Kent Academic Repository

Thompson, Simon, Laemmel, Ralf and Kaiser, Markus (2013) ***Programming errors in traversal programs over structured data.*** **Science of Computer Programming . ISSN 0167-6423.**

Downloaded from
https://kar.kent.ac.uk/31505/ The University of Kent's Academic Repository KAR

The version of record is available from
https://doi.org/10.1016/j.scico.2011.11.006

This document version
Author's Accepted Manuscript

DOI for this version

Licence for this version
UNSPECIFIED

Additional information

Versions of research works

Versions of Record
If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts
If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal* , Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries
If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our Take Down policy (available from https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies).

ARTICLE IN PRESS

Science of Computer Programming ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Programming errors in traversal programs over structured data☆

Ralf Lämmel [a,*], Simon Thompson [b], Markus Kaiser [a]

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

[b] *University of Kent, UK*

ARTICLE INFO

Article history:
Received 22 March 2010
Received in revised form 14 September 2011
Accepted 23 November 2011
Available online xxxx

Keywords:
Traversal strategies
Traversal programming
Term rewriting
Stratego
Strafunski
Generic programming
Scrap your boilerplate
Type systems
Static program analysis
Functional programming
XSLT
Haskell

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 traversals, prematurely terminated traversals, and traversals with dead code.

Based on an inventory of possible programming errors we explore options of static typing and static analysis so that some categories of errors can be avoided. This exploration generates suggestions for improvements to strategy libraries as well as their underlying programming languages. Haskell is used for illustrations and specifications with sufficient explanations to make the presentation comprehensible to the non-specialist. The overall ideas are language-agnostic and they are summarized accordingly.

© 2011 Elsevier B.V. All rights reserved.

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* [70,41,3]: we are specifically concerned with *programming errors* in this context.

Let us briefly indicate some application domains for such traversal programming. To this end, consider the illustrative scenarios for querying and transforming data as shown in Fig. 1. The listed queries and transformations benefit from designated support for traversal programming. The scenarios assume data models for *(a)* the abstract syntax of a

☆ The paper and accompanying source code are available online:
• http://userpages.uni-koblenz.de/~laemmel/syb42.
• http://code.google.com/p/strafunski/.

* Corresponding author. Tel.: +49 31 20 4447824; fax: +49 31 20 4447653.
E-mail address: rlaemmel@gmail.com (R. Lämmel).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.11.006

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS

Illustrative queries

The abstract syntax of a programming language as a data model

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

The organizational structure of a company as a data model

(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 abstract syntax of a programming language as a data model

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

The organizational structure of a company as a data model

(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. 1. Illustrative scenarios for traversal programming.

programming language and *(b)* the organizational structure of a company within an information system. Suitable data models are sketched in Fig. 2 in Haskell syntax.[1]

A querying traversal essentially visits a compound term, potentially layer by layer, to extract data from subterms of interest. For instance, salary terms are extracted from a company term when dealing with the scenarios of totaling salaries of all or some employees in Fig. 1.

A transforming traversal essentially copies a compound term, potentially layer by layer, except that subterms of interest may be replaced. For instance, function applications are enriched by logging code when dealing with the corresponding program transformation scenario of Fig. 1.

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]. For example, the first scenario for the company example requires a traversal strategy as follows:

- A basic (non-traversing) *rule* is needed to extract salary from an employee.
- The rule is to be iterated over a term by a *traversal scheme*.
- The traversal scheme itself is defined in terms of *basic combinators*.

The notion of traversal strategies was pioneered by Eelco Visser and collaborators [48,71,70] in the broader context of term rewriting. This seminal work also led to Visser et al.'s Stratego/XT [69,10]—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., [9,7,67,79].

Historically, strategic programming has been researched considerably in the context of software transformation. We refer the reader to [70,68,34,44,46,78] for some published accounts of actual applications of traversal programming with traversal strategies.

The Stratego approach inspired strategic programming approaches for other programming paradigms [43,41,72,3,4]. An early functional approach, known by the name 'Strafunski', inspired the 'Scrap your boilerplate' (SYB) form of generic

[1] *Note on Haskell*: ***type*** definitions are type synonyms. For instance, *Name* is defined to be a synonym for Haskell's *String* data type. In contrast, ***data*** types define new algebraic types with one or more constructors (say, cases). Consider the data type *Expr*; it declares a number of constructors for different expression forms. For instance, the constructor *Var* models variables; there is a constructor component of type *Name* (in fact, *String*) and hence the constructor function's type is *Name —> Expr*.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS

***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. 2. Illustrative data models (rendered in Haskell). Queries and transformation on such data may require traversal programming possibly prone to programming errors. The data models involve multiple types; recursion is exercised; there is also a type with multiple alternatives. As a result, traversal programmers need to carefully control queries and transformations—leaving room for programming errors. *Note on language agnosticism*: Haskell's algebraic data types are chosen here without loss of generality. Other data-modeling notations such as XML schemas or class diagrams are equally applicable.

functional programming [38–40,59,26,27], which is relatively established today in the Haskell community. SYB, in turn, inspired cross-paradigm variations, e.g., 'Scrap your boilerplate in C++' [53]. Traversal strategies à la Stratego were also amalgamated with attribute grammars [31]. Another similar form of traversal strategies are those of the rewriting-based approach of HATS [79,76,77].

There are also several, independently developed forms of traversal programming: TXL's strategies [11], adaptive traversal specifications [47,42,1], Cω's data access [5], XPath-like queries and XSLT transformations [36,15] as well as diverse (non-SYB-like) forms of generic functional programming [28,24,60].

Research topic: programming errors *in traversal programming*

Despite the advances in 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 in rewrite rules, but other errors are of a novel nature. Their appearance can be off-putting to the newcomer to the field, and indeed limit the productivity of more experienced strategists.

Regardless of the specifics of an approach for traversal programming—the potential for programming errors is relatively obvious. Here are two possible errors that may arise for the traversal scenarios of Fig. 1:

A programming error implying an incorrect result. In one of the scenarios for the company example, a query is assumed to total the salaries of all employees who are *not* managers. One approach is to use two type-specific cases in the traversal such that managers contribute a subtotal of zero whereas regular employees contribute with their actual salary. The query should combine the type-specific cases in a traversal that ceases when either of the two cases is applicable. A traversal could be incorrect in that it continues even upon successful application of either case. As a result, managers are

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

not effectively excluded from the total because the traversal would eventually encounter the employee term inside each manager term.

A programming error implying divergence. In one of the scenarios for the AST example, a transformation is assumed to *unfold or inline a specific function*. This problem involves two traversals: one to find the definition of the function, another one to actually affect references to the function. Various programming errors are conceivable here. For instance, the latter traversal may end up unfolding recursive function definitions indefinitely. Instead, the transformation should not attempt unfolding for any subterm that was created by unfolding itself. This can be achieved by bottom-up traversal order.

Research objective: advice on library and language improvements

We envisage that the traversal programming of the future will be easier and safer because it is better understood and better supported. Strategy libraries and the underlying programming languages need to be improved in ways that programming errors in the sense of incorrect strategy application and composition are less likely. Our research objective is to provide advice on such improvements.

We contend that *static typing* and *static program analysis* are key tools in avoiding programming errors by detecting favorable or unfavorable behaviors of strategies statically, that is, without running the program. Advanced types are helpful in improving libraries so that their abstractions describe more accurately the contracts for usage. Static program analysis is helpful in verifying correctness of strategy composition.

Accordingly, in this paper, we identify pitfalls of strategic programming and discover related properties of basic strategy combinators and common library combinators. Further, we research the potential of static typing and static program analysis with regard to our research objective of providing advice on improved libraries and languages for strategic programming.

Summary of the paper's contributions[2]

(1) We provide a fine-grained *inventory of programming errors* in traversal programming with traversal strategies; see Section 3. To this end, we reflect on the use of basic strategy combinators and library abstractions in the solution of traversal problems.
(2) We explore the *utility of static typing* as means of taming traversal strategies; see Section 4. This exploration clarifies what categories of programming errors can be addressed, to some extent, by available advanced static type systems. We use examples written in Haskell 2010, with extensions, for illustrations.
(3) We explore the *utility of static program analysis* as means of taming traversal strategies; see Section 5. Static analysis is an established technique for dealing with correctness and performance properties of software. We study success and failure behavior, dead code, and termination.

Haskell en route

In this paper, we use Haskell for two major purposes: *(a)* illustrations of static typing techniques for taming strategies; *(b)* specifications of algorithms for static program analysis. All ideas are also explained informally and specific Haskell idioms are explained—as they are encountered—so that the presentation should also be comprehensible to the non-specialist. The overall ideas of taming strategies by static typing and static program analysis are not tied to Haskell; they are largely agnostic to the language—as long as it supports strategic programming style. Accordingly, all subsections of Section 4 and Section 5 begin with a language-agnostic, informal advice for improving strategy libraries and the underlying languages.

Scope of this work

Our work ties into traversal strategies à la Stratego (also à la Strafunski and 'Scrap Your Boilerplate') in that it intimately connects to idiomatic and conceptual details of this class of approaches: 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 some of the results may be applicable to other forms of traversal programming, e.g., adaptive programming [56,47,42], (classic) generic functional programming [28,24,60], XML queries or transformations (based on mainstream languages). However, we do not explore such potential. As a brief indication of potential relevance,

[2] *Note on the relationship to previous work by the authors*: 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 present paper is generally more detailed and updated, but its distinctive contribution is research on static program analysis for traversal strategies in Section 5; only one of the analyses was sketched briefly in the short version. The present paper also takes advantage of other previously published work by two of the present authors in so far that some of the strategy properties discovered by [30] help steering research on programming errors in the present paper. Programming errors are not systematically discussed in [30]. Also, type systems and static program analysis played no substantial role in said work. Instead, the focus was on demonstrating the potential of theorem proving in the context of traversal strategies.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

s	::=	$t \rightarrow t$	(Rewrite rules as basic building blocks.)
	|	id	(Identity transformation; succeeds and returns input.)
	|	$fail$	(Failure transformation; fails and returns failure "↑".)
	|	$s; s$	(Left-to-right sequential composition.)
	|	$s \nleftarrow s$	(Left-biased choice; try left argument first.)
	|	$\square(s)$	(Transform all immediate subterms by s; maintain constructor.)
	|	$\diamond(s)$	(Transform one immediate subterm by s; maintain constructor.)
	|	v	(A variable strategy subject to binding or substitution.)
	|	$\mu v.s$	(Recursive closure of s referring to itself as v.)

Fig. 3. Syntax of strategy primitives for transformations.

let us consider XSLT [75] with its 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 which XML (sub-)trees to match and how to process them. Processing begins at the root node and proceeds by applying the best fitting template pattern, and recursively processing according to the template. Hence, traversal is implicit in the XSLT processing model, but the templates control the traversal. Several problems that we study in the present paper—such as dead code or divergence—can occur in XSLT and require similar analyses.

Road map of the paper

- Section 2 provides background on strategic programming.
- Section 3 makes an inventory of programming errors in strategic programming.
- Section 4 researches the potential of static typing for taming traversal strategies.
- Section 5 researches the potential of static analysis for taming traversal strategies.
- Section 6 discusses related work.
- Section 7 concludes the paper.

2. Background on strategic programming

This section introduces the notion of traversal strategy and the style of strategic programming. We integrate basic material that is otherwise scattered over several publications. The collection is also valuable in so far that we make an effort to point out properties of strategies as they will be useful eventually in the discussion of programming errors.

We describe traversal strategies in three different styles. First, we use a formal style based on a grammar and a natural semantics. Second, we use an interpreter style such that the semantics is essentially implemented in Haskell. Third, we embed strategies into Haskell—this is the style that is useful for actual strategic programming (in Haskell) as opposed to formal investigation.

2.1. A core calculus of transformations

Fig. 3 shows the syntax of a core calculus for transformations. Later we will also cover queries.[3] Transformations are sufficient to introduce strategies in a formal manner; we are confident that our approach easily extends to queries. The core calculus of Fig. 3 follows closely Visser *et al.*'s seminal work [70].

The calculus contains basic strategies as well as strategy *combinators*. The basic strategy *id* denotes the always succeeding identity transformation, which can be thought of as the identity function. The basic strategy *fail* denotes the always failing transformation. Here we note that strategies can either fail, succeed, or diverge. (We use the symbol "↑" to denote failure in the upcoming semantics.) There is also a basic strategy form for rewrite rules in the sense of term rewriting. All strategies—including rewrite rules—are applied at the root node of the input. Traversal into internal nodes of the tree is provided by designated combinators.

Turning to the combinators, the composed strategy $s; s'$ denotes the sequential composition of s and s'. The composed strategy $s \nleftarrow s'$ denotes left-biased choice: try s first, and try s' second—if s failed. The characteristic combinators of strategic programming are $\square$ and $\diamond$ (also known as '*all*' and '*one*'). These combinators model so-called *one-layer traversal*. The strategy $\square(s)$ applies s to all immediate subterms of a given term. If there is any subterm for which s fails, then $\square(s)$ fails, too. Otherwise, the result is the term that is obtained by the application of the original outermost constructor to the processed subterms. The strategy $\diamond(s)$ applies s only to one immediate subterm, namely the leftmost one, if any, for which s succeeds.

[3] *Note on terminology*: some of the strategic programming literature prefers the terms 'type-preserving strategies' for transformations and 'type-unifying strategies' for queries. We do not follow this tradition here—in the interest of a simple terminology.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS

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

$$id \,@\, t \rightsquigarrow t \quad [\text{id}^+]$$

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

$$\frac{s_1 \,@\, t \rightsquigarrow t'}{s_1 \leftarrow\!\!+ s_2 \,@\, t \rightsquigarrow t'} \quad [\text{choice}^+.1]$$

$$\frac{s_1 \,@\, t \rightsquigarrow \uparrow \;\wedge\; s_2 \,@\, t \rightsquigarrow t'}{s_1 \leftarrow\!\!+ s_2 \,@\, t \rightsquigarrow t'} \quad [\text{choice}^+.2]$$

$$\frac{\forall i \in \{1, \ldots, n\}.\, s \,@\, t_i \rightsquigarrow t_i'}{\square(s) \,@\, c(t_1, \ldots, t_n) \rightsquigarrow c(t_1', \ldots, t_n')} \quad [\text{all}^+]$$

$$\frac{\begin{array}{l} \exists i \in \{1, \ldots, n\}. \\ \quad\; s \,@\, t_i \rightsquigarrow t_i' \\ \wedge \quad \forall i' \in \{1, \ldots, i-1\}.\, s \,@\, t_{i'} \rightsquigarrow \uparrow \\ \wedge \quad \forall i' \in \{1, \ldots, i-1, i+1, \ldots, n\}.\, t_{i'} = t_{i'}' \end{array}}{\diamond(s) \,@\, c(t_1, \ldots, t_n) \rightsquigarrow c(t_1', \ldots, t_n')} \quad [\text{one}^+]$$

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

Fig. 4. Positive rules of natural semantics for the core calculus.

If s fails for all subterms, then $\diamond(s)$ fails, too. The one-layer traversal combinators, when used within a recursive closure, enable the key capability of strategic programming: to describe arbitrarily deep traversal into heterogeneously typed terms.

A comprehensive instantiation of strategic programming may require additional strategy primitives that we omit here. 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), and negation (i.e., a strategy to invert the success and failure behavior of a given strategy) [70]. As motivated earlier, we also omit specific primitives for queries.

Following [70] and subsequent work on the formalization of traversal strategies [35,30], we give the formal semantics of the core calculus as a big-step operational semantics using a success and failure model, shown in Figs. 4 and 5. (The version shown has been extracted from a mechanized model of traversal strategies, based on the theorem prover Isabelle/HOL [30].)

The judgment $s \,@\, t \rightsquigarrow r$ describes a relation between a strategy expression s, an input term t, and a result r, which is either a proper term or failure—denoted by '$\uparrow$'. Based on tradition, we separate positive rules (resulting in a proper term) and negative rules (resulting in '$\uparrow$'). Incidentally, this distinction already helps in understanding the success and failure behavior of strategies. For instance, the positive rule [all$^+$] models that the strategy application $\square(s) \,@\, c(t_1, \ldots, t_n)$ applies s to all the t_i such that new terms t_i' are obtained and used in the result 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 semantics shown uses variables only for the sake of recursive closures, and the semantics of these is modeled by substitution. We could also furnish variables for the sake of parameterized 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.

The semantics commits to a simple, deterministic model: each strategy application evaluates to one term deterministically, or it fails, or it diverges. Further, the semantics of the choice combinator is left-biased with only local backtracking. Non-deterministic semantics have been considered elsewhere such that results may be lists (or sets) of terms with the empty list (or set) representing failure [6,7]. Non-determinism is also enabled naturally by a monadic-style functional programming embedding when using the list monad.

2.2. Traversal schemes

Assuming an *ad hoc* notation for parameterized strategy definition—think, for example, of macro expansion—some familiar traversal schemes and necessary helpers are defined in Fig. 6. We refer to [33,59] for a more detailed discussion of the design space for traversal schemes.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

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

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

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

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

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

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

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

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

Fig. 5. Negative rules of natural semantics for the core calculus.

$full_td(s)$	$=$	$\mu v.s;\,\Box(v)$	-- Apply s to each subterm in top-down manner
$full_bu(s)$	$=$	$\mu v.\Box(v);\,s$	-- Apply s to each subterm in bottom-up manner
$once_td(s)$	$=$	$\mu v.s \nleftarrow\!\!+ \Diamond(v)$	-- Find one subterm (top-down) for which s succeeds
$once_bu(s)$	$=$	$\mu v.\Diamond(v) \nleftarrow\!\!+ s$	-- Find one subterm (bottom-up) for which s succeeds
$stop_td(s)$	$=$	$\mu v.s \nleftarrow\!\!+ \Box(v)$	-- Stops when s succeeds on a 'cut' through the tree
$stop_bu(s)$	$=$	$\mu v.\Box(v) \nleftarrow\!\!+ s$	-- An illustrative programming error; see the text.
$innermost(s)$	$=$	$repeat(once_bu(s))$	-- A form of innermost normalization à la term rewriting.
$repeat(s)$	$=$	$\mu v.try(s;\,v)$	-- Fixed point iteration; apply s until it fails.
$try(s)$	$=$	$s \nleftarrow\!\!+ id$	-- Recovery from failure of s with catch-all *id*.

Fig. 6. Familiar traversal schemes.

The traversal scheme $stop_bu(s)$ from Fig. 6 is in fact bogus; it is only included here to provide an early, concrete example of a conceivable programming error. That is, 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 is. A library author may notice the problem eventually. A 'regular' strategic programmer may make similar programming errors when developing problem-specific traversals.

2.3. 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 '□' traversals can be composed into one. Such a simple law does not hold for '◇', neither does it hold generally for traversal schemes. In fact, little is known about algebraic laws for traversal schemes, but see [29,58] for some related research.

We also illustrate three non-laws at the foot of Fig. 7, which show putative equalities. The first reflects that fact that sequential composition is (of course) not commutative, the second that choice is (indeed) left-biased, and the third that distributivity is limited. Finding a counterexample to the third non-law we leave as an exercise for the reader.

ARTICLE IN PRESS

[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 judgment *constant* that holds for all constant terms, i.e., terms with 0 subterms.

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

Fig. 8. Syntax according to Fig. 3 in Haskell.

Let us also discuss basic properties of success and failure behavior for strategies. According to [30], 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 is 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.

These properties can be confirmed based on induction arguments. For instance, (the all-based branch of the choice in) a stop-top-down 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 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 because the argument strategy could continuously increase the given term before traversing into it. This will be illustrated in Section 3. We will address termination by means of static program analysis in Section 5.

2.4. A Haskell-based interpreter for strategies

Let us also provide an interpreter-based model of the core calculus in Haskell. In fact, we provide this model in a way that we can easily refine it later on for the purpose of static program analysis in Section 5. The interpreter-based model is not very useful though for actual programming in Haskell because it is essentially untyped with regard to the terms being transformed. We will properly embed strategies in Haskell in Section 2.5.

Fig. 8 shows the algebraic data type T for the syntax of transformations according to the core calculus. There are constructors for the various basic strategies and combinators. There is, however, no strategy form for rewrite rules because we can easily represent rewrite rules as functions. Please note that type T is parameterized by the type x for variables. Here we

```
-- Representation of terms
data Term   = Term Constr [Term]
type Constr = String

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

-- Interpreter function
interpret  ::  T 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 combinator
fixProperty  ::  (x  -> x) -> x
fixProperty  f = f  (fixProperty  f)

-- Common helper for All and One
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. 9. Haskell-based interpreter for transformation strategies.

apply a modeling technique for syntax that allows us to model variables and binding constructs of the interpreted language with variables and binding constructs of the host language. Further, this particular style supports repurposing this syntax most conveniently during abstract interpretation in Section 5. Finally, the style frees us from implementing substitution.

Fig. 9 shows the actual interpreter, which is essentially a recursive function, *interpret*, on the syntactical domain—subject to a few auxiliary declarations as follows. There is an algebraic data type *Term* for terms to be transformed; constructors are assumed to be strings; see the type synonym *Constr*. There is a type synonym *Meaning* to make explicit the type of functions computed by the interpreter. That is, a strategy is mapped to a function from terms to terms where the function is partial in the sense of the *Maybe* type constructor.[4] The type *Meaning* hence also describes what variables are to be bound to.[5]

[4] *Note on Haskell*: *Maybe* is defined as follows: ***data*** *Maybe x = Just x | Nothing*. The constructor *Just* is applied when a value (i.e., a result in our case) is available whereas the constructor *Nothing* is applied otherwise. Maybe values can be inspected by regular pattern matching, but we also use the convenience function *maybe :: b —> (a —> b) —> Maybe a —> b* which applies the first argument if the given 'maybe' is *Nothing* and otherwise the second argument (a function) to the value.

[5] *Note on Haskell*: Haskell in all its glory has infinite and partial data structures, such as trees with undefined leaves, or indeed undefined subtrees. In principle, the data type *Term* can be used in such a manner. In the presence of infinite and partial structures, the discussion of strategy semantics and properties (most 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.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS


```
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_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. 10. Familiar traversal schemes for interpretation in Haskell.

```
—— Transformations as generic functions
type T m = forall x. Term x => x —> m x

—— Combinator types
idT :: Monad m => T m
failT :: MonadPlus m => T m
sequT :: Monad m => T m —> T m —> T m
choiceT :: MonadPlus m => T m —> T m —> T m
allT :: Monad m => T m —> T m
oneT :: MonadPlus m => T m —> T m
adhocT :: (Term x, Monad m) => T m —> (x —> m x) —> T m

—— Trivial combinator definitions
idT = return
failT = const mzero
sequT f g x = f x >>= g
choiceT f g x = f x 'mplus' g x

—— Non—trivial combinator definitions using pseudo—code
allT f (C t1 ... tn) =
     f t1 >>= \t1' —> ... f tn >>= \tn' —>
     return (C t1' ... tn')

oneT f (C t1 ... tn) =
     ... —— elided for brevity

adhocT s f x =
     if typeOf x == argumentTypeOf f
         then f x
         else s x
```

Fig. 11. Embedding strategies (transformations) in Haskell.

The equations of the *interpret* function combine the positive and negative rules of the natural semantics in a straightforward manner. The only special case is the approach to recursion. We use a fixed point combinator, *fixProperty*, to this end. Its name emphasizes that the definition of the operator is immediately the defining property of a fixed point.

The traversal combinators are interpreted with the help of auxiliary functions *transform*, *all*, and *one*. One-layer traversal is essentially modeled by means of mapping over the list of subterms while using the folklore list-processing function *map*. The auxiliary functions are otherwise only concerned with the manipulation of failure for processed subterms.

Fig. 10 shows familiar traversal schemes in the Haskell-based syntax.

2.5. Embedding strategies into Haskell

By embedding strategies into Haskell, they can be applied to programmer-defined data types such as those illustrative data models for companies and programming-language syntax in Fig. 2 in the introduction.

Fig. 11 defines the generic function type for transformations and the function combinators for the core calculus. A number of aspects of this embedding need to be carefully motivated.

The type *T* uses forall-quantification to emphasize that strategies are indeed generic (say, polymorphic) functions because they are potentially applied to many different types of subterms along traversal. The context *Term x => ...* emphasizes that strategies are not universally polymorphic functions, but they can only be applied to types that instantiate the Haskell class

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

A Haskell class is specified by a signature, which can be thought of as an interface. An implementation of the class—called an *instance* in Haskell—is given by defining the interface functions for a particular type. A typical example is the equality class, *Eq*, shown with an instance for the Boolean type, *Bool*.

```
class Eq a where
  (==) :: a —> a —> Bool
instance Eq Bool where
  (x == y)  =  if x then y else (not y)
```

A function may be polymorphic yet require that a type is a member of a particular class, as in the list membership function; the context *Eq a => ...* constrains *a* to be a member of the *Eq* class.

```
elem :: Eq a => a —> [a] —> Bool
elem x ys = or [ x==y | y<—ys ]
```

Since *Bool* is in the class *Eq*, then *elem* can be used over Boolean lists.

Fig. 12. A note on classes in Haskell.

Term.[6] Thus, *Term* is the *set* of types of terms. (For comparison, in the interpreter-based model, *Term* denotes the type of terms.) The operations of the *Term* class enable traversal and strategy extension (see below). The specifics of these operations are not important for the topic of the present paper; see the Strafunski/'Scrap Your Boilerplate' literature [43,38] for details.

There is the *adhocT* combinator that has no counterpart in the formal semantics and the interpreter-based model because it is specifically needed for static typing when different types can be traversed. The *adhocT* combinator enables so-called *strategy extension* as follows. The strategy *adhocT s f* constructs a new strategy from the given strategy *s* such that the result behaves like the type-specific case *f*, when *f* is applicable and like *s* otherwise. This is also expressed by the pseudo-code. We omit technical details that are not important for the topic of the present paper; see, again, [43,38] for details. It is important though to understand that an operation like *adhocT* combinator is essential for a (conservative) typeful embedding. This will be illustrated shortly.

When compared to the interpreter of Fig. 9, the embedding does not refer to the *Maybe* type constructor for the potential of failure in strategy application. Instead, a type-constructor parameter for a monad *m* is used.[7] The use of monads is a strict generalization of the use of *Maybe* because *(a) Maybe* is a specific monad and *(b)* other monads can be chosen to compose traversal behavior with additional computational aspects. This generalization has been found to be essential in practical, strategic programming in Haskell. By instantiating *m* as *Maybe*, we get these types:

```
idT :: T Maybe
failT :: T Maybe
sequT :: T Maybe —> T Maybe —> T Maybe
... .
```

We note that the choice of *Monad* versus *MonadPlus* in the original function signatures of Fig. 11 simply expresses what is required by the combinators' definitions. For instance, *idT* does not refer to any members of the *MonadPlus* class whereas *choiceT* does.

The pseudo-code for the *allT* combinator expresses that the argument function (say, strategy) is applied to all immediate subterms, the various computations are sequenced, and a term with the original outermost constructor is constructed from the intermediate results—unless failure occurred.

Fig. 14 expresses familiar traversal schemes with the embedding. The function definitions are entirely straightforward, but two details are worth noticing as they relate to the central topic of programming errors. First, the function definitions use general recursion, thereby implying the potential for divergence. Second, the distinction of *Monad* and *MonadPlus* in the function signatures signals whether control flow can be affected by fallible arguments of the schemes, but the types do not imply rejection of infallible arguments, thereby implying the potential for degenerated control flow.

Fig. 15 composes a traversal for the transformation of companies such that the salaries of all employees are increased by 1 Euro. The implementation of the traversal is straightforward: we select 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 local function *f* can be viewed as a rewrite rule in that it rewrites employees—there is pattern matching on the left-hand side and term construction on the right-hand side. The trivial identity monad, *Id*, is used here because the traversal is a pure function—without even the potential of failure. Fig. 1 also proposed a slightly more involved transformation scenario for companies: decrease the salaries of all non-top-level managers. We leave this scenario as an exercise for the reader.

[6] *Note on Haskell*: Fig. 12 gives a brief overview of Haskell classes (say, type classes) for those unfamiliar with this aspect of the Haskell language.

[7] *Note on Haskell*: Fig. 13 gives a brief overview of monads for those unfamiliar with the concept.

Classes can also describe interfaces over type constructors, that is, functions from types to types. For instance:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Monads encapsulate an interface for 'computations over *a*', since *return x* gives the trivial computation of the value *x* and *(>>=)* or 'bind' allows computations to be sequenced. The simplest implementation of *Monad* is the identity type function:

```
data Id a = Id { getId :: x }

instance Monad Id where
  return x = Id x
  c >>= f = f (getId c)
```

Other instances of *Monad* provide for non-deterministic or stateful computation, which can be used to good effect in traversals, e.g., to accumulate context information during the traversal.

In a similar way, *MonadPlus* encapsulates the concept of computations that might fail, witnessed by the *mzero* binding, and *mplus* combines together the results of two computations that might fail, transmitting failure as appropriate. The simplest instance of *MonadPlus* is the *Maybe* type:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  (Just x) >>= f = f x

instance MonadPlus Maybe where
  mzero = Nothing
  mplus Nothing y = y
  mplus x _ = x
```

Every instance of *MonadPlus* presupposes an instance of *Monad*, but not *vice versa*.

Fig. 13. A note on monads in Haskell.

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

full_td s    = s `sequT` allT (full_td s)
full_bu s    = allT (full_bu s) `sequT` s
once_td s    = s `choiceT` oneT (once_td s)
once_bu s    = oneT (once_bu s) `choiceT` s
stop_td s    = s `choiceT` allT (stop_td s)
innermost s  = repeat (once_bu s)
repeat s     = try (s `sequT` repeat s)
try s        = s `choiceT` idT
```

Fig. 14. Familiar traversal schemes embedded in Haskell.

2.6. A note on queries

For most of the paper we focus on transformations since queries do not seem to add any additional, fundamental challenges. However, we extend the embedding approach here to include queries for a more complete illustration of strategic programming. Fig. 16 defines the generic function type for queries and corresponding function combinators.

ARTICLE IN PRESS


```
-- Increase the salaries of all employees.
increase_all_salaries :: T Id
increase_all_salaries = full_td (adhocT idT f)
 where
  f (Employee n s) = Id (Employee n (s+1))
```

Fig. 15. Implementation of a transformation scenario from Fig. 1.

```
-- Queries as generic functions
type Q r = forall x. Term x => x -> r

-- Combinator types
constQ :: r -> Q r
failQ :: MonadPlus m => Q (m r)
bothQ :: Q u -> Q u' -> Q (u,u')
choiceQ :: MonadPlus m => Q (m r) -> Q (m r) -> Q (m r)
allQ :: Q r -> Q [r]
adhocQ :: Term x => Q r -> (x -> r) -> Q r

-- Trivial combinator definitions
constQ r = const r
failQ = const mzero
bothQ f g x = (f x, g x)
choiceQ f g x = f x `mplus` g x

-- Non-trivial combinator definitions using pseudo-code
allQ f (C t1 ... tn) =
    [f t1, ..., f tn]

adhocQ s f x =
    if typeOf x == argumentTypeOf f
       then f x
       else s x
```

Fig. 16. Embedding strategies (queries) in Haskell.

The generic type *Q* models that queries may be applied to terms of arbitrary types while the result type *r* of the query is a parameter of the query; it does not depend on the actual type of the input term. Here, we note that *Q* is not parameterized by a monad-type constructor, as it is the case for *T*. This design comes without loss of generality because the result type *r* may be instantiated also to the application of a monad-type constructor, if necessary.

The basic strategy *constQ r* denotes the polymorphic constant function, which returns *r*—no matter the input term or its type. The basic strategy *failQ* is the always failing query. There is no special sequential composition for queries because regular function composition is appropriate—given that the result of a query is of a fixed type. However, there is the combinator *bothQ* which applies two queries to the input and returns both results as a pair. Further, there is also a form of choice, *choiceQ*, for queries. Ultimately, there is also one-layer traversal for queries. We only show the combinator *allQ*, which essentially constructs a list of queried subterms. Finally, there is also a form of strategy extension for queries.

Fig. 17 expresses useful traversal schemes with the embedding. The first two schemes are parameterized over a monoid to allow for the collection of data in a general manner.[8] (The postfix "cl" hints at "collection".) That is, the monoid's type provides the result type of queries and the monoid's binary operation is used to combine results from querying many subterms. The third traversal scheme in Fig. 17 deals with finding a single subterm of interest as opposed to collecting data from many subterms of interest.

Fig. 19 composes traversal for the query scenarios on companies: total salaries of all employees or non-managers, only. The implementation of the former is straightforward; perhaps surprisingly, the implementation of the latter is significantly more involved. The simple collection scheme *full_cl* is inappropriate for totaling all non-managers because the scheme would reach all nodes eventually—including the employee subterms that are part of manager terms, from which salaries must not be extracted though. Hence, a traversal with 'stop' is needed indeed. Further, an always failing default is needed here—again, in contrast to the simpler case of totaling all salaries. Finally, the solution depends on the style of data modeling. That is,

8 *Note on Haskell*: Fig. 18 gives a brief overview of monoids for those unfamiliar with the concept.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

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

-- Collection with stop
stop_cl :: Monoid u => Q (Maybe u) -> Q u
stop_cl s = maybe mempty id
            . (s `choiceQ` (Just . mconcat . allQ (stop_cl s)))

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

Fig. 17. Traversal schemes for queries embedded in Haskell.

The *Monoid* type class encapsulates a type with a binary operation, *mappend*, and a unit, *empty*, for that operation:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

The simplest instance is the list monoid, which indeed suggests the names used in the class.

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

Other instances are given by addition and zero (or multiplication and one) over numbers, wrapped by the *Sum* constructor:

```
newtype Sum a = Sum { getSum :: a }
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum x `mappend` Sum y = Sum (x + y)
```

In each case *mconcat* is used to accumulate a list of values into a single value. This value will be independent of the way in which the accumulation is done if the instance satisfies the *Monoid* laws:

```
mappend mempty x       = x
mappend x mempty       = x
mappend x (mappend y z) = mappend (mappend x y) z
```

Fig. 18. A note on the *Monoid* class in Haskell.

```
-- Total the salaries of all employees.
total_all_salaries :: Q Float
total_all_salaries = getSum . full_cl (adhocQ (constQ mempty) f)
 where
  f (Employee _ s) = Sum s

-- Total the salaries of all employees who are not managers.
total_all_non_managers :: Q Float
total_all_non_managers = getSum . stop_cl type_case
 where
  type_case :: Q (Maybe (Sum Float))
  type_case = adhocQ (adhocQ (constQ Nothing) employee) manager
  employee (Employee _ s) = Just (Sum s)
  manager (Manager _) = Just (Sum 0)
```

Fig. 19. Implementation of two query scenarios from Fig. 1.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

the assumed data model distinguishes the types of managers and employees. Hence, we can use an extra type-specific case for managers to stop collection at the manager level. Without the type distinction in the data model, the traversal program would need to exploit the special *position* of managers within department terms.

The transformation scenario for decreasing the salaries of all non-top-level managers, which we left as an exercise for the reader, calls for similarly involved considerations. These illustrations may help to confirm that programming errors are quite conceivable in strategic programming—despite the conciseness of the programming style.

3. Inventory of strategic programming errors

The implementation of a strategic programming (sub-) problem (say, a traversal problem) is normally centered around some *problem-specific ingredients* ('rewrite rules') that have to be organized in a more or less complex strategy. There are various decisions to be made and accordingly, there are opportunities for misunderstanding and programming errors. This section presents a fine-grained inventory of programming errors by reflecting systematically on the use of basic strategy combinators and library abstractions in the implementation of traversal problems. We use a deceptively simple scenario as the running example. We begin with a short proposal of the assumed process of designing and implementing traversal programs, which in itself may improve understanding of strategic programming and help reducing programming errors. The following discussion is biased towards transformations, but coverage of queries would require only a few, simple adaptations.

3.1. Design of traversal programs

Traversal programming is based on the central notion of *terms of interest*—these are the terms to be affected by a transformation. When *designing a traversal*, the terms of interest are to be identified along several axes:

Types Most obviously, terms of interest are of certain types. For instance, a transformation for salary increase may be concerned with the type of employees.

Patterns and conditions Terms of interest often need to match certain patterns. For instance, a transformation for the application of a distributive law deals with the pattern $x * (y + z)$. In addition, the applicability of transformations is often subject to (pre-) conditions.

Position-based selection A selection criterion may be applied if the transformation should not affect all terms with fitting patterns and conditions. In an extreme case, a single term is to be selected. Such selection typically refers to the position of these terms; think of top-most versus bottom-most.

Origin The initial input is expected to contribute terms with fitting patterns and conditions, but previous applications of rewrite rules may contribute terms as well. Hence, it must be decided whether the latter kind of origin also qualifies for terms of interest. For instance, an unfolding transformation for recursive function definitions may specifically disregard function applications that were introduced by unfolding.

3.2. Implementation of traversal programs

When *implementing a traversal*, the types and patterns of terms of interest are modeled by the left-hand sides of rewrite rules. Conditions are typically modeled by the rewrite rules, too, but the choice of the traversal scheme may be essential for being able to correctly check the conditions. For instance, the traversal scheme may need to pass down information that is needed by a condition eventually. The axes of selection and origin (of terms of interest) are expressed through the choice of a suitable traversal scheme.

Let us provide a summary of basic variation points in traversal implementation. For simplicity, let us focus here on traversal problems whose implementation corresponds to a strategy that has been built by applying one or more traversal schemes from a library to the problem-specific rewrite rules, possibly subject to composition of rewrite rules or sub-traversals.

Organizing the strategy involves the following decisions:

Scheme Which traversal scheme is to be used?
- Is a full or a limited traversal required?
- Does top-down versus bottom-up order matter?
- Does a strategy need to be iterated?
- ...

Default What polymorphic and monomorphic defaults are to be used?
- The identity transformation.
- The always failing transformation.
- Another, more specific, behavior.

Composition How to compose a strategy from multiple parts?
- Use type case (strategy extension) at the level of rewrite rules.
- Combine arguments of a traversal scheme in a sequence.

ARTICLE IN PRESS

- Combine arguments of a traversal scheme in a choice.
- Combine traversals in a sequence.
- Combine traversals in a choice.

Based on an appropriate running example we shall exercise these choices, and see the consequences of incorrect decisions as they cause programming errors in practice. In our experience, wrong choices are the result of insufficiently understanding (i) the variation points of traversal schemes, (ii) the subtleties of control flow, and (iii) the axes of terms of interest in practical scenarios.

3.3. 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 iterate ℓ over any term.

For concreteness' sake, we operate on n-ary trees of natural numbers. Further, we assume a Peano-like definition of the data type for numbers. Here are the data types for numbers and trees:

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

The Peano-induced recursion implies a simple form of nesting. 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-modeling level—think of nesting of departments in the company example, or nesting of expressions or function-definition blocks in the AST example given in the introduction.

Here are simple tree samples:

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

The rewrite rule for incrementing numbers is represented as follows:

```
increment n = Succ n.
```

In fact, let us use monadic style because the basic Strafunski-like library introduced in Section 2.5 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 numbers in an arbitrary term. That is, we need to make decisions regarding traversal scheme, default, and composition for the implementation.

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? Also, how to exactly apply the chosen scheme to the given rewrite rule? 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* numbers, while *once_bu* would only affect one number. In the remainder of the section, we will attempt different schemes and vary other details, thereby showcasing potential programming errors.

3.4. Strategies going wrong

The composed strategy may go wrong in different ways:

- It diverges.
- It transforms incorrectly, i.e., numbers are not exactly incremented.
- It does not modify the input, i.e., numbers are not incremented at all.
- It fails even when the transformation is assumed never to fail.
- It succeeds even when failure is preferred for terms without numbers.

Let us consider specific instances of such problems.

3.4.1. Divergent traversal

Let us attempt a full top-down traversal. Alas, the traversal diverges:[9]

[9] *Note on Haskell*: Throughout the section, we operate at the Haskell prompt. That is, we show input past the '>' prompt sign and resulting output, if any, right below the input.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS

> full_td (adhocT idT increment) tree1
... an infinite tree is printed ... .

The intuitive reason for non-termination is that numbers are incremented prior to the traversal's descent. Hence, the term under traversal grows and each increment enables another increment.

Let us attempt instead the *innermost* scheme. Again, traversal diverges:

> innermost (adhocT failT increment) tree1
... no output ever is printed ... .

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.

Both decisions here illustrate the case of choosing the wrong traversal scheme which in turn may be the result of insufficiently understanding some axes of terms of interest (see Section 3.1) and associated properties of rewrite rules. In particular, the traversal schemes used here support the *origin* axis in a way terms of interest are created by the traversal.

3.4.2. Incorrect transformation

Let us attempt instead the *full_bu* scheme:

> full_bu (adhocT idT increment) tree1
Just (Node {rootLabel = Succ Zero, subForest = []}).

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 numbers, then we would learn that the composed strategy replaces each number n by $2n + 1$ as opposed to $n + 1$. To see this, one should notice that a number n is represented as a term of depth $n + 1$, and the choice of the scheme *full_bu* implies that *increment* applies to each 'sub-number'.

More generally, we see an instance of an overlooked *applicability condition* (see Section 3.1) in that numbers are terms of interest, but not subterms thereof. The same kind of error could occur in the implementation of any other scenario as long as it involves nesting. In real-world scenarios, nesting may actually arise also through mutual (data type-level) recursion.

3.4.3. No-op traversal

Finally, let us attempt the *stop_td* scheme. Alas, no incrementing happens:

> stop_td (adhocT idT 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 number is hit, but the polymorphic default *idT* makes the strategy stop for any subterm that is not a number. Let us replace *idT* by *failT*. Finally, we arrive at a proper solution for the original problem statement:

> stop_td (adhocT failT increment) tree1
Just (Node {rootLabel = Succ Zero, subForest = []}).

Hence, *stop_td* is the correct traversal scheme for the problem at hand, but we also need to be careful about using the correct polymorphic default for lifting the rewrite rule to the strategy level; see Section 3.2.

3.4.4. Unexpected failure

failT 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 number n by $2n + 1$, as we accidentally ended up doing in Section 3.4.2. Back then, the polymorphic default *idT* was appropriate for *full_bu*. In contrast, the default *failT* is not appropriate:

> full_bu (adhocT failT increment) tree1
Nothing.

This is a case of unexpected failure in the sense that we expect the traversal for incrementing numbers to succeed for all possible input terms. The problem is again due to the wrong choice of default.

3.4.5. Unexpected success

Let us apply the confirmed scheme and default to a tree of Booleans:

> stop_td (adhocT failT increment) tree2
Just (Node { rootLabel = True, subForest = [] }).

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

Of course, no incrementing happens; the output equals the input. Arguably, a strategic programmer could expect that the traversal should fail, if the rewrite rule for incrementing never applies. For comparison, the traversal scheme *once_bu* does indeed fail in case of inapplicability of its argument. Defining a suitable variation on *stop_td* that indeed fails in the assumed way we leave as an exercise for the reader. Misunderstood success and failure behavior may propagate as a programming error as it may affect the control flow in the strategic program.

3.5. Subtle control flow

Arguably, several of the problems discussed involve subtleties of control flow. It appears to be particularly difficult to understand and to correctly configure control flow of strategies on the grounds of success and failure behavior for operands in strategy composition.

Let us modify the running example slightly to provide another illustration. We consider the refined problem statement that only *even* numbers are to be incremented. In the terminology of rewriting, this statement calls for a conditional rewrite rule:[10]

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

-- Haskell code (monadic notation)
increment_even n = do guard (even n); increment n.
```

We use the same traversal scheme as before:

```
> stop_td (adhocT failT increment_even) tree1
Just (Node {rootLabel = Succ Zero, subForest = []}).
```

This particular test case looks fine, but if we were testing the same strategy with trees that contain odd numbers, 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 numbers, we will hit an even sub-number in the next step, which is hence incremented. So we need to make sure that recursion ceases for *all* numbers. Thus:

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

The example shows the subtleties of control flow in strategic programming: committing to the specific monomorphic type in *adhocT* can still fail, and so lead to further traversal.

3.6. Dead code

So far we have mainly spotted programming errors through comparison of expected with actual output, if any. Let us now switch to the examination of composed strategies. There are recurring patterns of producing dead code in strategic programming. We take the position here that dead code is a symptom of programming errors.

Consider the following patterns of strategy expressions:

- *adhocT* (*adhocT* s f_1) f_2
- *choiceT* s_1 s_2
- *sequT* s_1 s_2.

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

Let us illustrate the first kind of programming error: two type-specific cases of the same type that are composed with *adhocT*. Let us consider a refined problem statement such that incrementing of numbers is to be replaced by (i) increment *by one* all odd numbers, (ii) increment *by two* all even numbers. Here are the basic building blocks that we need:

[10] 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 number.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

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

atEven n  | even n      = Just  (Succ (Succ n))
          | otherwise   = Nothing.
```

Arguably, both rewrite rules could also have been combined in a single function to start with, but we assume here a modular decomposition as the starting point. We also leave it as an exercise to the reader to argue that the monomorphic default *Nothing* is appropriate for the given problem. Intuitively, we wish to compose these type-specific cases so that both of them are tried.

Let us attempt a composition that uses *adhocT* twice:

```
> stop_td (adhocT (adhocT failT 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 numbers, and the case for odd numbers dominates the one for even numbers. The case for even numbers is effectively dead code. In the sample tree, the number, *Zero*, is even.

The two rewrite rules need to be composed at the monomorphic level of the number type—as opposed to the polymorphic level of strategy extension. To this end, we need composition combinators that can be applied to functions of specific types as opposed to generic functions:

```
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 (adhocT failT (mchoice atEven atOdd)) tree1
Just (Node {rootLabel = Succ (Succ Zero), subForest = []}).
```

We face a more subtle form of dead code when the root type for terms in a traversal implies that the traversal cannot encounter subterms of the type expected by a type-specific case. Consider again the strategy application that we already used for the illustration of potentially unexpected success in Section 3.4.5:

```
> stop_td (adhocT failT increment) tree2
Just (Node { rootLabel = True, subForest = [] }).
```

The output equals the input. In this application, the rewrite rule *increment* is effectively dead code. In fact, it is not important what actual input is passed to the strategy. It suffices to know that the input's type is *Tree Boolean*. Terms of interest, i.e., numbers, cannot possibly be found below any root of type *Tree Boolean* and the given strategy is a no-op in such a case. This may be indeed a symptom of a programming error: we either meant to traverse a different term (i.e., one that contains numbers) or we meant to invoke a different strategy (i.e., one that affects Boolean literals or polymorphic trees). Accordingly, one could argue that the strategy application at hand should be rejected statically.

3.7. Options of composition

As a final exercise on the matter of strategy composition, let us study one more time the refined example for incrementing odd and even numbers as introduced in Section 3.6. We take for granted the following decisions: *stop_td* for the traversal scheme and *failT* for the polymorphic default. Given all the principal options for composition, as of Section 3.2, there are the following concrete options for the example:

1. *stop_td (adhocT (adhocT failT atEven) atOdd)*
2. *stop_td (adhocT failT (mchoice atEven atOdd))*
3. *stop_td (adhocT failT (msequ atEven atOdd))*
4. *stop_td (choiceT (adhocT failT atEven) (adhocT failT atOdd))*
5. *stop_td (sequT (adhocT failT atEven) (adhocT failT atOdd))*
6. *choiceT (stop_td (adhocT failT atEven)) (stop_td (adhocT failT atOdd))*
7. *sequT (stop_td (adhocT failT atEven)) (stop_td (adhocT failT atOdd)).*

(We do not exercise all variations on the order of operands.) 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. Demonstrating and explaining the issues with these strategies we leave as an exercise for the reader.

4. Static typing of traversal strategies

We use established means of static typing to curb the identified programming errors, to the extent possible, in a way that basic strategy combinators and library abstractions are restricted in generality. In particular, we use static typing to avoid wrong decisions regarding strategy composition, to reduce subtleties of control flow, and to avoid some forms of dead code. We do not design new type systems here. Instead, we attempt to leverage established means, as well as we can.

The section is organized as a sequel of contributions—each of them consisting of language-agnostic advice for improving strategic programming and an illustration in Haskell. We use Haskell for illustrations because it is an established programming language for statically typed strategic programming and its type system is rather powerful in terms of supporting different forms of polymorphism and simple forms of dependent typing [50,65,63,32,60]. A basic 'reading knowledge' of Haskell, supplemented with the background notes in Section 2, should be sufficient to understand the essence of the Haskell illustrations.

We provide language-agnostic advice because different languages may need to achieve the suggested improvements in different ways, if at all. In fact, not even Haskell's advanced type system achieves the suggested improvements in a fully satisfactory manner. Hence, the section may ultimately suggest improvements of practical type systems or appropriate use of existing proposals for type-system improvements, e.g., [14,64,51,57,45,80,8,16].

4.1. Hard-wire defaults into traversal schemes

Advice 1. *By hard-wiring a suitable default into each traversal scheme, rule out wrong decisions regarding the polymorphic default during strategy composition (see Section 3.2). Here we assume that the default can either be statically defined for each scheme or else that it can be determined at runtime by observing other arguments of the scheme.*

We can illustrate the advice in Haskell in a specific manner by reducing the polymorphism of traversal schemes as follows. While the general schemes of Section 2.5 are essentially parameterized by polymorphic functions on terms (in fact, rank-2 polymorphic functions [38,57]), the restricted schemes are parameterized by specific type-specific cases. There is also a proposal for a variation of 'Scrap Your Boilerplate' that points in this direction [52].

The following primed definitions take a type-specific case *f*, which is then generalized *within the definition* by means of the appropriate polymorphic default, *idT* or *failT*. We delegate to the more polymorphic schemes otherwise.

```
full_td' :: (Term x, Term y, Monad m) => (x —> m x) —> y —> m y
once_bu' :: (Term x, Term y, MonadPlus m) => (x —> m x) —> y —> m y
stop_td' :: (Term x, Term y, MonadPlus m) => (x —> m x) —> y —> m y
...


full_td' f = full_td (adhocT idT f)
once_bu' f = once_td (adhocT failT f)
stop_td' f = stop_td (adhocT failT f)
... .
```

These schemes reduce programming errors as follows. Most obviously, polymorphic defaults are correct by design because they are hard-wired into the definitions. A side effect is that the use of strategy extension is now limited to the library, and hence strategy composition is made simpler by reducing the number of options (see Section 3.7).

However, there are scenarios that require the general schemes; see [70,43,44] for examples. The problem is that we may need a variable number of type-specific cases. Some scenarios of strategies with multiple cases can be decomposed into multiple traversals, but even when it is possible, it may be burdensome and negatively affect performance. Further, there are cases, when the hard-wired default is not applicable. Hence, the default should be admissible to overriding.

As a result, the restricted schemes cannot fully replace the general schemes. Therefore, a strategic programming library would need to provide both variants and stipulate usage of the restricted schemes whenever possible.

In principle, one can think of unified schemes that can be applied to single type-specific cases, collections thereof, and polymorphic functions that readily incorporate a polymorphic default. Those schemes would need to coerce type-specific cases to polymorphic functions. We will illustrate this idea in a limited manner in Section 4.4.

4.2. Declare and check fallibility contracts

Advice 2. *Curb programming errors due to subtle control flow (see Section 3.5) by declaring and checking contracts regarding fallibility. These contracts convey whether the argument of a possibly restricted traversal scheme is supposed to be fallible and whether the resulting traversal is guaranteed to be infallible (subject to certain preconditions).*

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

The advice is meant to improve strategic programming so that more guidance is provided as far as the success and failure behavior of traversal schemes and their arguments is concerned. According to Section 2.3, traversal schemes differ in terms of their fallibility properties and the dependence of these properties on fallibility properties of the arguments. For instance, *full_td* preserve infallibility, that is, a composed traversal *full_td s* is infallible if the argument *s* is infallible. In contrast, *stop_td s* is infallible regardless of *s*.

The function signatures of the schemes of Section 2.5 hint at fallibility properties: see the distinguished use of *Monad* vs. *MonadPlus*. For instance:

```
full_td :: Monad m => T m -> T m
once_bu :: MonadPlus m => T m -> T m
stop_td :: MonadPlus m => T m -> T m.
```

However, such hinting does not imply checks. 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. Such hinting may also be misleading. For instance, the appearance of *MonadPlus* in the type of *stop_td* may suggest that such a traversal may fail, but, in fact, it cannot. Instead, the appearance of *MonadPlus* hints at the fact that the argument is supposed to be fallible.

We can illustrate the advice in Haskell in a specific manner by providing infallible variations on the traversal schemes of Section 2.5. To this end, we use the identity monad whenever we want to require or imply infallibility. We use the maybe monad whenever the argument of such a scheme is supposed to be fallible. Thus:

```
full_td' :: T Id -> T Id
full_bu' :: T Id -> T Id
stop_td' :: T Maybe -> T Id
innermost' :: T Maybe -> T Id
repeat' :: T Maybe -> T Id
try' :: T Maybe -> T Id.
```

Applications of these restricted schemes are hence guaranteed to be infallible. We cannot provide an infallible variation on *once_bu* due to its nature. It is also instructive to notice that *try* models transition from a fallible to an infallible strategy—not just operationally, as before, but now also at the type level. The inverse transition is not served.

The primed definitions *full_td'* and *full_bu'* simply delegate to the original schemes, but the other primed definitions need to be defined from scratch because they need to compose infallible and fallible strategy types in a manner that requires designated forms of sequence and choice. These new definitions can be viewed as constructive proofs for fallibility properties.

```
full_td' s = full_td s
full_bu' s = full_bu s
stop_td' s = s `choiceT'` allT (stop_td' s)
innermost' s = repeat' (once_bu s)
repeat' s = try' (s `sequT'` repeat' s)
try' s = s `choiceT'` idT

choiceT' :: T Maybe -> T Id -> T Id
choiceT' f g x = maybe (g x) Id (f x)

sequT' :: T Maybe -> T Id -> T Maybe
sequT' f g = f `sequT` (Just . getId . g).
```

The type of *choiceT'* is interesting in so far that it allows us to compose a fallible strategy with an infallible strategy to obtain an infallible strategy. That is, the scope of fallibility is made local.

The fallibility properties were modeled at the expense of eliminating the general monad parameter. Generality could be recovered though by consistently parameterizing all infallible schemes with a plain monad and adding an application of the monad transformer for *Maybe* whenever the argument of such a scheme is supposed to be fallible. Thus:

```
full_td'' :: Monad m => T m -> T m -- equals original type
full_bu'' :: Monad m => T m -> T m -- equals original type
stop_td'' :: Monad m => T (MaybeT m) -> T m
innermost'' :: Monad m => T (MaybeT m) -> T m
repeat'' :: Monad m => T (MaybeT m) -> T m
try'' :: Monad m => T (MaybeT m) -> T m.
```

The definitions are omitted here as they require non-trivial knowledge of monad transformers; see though the paper's online code distribution. These definitions declare the fallibility contracts better than the original schemes, but enforcement

ARTICLE IN PRESS

is limited. The monad-type parameter may be still (accidentally) instantiated to an instance of *MonadPlus*. For instance, the types of *full_td''* and *full_bu''* are not at all constrained, when compared to the original schemes.

4.3. Reserve fallibility for modeling control flow

Advice 3. *Curb programming errors due to subtle control flow (see Section 3.5) by reserving fallibility, as discussed so far, for modeling control flow. If success and failure behavior is needed for other purposes, such as assertion checking, then strategies shall use effects that cannot be confused with efforts to model control flow. The type system must effectively rule out such confusion.*

We can illustrate the advice in Haskell in a specific manner by defining the traversal schemes of Section 2.5 from scratch in terms of two distinct types for infallible versus fallible types:

```
data T m = T { getT :: forall x. Term x => x —> m x }
data T' m = T' { getT' :: forall x. Term x => x —> MaybeT m x }.
```

We use data types here (as opposed to type synonyms) so that the two types cannot possibly be confused. This is discussed in more detail below.

If an infallible strategy needs to fail for reasons other than affecting regular control flow, then the monad parameter can still be used to incorporate the maybe monad or an exception monad, for example. In this manner, strategies may perform assertion checking, as often needed for preconditions of nontrivial transformations, without running a risk of failure to be consumed by the control-flow semantics of the strategic program. In this manner, strategic programming is updated to reliably separate control flow and exceptions (or other effects), as it is common in the general programming field [61,49].

The types of the 'full' traversal schemes reflect that control flow is hard-wired:

```
full_td :: Monad m => T m —> T m
full_bu :: Monad m => T m —> T m.
```

The types of the 'once' traversal schemes reflect that fallibility is essential:

```
once_td :: Monad m => T' m —> T' m
once_bu :: Monad m => T' m —> T' m.
```

The other library combinators construct infallible strategies from fallible ones:

```
stop_td :: Monad m => T' m —> T m
innermost :: Monad m => T' m —> T m
repeat :: Monad m => T' m —> T m
try :: Monad m => T' m —> T m.
```

At first sight, these types look deceptively similar to those that we defined for fallibility contracts in Section 4.2. However, the important difference is that *T m* and *T' m'* cannot be confused whereas this is possible for *T m* and *T (MaybeT m')* if *m* and *MaybeT m'* are unifiable.

The definitions of the new schemes are omitted here as they rely on a designated, non-trivial suite of basic strategy combinators; see though the paper's online code distribution. It is fair to say that the present illustration also addresses Advice 2 regarding fallibility contracts.

4.4. Enable families of type-specific cases

Advice 4. *Rule out dead code due to overlapping type-specific cases (see Section 3.6) by enabling strongly typed families of type-specific cases as arguments of traversal schemes. Such a family is a non-empty list of functions the types of which are pairwise non-unifiable but they all instantiate the same generic function type for strategies.*

We can illustrate the advice in Haskell in a specific manner by making use of advanced type-class-based programming. More specifically, we leverage existing library support for strongly typed, heterogeneous collections—the HList library [32].

Consider the pattern of composing a strategy from several (say, two) type-specific cases and a polymorphic default:

adhocT (adhocT s f_1*)* f_2.

The type-specific cases, f_1 and f_2, are supposed to override the polymorphic default *s* in a point-wise manner. Ignoring static typing for a second, we can represent the two type-specific cases instead as a list $[f_1, f_2]$. Conceptually, it is a heterogeneous list in the sense that the types of the functions for type-specific cases are supposed to be distinct (in fact, non-unifiable); otherwise dead code is admitted. Using HList's constructors for heterogeneous lists, the type-specific cases are represented as follows:

HCons f_1 *(HCons* f_2 *HNil)*.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

```
-- Type-class-polymorphic type of familyT
class (Monad m, HTypeIndexed f) => FamilyT f m
  where
    familyT :: T m -> f -> T m

-- Empty list case
instance Monad m => FamilyT HNil m
  where
    familyT g _ = g

-- Non-empty list case
instance ( Monad m
         , FamilyT t m
         , Term x
         , HOccursNot (x -> m x) t
         )
           => FamilyT (HCons (x -> m x) t) m
  where
    familyT g (HCons h t) = adhocT (familyT g t) h
```

The list of type-specific cases is constrained to only hold elements of distinct types; see the constraint *HTypeIndexed*, which is provided by the HList library. Also notice that the element types are constrained to be function types for monadic transformations; see the pattern *x —> m x* in the head of the last instance. 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; see the constraint *HOccursNot*, which is again provided by the HList library.

Fig. 20. Derivation of a transformation from type-specific cases and a default.

Now suppose that the types of the type-specific cases all instantiate the polymorphic type *T*. Further suppose that there is a function *familyT* for strategy construction; it takes two arguments: the heterogeneous collection and a transformation *s* which serves as polymorphic default. The original strategy is constructed as follows:

familyT (HCons f_1 (HCons f_2 HNil)) s.

The function *familyT* must be polymorphic in a special manner such that it can process all heterogeneous collections of type-specific cases. To this end, the function must be overloaded on all possible types of such collections, which is achieved by type-class-based programming; see Fig. 20.

The family-enabled traversal schemes are defined as follows:

```
full_td' s = full_td (familyT idT s)
full_bu' s = full_bu (familyT idT s)
once_td' s = once_td (familyT failT s)
once_bu' s = once_bu (familyT failT s)
stop_td' s = stop_td (familyT failT s)
innermost' s = innermost (familyT failT s).
```

That is, the family-enabled schemes invoke the *familyT* function to resolve the heterogeneous collection into a regular generic function subject to the polymorphic default that is known for each traversal scheme. In this manner, we do not just avoid dead code for type-specific cases; we also address the issue of Advice 1 in that the polymorphic default is hard-wired into the schemes—though without the restriction to a single type-specific case, as was the case in Section 4.1.

Admittedly, the illustration involves substantial encoding. For example, type errors in type-class-based programming are rather involved, and tend to be couched in terms well below the abstraction level of the strategic programmer. Hence, future type systems should provide first-class support for the required form of type case.

4.5. Declare and check reachability contracts

Advice 5. *Curb programming errors due to type-specific cases not exercised during traversal (see Section 3.6) by declaring and checking contracts regarding reachability. These contracts describe that the argument types of type-specific cases can possibly be encountered on subterm positions of the traversal's input whose root type is known. The type system shall enforce these contracts for composed traversals.*

We can illustrate the advice in Haskell in a specific manner by making use again of advanced type-class-based programming. That is, we leverage a type-level relation on types, which captures whether or not terms of one type may occur within terms of another type. The relation is modeled by the following type class:

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

The above instance makes sure that relation *ReachableFrom* is reflexive. All remaining instances must be derived from the declarations of data types that are term types. The instances essentially rephrase the constructor components of the datatype declarations. 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).
```

Recall the example of dead code due to unreachable types in Section 3.6. The relation *ReachableFrom* clearly demonstrates that numbers can be reached from a tree of numbers (using the second instance) but not from a tree of Booleans.

The relation *ReachableFrom* can be leveraged directly for the declaration of contracts regarding reachability. To this end, appropriate constraints must be added to the function signatures of the traversal schemes. For simplicity, we focus here on the simpler function signatures for the traversal schemes of Section 4.1 with hard-wired defaults. Thus:

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

Compared to *full_td'* of Section 4.1, the *ReachableFrom* constraint has been added. To this end, the application of the forall-quantified type synonym for the resulting strategy had to be inlined so that the constraint can address the type variable *y* for the strategy type. Of course, the 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. Here, we consider the deep identity traversal or the completely undefined traversal as 'useless'.

While the explicit declaration of contracts provides valuable documentation, it is possible, in principle, to infer reachability constraints from the traversal programs themselves. This may be difficult with type-class-based programming, but we consider a corresponding static program analysis in Section 5.2.

The idea underlying this illustration has also been presented in [37] in the context of applying 'Scrap Your Boilerplate' to XML programming. As we noted before in Section 4.4, the use of type-class-based programming involves substantial encoding, and hence, future type systems should provide more direct means for expressing reachability contracts.

5. Static analysis of traversal strategies

We go beyond the limitations of established means of static typing by proposing designated static analyses to curb the identified programming errors. In this manner, we can address some problems more thoroughly than with established means of static typing. For instance, we can infer contracts regarding fallibility and reachability—as opposed to checking them previously. Also, the encoding burden of the previous section would be eliminated by a practical type system which includes the proposed analyses. Further, we can address additional problems that were out of reach so far. In particular, we can perform designated forms of termination analysis to rule out divergent traversal.

The section is organized as a sequel of contributions—each of them consisting of a piece of language-agnostic advice for improving strategic programming and an associated static analysis to support the advice.

We use abstract interpretation and special-purpose type systems for the specification and implementation of the analyses. We have included a representative example of a soundness proof. In all cases, we have modeled the analyses algorithmically in Haskell. All but some routine parts of the analyses are included into the text; a cursory understanding of the analyses does not require Haskell proficiency.

5.1. Perform fallibility analysis

Advice 6. *Curb programming errors due to subtle control flow (see Section 3.5) by statically analyzing strategy properties regarding fallibility, i.e., success and failure behavior. Favorable properties may be stated as contracts in programs which are verified by the proposed analysis.*

Without loss of generality, we focus here on an analysis that determines whether a strategy can be guaranteed to succeed (read as 'always succeeds' or 'infallible'). Similar analyses are conceivable for cases such as 'always fails', 'sometimes succeeds', and others .

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 apply a special-purpose type system; 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.

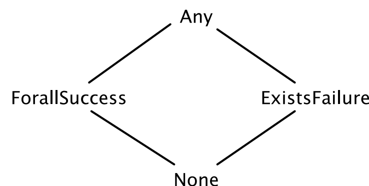
Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

```
-- 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 and 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. 21. The abstract domain for success and failure behavior.

5.1.1. An abstract interpretation-based approach

We use the following lattice for the abstract domain for a simple success and failure analysis.[11]



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

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

In the former case, we also speak of an *infallible* strategy according to Section 2.3. Note that this is a *partial correctness* analysis. Hence, in none of the cases is it implied that the program terminates for all arguments. 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. 21 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. 22. It re-interprets the abstract syntax of Section 2.4 to perform abstract interpretation on the abstract domain as opposed to regular interpretation. We discuss the analysis case by case now.

[11] We use the general framework of abstract interpretation by Cousot and Cousot [13,12]; we are specifically guided by Nielson and Nielson's style as used in their textbooks [54,55].

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS


```
-- The actual analysis
analyse :: T 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. 22. Abstract interpretation for analyzing the success and failure behavior of traversal programs.

	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_td s	*ForallSuccess*	*None*	*None*
innermost s	*ForallSuccess*	*ForallSuccess*	*ForallSuccess*

Fig. 23. Exercising success and failure analysis on traversal schemes.

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 as opposed to a many-sorted 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. 23 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

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS

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

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

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

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

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

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

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

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

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

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

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

Fig. 24. Typing rules for success and failure behavior

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, which means that the analysis concluded with an imprecise result: we do not get to know anything of value about 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 (see Section 2.3), but the analysis reports *None*.

One may want to improve the abstract interpretation-based approach so that it computes more useful results. Here we note that the informal arguments in support of fallibility and infallibility of traversal schemes typically rely on induction over all possible terms. It is not straightforward to adjust abstract interpretation in a way to account for induction.

We abandon abstract interpretation for now. It turns out that a type-system-based approach provides useful results rather easily because it has fundamentally different characteristics of dealing with recursion. In Section 5.2, we will revisit abstract interpretation and apply it successfully to a reachability analysis for type-specific cases.

5.1.2. A type system-based approach

Let us use deduction based on a special-purpose type system to infer when a strategy 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. 24 describe a typing judgment such that $\Gamma \vdash s : \mathsf{True}$ is intended to capture the judgment that the strategy s does not fail for any argument t, in the context Γ. That is, there does not exist any t such that $s\,@\,t \rightsquigarrow \uparrow$, according to the semantics in Figs. 4 and 5.

Fig. 25 rephrases Fig. 24 in a directly algorithmic manner in Haskell—also providing type inference. The context Γ, which carries fallibility information about free variables in a term, is needed so that the analysis can deal with recursively-defined strategies.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

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

-- Type inference
typeOf :: T 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
rec :: (Type -> T Type) -> Type -> Maybe Type
rec f t = typeOf (f t) >>= \t' ->
            if t==t' then Just t else Nothing
```

Fig. 25. Type inference for success and failure behavior

	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_td s	True	True
innermost s	True	True

Fig. 26. Exercising success and failure types on traversal schemes.

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 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*. 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. 24 (and the equations of Fig. 25) are very similar to the equations of Fig. 22. For instance, the rules for base cases id^{SF} and $fail^{SF}$ 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 ($sequ.1^{SF}$ to $sequ.3^{SF}$). If either component of a choice is infallible, the choice is too ($choice.1^{SF}$, $choice.3^{SF}$). 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. 26 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. (We use the context parameter of the type system, or, in fact, the *Var* form of strategy terms, to capture and propagate such assumptions; see the paper's online code distribution.) When compared to Fig. 23, 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. Also, the scheme for stop-top-down traversal is found to universally succeed, no matter what the argument strategy. The abstract interpretation-based approach could not make a useful prediction for these cases.

5.1.3. Simple dead-code detection

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 that involve dead code in the sense of Section 3.6. Specifically, we could remove the following rule:

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

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

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 the composition is equivalent to s_1 with s_2 being dead code.

5.1.4. Soundness of the type system

We prove soundness of the type system in Fig. 24 relative to the established, natural semantics in Figs. 4 and 5.

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

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 *id* then there is no evaluation rule deriving $id @ t \rightsquigarrow \uparrow$ for any t, contradicting the hypothesis.

Failure If s is *fail* then there is no typing rule deriving $\vdash fail : \mathsf{True}$, 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 $sequ.1^{\mathsf{SF}}$ 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 $seq^-.1$ or from $s_2 @ t \rightsquigarrow \uparrow$ by rule $seq^-.2$.

We choose i such that $s_i @ t \rightsquigarrow \uparrow$; the corresponding derivation is shorter than $s @ t \rightsquigarrow \uparrow$, a contradiction to the minimality of the derivation for s.

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

Now, by hypothesis we also have a term t so that $s_1 \nleftarrow s_2 @ t \rightsquigarrow \uparrow$: 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 $\Box(s')$ then $\vdash \Box(s') : \mathsf{True}$ 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 rec^{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, $s'[v \mapsto \mu v.s']$ 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.

5.2. Perform reachability analysis

Advice 7. *Curb programming errors due to type-specific cases not exercised during traversal (see Section 3.6) by statically analyzing reachability of the cases within strategies that are applied to a term of a statically known type. Such dead code detection does not require programmer-provided reachability contracts; instead it is a general analysis of strategy applications.*

For instance, using again the introductory company example, we would like to obtain the kind of information in Fig. 27 by a reachability analysis. In this example, we assume one type-specific case, *incSalary*, which is used in the traversal program subject to the analysis. 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* (see the first line of 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 though apply *All (try incSalary)* to an employee (fourth line) because a salary may be encountered as an immediate subterm of an employee. The last two lines in the table illustrate the reachability behavior of a traversal scheme, in comparison to a one-layer traversal.

The abstract interpretation relies on many-sorted signatures for the terms to be traversed, as shown in Fig. 28; we omit the straightforward definition of various constructors and observers. In the code for the abstract interpretation for reachability, we will only use the observer *sorts* of type *Signature −> Set Sort*, to retrieve all possible sorts of a signature, and the observer *argSortsOfSort*, to retrieve all sorts of immediate subterms for all possible terms of a given sort.

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

Fig. 27. Exercising the reachability analysis for companies.

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

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

Fig. 28. An abstract data type for signatures.

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.
```

For each case, we need to capture its 'sort'. Only when a type-specific case is applied to a term of the designated sort, then the case should be counted as being exercised. More generally: *the abstract interpretation computes what type-specific cases are exercised by a given strategy when faced with terms of different sorts*. To this end, we use the following abstract domain:

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

Here, *Map* is a Haskell type for finite maps: sorts are associated with type-specific cases. The analysis associates each strategy with such a map—as evident from the following function signature:

```
analyse :: Signature -> T Abs -> Abs.
```

That is, for each *Sort* in the given signature the analysis is supposed to return a set of (named) type-specific *cases* which *may* be executed by the traversal if the traversal is applied to a term of the sort. 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 which is represented here as a trivial map of type *Abs*. 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*. 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 returned cases are actually executed.

The analysis proceeds by induction over the structure of strategies, and is parameterized 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 [54,55].

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

ARTICLE IN PRESS


```
$analyse :: Signature -> T Abs -> Abs
analyse sig = analyse'
 where
  analyse' :: T 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. 29. Abstract interpretation for analyzing the reachability of type-specific cases relative to a given signature.

so, the recursive call of the analysis, *analyse' 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.

The analysis is conservative in so far that it 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 and their recursive application—think of the patterns in rewrite rules. As a result, certain reachability-related programming errors will go unnoticed. Consider the following example:

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

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* and it succeeds for all such terms. 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 that are only to be found below departments.

This illustration clearly suggests that a success and 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 should treat sequence differently from choice, and an 'all' traversal differently from a 'one' traversal. We omit such elaborations here.

5.3. Perform termination analysis

Advice 8. *Curb various kinds of programming errors that may manifest themselves as divergent traversals. These includes wrong decisions regarding the traversal scheme and misunderstood properties of rewrite rules. To this end, perform a static termination analysis that leverages appropriate measures in proving termination of recursive traversals.*

That is, 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.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

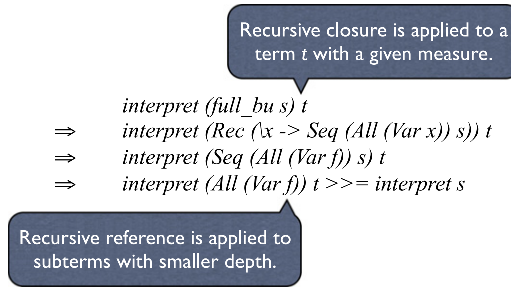


Fig. 30. Illustration of termination checking.

Based on experiences with modeling strategies in Isabelle/HOL [30], our analysis for termination checking essentially leverages an induction principle; see Fig. 30 for an illustration. That is, the analysis is 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. However, many traversals in strategic programming cannot be proven to converge just on the grounds of depth. For instance, a program transformation for macro expansion could be reasonably implemented with a full top-down traversal, but such a traversal clearly increases term depth. Accordingly, we will generalize the analysis to deal with measures other than depth.

5.3.1. 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. 30, 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 '≤' (or '=' if the abstract domain provided this option) as we enter the recursive closure, and it is accordingly updated as we symbolically interpret the body of the closure. Ultimately, we are interested in the relation at the point of recursive reference. These are the values of *Rel*; see Fig. 31 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 during 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 whether the depth was preserved, increased, or decreased.

So far we have emphasized a depth relation 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 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 :: T SRel -> Abs .
```

In fact, we would like to use variables of the *T* 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. Thus:

```
—— 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. 31. Relation on measures such as depth of terms.

```
analyse :: T (SRel,Bool) —> Abs .
```

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

```
typeOf :: T (SRel,Bool) —> Maybe SRel
typeOf s = analyse s Leq .
```

5.3.2. Termination analysis with the depth measure

The analysis is defined in Fig. 32. 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* decreases depth in a vacuous sense. The case for *Seq* sequentially composes 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* checks whether we face a recursive reference (i.e., *b == True*) because in this case we must insist on the current *TRel* value to be *Less*. If this precondition holds, then the strategy property for the variable is combined with the current term property using the *plus* operation (say, addition) for type *Rel*.

The case for *Rec* essentially resets the *TRel* value to *Leq*; see *analyse ... Leq*, and attempts the analysis of the body for all possible assumptions about the effect of the recursive references; see *[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; see '<='. The ordering on the attempts implies that the least constraining type is inferred (i.e., the largest value of *SRel*).

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

```
analyse :: T (SRel, Bool) —> Abs
analyse Id = Just
analyse Fail = const (Just Less)
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. 32. A static analysis for termination checking relative to the depth measure for terms.

	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. 33. Exercising the termination analysis on 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 operations increase and decrease are highly constrained. In particular, once the *TRel* value has reached *Any*, decrease does not get us back onto a termination-proven path.

The analysis is able to infer termination types for a range of interesting traversal scenarios; see Fig. 33. 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; see 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 is useful once we take into account other measures.

At this point, we are not yet able 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. Compound measures, as discussed below, come to the rescue.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

	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. 34. Exercising compound measures.

5.3.3. Termination analysis with compound measures

The static analysis can be elaborated to deal with measures other than depth. In this paper, we demonstrate a measure of *the number of occurrences of a specific constructor* in a term, i.e., the constructor count. This sort measure is applicable to transformation scenarios where a specific constructor is systematically eliminated, e.g., in the sense of macro expansion.

This approach could also be generalized to deal with a measure for the *number of matches for a specific pattern* in a term, such as the LHS of a rewrite rule. This elaboration is not discussed any further though.

The general idea is to associate strategy expressions and argument strategies of schemes in a traversal program with a suitable measure. In this paper, we require that the programmer provides the measure, but ultimately such measures may be inferred. We need a new type, *Measure*, to represent compound measure in a list-like structure. 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 type of measure transformation and the signature of the program analysis must be changed such that we use non-empty lists of values of type *TRel* as opposed to singletons before, thereby accounting for compound measures. Thus:

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

The initial list of type *[TRel]* is trivially obtained from (the length of) the measure by a function like this:

```
leqs Depth = [Leq]
leqs (Count _ m) = Leq : leqs m .
```

Thus, the analysis computes the effects of the strategy while assuming that the declared measure holds for the argument. At this level of development, the analysis is oblivious to the actual constructor names because *T* does not involve any expressiveness for dealing with specific constructors. Measure claims are to be verified for given rewrite rules that serve as arguments, but this part is omitted here.

The power of such termination types is illustrated in Fig. 34. 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 to the paper's online code distribution for details.

6. Related work

We will focus here on related work that deals with or is directly applicable to programming errors in traversal programming with traversal strategies or otherwise.

Simplified traversal programming In an effort to manage the relative complexity of traversal strategies or the generic functions of "Scrap Your Boilerplate", simplified forms of traversal programming have been proposed. The functional programming-based work of [52] describes less generic traversal schemes (akin to those of Section 4.1), and thereby suffices with simpler types, and may achieve efficiency of traversal implementation more easily. The rewriting-based work of [67] 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.

Implicit strategy extension It is fair to say that several challenges relate to strategy extension. When traversal schemes are not parameterized with generic functions, as mentioned above, then strategy extension is no longer needed by the programmer, but expressiveness will be limited. There exists a proposal [17] for a language design that essentially makes strategy extension implicit in an otherwise typeful setting. This approach may imply a slightly simpler programming model, but it is not obvious that it necessarily reduces programming errors. This question remains open, but we refer to [35] for a (dated) discussion of the matter.

ARTICLE IN PRESS

Runtime checks With respect to our various efforts to declare, infer, and enforce fallibility properties it is useful to note that Stratego supports the *with* operator as alternative to the *where* clause for conditional rules (and as strategy combinator as well), which indicates that its argument should be a transformation that always succeeds. When it does not succeed, a run-time exception is raised. It is reported, anecdotally, for example, by a reviewer of this paper, that this feature has been helpful for detection of programming errors. Such annotations can be useful for expressing the intent of programmers and may be useful both for runtime checking and as input to static analyses in future systems.

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 [56,47,42]. 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 Section 4.5 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 [73], XSLT [75], and XQuery [74]. Some limited cross-paradigmatic comparison of traversal strategies and XML programming has been presented in [36,15]. 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 [74] 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 an additional challenge due to its template mechanism. Default templates, in particular, provide a kind of traversal capability. Let us mention related work on analyzing XML programs. [18] 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 [19] to perform static analysis of XML paths and types [21], and a dead code elimination for XQuery programs [20].

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 [70,41,72,67,29,38,35,58,15,30]. In [15], 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. We contend that better understanding of properties of traversal strategies, just by itself, curbs programming errors, but 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.

Termination analysis Our termination analysis is arguably naive in that it focuses on recursion patterns of traversals. A practical system for a full-fledged strategic programming language would definitely need to include existing techniques for termination analysis, as they are established in the programming languages and rewriting communities. We mention some recent work in the adjacency of (functional) traversal strategies. [23,66] address termination analysis for rewriting with strategies (but without covering programmable traversal strategies). [62,22] address termination analysis for higher-order functional programs; it may be possible to extend 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 [25]; 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.

7. Concluding remarks

The ultimate motivation for the work presented here is to make traversal programming with strategies easier and safer. To this end, strategy libraries and the underlying programming languages need to improve so that contracts of traversal strategies are accurately captured and statically verified. That is, we envisage that "design by contract" is profoundly instantiated for traversal programming with strategies. These contracts would deal, for example, with (in)fallibility or measures for termination.

Throughout the paper, we have revealed pitfalls of strategic programming and discovered related properties of basic strategy combinators and common library combinators. To respond to these, we have developed concrete advice on suggested improvements to strategy libraries and the underlying programming languages:

- Hard-wire defaults into traversal schemes. (Section 4.1)
- Declare and check fallibility contracts. (Section 4.2)
- Reserve fallibility for modeling control flow. (Section 4.3)
- Enable families of type-specific cases. (Section 4.4)

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

- Declare and check reachability contracts. (Section 4.5)
- Perform fallibility analysis. (Section 5.1)
- Perform reachability analysis. (Section 5.2)
- Perform termination analysis. (Section 5.3)

Such advice comes without any claim of completeness. Also, such advice must not be confused with a proper design for the ultimate library and language.

Arguably, some improvements can be achieved by revising strategy libraries so that available static typing techniques are leveraged. We demonstrated this path with several Haskell-based experiments. This path is limited in several respects. First, only part of the advice can be addressed in this manner. Second, the typing techniques may not be generally available for languages used in traversal programming. Third, substantial encoding effort is needed in several cases.

Hence, our work suggests that type systems of programming languages need to become more expressive so that all facets of traversal contracts can be captured at an appropriate level of abstraction and statically verified in a manner that also accounts for language usability. We assert that, in fact, strategic programming is in need of a form of dependent types, an extensible type system, or, indeed, an extensible language framework that admits pluggable static analysis.

The development of the paper is based on a series of programming errors as they arise from a systematic discussion of the process of design and implementation of strategic programs. Empirical research would be needed to confirm the relevance of the alleged pitfalls. However, based on the authors' experience with strategic programming in research, education, and software development, the authors can confirm that there is anecdotal evidence for the existence of the presented problems.

In this paper, we have largely ignored another challenge for strategic programming, namely *performance*. In fact, disappointing performance may count as another kind of programming error. Better formal and pragmatic understanding of traversal programming is needed to execute traversal strategies more efficiently. Hence, performance is suggested as a major theme for future work. There is relevant, previous work on fusion-like techniques for traversal strategies [29], calculational techniques for the transformation of traversal strategies [15], and complementary ideas from the field of adaptive programming [47].

Acknowledgements

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. Thanks are due to the reviewers of the initial journal submission who provided substantial advice. Part of the material has been used in Ralf Lämmel's invited talk at LOPSTR/PPDP 2009, and all feedback is gratefully acknowledged. Much of the presented insights draw from past collaboration and discussions with Simon Peyton Jones, Karl Lieberherr, Claus Reinke, Eelco Visser, Joost Visser, and Victor Winter.

References

[1] A. Abdelmeged, K.J. Lieberherr, Recursive adaptive computations using per object visitors, in: Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, ACM, 2007, pp. 825–826.
[2] A. Abel, Type-based termination of generic programs, Science of Computer Programming 74 (8) (2009) 550–567.
[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: piggybacking rewriting on java, in: Term Rewriting and Applications, 18th International Conference, RTA 2007, Proceedings, in: LNCS, vol. 4533, Springer, 2007, pp. 36–47.
[4] E. Balland, P.-E. Moreau, A. Reilles, Rewriting strategies in java, ENTCS 219 (2008) 97–111.
[5] G. Bierman, E. Meijer, W. Schulte, The essence of data access in Cω, in: ECOOP'05, Object-Oriented Programming, 19th European Conference, Proceedings, in: LNCS, vol. 3586, Springer, 2005, pp. 287–311.
[6] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, An overview of ELAN, in: C. Kirchner, H. Kirchner (Eds.), Proceedings of the International Workshop on Rewriting Logic and its Applications, WRLA'98, in: ENTCS, vol. 15, Elsevier Science, 1998.
[7] P. Borovanský, C. Kirchner, H. Kirchner, C. Ringeissen, Rewriting with strategies in ELAN: a functional semantics, International Journal of Foundations of Computer Science (2001).
[8] A. Bove, P. Dybjer, U. Norell, A brief overview of agda — a functional language with dependent types, in: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs '09, in: LNCS, vol. 5674, Springer, 2009, pp. 73–78.
[9] M. Brand, M. Sellink, C. Verhoef, Generation of components for software renovation factories from context-free grammars, in: I. Baxter, A. Quilici, and C. Verhoef (Eds.), Proceedings Fourth Working Conference on Reverse Engineering, 1997, pp. 144–153.
[10] M. Bravenboer, K.T. Kalleberg, R. Vermaas, 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, ACM, 2006, pp. 95–99.
[11] J.R. Cordy, The TXL source transformation language, Science of Computer Programming 61 (3) (2006) 190–210.
[12] P. Cousot, Abstract Interpretation, ACM Computing Surveys 28 (2) (1996) 324–328.
[13] P. Cousot, R. Cousot, Basic concepts of abstract interpretation, in: Building the Information Society, IFIP 18th World Computer Congress, 2004, Topical Sessions, Proceedings, Kluwer, 2004, pp. 359–366.
[14] K. Crary, S. Weirich, Flexible type analysis, in: ICFP '99: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, ACM, 1999, pp. 233–248.
[15] A. Cunha, 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, ACM, 2007, pp. 11–20.
[16] B.C.d.S. Oliveira, A. Moors, M. Odersky, Type classes as objects and implicits, in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, ACM, 2010, pp. 341–360.
[17] E. Dolstra, E. Visser, First-class rules and generic traversal, Technical Report UU-CS-2001-38, Institute of Information and Computing Sciences, Utrecht University, 2001.
[18] C. Dong, J. Bailey, Static analysis of XSLT programs, in: K.-D. Schewe, H. Williams (Eds.), Fifteenth Australasian Database Conference (ADC2004), Conferences in Research and Practice in Information Technology, Australian Computer Society, Inc, 2004.

[19] P. Genevès, Logics for XML, PhD thesis, Institut National Polytechnique de Grenoble, 2006.
[20] P. Genevès, N. Layaïda, Eliminating dead-code from XQuery programs, in: ICSE'10, Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering, ACM, 2010.
[21] P. Genevès, N. Layaïda, 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, ACM, 2007, pp. 342–351.
[22] J. Giesl, S. Swiderski, P. Schneider-Kamp, R. Thiemann, Automated termination analysis for haskell: from term rewriting to programming languages, in: Term Rewriting and Applications, 17th International Conference, RTA 2006, Proceedings, in: LNCS, vol. 4098, Springer, 2006, pp. 297–312.
[23] I. Gnaedig, H. Kirchner, Termination of rewriting under strategies, ACM Transactions on Computational Logic 10 (2) (2009).
[24] R. Hinze, A new approach to generic functional programming, in: T. Reps (Eds.), Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19–21, 2000, pp. 119–132.
[25] 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, ACM, 2000, pp. 119–132.
[26] R. Hinze, A. Löh, "Scrap Your Boilerplate" revolutions, in: Proceedings, Mathematics of Program Construction, 8th International Conference, MPC 2006, in: LNCS, vol. 4014, Springer, 2006, pp. 180–208.
[27] R. Hinze, A. Löh, B.C.D.S. Oliveira, "Scrap Your Boilerplate" reloaded, in: FLOPS'06: Proceedings of Functional and Logic Programming, 8th International Symposium, in: LNCS, vol. 3945, Springer, 2006, pp. 13–29.
[28] P. Jansson, J. Jeuring, PolyP - a polytypic programming language extension, in: POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1997, pp. 470–482.
[29] P. Johann, 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, 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, ACM, 2009, pp. 93–104.
[31] L.C.L. Kats, A.M. Sloane, E. Visser, Decorated attribute grammars: attribute evaluation meets strategic programming, in: Compiler Construction, 18th International Conference, CC 2009, Proceedings, in: LNCS, vol. 5501, Springer, 2009, pp. 142–157.
[32] O. Kiselyov, R. Lämmel, K. Schupke, Strongly typed heterogeneous collections, in: Haskell'04: Proceedings of the ACM SIGPLAN Workshop on Haskell, ACM, 2004, pp. 96–107.
[33] R. Lämmel, The sketch of a polymorphic symphony, in: WRS'02: Proceedings of International Workshop on Reduction Strategies in Rewriting and Programming, in: ENTCS, vol. 70, Elsevier Science, 2002, p. 21.
[34] R. Lämmel, Towards generic refactoring, in: Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, ACM, 2002, pp. 15–28.
[35] R. Lämmel, Typed generic traversal with term rewriting strategies, Journal Logic and Algebraic Programming 54 (1–2) (2003) 1–64.
[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, ACM, 2007, pp. 137–142.
[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, ACM, 2007, pp. 137–142.
[38] R. Lämmel, 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, ACM, 2003, pp. 26–37.
[39] R. Lämmel, 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, ACM, 2004, pp. 244–255.
[40] R. Lämmel, 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, ACM, 2005, pp. 204–215.
[41] R. Lämmel, E. Visser, J. Visser, The essence of strategic programming – an inquiry into trans-paradigmatic genericity. Draft, 2002–2003.
[42] R. Lämmel, E. Visser, J. Visser, Strategic programming meets adaptive programming, in: AOSD'03: Conference Proceedings of Aspect-Oriented Software Development, ACM, 2003, pp. 168–177.
[43] R. Lämmel, J. Visser, Typed combinators for generic traversal, in: PADL'02: Proceedings of Practical Aspects of Declarative Programming, in: LNCS, vol. 2257, Springer, 2002, pp. 137–154.
[44] R. Lämmel, J. Visser, A Strafunski application letter, in: PADL'03: Proceedings of Practical Aspects of Declarative Programming, in: LNCS, vol. 2562, Springer, 2003, pp. 357–375.
[45] D. Leijen, HMF: simple type inference for first-class polymorphism, in: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, ACM, 2008, pp. 283–294.
[46] H. Li, S. Thompson, C. Reinke, The Haskell Refactorer, HaRe, and its API, ENTCS 141 (4) (2005) 29–34.
[47] K.J. Lieberherr, B. Patt-Shamir, D. Orleans, Traversals of object structures: specification and efficient implementation, ACM Transactions on Programming Languages and Systems 26 (2) (2004) 370–412.
[48] B. Luttik, E. Visser, Specification of rewriting strategies, in: M.P.A. Sellink (Ed.), 2nd International Workshop on the Theory and Practice of Algebraic Specifications, ASF+SDF'97, Springer, 1997, Electronic Workshops in Computing.
[49] S. Marlow, An extensible dynamically-typed hierarchy of exceptions, in: Haskell'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell, ACM, 2006, pp. 96–106.
[50] C. McBride, Faking it: simulating dependent types in Haskell, Journal of Functional Programming 12 (4–5) (2002) 375–392.
[51] C. McBride, Epigram: practical programming with dependent types, in: Advanced Functional Programming, 5th International School, AFP 2004, Revised Lectures, in: LNCS, vol. 3622, Springer, 2005, pp. 130–170.
[52] N. Mitchell, C. Runciman, Uniform boilerplate and list processing, in: Haskell'07: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop, ACM, 2007, pp. 49–60.
[53] G. Munkby, A.P. Priesnitz, S. Schupp, M. Zalewski, Scrap++: scrap your boilerplate in C++, in: Proceedings of the ACM SIGPLAN Workshop on Genetic Programming, WGP 2006, ACM, 2006, pp. 66–75.
[54] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 2005.
[55] H.R. Nielson, F. Nielson, Semantics with Applications (An Appetizer), Springer, 2007.
[56] J. Palsberg, B. Patt-Shamir, K.J. Lieberherr, A new approach to compiling adaptive programs, Science of Computer Programming 29 (3) (1997) 303–326.
[57] S.L. Peyton Jones, D. Vytiniotis, S. Weirich, M. Shields, Practical type inference for arbitrary-rank types, J. Funct. Program. 17 (1) (2007) 1–82.
[58] F. Reig, Generic proofs for combinator-based generic programs, in: Trends in Functional Programming, 2004, pp. 17–32.
[59] D. Ren, 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, ACM, 2006, pp. 13–24.
[60] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, B.C.d.S. Oliveira, Comparing libraries for generic programming in Haskell, in: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, ACM, 2008, pp. 111–122.
[61] B.G. Ryder, 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) (2003) 16–22.
[62] D. Sereni, N.D. Jones, Termination analysis of higher-order functional programs, in: Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Proceedings, in: LNCS, vol. 3780, Springer, 2005, pp. 281–297.
[63] C.-c. Shan, Sexy types in action, SIGPLAN Notices 39 (2004) 15–22.
[64] M. Shields, E. Meijer, Type-indexed rows, in: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'01, ACM, 2001, pp. 261–275.
[65] P. Thiemann, Programmable type systems for domain specific languages, ENTCS 76 (2002) 233–251.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006

ARTICLE IN PRESS

[66] R. Thiemann, C. Sternagel, Loops under strategies, in: Rewriting Techniques and Applications, 20th International Conference, Proceedings, in: LNCS, vol. 5595, Springer, 2009, pp. 17–31.
[67] M. van den Brand, P. Klint, J.J. Vinju, Term rewriting with traversal functions, ACM Transactions Software Engineering Methodology 12 (2) (2003) 152–190.
[68] E. Visser, Language independent traversals for program transformation, in: Proceedings of WGP'2000, Technical Report, Universiteit Utrecht, 2000, pp. 86–104.
[69] 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, in: LNCS, vol. 3016, Springer, 2004, pp. 216–238.
[70] E. Visser, Z. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in: ICFP'98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ACM, 1998, pp. 13–26.
[71] E. Visser, Z.-e.-A. Benaissa, A core language for rewriting, in: Second International Workshop on Rewriting Logic and its Applications, WRLA 1998, in: ENTCS, vol. 15, Elsevier Science, 1998.
[72] J. Visser, Generic traversal over typed source code representations, PhD thesis, University of Amsterdam, 2003.
[73] W3C. XML Path Language (XPath) Version 1.0. Available online at http://www.w3.org/TR/xpath/, W3C Recommendation 16 November 1999.
[74] W3C. XQuery 1.0: An XML Query Language. Available online at http://www.w3.org/TR/xquery/, W3C Recommendation 23 January 2007.
[75] W3C. XSL Transformations (XSLT) Version 2.0. Available online at http://www.w3.org/TR/xslt20/, W3C Recommendation 23 January 2007.
[76] V.L. Winter, Strategy construction in the higher-order framework of TL, ENTCS 124 (1) (2005) 149–170.
[77] V.L. Winter, J. Beranek, Program transformation using HATS 1.84, in: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Revised Papers, in: LNCS, vol. 4143, Springer, 2006, pp. 378–396.
[78] V.L. Winter, J. Beranek, F. Fraij, S. Roach, G.L. Wickstrom, A transformational perspective into the core of an abstract class loader for the SSP, ACM Trans. Embedded Comput. Syst. 5 (4) (2006) 773–818.
[79] V.L. Winter, M. Subramaniam, The transient combinator, higher-order strategies, and the distributed data problem, Science of Computer Programming 52 (2004) 165–212.
[80] D.N. Xu, S.L. Peyton Jones, K. Claessen, Static contract checking for Haskell, in: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, ACM, 2009, pp. 41–52.

Please cite this article in press as: R. Lämmel, et al., Programming errors in traversal programs over structured data, Science of Computer Programming (2012), doi:10.1016/j.scico.2011.11.006