polymorphic defaults are correct by design because they are hard-wired into the definitions. 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" (c.f., §3.9) 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. [63,43,44] for some concrete examples; also, see Fig. 10 for a simple example. The problem is that we may need *multiple* type-specific cases. 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.

A future language design 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 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 and failure behavior of traversal schemes and their arguments.

At this point, some signatures hint at the potential use of the associated traversal schemes with "occasionally failing vs. potentially never failing" arguments. For instance, see the distinguished use of *Monad* vs. *MonadPlus* in the following signatures:

```
full_td :: Monad m => TP m -> TP m
once_bu :: MonadPlus m => TP m -> TP m
stop_td :: MonadPlus m => TP m -> TP m
```

However, such hinting does not imply checks, and not even the hints are comprehensive. For instance, a programmer may still pass a notoriously failing argument to *full_td* despite the signature's hint that a universally succeeding argument may be perfectly acceptable. Related to this problem is the programmer's expectation to effectively document that a *specific* full top-down traversal is meant to universally succeed. The general traversal scheme is polymorphic in the monad-type constructor, and hence both "universal success and potential failure" are options.

We can provide infallible variations on the generic traversal schemes; remember the discussion of infallibility in §2.4. To this end, we use the identity monad whenever we want to require or imply infallibility. For the type-preserving traversal schemes of Fig. 9 the following more restrictive types of infallible traversals make sense:

```
full_td '    :: TP Id -> TP Id
full_bu '    :: TP Id -> TP Id
stop_td '    :: TP Maybe -> TP Id
innermost' :: TP Maybe -> TP Id
repeat '     :: TP Maybe -> TP Id
try '        :: TP Maybe -> TP Id
```

(There is no less fallible variation on *once_bu*.) The primed definitions *full_td'* and *full_bu'* simply delegate, but the other primed definitions need to be redefined from scratch because they need to compose infallible and fallible strategy types—also subject to a designated form of choice. Thus:

```
full_td ' s     = full_td s
full_bu ' s     = full_bu s
stop_td ' s     = s `choiceTP'` allTP (stop_td ' s)
innermost' s    = repeat' (once_bu s)
repeat ' s      = try' (s `sequTP` (Just . getId . repeat ' s))
try ' s         = s `choiceTP'` Id

choiceTP' :: TP Maybe -> TP Id -> TP Id
choiceTP' f g x = maybe (g x) Id (f x)
```

A programmer should favor the infallible schemes, whenever possible, and fall 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. Just like the previous refinement, this variation on the general library increases the API surface since it does not cover all use cases of the general definitions. For instance, we cannot use the new variations if we need a monadic effect other than failure.

A future language design 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. As a result, a different type system may be needed. Also, it may be beneficial to separate success and failure behavior from the ordinary monadic layer of the strategy types. More specifically, in the context of Haskell, it may be interesting to leverage existing concepts for type-checked exception handling [48] for the benefit of documenting and checking claims about success and failure behavior.

4.3 Checked ad-hoc chains

The problem of unreachable cases in ad-hoc chains (c.f., §3.6) can be avoided by a type check that specifically establishes that the cases cover distinct 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. Here, we note that we gain similar benefits as in §4.1—except that we do not need to restrict the arguments of traversal schemes to be monomorphic.

In the following, we use an advanced Haskell library, HList [32], to describe the constituents of a traversal scheme as a *family of monomorphic cases*, which is, in fact, an appropriately constrained, heterogeneous list of functions. Consider

```
-- Type-class-polymorphic type of familyTP
class (Monad m, HTypeIndexed f) => FamilyTP f m
  where
    familyTP :: GenericM m -> f -> GenericM m

-- Empty list case
instance Monad m => FamilyTP HNil m
  where
    familyTP g _ = g

-- Non-empty list case
instance ( Monad m
         , FamilyTP t m
         , Data x
         , HOccursNot (x -> m x) t
         )
           => FamilyTP (HCons (x -> m x) t) m
  where
    familyTP g (HCons h t) = adhocTP (familyTP g t) h
```

Fig. 11. Derivation of a polymorphic strategy from a list of type-specific cases.

the pattern that we used so far: *adhocTP* (*adhocTP 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. Such an ad-hoc chain can be represented as the heterogeneous list *HCons* s_1 (*HCons* s_2 *HNil*) where *HNil* is the representation of the empty heterogeneous list, and *HCons* is the list constructor for non-empty heterogeneous lists. Using established type-class-based programming techniques, a heterogeneous list can be converted to a plain ad-hoc chain by a type-class-polymorphic function, *familyTP*, 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 delegate to the original schemes:

```
full_td' s = full_td (familyTP id s)
stop_td' s = stop_td (familyTP fail s)
once_bu' s = once_bu (familyTP fail s)
...
```

The function *familyTP* is defined in Fig. 11. The list of type-specific cases is constrained to only hold elements of distinct types; c.f., the constraint *HTypeIndexed*, which is provided by the HList library. Also notice that the element types are constrained to be function types following the monadic, type-preserving scheme; c.f., *x -> m x*. As a proof obligation for the *HTypeIndexed* constraint, the instance for non-empty lists must establish that the head's type does not occur again in the tail of the family; cf. the constraint *HOccursNot*, which is again provided by the HList library.

A future language design may provide a general enough form of type case more directly [13], and hence the above approach may be achievable without the current encoding burden. For instance, type errors in type-class-based programming are rather involved; they are not referring to the abstraction level of the programmer.

4.4 Reachable type constraints

Let us consider again the subtle form of unreachable (monomorphic) cases, where they turn out to be unreachable just because their *types* cannot be reached by the traversal that is assumed to start from a certain root type. For instance, we cannot reach a natural number in a tree of Booleans with any type-preserving traversal scheme; c.f., §3.7.

To avoid related programming errors, we may impose more restrictive types on the traversal schemes such that the types of type-specific cases are compared with the root type such that reachability is checked. This idea has also been presented in [37] in the context of applications of "Scrap Your Boilerplate" to the XML-programming context.

Using type-class-based programming again, we can define a type-level relation on types, which captures whether or not terms of one type may occur within terms of another type. (In fact, we prefer to make this relation reflexive.) The relation is modelled by the following type class:

```
class ReachableFrom x y
instance ReachableFrom 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 *ReachableFrom*:

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

Now we have to use *ReachableFrom* within a more constrained type for the traversal schemes. Without loss of generality, we only consider the less generic schemes of §4.1. For instance, the rank-1 variation of full top-down traversal would be constrained as follows:

```
full_td'' :: (Data x, Data y, Monad m, ReachableFrom x y)
          => (x -> m x)
          -> y -> m y
full_td'' = full_td'
```

Compared to *full_td'* of §4.1, we have simply added the *ReachableFrom* constraint. Of course, this constraint does not change the behavior of full top-down traversal, and hence, there is no correctness issue. However, it is not straightforward to see (or to prove) that the additional constraint does not remove any useful behavior. For a library scheme, it may be acceptable to spend effort on these constraints. For problem-specific traversals, this effort may not always be acceptable.

A future language design should support reachability constraints more directly—without the current need for a user-defined type class *ReachableFrom*. Constraint annotation and checking should also work easily with polymorphic arguments of traversal programs. In fact, in some cases, it should be possible to infer reachability constraints from the traversal programs themselves—one of the analyses from the next section demonstrates a corresponding approach in a simplified setting.

5 Static program analysis

Arguably, the refinement experiments of the previous section suggest that the embedding approach to traversal strategies is severely limited. First, certain categories of programming errors may be very hard to address, e.g., divergent traversal. Second, other categories require considerable encoding overhead and imply increased API surface. Third, a programmer of problem-specific traversals will not benefit from the refined types, if (s)he bypasses the library with refined types for traversal schemes.

Hence, deep support for traversal strategies may be preferable. Accordingly, we engage in an effort for static program analysis in determining selected properties of traversal programs—both traversal schemes in a library and arbitrary, problem-specific traversal programs. We see work on static analysis as an important cornerstone of a future language design for strategic programming, or, alternatively, as a strong test case for a language with a more programmable type system capable of seamlessly integrating special-purpose program analyses with the host language's type system.

We are interested here in different properties of traversal programs:

- their success and failure behavior,

- their termination behavior, and
- the reachability of type-specific cases (as a kind of dead code detection).

The first and the second properties are applicable even to fully generic traversal programs (most notably, traversal schemes); in contrast, the third property necessarily involves a problem-specific signature of terms to be traversed.

Some analyses may process the traversal programs *as is*, other analyses may require the programmer to provide "contracts" describing the intended behaviour of the programs.

We use abstract interpretation and deductive techniques (in particular, special-purpose type systems) for the specification and implementation of the analyses. In all cases, we have modeled the analyses algorithmically in Haskell. All but the routine parts of the analyses are included into the text, a cursory understanding of the analyses does not require Haskell proficiency.

5.1 A model of type-preserving strategies

All the subsequent, algorithmic specifications of analyses rely on a simplified, Haskell-based model of (type-preserving) strategies; c.f., Fig. 12 for the syntax of strategies which is defined in terms of an algebraic data type for strategies.

Clearly, the data type covers the known strategy combinators; *Id*, *Fail*, and so forth. The constructors *Rec* and *Var* model fixed-point combinator and recursive reference, respectively. We can model familiar traversal schemes as *TP* terms; see at the bottom of Fig. 12. A reference semantics, modelling the strategies as possibly failing functions on a homogeneous type of terms, is given in Fig. 13.

The chosen model of recursive closures relies on a designated type parameter *x* for the type of the fixed point. A type parameter is needed because the actual type differs for the standard semantics, an abstract interpretation, or a type system. In the case of a standard semantics, the type parameter would need to be instantiated to the semantic domain for strategies, such as *Term −> Maybe Term*. In contrast, in the case of a type system, the type parameter would need to be instantiated to the type of type expressions, thereby modeling, for example, assumptions about recursive references.

5.2 Analysis of success and failure behavior

Given a strategy expression, we would like to determine whether the strategy can be guaranteed to succeed (read as "always succeeds"). Similar analyses are conceivable for cases such as "always' fails" and "sometimes succeeds".

```
-- Syntactical domain for strategies
data TP x
  = Id
  | Fail
  | Seq (TP x) (TP x)
  | Choice (TP x) (TP x)
  | Var x
  | Rec (x -> TP x)
  | All (TP x)
  | One (TP x)

-- Traversal schemes and helpers
full_bu  s    = Rec (\x -> Seq (All (Var x)) s)
full_td  s    = Rec (\x -> Seq s (All (Var x)))
once_bu  s    = Rec (\x -> Choice (One (Var x)) s)
once_td  s    = Rec (\x -> Choice s (One (Var x)))
stop_bu  s    = Rec (\x -> Choice (All (Var x)) s)
stop_td  s    = Rec (\x -> Choice s (All (Var x)))
innermost s = repeat (once_bu s)
try s         = Choice s Id
repeat s      = Rec (\x -> try (Seq s (Var x)))
```

Fig. 12. **Haskell-based model of syntax of type-preserving strategies.**

We will first apply abstract interpretation to the problem, but come to the conclusion that the precision of the analysis is insufficient to yield any non-trivial results. We will then retry by using a type-system-based approach; the latter approach provides sufficient precision. The first approach nevertheless provides insight into success and failure behavior, and the overall framework for abstract interpretation can be later re-purposed for another analysis.

5.2.1 An abstract interpretation-based approach

We use the following lattice for the abstract domain for a simple success/failure analysis. [6]

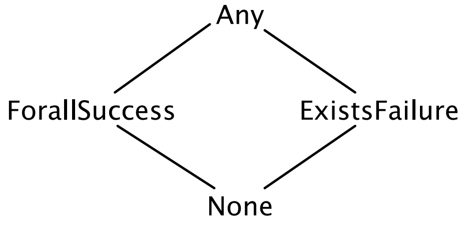
[6] We use the general framework of abstract interpretation by Cousot and Cousot [12,11]; we are specifically guided by Nielson and Nielson's style as used in their textbooks [51,52].

```
-- Terms
data Term  = Term Constr [Term]
type Constr = String

-- The semantic domain for strategies
type Meaning = Term -> Maybe Term

-- Compositional semantics
interpret  :: TP Meaning -> Meaning
interpret  Id              = Just
interpret  Fail            = const Nothing
interpret  (Seq s s')      = maybe Nothing (interpret s') . interpret s
interpret  (Choice s s')   = \t -> maybe (interpret s' t) Just (interpret s t)
interpret  (Var x)         = x
interpret  (Rec f)         = fixProperty (interpret . f)
interpret  (All s)         = transform (all (interpret s))
interpret  (One s)         = transform (one (interpret s))

-- Fixed-point property-based fixed-point combinator
fixProperty  :: (x -> x) -> x
fixProperty f = f (fixProperty f)

-- Transform immediate subterms
transform :: ([Term] -> Maybe [Term]) -> Meaning
transform f (Term c ts)
 = maybe Nothing (Just . Term c) (f ts)

-- Transform all terms in a list
all  :: Meaning -> [Term] -> Maybe [Term]
all f ts = kids ts'
 where
  ts' = map f ts
  kids [] = Just []
  kids (Just t':ts') = maybe Nothing (Just . (:) t') (kids ts')

-- Transform one term in a list
one :: Meaning -> [Term] -> Maybe [Term]
one f ts =  kids ts ts'
 where
  ts' = map f ts
  kids [] [] = Nothing
  kids (t:ts) (Nothing:ts') = maybe Nothing (Just . (:) t) (kids ts ts')
  kids (_:ts) (Just t':ts') = Just (t':ts)
```

Fig. 13. **Haskell-based interpreter of type-preserving strategies.**

```
-- General framework for abstract domains

class Eq x => POrd x
 where
  (<=) :: x -> x -> Bool
  (<) :: x -> x -> Bool
  x < y = not (x==y) && x <= y

class POrd x  => Bottom x   where bottom :: x
class POrd x  => Top x      where top :: x
class POrd x  => Lub x      where lub :: x -> x -> x



-- The abstract domain for success/failure  analysis

data Sf = None | ForallSuccess |  ExistsFailure  | Any

instance POrd Sf
 where
  None <= _     = True
  _     <= Any = True
  x     <= y     = x == y

instance Bottom Sf  where bottom = None
instance Top Sf       where top = Any

instance Lub Sf
 where
  lub None x        = x
  lub x      None   = x
  lub Any  x        = Any
  lub x      Any    = Any
  lub x      y      = if x == y then x else Any
```

Fig. 14. **The abstract domain for success and failure behavior.**



The bottom element *None* represents the absence of any information, and gives the starting point for fixed point iteration. The two data values above represent the two cases of:

- there is no value where the strategy fails: *ForallSuccess*
- there is a failure point for the strategy: *ExistsFailure*.

Note that this is a *partial correctness* analysis. Hence, in none of the cases is it implied that the program terminates for all arguments. In the former case, we also speak of an "*infallible*" strategy; c.f., §2.4. The "top" value, *Any*, represents the result of an analysis that concludes with both of the above cases as being possible. Such a result tells us nothing of value.

We refer to Fig. 14 for the Haskell model of the abstract domain. We assume appropriate type classes for partial orders (*POrd*), least elements (*Bottom*), greatest elements (*Top*), and least upper bounds (*Lub*).

The actual analysis is shown in full detail in Fig. 15. We discuss the analysis case by case now. The base cases can be guaranteed to succeed (*Id*) and to fail (*Fail*). The composition functions for the compound strategy combinators can easily be verified to be monotone.

In the case of sequential composition, we infer success if both of the operands succeed, while (definite) failure can be inferred only if the first operand fails. The reason for this is that the failure point in the domain of the second operand may not be in the range of the first. Hence, we conclude with *Any* in some cases. In the case of choice, we infer success if either of the components succeeds, while failure cannot be inferred at all. The reason for this is that two failing operands may have different failure points. Hence, we have to conclude again with the imprecise *Any* value in some cases.

A reference to a *Var* simply uses the information it contains, and *fixEq* is the standard computation of the least fixed point within a lattice, by iteratively applying the function to the *bottom* element (in our analysis *None*). We infer success for an "all" traversal if the argument strategy is guaranteed to succeed; likewise, the "all" traversal has a failure point if the argument strategy has a failure point (because we could construct a term to exercise the failure point on an immediate subterm position, assuming a homogeneous set of terms). We infer *ExistsFailure* for a "one" traversal because it has a failure point for every constant term, regardless of the argument strategy. Fig. 16 shows the results of the analysis.

The columns are labeled by the assumption for the success and failure behavior of the argument strategy s of the traversal scheme. There is no column for *None* since this value is only used for fixed-point computation. There are a number of cells with the *None* value, which means that the analysis was not able to make any progress during the fixed point computation. The analysis is patently useless in such cases. There are a number of cells with the *Any* value,

```
-- The actual analysis
analyse :: TP Sf -> Sf
analyse Id           = ForallSuccess
analyse Fail         = ExistsFailure
analyse (Seq s s')   = analyse s `seq` analyse s'
analyse (Choice s s') = analyse s `choice` analyse s'
analyse (Var x)      = x
analyse (Rec f)      = fixEq (analyse . f)
analyse (All s)      = analyse s
analyse (One s)      = ExistsFailure

-- Equality-based fixed-point combinator
fixEq :: (Bottom x, Eq x) => (x -> x) -> x
fixEq f = iterate bottom
 where
  iterate x = let x' = f x
              in if (x==x') then x else iterate x'

-- Abstract interpretation of sequential composition
seq :: Sf -> Sf -> Sf
seq None          _             = None
seq ForallSuccess None          = None
seq ForallSuccess ForallSuccess = ForallSuccess
seq ForallSuccess _             = Any
seq ExistsFailure _             = ExistsFailure
seq Any           _             = Any

-- Abstract interpretation of left-biased choice
choice :: Sf -> Sf -> Sf
choice ForallSuccess _             = ForallSuccess
choice _             ForallSuccess = ForallSuccess
choice None          _             = None
choice _             None          = None
choice _             _             = Any
```

Fig. 15. **Abstract interpretation for analysing the success and failure behavior of traversal programs.**

which means that the analysis concluded with an imprecise result: we do not get to know anything of value about the the success and failure behavior in such cases.

All the cells with values *ForallSuccess* and *ExistsFailure* are as expected, but overall the analysis fails to recover behavior in most cases. For instance, we know that a stop-top-down traversal is guaranteed to succeed. That is, such

	s :: *ForallSuccess*	*s* :: *ExistsFailure*	*s* :: *Any*
full_bu s	*None*	*None*	*None*
full_td s	*None*	*ExistsFailure*	*Any*
once_bu s	*ForallSuccess*	*Any*	*Any*
once_td s	*ForallSuccess*	*Any*	*Any*
stop_bu s	*ForallSuccess*	*None*	*None*
stop_td s	*ForallSuccess*	*None*	*None*
innermost s	*ForallSuccess*	*ForallSuccess*	*ForallSuccess*

Fig. 16. **Exercising success/failure analysis on common traversal schemes.**

traversal succeeds eventually for "leaves", i.e., terms without subterms, and it succeeds as well for every term for which the traversal succeeds for all immediate subterms. Hence, by induction over the depth of the term, the traversal succeeds universally.

The actual recursion over the *structure* of a term is different from the iteration used by *fixEq*, which fails to capture global success. It is not straightforward to improve the abstract interpretation-based approach so that it would compute more useful results. Hence, we will investigate a type-system-based approach now, which has different characteristics of dealing with recursion. Later on, we will revisit abstract interpretation in the context of a reachability analysis for type-specific cases. For that sort of application of abstract interpretation it is appropriate to incorporate a specific signature into the analysis; this line of refinement is not generally useful for analysing the success and failure behavior, which ought to be applicable to fully generic traversal programs.

5.2.2 A type system-based approach

Our intention is to capture when a strategic program can be guaranteed always to yield a value *if it terminates*, that is, it fails for no input. We use the type True for such a situation and False for the lack thereof.

The rules in Fig. 17 describe a typing judgement such that $\Gamma \vdash s : \mathsf{True}$ is intended to capture infallible strategies, i.e., (in the context Γ) the *strategy s does not fail for any argument t*. That is, for no t is it the case that $s\,@\,t \rightsquigarrow \uparrow$, according to the semantics in Fig. 4–Fig. 5. Fig. 18 rephrases Fig. 17 in a directly algorithmic manner in Haskell—also providing type inference.

The property of infallibility is undecidable, and hence, the type system will not identify all strategies of type True, but it is guaranteed to be sound, in

$\Gamma \vdash \mathit{id} : \mathsf{True}$ [id$^{\mathrm{SF}}$]

$\Gamma \vdash \mathit{fail} : \mathsf{False}$ [fail$^{\mathrm{SF}}$]

$$\frac{\Gamma \vdash s_1 : \mathsf{True} \ \wedge \ \Gamma \vdash s_2 : \mathsf{True}}{\Gamma \vdash s_1; s_2 : \mathsf{True}} \quad [\mathsf{sequ.1}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s_1 : \mathsf{False} \ \wedge \ \Gamma \vdash s_2 : \tau}{\Gamma \vdash s_1; s_2 : \mathsf{False}} \quad [\mathsf{sequ.2}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s_1 : \tau \ \wedge \ \Gamma \vdash s_2 : \mathsf{False}}{\Gamma \vdash s_1; s_2 : \mathsf{False}} \quad [\mathsf{sequ.3}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s_1 : \mathsf{False} \ \wedge \ \Gamma \vdash s_2 : \mathsf{True}}{\Gamma \vdash s_1 \nleftarrow\!\!\!+ s_2 : \mathsf{True}} \quad [\mathsf{choice.1}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s_1 : \mathsf{False} \ \wedge \ \Gamma \vdash s_2 : \mathsf{False}}{\Gamma \vdash s_1 \nleftarrow\!\!\!+ s_2 : \mathsf{False}} \quad [\mathsf{choice.2}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s_1 : \mathsf{True} \ \wedge \ \Gamma \vdash s_2 : \tau}{\Gamma \vdash s_1 \nleftarrow\!\!\!+ s_2 : \mathsf{True}} \quad [\mathsf{choice.3}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash \Box(s) : \tau} \quad [\mathsf{all}^{\mathrm{SF}}]$$

$$\frac{\Gamma \vdash s : \tau}{\Gamma \vdash \Diamond(s) : \mathsf{False}} \quad [\mathsf{one}^{\mathrm{SF}}]$$

$$\frac{v : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash \mu v.s : \tau} \quad [\mathsf{rec}^{\mathrm{SF}}]$$

Fig. 17. **Typing rules for success and failure behaviour**

that no strategy is mis-identified as being infallible by the type system when it is not.

When we compare this approach to the abstract interpretation-based approach, then False should be compared with *Any* as opposed to *ExistsFailure*.

```
-- Type expressions
type Type = Bool -- Can we conclude that there is definitely no failure ?

-- Type inference
typeOf :: TP Type -> Maybe Type
typeOf Id              = Just True
typeOf Fail            = Just False
typeOf (Seq s s')      = liftM2 (&&) (typeOf s) (typeOf s')
typeOf (Choice s s')   = liftM2 (||) (typeOf s) (typeOf s')
typeOf (Var x)         = Just x
typeOf (Rec f)         = rec f True `mplus` rec f False
typeOf (All s)         = typeOf s
typeOf (One s)         = typeOf s >> Just False

-- Infer type of recursive closure by exhaustion
rec :: (Type -> TP Type) -> Type -> Maybe Type
rec f t = typeOf (f t) >>= \t' ->
          if t==t' then Just t else Nothing
```

Fig. 18. **Type inference for success and failure behaviour**

That is, True represents guarantee of success, while False represents lack of such a guarantee, as opposed to existence of a failure point. There is no counterpart for *ExistsFailure* in the type system. There is certainly no counterpart for *None* either, because this value is an artifact of fixed point iteration, which is not present in the type system.

With this comparison in mind, the deduction rules of Fig. 17 (say, the equations Fig. 18) of are very similar to the equations of Fig. 15. For instance, the rules for base cases id^{SF} and $fail^{\mathrm{SF}}$ state that the identity, *id*, is infallible, but that the primitive failure, *fail*, is not. For a sequence to be infallible, both components need to be infallible ($sequ.1^{\mathrm{SF}}$ to $sequ.3^{\mathrm{SF}}$), while if either component of a choice is infallible, the choice is too ($choice.1^{\mathrm{SF}}$, $choice.3^{\mathrm{SF}}$). A choice between two potentially fallible programs might well be infallible, but this analysis can only conclude that this is not guaranteed, and it is here that imprecision comes into the analysis. The type of an "all" traversal coincides with the type of argument strategy (all^{SF}). There is no guarantee of success for a "one" traversal (one^{SF}).

Finally, in dealing with the recursive case it is necessary to introduce a type context, Γ, containing typing assertions on variables. To conclude that a recursive definition $\mu v.s$ is infallible, it is sufficient to show that the body of the recursion, s, is infallible assuming that the recursive call, v, is too. Fig. 19

	s :: False	s :: True
full_bu s	False	True
full_td s	False	True
once_bu s	False	True
once_td s	False	True
stop_bu s	True	True
stop_td s	True	True
innermost s	True	True

Fig. 19. **Exercising success/failure types on common traversal schemes.**

presents the results of using the type system for some common traversals.

Again, the columns label the assumption for the success and failure behavior of the argument strategy s of the traversal scheme. (As an aside, operationally, we use the context parameter of the type system, or, in fact, the *Var* form of *TP* terms, to capture and propagate such assumptions; see the test cases that comes with source-code distribution.) When compared to Fig. 16, guarantee of success is inferred for several more cases. For instance, such a guarantee is inferred for the schemes of full top-down and bottom-up traversal, subject to the guarantee for the argument, whereas the abstract interpretation-based approach could not make an prediction for these cases. Also, the scheme for stop-top-down traversal is found to universally succeed, no matter what the argument strategy.

An improved type system As an aside, there is actually a trivial means to improve the usefulness of the type system. That is, we can easily exclude certain strategies which do not make sense. Specifically, we could remove the following rule:

$$\frac{\Gamma \vdash s_1 : \mathsf{True} \ \wedge \ \Gamma \vdash s_2 : \tau}{\Gamma \vdash s_1 \nleftarrow s_2 : \mathsf{True}} \qquad [\mathsf{choice.3}^{\mathrm{SF}}]$$

In this way, we classify a choice construct $s_1 \nleftarrow s_2$ with an infallible left operand as ill-typed: the point being that this is equivalent to s_1, with the s_2 being unreachable or ‘dead’ code.

Soundness of the type system We prove soundness of the type system in Fig. 17 relative to the established, natural semantics in Fig. 4–Fig. 5.

Theorem 1 *For all strategic programs s if $\vdash s : \mathsf{True}$ then for no term t $s \,@\, t \rightsquigarrow \uparrow$.*

Soundness proof We use a proof by contradiction. We suppose that there is some program s such that $\vdash s : \mathsf{True}$ and that there is an argument t so that $s \,@\, t \rightsquigarrow \uparrow$, and we choose s and t so that the depth of the derivation of $s \,@\, t \rightsquigarrow \uparrow$ is minimal; from this we derive a contradiction. We work by cases over s.

Identity If s is ϵ then there is no evaluation rule deriving $\epsilon \,@\, t \rightsquigarrow \uparrow$ for any t, contradicting the hypothesis.

Failure If s is δ then there is no typing rule deriving $\vdash \delta : \mathsf{True}$, contradicting the hypothesis.

Sequence If s is $s_1; s_2$ then the only way that $\vdash s_1; s_2 : \mathsf{True}$ can be derived is for the typing rule $\mathit{sequ}.1^{\mathrm{SF}}$ to be applied to derivations of $\vdash s_1 : \mathsf{True}$ and $\vdash s_2 : \mathsf{True}$.

Now, by hypothesis we also have a term t so that $s_1; s_2 \,@\, t \rightsquigarrow \uparrow$: examining the evaluation rules we see that this can only be deduced from $s_1 \,@\, t \rightsquigarrow \uparrow$ by rule $\mathit{seq}^{-}.1$ or from $s_2 \,@\, t \rightsquigarrow \uparrow$ by rule $\mathit{seq}^{-}.2$.

We choose s_i to be the divergent case: whichever we choose, the derivation of $s_i \,@\, t \rightsquigarrow \uparrow$ is shorter than $s \,@\, t \rightsquigarrow \uparrow$, a contradiction to the minimality of the derivation for s.

Choice If s is $s_1 \leftarrow\!\!\!+ s_2$ then there are two ways that $\vdash s_1 \leftarrow\!\!\!+ s_2 : \mathsf{True}$ can be derived: using $\mathit{choice}.3^{\mathrm{SF}}$ from a derivation of $\vdash s_1 : \mathsf{True}$ or using $\mathit{choice}.1^{\mathrm{SF}}$ from a derivation of $\vdash s_2 : \mathsf{True}$.

Now, by hypothesis we also have a term t so that $s_1 \leftarrow\!\!\!+ s_2 \,@\, t \rightsquigarrow \uparrow$: examining the evaluation rules we see that this can only be deduced from $s_1 \,@\, t \rightsquigarrow \uparrow$ and $s_2 \,@\, t \rightsquigarrow \uparrow$ by rule choice^{-}.

We choose s_i to be the case where $\vdash s_i : \mathsf{True}$. Whichever we choose, the derivation of $s_i \,@\, t \rightsquigarrow \uparrow$ is shorter than $s \,@\, t \rightsquigarrow \uparrow$, a contradiction to the minimality of the derivation for s.

All If s is $\square(s')$ then $\vdash \square(s') : \mathsf{True}$ is derived from $\vdash s' : \mathsf{True}$. From the negative rules for evaluation we conclude that t is of the form $c(t_1, \ldots, t_n)$ and for some i we have $s' \,@\, t_i \rightsquigarrow \uparrow$, and the derivation of this will be shorter than that of $s \,@\, t \rightsquigarrow \uparrow$, in contradiction to the hypothesis.

One If s is $\Diamond(s')$ then $\vdash \Diamond(s') : \mathsf{True}$ cannot be derived, directly contradicting the hypothesis.

Recursion Finally we look at the case that s is of the form $\mu v.s'$. We have a derivation (d_1, say) of $\vdash \mu v.s' : \mathsf{True}$, and this is constructed by applying rule $\mathit{rec}^{\mathrm{SF}}$ to a derivation d_2 of $v : \mathsf{True} \vdash s' : \mathsf{True}$.

We also have the argument t so that $\mu v.s' \,@\, t \rightsquigarrow \uparrow$. This in turn is derived from a derivation $s'[v \mapsto \mu v.s'] \,@\, t \rightsquigarrow \uparrow$, shorter than the former. So, $s'[v \mapsto$

	Strategy	Root sort	Reachable type-specific cases
1.	*Id*	*Company*	$\emptyset$
2.	*incSalary*	*Salary*	$\{$*incSalary*$\}$
3.	*try incSalary*	*Employee*	$\emptyset$
4.	*All (try incSalary)*	*Employee*	$\{$*incSalary*$\}$
5.	*All (try incSalary)*	*Department*	$\emptyset$
6.	*once_bu (try incSalary)*	*Department*	$\{$*incSalary*$\}$

Fig. 20. **Exercising the reachability analysis for companies.**

$\mu v.s']$ will be our counterexample to the minimality of $\mu v.s'$, so long as we can derive $\vdash s'[v \mapsto \mu v.s'] : \mathsf{True}$.

We construct a derivation of this from d_2, replacing each occurrence in d_2 of the variable rule applied to $v : \mathsf{True}$ by a copy of the derivation d_1, which establishes that the value substituted for v, $\mu v.s'$, has the type True, thus giving a derivation of $\vdash s'[v \mapsto \mu v.s'] : \mathsf{True}$, as required to prove the contradiction.

5.3 Reachability of type-specific cases

We seek an analysis that determines (conservatively) all type-specific cases that may be exercised during the evaluation of a given strategy when applied to a term of a given sort. As discussed in Section 3, the idea is that if it is statically evident that a type-specific component of a given traversal program will never be exercised, then we have strong evidence of a programming error.

Illustrative examples For instance, using again the introductory company example, we would like to obtain the kind of information in Fig. 20 by a reachability analysis. In this example we assume one type-specific case, *incSalary* that is used in some of the strategy expressions. The case increases salaries, and we assume that it is applicable to salary terms, only.

Let us motivate some of the expected results in detail. When applying the strategy *Id* (c.f., first line in the figure), which clearly does not involve any type-specific case, we obtain the empty set of reachable cases. When applying the type-specific *incSalary* to a salary term (second line), then the case is indeed applied; hence the result is $\{$*incSalary*$\}$. We cannot usefully apply *incSalary* to an employee (third line); hence, we obtain the empty set of reachable cases. We may apply though *All (try incSalary)* to an employee (fourth line) because a salary may be encountered on an immediate subterm posi-

```
-- Representation of signatures
type Sort       = String
type Constr     = String
type Symbol     = (Constr,[Sort],Sort)
data Signature = Signature { sorts    :: Set Sort
                           , symbols :: Set Symbol
                           }

-- Additional access functions
argSortsOfSort :: Signature -> Sort -> Set Sort
...
```

Fig. 21. **An abstract data type for signatures.**

tion of the employee. The last two lines in the table illustrate the reachability behaviour of a traversal scheme, in comparison to a one-layer traversal.

The abstract interpretation-based setup We assume many-sorted signatures for the terms to be traversed, as shown in Fig. 21, where we omit the straightforward definition of constructors and observers. In the abstract interpretation of the present section, we will only use basic accessors *sorts*, to retrieve all possible sorts of a signature, and *argSortsOfSort*, to retrieve all sorts of immediate subterms for all possible terms of a given sort.

For simplicity's sake, we formally represent type-specific cases simply by their name. Reachability analysis returns sets of such cases (say, names or strings). Hence we define:

```
type Case  = String
type Cases = Set Case
```

We assume that each type-specific case applies to a specific sort. We do not encode this type with the case itself but by attaching a property to the case. In fact, we use the following abstract domain both for the abstract interpretation and the property of type-specific cases:

```
type Abs = Map Sort Cases
```

Here, *Map* is a Haskell type for finite maps: sorts are associated with type-specific cases. We trivially associate such a map with each type-specific case, and the analysis associates such a map with a given strategy, as evident from the function type for the static program analysis:

```
analyse :: Signature -> TP Abs -> Abs
```

That is, for each *Sort* in the signature, the analysis returns a set of (named) type-specific *cases* which *may* be executed by the traversal. For instance:

```
> let incSalary = fromList [("Salary",Set.fromList ["incSalary"])]
> analysis companySignature (All (All (Var incSalary)))
[("Unit",fromList ["incSalary"]),("Manager",fromList ["incSalary"])]
```

The first input line shows the assembly of a type-specific case. The second input line starts a reachability analysis. The printed map for the result of the analysis states that *incSalary* can be reached from both *Unit* and *Manager*, but not from any data type. Indeed, salary components occur exactly two constructor levels below *Unit* and *Manager*.

The analysis is safe in that it is guaranteed to return all cases which are executed on some input; it is however an over-approximation, and so no guarantee is provided that all cases that are returned will be executed.

Definition of the analysis The analysis proceeds by induction over the structure of strategies, and is parametrized by the *Signature* over which the strategy is evaluated. The analysis crucially relies on the algebraic status of finite maps to define partial orders with general least upper bounds subject to the co-domain of the maps being a lattice itself. (We use the set of all subsets of type-specific cases as the co-domain.) The bottom value of this partial order is the map that maps all values of the domain to the bottom value of the co-domain. Here we note that such maps are an established tool in static program analysis. For instance, maps may be used in the analysis of imperative programs for *property states* as opposed to concrete states for program variables as in [51,52].

The central part of the analysis is the treatment of the one-layer traversals *All s* and *One s*. The reachable cases are determined separately for each possible sort, and these per-sort results are finally combined in a map. For each given sort *so*, the recursive call of the analysis, *analyse sig s*, is exercised for all possible sorts of immediate subterms of terms of sort *so*. Fixed point iteration will eventually reach all reachable sorts in this manner.

Discussion The analysis distinguishes neither *Seq* from *Choice* nor *All* from *One* in any manner. Also, the analysis assumes all constructors to be universally feasible, which is generally not the case due to type-specific cases (such as rewrite rules) and their recursive application. As a result, certain reachability-related programming errors will go unnoticed. Consider the following example:

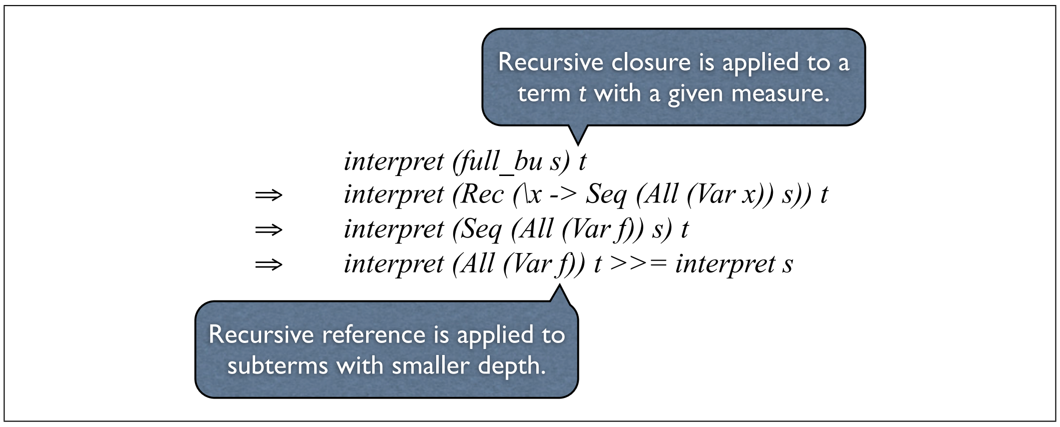
```
stop_td (Choice leanDepartment (Choice incSalary Id)) myCompany
```

```
analyse :: Signature -> TP Abs -> Abs
analyse sig = analyse'
 where
  analyse' :: TP Abs -> Abs
  analyse' Id              = bottom
  analyse' Fail            = bottom
  analyse' (Seq s s')      = analyse' s `lub` analyse' s'
  analyse' (Choice s s')   = analyse' s `lub` analyse' s'
  analyse' (Var x)         = x
  analyse' (Rec f)         = fixEq (analyse' . f)
  analyse' (All s)         = transform sig $ analyse' s
  analyse' (One s)         = transform sig $ analyse' s

transform :: Signature -> Abs -> Abs
transform sig abs
 = Map.fromList
 $ map perSort
 $ Set.toList
 $ sorts sig
 where
  perSort :: Sort -> (Sort, Cases)
  perSort so = (so,cases)
   where
    cases = lubs
          $ map perArgSort
          $ Set.toList argSorts
     where
      argSorts = argSortsOfSort sig so
      perArgSort = flip Map.lookup abs
```

Fig. 22. **Abstract interpretation for analysing the reachability of type-specific cases relative to a given signature.**

For the sake of a concrete intuition, we assume that *leanDepartment* will pension off all eligible employees, if any. Here we assume that *leanDepartment* applies to terms of sort *Department* for which it also succeeds. As a result of *leanDepartment*'s success at the department level of company terms, the stop-top-down traversal will never actually hit employees or salaries.

This illustration clearly suggests that a success/failure analysis should be incorporated into the reachability analysis for precision's sake. To this end, it would be beneficial to know whether a type-specific case universally succeeds for all terms of the sort in question. Further, the analysis would treat sequence differently from choice, and "all" traversal differently from "one" traversal. We omit this elaboration here.


Recursive closure is applied to a term *t* with a given measure.
Recursive reference is applied to subterms with smaller depth.

```
      interpret (full_bu s) t
⇒     interpret (Rec (\x -> Seq (All (Var x)) s)) t
⇒     interpret (Seq (All (Var f)) s) t
⇒     interpret (All (Var f)) t >>= interpret s
```

Fig. 23. **Illustration of termination checking.**

5.4 Termination analysis

We seek an analysis that determines, conservatively, whether a given recursive strategy is guaranteed to converge. For instance, the intended analysis should infer the following convergence properties. A full bottom-up traversal converges regardless of the argument strategy, as long as the argument strategy itself converges universally. A full top-down traversal converges as long as the argument strategy converges universally and does not increase some suitable measure such as the depth of the term.

Based on experiences with modelling terminating strategies in Isabelle/HOL [30], our analysis for termination checking essentially leverages an induction principle; see Fig. 23 for an illustration. That is, the analysis is basically meant to verify that a measure of the term (such as the depth of the term), as seen by the recursive closure as a whole, is *decreased* throughout the body of the recursive closure until the recursive reference is invoked. The figure shows a few steps of interpretation when applying a full bottom-up traversal to a term. The recursive reference is eventually applied to immediate subterms of the original term. Hence, the mere depth measure of terms is sufficient here for the induction principle.

We will first develop a basic termination analysis that leverages the depth measure in an essential manner. However, many traversals in strategic programming cannot be proven to converge just on the grounds of depth. For instance, macro expansion based on a full top-down traversal, or normalization based on an innermost traversal actually increase depth (potentially). Accordingly, we will generalize the analysis to leverage compound measures with components other than depth.

A note on related work Termination analysis is an established technique in the programming languages and rewriting communities. We mention some recent work in the adjacency of (functional) traversal strategies. [22,59] address termination analysis for rewriting with strategies (but without covering programmable traversal strategies). [58,21] address termination analysis for higher-order functional programs; it may be possible to extent these systems with awareness for one-layer traversal and generic functions for traversal. [2] addresses termination analysis for generic functional programs of the kind of Generic Haskell [24]; the approach is based on type-based termination and exploits the fact that generic functions are defined by induction on types, which is not directly the case for traversal strategies, though. Our termination analysis is arguably naive in that focuses on recursion pattern of traversals. A practical system for a full-fledged strategic programming language would definitely need to include some established termination approach.

5.4.1 Measure relations on terms and strategies

The key concept of the termination analysis is a certain style of manipulating measures symbolically. We will explain this concept here for the depth measure for ease of understanding, but the style generalizes easily for other measures.

Based on the intuition of Fig. 23 the analysis must track the depth measure of terms throughout the symbolic execution of a strategy so that the depth can be tested at the point of recursive reference. That is, *the depth must be shown to be smaller at the point of recursive reference than the point of entering the recursive closure.* In this manner, we establish that the depth measure is smaller for each recursive unfolding, which implies convergence.

Obviously, the analysis cannot use actual depth, but it needs to use an abstract domain. We use a finite domain *Rel* whose values describe the relation between the depths of two terms: i) the term in a given position of the body of a recursive closure; ii) the term at the entrance of the recursive closure. The idea is that the relation is initialized to "=" (in fact, "≤"—because we do not have a representation of "=") as we enter the recursive closure, and it is accordingly updated as we symbolically interpret the body of the closure. Ultimately, we are only interested in the relation—as it holds for the recursive reference of the body. These are the values of *Rel*; see Fig. 24 for a full specification:

```
-- Relation on measures
data Rel = Leq | Less | Any
```

One can think of the values as follows. The value *Leq* models that the depth of the term was not increased along the execution of the body of the recursive closure so far. The value *Less* models that the depth of the term was strictly decreased instead. This is the relation that must hold at the point of the recursive reference. The value *Any* models that we do not know for certain

```
-- Relation on measures
data Rel = Leq | Less | Any

-- Partial order and LUB for Rel
instance POrd Rel
 where
  _    <= Any = True
  Less <= Leq = True
  r    <= r'  = r == r'

instance Lub Rel
 where
  lub Less Leq  = Leq
  lub Leq  Less = Leq
  lub r    r'   = if r == r' then r else Any

-- Rel arithmetic
plus :: Rel -> Rel -> Rel
plus Less Less = Less
plus Less Leq  = Less
plus Leq  Less = Less
plus Leq  Leq  = Leq
plus _    _    = Any

decrease :: Rel -> Rel
decrease Leq  = Less
decrease Less = Less
decrease Any  = Any

increase :: Rel -> Rel
increase Less = Leq
increase Leq  = Any
increase Any  = Any
```

Fig. 24. **Relation on measures such as depth of terms.**

whether the depth was preserved, increased, or decreased.

So far we have emphasized a depth relations for *terms*. However, the analysis also relies on depth relations for *strategies*. That is, we can use the same domain *Rel* to describe *the effect of a strategy on the depth*. In this case, one can think of the values now as follows. The value *Leq* models that the strategy does not increase the depth of the term. The value *Less* models that the strategy strictly decreases the depth of term. The value *Any* models that we do not know for certain whether the strategy preserves, increases, or decreases depth.

For clarity, we use these type synonyms:

```
type TRel = Rel -- Term property
type SRel = Rel -- Strategy property
```

We set up the type of the analysis as follows:

```
type Abs = TRel -> Maybe TRel
analyse :: TP SRel -> Abs
```

In fact, we would like to use variables of the *TP* type again to capture effects for *arguments* of traversal schemes. Hence, we should distinguish recursive references from other references; we use an extra Boolean to this end; *True* encodes recursive references.

```
analyse :: TP (SRel,Bool) -> Abs
```

At the top level, we begin analysing a strategy expression (presumably a recursive closure) by assuming *TRel* value of *Leq*. Also, we effectively provide type inference in that we compute an *SRel* value for the given strategy expression. Thus:

```
typeOf :: TP (SRel,Bool) -> Maybe SRel
typeOf s = analyse s Leq
```

5.4.2 Termination analysis with the depth measure

The analysis is defined in Fig. 25. The analysis can be viewed as an algorithmically applicable type system which tracks effects on measures as types. Just in the same way as the standard semantics threads a term through evaluation, this analysis threads a term property of type *TRel* through symbolic evaluation. The case for *Id* preserves the property (because *Id* preserves the depth, in fact, the term). The case for *Fail* preserves the delta, too, in a vacuous sense. The case for *Seq* sequentially compose the effects for the operand strategies. The case for *Choice* takes the least upper bound of the effects for the operand strategies. For instance, if one operand possibly increases depth, then the composed strategy is stipulated to potentially increase depth.

The interesting cases are those for variables (including the case of recursive references), recursive closures and one-layer traversal. The case for *Var* essentially applies the strategy property for the variable (i.e., an *SRel* value) to the current term property. This is a sort of addition. In addition, we check whether we are facing a recursive reference (i.e., *b == True*) because in this case we must insist on the current *TRel* value to be *Less*.

```
analyse :: TP (SRel,Bool) -> Abs
analyse Id = Just
analyse Fail = Just
analyse (Seq s s') = maybe Nothing (analyse s') . analyse s

analyse (Choice s s')
 = \r ->
     case (analyse s r, analyse s' r) of
       (Just r1, Just r2) -> Just (lub r1 r2)
       _                  -> Nothing

analyse (Var (r,b))
 = \r' ->
     if not b || r' < Leq
       then Just (plus r' r)
       else Nothing

analyse (Rec f)
 = \r -> maybe Nothing (Just . plus r) (typeOfClosure r)
 where
  typeOfClosure r = if null attempts
                      then Nothing
                      else Just (head attempts)
   where
    attempts = catMaybes (map wtClosure' [Less,Leq,Any])
    wtClosure r = maybe False (<=r) (analyse (f (r,True)) Leq)
    wtClosure' r = if wtClosure r then Just r else Nothing

analyse (All s) = transform (analyse s)
analyse (One s) = transform (analyse s)

transform :: Abs -> Abs
transform f r = maybe Nothing (Just . increase) (f (decrease r))
```

Fig. 25. **A static analysis for termination checking relative to the depth measure for terms.**

The case for *Rec* essentially resets the *TRel* value to *Leq*; c.f., *analyse ... Leq*, and attempts the analysis of the body for all possible assumptions about the effect of the recursive references; c.f., *[Less,Leq,Any]*. If the analysis returns with any computed effect, then this result is required to be less or equal to the one assumed for the recursive reference; c.f., "<=". The ordering on the attempts implies that the least constraining type is inferred (i.e., the largest value of *SRel*).

	s :: Any	*s :: Leq*	*s :: Less*
full_bu s	*Just Any*	*Just Leq*	*Just Less*
full_td s	*Nothing*	*Just Leq*	*Just Leq*
stop_td s	*Just Any*	*Just Leq*	*Just Leq*
once_bu s	*Just Any*	*Just Leq*	*Just Leq*
repeat s	*Nothing*	*Nothing*	*Just Leq*
innermost s	*Nothing*	*Nothing*	*Nothing*

Fig. 26. **Exercising the termination analysis on common traversal schemes.**

Finally, the cases for *All* and *One* are handled identically as follows. The term property is temporarily decreased as the argument strategy is symbolically evaluated, and the resulting term property is again increased on the way out. This models the fact that argument strategies of "all" and "one" are only applied to subterms. It is important to understand that the symbolic range for increase and decrease are limited. That is, once the *TRel* value has reached *Any*, there is no way to get back onto a termination-proven path.

Illustrative examples Despite the limitations of the approach so far, especially its restriction to term depth, the analysis is already able to infer termination types for a range of interesting traversal scenarios; c.f., Fig. 26.

The table clarifies that a full bottom-up traversal makes no assumption about the measure effect of the argument strategy, while a full top-down traversal does not get assigned a termination type for an unconstrained argument; c.f., the occurrence of *Nothing*. The depth measure is practically useless for typical applications of *repeat*, but we can observe nevertheless that an application of *repeat* will only terminate, if its argument is "strictly decreasing". This property will also be inherited by other measures. Finally, our present analysis is unable to find any terminating use case for *innermost*. This is not surprising because *innermost* composes *repeat* and *once_bu*, where the former requires a strictly decreasing strategy (i.e., *Less*), and the latter can only be type-checked with *Leq* as the termination type.

5.4.3 Termination analysis with compound measures

There is a non-trivial elaboration of the static analysis that can deal with measures other than depth. We have explored the measure of constructor counts (i.e., the number of occurrences of a specific constructor in a given term) in detail; c.f., the online code distribution for the paper. The approach

	s ::[Less, Any]
full_td s	*Just [Less, Any]*
once_bu s	*Just [Less, Any]*
repeat s	*Just [Leq, Any]*
innermost s	*Just [Leq, Any]*

Fig. 27. **Exercising compound measures.**

may also be generalized to deal with number of matches for a specific pattern, such as the LHS of a rewrite rule.

Let us sketch the generalization. We need a new type, *Measure*, to represent the measure. Here, we assume that the depth measure always appears in the last (least significant) position.

```
data Measure = Depth | Count Constr Measure
type Constr = String
```

The idea is to use a single, composed measure throughout the traversal program. Type-specific ingredients of a traversal program, i.e., rewrite rules, in particular, would need to be type-checked against a user-defined measure, or such a measure would need to be inferred.

The existing type of measure transformation and the signature of the program analysis changes as follows: we basically account for non-empty lists of values of type *Rel* as opposed to singletons before:

```
type Abs = [TRel] -> Maybe [TRel]
analysis :: TP ([SRel],Bool) -> Abs
```

The power of such termination types is illustrated in Fig. 27. The new type for *full_td* shows that we do not rely on the argument strategy to be non-increasing on the depth; we may as well use a depth-increasing argument, as long as it is non-increasing on some constructor count. The new type for *once_bu* is strictly decreasing, and hence, its iteration with *repeat* results in a termination type for *innermost*. We refer again to the online code distribution for this paper for details.

6 Related work

We will focus here on related work that deals more or less directly with programming errors in traversal programming with traversal strategies or otherwise. We do not collect any groups of related work on the more diverse subjects of strategic programming models, application of strategic programming, or general programming-language concepts that are potentially useful in revising strategic programming. Those references were given in more, appropriate contexts already.

Simplified traversal programming In an effort to manage the relative complexity of traversal strategies (or the general generic functions of "Scrap Your Boilerplate"), simplified forms of traversal programming have been proposed. The functional programming-based work of [49] describes less generic traversal schemes (akin to those of § 4.1), and thereby suffices with simpler types, and may achieve efficiency of traversal implementation more easily. The rewriting-based work of [60] follows a different route; it limits programmability of traversal by providing only a few schemes and few parameters. Again, such a system may be easier to grasp for the programmer, and efficiency of traversal implementation may be achievable more easily. Our work is best understood as an attempt to uphold the generality or flexibility and relative simplicity (in terms of language constructs involved) of traversal strategies while using orthogonal means such as advanced typing or static analysis for helping with program comprehension.

Adaptive programming While the aforementioned related work is (transitively) inspired by traversal strategies à la Stratego, there is the independent traversal programming approach of adaptive programming [53,46,42]. Traversal specifications are more disciplined in this paradigm. Generally, the focus is more on problem-specific traversal specifications as opposed to generic traversal schemes. Also, there is a separation between traversal specifications and computations or actions, thereby simplifying some analyses, e.g., a termination analysis. The technique of §4.4 to statically check for reachable types is inspired by adaptive programming. Certain categories of programming errors are less of an issue, if any, in adaptive programming. Interestingly, there are recent efforts to evolve adaptive programming, within the bounds of functional object-oriented programming, to a programming paradigm that is appears to be more similar to strategic programming [1].

The XML connection Arguably, the most widely used kinds of traversal programs in practice are XML queries and transformations such as those based on XPath [66], XSLT [68], and XQuery [67]. Some limited cross-paradigmatic comparison of traversal strategies and XML programming has been presented in [36,14]. Most notably, there are related problems in XML programming. For instance, one would like to know that an XPath query does not necessarily return the empty node set. The XQuery specification [67] even addresses this issue, to some extent, in the XQuery type system. While XSLT also inherits all XPath-related issues (just as much as XQuery), there is the additional challenge due to its template mechanism. Default templates, in particular, provide a kind of traversal capability. Let us mention related work on analysing XML programs. [16] describes program analysis for XSLT based on an analysis of the call graph of the templates and the underlying DTD for the data. In common with our work is a conservative estimation of sufficient conditions for program termination as well as some form "dead code" analysis. There is recent work on logics for XML [18] to perform static analysis of XML paths and types [20], and a "dead code" elimination for XQuery programs [19].

Properties of traversal programs Mentions of algebraic laws and other properties of strategic primitives and some traversal schemes appear in the literature on Stratego-like strategies [63,41,65,60,29,38,35,54,14,30]. In [14], laws feed into automated program calculation for the benefit of optimization ("by specialization") and reverse engineering (so that generic programs are obtained from boilerplate code). In [29], specialized laws of applications of traversal schemes are leveraged to enable fusion-like techniques for optimizing strategies. None of these previous efforts have linked properties to programming errors. Also, there is no previous effort on performing static analysis for the derivation of general program properties about termination, metadata-based reachability, or success and failure behavior. We hypothesize that a better understanding of properties of traversal strategies, just by itself, would reduce the likelihood of programming errors. Obviously, we do not claim that the present paper has reached a limit in this respect, as we will clarify below.

7 Concluding remarks

The ultimate motivation for the presented work is to provide input for the next generation of strategic programming, in particular, and for future language designs and implementations with traversal capabilities, in general. In fact, we hope for future programming languages to be expressive and extensible enough to provide a safe and easy to use form of traversal strategies.

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 [17,25]. 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 [29], and calculational techniques for the transformation of traversal strategies [14].

Acknowledgments. 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 and the discussion at the workshop. Part of the material has also been used in Ralf Lämmel's invited talk at LOPSTR/PPDP 2009, and all feedback is gratefully acknowledged. Much of the presented insights also draw from collaboration and discussions with Simon Peyton Jones, Karl Lieberherr, Claus Reinke, Eelco Visser, Joost Visser, and Victor Winter.

References

[1] A. Abdelmeged and K. J. Lieberherr. Recursive adaptive computations using perobject visitors. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, pages 825–826. ACM, 2007.

[2] A. Abel. Type-based termination of generic programs. *Science of Computer Programming*, 74(8):550–567, 2009.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In *Term Rewriting and Applications, 18th International Conference, RTA 2007, Proceedings*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.

[4] E. Balland, P.-E. Moreau, and A. Reilles. Rewriting strategies in java. *ENTCS*, 219:97–111, 2008.

[5] G. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Cω. In *ECOOP'05, Object-Oriented Programming, 19th European Conference, Proceedings*, volume 3586 of *LNCS*, pages 287–311. Springer, 2005.

[6] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*, Pont-à-Mousson, France, Sept. 1998. Elsevier Science.

[7] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[8] M. Brand, M. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.

[9] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: components for transformation systems. In *PEPM'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 95–99. ACM, 2006.

[10] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[11] P. Cousot. Abstract Interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[12] P. Cousot and R. Cousot. Basic concepts of abstract interpretation. In *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, pages 359–366. Kluwer, 2004.

[13] K. Crary and S. Weirich. Flexible type analysis. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 233–248. ACM, 1999.

[14] A. Cunha and J. 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.

[15] E. Dolstra and E. Visser. First-class Rules and Generic Traversal. Technical Report UU-CS-2001-38, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2001.

[16] C. Dong and J. Bailey. Static Analysis of XSLT Programs. In K.-D. Schewe and H. Williams, editors, *Fifteenth Australasian Database Conference (ADC2004)*, Conferences in Research and Practice in Information Technology. Australian Computer Society, Inc., 2004.

[17] R. Findler and M. 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.

[18] P. Genevès. *Logics for XML.* PhD thesis, Institut National Polytechnique de Grenoble, December 2006.

[19] P. Genevès and N. Layaïda. Eliminating Dead-Code from XQuery Programs. In *ICSE'10, Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering.* ACM, 2010.

[20] P. Genevès, N. Layaïda, and A. Schmitt. Efficient Static Analysis of XML Paths and Types. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 342–351. ACM Press, 2007.

[21] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Term Rewriting and Applications, 17th International Conference, RTA 2006, Proceedings*, volume 4098 of *LNCS*, pages 297–312. Springer, 2006.

[22] I. Gnaedig and H. Kirchner. Termination of rewriting under strategies. *ACM Transactions on Computational Logic*, 10(2), 2009.

[23] R. Hinze. A New Approach to Generic Functional Programming. In T. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21*, pages 119–132, Jan. 2000.

[24] R. Hinze. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132. ACM, 2000.

[25] R. Hinze, J. Jeuring, and A. 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.

[26] R. Hinze and A. Löh. "Scrap Your Boilerplate" Revolutions. In *Proceedings, Mathematics of Program Construction, 8th International Conference, MPC 2006*, volume 4014 of *LNCS*, pages 180–208. Springer, 2006.

[27] R. Hinze, A. Löh, and B. C. D. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In *FLOPS'06: Proceedings of Functional and Logic Programming, 8th International Symposium*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

[28] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[29] P. Johann and E. 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.

[30] M. Kaiser and R. Lämmel. An Isabelle/HOL-based model of stratego-like traversal strategies. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 93–104. ACM, 2009.

[31] L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In *Compiler Construction,*

18th International Conference, CC 2009, Proceedings, volume 5501 of *LNCS*, pages 142–157. Springer, 2009.

[32] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell'04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[33] R. 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.

[34] R. Lämmel. Towards generic refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming*, pages 15–28. ACM, 2002.

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

[36] R. 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, 2007.

[37] R. 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.

[38] R. Lämmel and S. 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.

[39] R. Lämmel and S. L. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP'04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255. ACM Press, 2004.

[40] R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP'05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 204–215, New York, NY, USA, 2005. ACM Press.

[41] R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming – an inquiry into trans-paradigmatic genericity. Draft, 2002–2003.

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

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

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

[45] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. *ENTCS*, 141(4):29–34, 2005.

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

[47] B. Luttik and E. Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer.

[48] S. Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 96–106. ACM, 2006.

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

[50] G. Munkby, A. P. Priesnitz, S. Schupp, and M. Zalewski. Scrap++: scrap your boilerplate in C++. In *Proceedings of the ACM SIGPLAN Workshop on Genetic Programming, WGP 2006*, pages 66–75. ACM, 2006.

[51] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[52] H. R. Nielson and F. Nielson. *Semantics with Applications (An Appetizer)*. Springer, 2007.

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

[54] F. Reig. Generic proofs for combinator-based generic programs. In *Trends in Functional Programming*, pages 17–32, 2004.

[55] D. Ren and M. 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.

[56] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008*, pages 111–122. ACM, 2008.

[57] B. G. Ryder and M. L. 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.

[58] D. Sereni and N. D. Jones. Termination Analysis of Higher-Order Functional Programs. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Proceedings*, volume 3780 of *LNCS*, pages 281–297. Springer, 2005.

[59] R. Thiemann and C. Sternagel. Loops under Strategies. In *Rewriting Techniques and Applications, 20th International Conference, Proceedings*, volume 5595 of *LNCS*, pages 17–31. Springer, 2009.

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

[61] E. Visser. Language Independent Traversals for Program Transformation. In *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 86–104, 2000.

[62] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In *Domain-Specific Program Generation, Dagstuhl Seminar, 2003, Revised Papers*, volume 3016 of *LNCS*, pages 216–238. Springer, 2004.

[63] E. Visser, Z. Benaissa, and A. 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.

[64] E. Visser and Z.-e.-A. Benaissa. A Core Language for Rewriting. In *Second International Workshop on Rewriting Logic and its Applications (WRLA 1998)*, volume 15 of *ENTCS*. Elsevier Science Publishers, 1998.

[65] J. Visser. *Generic Traversal over Typed Source Code Representations.* PhD thesis, University of Amsterdam, Feb. 2003.

[66] W3C. XML Path Language (XPath) Version 1.0. Available online at `http://www.w3.org/TR/xpath/`, W3C Recommendation 16 November 1999.

[67] W3C. XQuery 1.0: An XML Query Language. Available online at `http://www.w3.org/TR/xquery/`, W3C Recommendation 23 January 2007.

[68] W3C. XSL Transformations (XSLT) Version 2.0. Available online at `http://www.w3.org/TR/xslt20/`, W3C Recommendation 23 January 2007.

[69] V. L. Winter. Strategy Construction in the Higher-Order Framework of TL. *ENTCS*, 124(1):149–170, 2005.

[70] V. L. Winter and J. Beranek. Program Transformation Using HATS 1.84. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Revised Papers*, volume 4143 of *LNCS*, pages 378–396. Springer, 2006.

[71] V. L. Winter, J. Beranek, F. Fraij, S. Roach, and G. L. Wickstrom. A transformational perspective into the core of an abstract class loader for the SSP. *ACM Trans. Embedded Comput. Syst.*, 5(4):773–818, 2006.

[72] V. L. Winter and M. Subramaniam. The transient combinator, higher-order strategies, and the distributed data problem. *Science of Computer Programming*, 52:165–212, 2004.