

An Isabelle/HOL-based Model of Stratego-like Traversal Strategies

Markus Kaiser and Ralf Lämmel

Software Languages Team, Universität Koblenz-Landau

Abstract

Traversal strategies are at the heart of transformational programming with rewriting-based frameworks such as Stratego/XT or Tom and specific approaches for generic functional programming such as Strafunski or “Scrap your boilerplate”. Such traversal strategies are distinctively based on one-layer traversal primitives from which traversal schemes are derived by recursive closure.

We describe a mechanized, formal model of such strategies. The model covers two different semantics of strategies, strategic programming laws, termination conditions for strategy combinators as well as properties related to the success/failure behavior of strategies. The model has been mechanized in Isabelle/HOL.

Categories and Subject Descriptors D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features; D.1.1 [PROGRAMMING TECHNIQUES]: Applicative (Functional) Programming; D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification

General Terms Design, Languages, Theory, Verification

Keywords Stratego, Traversal Strategies, Software Transformation, Domain Specific Languages, Rewriting, Generic Functional Programming, Isabelle/HOL

1. Introduction

Traversals are becoming an established programming idiom in programming domains that involve large, heterogeneous, structured data. For instance, the implementation of software transformations for general-purpose or domain-specific languages routinely involves traversals over data based on abstract or concrete syntaxes as well as intermediate or exchange formats.

In the declarative and transformational programming communities, a well-known approach to traversal programming is based on *traversal strategies* as they were pioneered by work on the rewriting-based Stratego/XT framework [36, 34, 6]. There are also other rewriting-based incarnations of traversal strategies, e.g., [33, 4]. Stratego inspired strategic programming approaches for other programming paradigms [21, 37], which in turn inspired the “Scrap your boilerplate” form of generic functional programming [18, 12]. In this paper, strategic functional programming with “Strafun-

ski” [21] will play a certain role because Strafunski’s implementation provides some guidance for the derivation of the formal model.

Stratego-like traversal strategies are distinctively based on *one-layer traversal primitives* (also known as “all”, “one”, etc.) from which *traversal schemes* (such as “full top-down” or “once bottom-up”) are derived by recursive closure. Rewrite rules (or strategies) are passed as arguments to the schemes and applied to all nodes that are encountered along the traversal. Success/failure behavior may form an important element of traversal control.

Expressiveness and abstraction level provided by traversal strategies come with a cost: *traversal programs can go wrong in new ways*. For instance, a traversal may suddenly fail or diverge; it may also be “surprisingly” inefficient. In a recent paper [19], we illustrated options for traversal programs going wrong, and we indicated opportunities for improving the strategic programming notion. We also concluded that, above all, it is important to gain better understanding of the *properties of traversal programs* so that programmers make more knowledgeable choices of traversal schemes, and programming environments may provide improved documentation, targeted checks, and other support.

The present paper makes a contribution towards the goal of a matured strategic programming notion by means of a mechanized, formal model for traversal strategies which is shown to be useful in studying laws and properties of traversal strategies. While there exists work on operational semantics (SOS) and type systems for Stratego-like traversal strategies [36, 17], there is no mechanized, formal model of such strategies and their laws and properties that predates our work.

Contribution This paper delivers the first machine-checked, formal model of Stratego-like traversal strategies. The core of the model adopts Strafunski’s functional programming-based implementation of traversal strategies [21], but this core is then shown to match the existing SOS-style semantics of Stratego’s strategy primitives [36].

The model systematically collects and proves strategic programming laws, termination conditions for strategy combinators as well as properties related to the success/failure behavior of strategies. The formal model has been mechanized in Isabelle/HOL [30, 26]. Many of these laws, conditions and properties are properly formalized for the first time.

Limitations The formal model (the semantics in particular) does not include a recursion combinator or an abstraction form (for general recursive strategies), but we rely on Isabelle/HOL’s forms of recursion and abstraction instead. The formal model does not define a strategy primitive for rewrite rules, but we assume them to be encoded as Isabelle/HOL functions.

Technically, the biggest challenge is to agree on an appropriate model of strategies that are not total, e.g., traversal schemes that may diverge—while Isabelle/HOL’s functions are total by defini-

$s ::=$	$t \rightarrow t$	(Rewrite rules as basic building blocks.)
	ϵ	(Identity strategy; succeeds and returns input.)
	δ	(Failure strategy; fails and returns failure “↑”.)
	$s; s$	(Left-to-right sequential composition.)
	$s \dot{+} s$	(Left-biased choice; try first argument first.)
	$\square(s)$	(Transform term by applying s to all children.)
	$\diamond(s)$	(Transform term by applying s to one child.)

Figure 1. Syntax of strategy primitives

tion. To this end, we model the non-diverging part of such a strategy combinator as a relation (between arguments and results) from which we derive a function which is left unspecified for the rest.

Road-map The remainder of this paper is organized as follows:

- Sec. 2 provides background on strategic programming.
- Sec. 3 develops a basic model of strategy primitives.
- Sec. 4 derives the familiar SOS-style semantics formally.
- Sec. 5 collects strategic programming laws (as lemmas again).
- Sec. 6 describes our treatment of partial (diverging) strategies.
- Sec. 7 develops a basic model of typical traversal schemes.
- Sec. 8 investigates the success/failure behavior of strategies.
- Sec. 9 discusses related work.
- Sec. 10 concludes the paper.

Prerequisites for the reader Prior knowledge of traversal strategies á la Stratego or Strafunski is helpful but not required (due to the next section).

While the formal model is based on Isabelle/HOL, we do not assume, in any way, that the reader is familiar with Isabelle/HOL. We only assume basic knowledge of functional programming, rewriting, predicate logic (of higher order), proofs by induction, and SOS style of operational semantics.

We emphasize that the presentation in this paper is focused on the *motivation of formal properties*. Proofs are hardly sketched, and if so, at an abstract level.¹ We simply use Isabelle/HOL as a modeling method to gain deeper understanding of traversal strategies.

A special focus lies on modeling partial strategies in a HOL framework where all functions are total.

2. Background

Strategic programming [36, 34, 37, 20] is based on the premise that ‘traversal control’ should be in the hands of the programmer; it should be *programmable* based on designated abstractions and language constructs — as opposed to encodings that require heavy boilerplate code. In particular, traversals are typically composed from reusable traversal schemes and problem-specific argument strategies. At a lower level, the traversal schemes are composed themselves from strategy primitives.

2.1 An illustrative scenario

The following XML processing problem illustrates the gist of traversal strategies. Let c be a term that represents the organizational structure of a company with *hierarchical* (say, nested) departments. As a reply to recession, all top-level managers decide to cut back their salary to 1\$ for some time.

We need a few building blocks: d is a test which succeeds for a (sub)term that represents a department and it fails for all other terms; likewise, m is a test for managers; l is a rewrite rule to replace a salary term by 1\$.

The following strategy application locates all top-level managers in c , and adjusts their salaries — without going deeper into c , i.e., without affecting salaries of subordinate managers (who did not agree on the aggressive measure for themselves):

$$\text{stoptd}(d; \text{stoptd}(m; \text{stoptd}(l)))@c$$

Read as: “Find all top-level departments in c ; below each department, find a manager; below the manager, find the salary, and adjust it.” Here, “ $;$ ” denotes sequential composition of two strategies, “ $@$ ” denotes application of a strategy to a term, and *stoptd* denotes the traversal scheme of top-down traversal (c.f., ... td) where traversal ceases (“stops”) for any given branch once the argument strategy was applied successfully.

The *stoptd* scheme can be defined in terms of strategy primitives and recursion as we will show below. While *stoptd* is appropriate for the situation at hand, other scenarios call for different schemes.

2.2 Strategy primitives

The development of the present paper is limited to type-preserving strategies (also known as transformations) as opposed to type-unifying strategies (also known as queries) [21, 17, 18]. Fig. 1 shows the primitives for treatment in this paper.

Most of these operators should be familiar because they occur in process calculi, rewriting calculi, tactic languages and that alike. The *distinctive primitives of strategic programming* are $\square(\cdot)$ and $\diamond(\cdot)$ (read as “all” and “one”). $\square(s)$ applies the argument strategy s to all direct subterms (“children”) of a given term, and then it reconstructs a term from the input term’s constructor and the intermediate results. If s fails for some child, then $\square(s)$ fails entirely. $\diamond(s)$ suffices to apply the argument strategy s to one child for which s succeeds; it reconstructs a term from the input term’s constructor, the single intermediate result, and the original children for all remaining positions. If s fails for all of the children (or if there are no children at all), then $\diamond(s)$ fails entirely.

Fig. 2 gives a big-step semantics for the primitives. We use a judgement (say, a relation) $s @ t \rightsquigarrow r$, where s is a strategy, t is a term, and r is a result (i.e., either a term or failure “↑”). Terms are of the form $c(t_1, \dots, t_n)$. For $n = 0$, a term is called a constant term. *Positive rules* reduce strategy applications to terms; *negative rules* model reductions that end in failure.

The SOS rules follow Stratego’s reference semantics [35, 36] very closely. The only noteworthy deviation is that we pick a deterministic semantics for $\diamond(s) @ t \rightsquigarrow r$ to better match the actual operational semantics of implementations including Stratego in particular. We assume a left-to-right order for finding the position of t to be affected. That is, the first position for which s succeeds is chosen.

2.3 Strategy library

Fig. 3 lists a small set of the traversal schemes and helper combinators for traversal. The combinators are defined by the notation of “general recursion”. For instance, in Strafunski, these schemes are defined just like that.² All the combinators at hand take one strategy argument and compose a new strategy from it. The idea is that a combinator for a traversal scheme completes the argument strategy into a traversal that applies the argument to some or all “nodes” of the input according to some rules.

¹The Isabelle/HOL theories from this paper are available online from the paper’s website: <http://www.uni-koblenz.de/~laemme1/isabelle2/>.

²In the figure, we use Stratego-like syntax. Our formal development in Isabelle/HOL uses curried function application — such as in *stoptd* s t .

Common traversal schemes

$topdown(s)$	$= s; \square(topdown(s))$	– Apply s in a full pass over the input in top-down order.
$bottomup(s)$	$= \square(bottomup(s)); s$	– Apply s in a full pass over the input in bottom-up order.
$oncedd(s)$	$= s + \diamond(oncedd(s))$	– Find the top-most position to apply s successfully.
$oncebu(s)$	$= \diamond(oncebu(s)) + s$	– Find the bottom-most position to apply s successfully.
$stoptd(s)$	$= s + \square(stoptd(s))$	– Attempt s in top-down manner until success.
$stopbu(s)$	$= \square(stopbu(s)) + s$	– <i>What's that? An exercise for the reader, but see the appendix.</i>
$innermost(s)$	$= repeat(oncebu(s))$	– An implementation of innermost normalization.

Common helpers

$repeat(s)$	$= try(s; repeat(s))$	– Iterate s until it fails; see <i>innermost</i> .
$try(s)$	$= s + id$	– Recover from the failure of s ; see <i>repeat</i> .

Figure 3. A small strategy library (general recursive definitions; Stratego-like syntax)

Positive rules

$\frac{\exists \theta. (\theta(t_l) = t \wedge \theta(t_r) = t')}{t_l \rightarrow t_r @ t \rightsquigarrow t'}$	[rule ⁺]
$e @ t \rightsquigarrow t$	[id ⁺]
$\frac{s_1 @ t \rightsquigarrow t' \wedge s_2 @ t' \rightsquigarrow t''}{s_1; s_2 @ t \rightsquigarrow t''}$	[sequ ⁺]
$\frac{s_1 @ t \rightsquigarrow t'}{s_1 + s_2 @ t \rightsquigarrow t'}$	[choice ⁺ .1]
$\frac{s_1 @ t \rightsquigarrow \uparrow \wedge s_2 @ t \rightsquigarrow t'}{s_1 + s_2 @ t \rightsquigarrow t'}$	[choice ⁺ .2]
$\frac{\forall i \in \{1, \dots, n\}. s @ t_i \rightsquigarrow t'_i}{\square(s) @ c(t_1, \dots, t_n) \rightsquigarrow c(t'_1, \dots, t'_n)}$	[all ⁺]
$\frac{\begin{array}{l} \exists i \in \{1, \dots, n\}. \\ s @ t_i \rightsquigarrow t'_i \\ \wedge \forall i' \in \{1, \dots, i-1\}. s @ t_{i'} \rightsquigarrow \uparrow \\ \wedge \forall i' \in \{1, \dots, i-1, i+1, \dots, n\}. t_{i'} = t'_{i'} \end{array}}{\diamond(s) @ c(t_1, \dots, t_n) \rightsquigarrow c(t'_1, \dots, t'_n)}$	[one ⁺]

Negative rules

$\frac{\nexists \theta. \theta(t_l) = t}{t_l \rightarrow t_r @ t \rightsquigarrow \uparrow}$	[rule ⁻]
$\delta @ t \rightsquigarrow \uparrow$	[fail ⁻]
$\frac{s_1 @ t \rightsquigarrow \uparrow}{s_1; s_2 @ t \rightsquigarrow \uparrow}$	[seq ⁻ .1]
$\frac{s_1 @ t \rightsquigarrow t' \wedge s_2 @ t' \rightsquigarrow \uparrow}{s_1; s_2 @ t \rightsquigarrow \uparrow}$	[seq ⁻ .2]
$\frac{s_1 @ t \rightsquigarrow \uparrow \wedge s_2 @ t \rightsquigarrow \uparrow}{s_1 + s_2 @ t \rightsquigarrow \uparrow}$	[choice ⁻]
$\frac{\exists i \in \{1, \dots, n\}. s @ t_i \rightsquigarrow \uparrow}{\square(s) @ c(t_1, \dots, t_n) \rightsquigarrow \uparrow}$	[all ⁻]
$\frac{\forall i \in \{1, \dots, n\}. s @ t_i \rightsquigarrow \uparrow}{\diamond(s) @ c(t_1, \dots, t_n) \rightsquigarrow \uparrow}$	[one ⁻]

Figure 2. Semantics of strategy primitives

3. Modeling strategy primitives

We develop a basic formal model that is inspired by Strafunski's functional programming incarnation of Stratego-like traversal strategies [21, 37]. (Here we note that Strafunski is essentially a *library* whose core provides combinators for all strategy primitives.)

3.1 The term representation

For simplicity, we use untyped terms just like in basic Stratego [36], or its underlying calculus [35]. We leave it to future work to cover many-sorted terms [17], or the kind of type-annotated terms of the universal term representation of Strafunski [21]. Hence, terms are modeled by the following recursive data type *cterm*³:

```
{* An algebraic data type *}
datatype
  cterm = C con (cterm list)
```

```
{* A type synonym *}
types
  con = nat
```

That is, a *cterm* consists of a *constructor* (in fact, a *natural number*) and a *list of cterms* — the children. We also provide functions to take terms apart:⁴

```
{* Function signatures *}
consts
  con_of   :: cterm → con
  children :: cterm → cterm list
```

```
{* Function definitions *}
primrec
```

```
  con_of (C c ts) = c
  children (C c ts) = ts
```

3.2 The strategy type

A strategy is essentially a function on *cterm*. However, we need to anticipate the potential of failure. We enable this form of partiality by means of an application of the type constructor *option*. Thus:⁵

```
types
  strategy = cterm → result
  result   = cterm option
```

Our simple model follows the letter of the SOS judgement and the original semantics of Stratego: the result of a strategy application is either a term or “failure”. Strafunski generalizes this

³All quotations required by the Isabelle/HOL system are omitted in the code presentations of this paper.

⁴Isabelle/HOL notation: In Isabelle, just like in SML, type parameters precede the type constructor; c.f., *cterm list*. The type *nat* and the type constructor *list* are provided by the Isabelle library. For readability, we typeset function types in terms of “ \rightarrow ” (whereas native Isabelle notation would use “ \Rightarrow ”).

⁵Isabelle/HOL notation: the type constructor *option* is provided by the Isabelle library. An *optional* value is either of the form *Some x* denoting the presence of a value x , or *None* denoting the absence of any value.

view by parametrizing the strategy type by an arbitrary monad, or in a monad with “+” — where necessary.

The function symbols for the strategy primitives receive the following types:

```

consts
  id      :: strategy
  fail    :: strategy
  sequ    :: strategy → strategy → strategy
  choice  :: strategy → strategy → strategy
  all     :: strategy → strategy
  one     :: strategy → strategy

```

3.3 Modeling rewrite rules

The above list of strategy primitives left out the strategy form of rewrite rules from Fig. 1. Indeed, in the present paper, we do not provide any intensional or explicit model of rewrite rules. This is entirely possible because rewrite rules can be represented by regular Isabelle/HOL functions. In fact, Strafunski also uses regular pattern-matching functions to represent rewrite rules. A designated model of rewrite rules would be needed for some profound properties of traversal programming. For instance, any sort of *analysis* of a rewrite system — e.g., an analysis to establish that it is strongly normalizing — would benefit from such a model. However, the trivial model of “rewriting rules as functions” is sufficient for the present paper.

3.4 Definitions for classic control primitives

Here are the canonical definitions for all primitives except “all” and “one”; the definitions are appropriate specializations of the Strafunski combinators (as far as the strategy type is concerned):⁶

```

defs
  id_def: id t = Some t
  fail_def: fail t = None
  sequ_def: sequ s s' t = case s t of None → None | Some t' → s' t'
  choice_def: choice s s' t = if (s t) ≠ None then s t else s' t

```

3.5 Definitions for one-layer traversal

Fig. 4 shows the definitions of “all” and “one”. Again, these definitions paraphrase those of Strafunski, but we have adjusted the definitions for clarity and appropriate modularity so that some of the subsequent discussions and properties can be delivered more easily. In particular, it is helpful that all involved functions can be defined in a primitive recursive fashion.

The definitions emphasize two phases: (i) map the argument strategy over the children using the folklore list *map*; (ii) post-process those intermediate results; c.f., *postMapAll* and *postMapOne*. By making these two phases explicit, we prepare for the separation of recursion into terms vs. the composition of intermediate results.

The post-processor for “all” maps a list of optional terms (the intermediate results) to an optional list of terms, where the result is *None* if some of the optional terms was *None*, and it is the list of present terms otherwise.⁷

The post-processor for “one” maps a list of terms (the original children) and a list of optional terms (the intermediate results) to an optional list, where the result is *None* if all of the intermediate results were *None*; otherwise, it is the list of original terms with one position replaced by a term from the list of intermediate results — the *leftmost* position that is not *None*.

⁶ Isabelle/HOL notation: we make use of *case ... of ...* expressions for pattern matching (or case discrimination) where each case is of the form *Pattern* → *Expression*, and cases are separated by “|”.

⁷ Our function *postMapAll* is an instance of Haskell’s *sequence* operator for sequential evaluation of monadic computations.

```

defs
  all_def: all s t =
    case (postMapAll (map s (children t))) of
      None → None
    | Some l → Some (C (con_of t) l)

  one_def: one s t =
    case (postMapOne (children t) (map s (children t))) of
      None → None
    | Some l → Some (C (con_of t) l)

consts
  postMapAll :: ('a option) list → ('a list) option
  postMapOne :: 'a list → ('a option) list → ('a list) option

primrec
  postMapAll [] = Some []
  postMapAll (r#rs) =
    case r of
      None → None
    | Some x → (case (postMapAll rs) of
        None → None
      | Some xs → Some (x#xs))

primrec
  postMapOne [] rs = None
  postMapOne (x#xs) rs =
    if rs = [] then None
    else case hd rs of
      None → (case postMapOne xs (tl rs) of
        None → None
      | Some xs' → Some (x#xs'))
    | Some x' → Some (x'#xs)

```

Figure 4. Functional model of one-layer traversal combinators

4. SOS lemmas for strategy primitives

We can show now that the SOS-style semantics of traversal strategies (as of Fig. 2) is obeyed by our basic, formal model. In essence, this means that we show the correctness of a functional implementation of the traversal strategies with regard to an operational semantics which we consider indeed as the reference semantics. Admittedly, there is little conceptual gap between the two forms: functional “interpreter style” appears to be very similar to big-step SOS style — except for the separation of positive and negative cases in the SOS specification. However, some efforts are needed due to peculiarities of the traversal primitives.

4.1 Basics of SOS transliteration

In Fig. 5, we begin to transliterate the original SOS rules as Isabelle/HOL lemmas.⁸ We cover all rules but those for “all” and “one” (and rewrite rules). This process of transliteration is systematic: each instance of the judgement is replaced by the application of the corresponding combinator to a term; SOS rules become implications; premises are combined in conjunctions. We can also preserve the style of implicit universal quantification that was used in the SOS rules.

4.2 Transliteration of index-bounded quantification

Interpreter style and SOS style notably differ with regard to the one-layer traversal primitives. The functional implementation of Fig. 4 leverages list-processing functions *map*, *postMapAll*, and *postMapOne*. In contrast, the SOS-style semantics of Fig. 2 lever-

⁸ For readability, we typeset the Isabelle/HOL formulae in normalized predicate-logical notation: “∨”, “∧”, “⇒”, “⇔”, “∀”, “∃”.

<p>lemma <i>id_pos_sos</i>: $id\ t = Some\ t$</p> <p>lemma <i>fail_neg_sos</i>: $fail\ t = None$</p> <p>lemma <i>sequ_pos_sos</i>: $s\ t = Some\ t' \wedge s' t' = Some\ t'' \implies sequ\ s\ s' t = Some\ t''$</p> <p>lemma <i>sequ_neg_1_sos</i>: $s\ t = None \implies sequ\ s\ s' t = None$</p> <p>lemma <i>sequ_neg_2_sos</i>: $s\ t = Some\ t' \wedge s' t' = None \implies sequ\ s\ s' t = None$</p> <p>lemma <i>choice_pos_1_sos</i>: $s\ t = Some\ t' \implies choice\ s\ s' t = Some\ t'$</p> <p>lemma <i>choice_pos_2_sos</i>: $s\ t = None \wedge s' t' = Some\ t' \implies choice\ s\ s' t = Some\ t'$</p> <p>lemma <i>choice_neg_sos</i>: $s\ t = None \wedge s' t' = None \implies choice\ s\ s' t = None$</p>

Figure 5. SOS as lemmas (part I/II)

<p>lemma <i>all_pos_sos</i>: $(\forall (i::nat). 1 \leq i \wedge i \leq n \implies s\ (ts\ i) = Some\ (ts' i))$ $\implies (all\ s\ (C\ c\ (vector\ n\ ts)) = Some\ (C\ c\ (vector\ n\ ts')))$</p> <p>lemma <i>all_neg_sos</i>: $(\exists (i::nat). 1 \leq i \wedge i \leq n \wedge s\ (ts\ i) = None)$ $\implies all\ s\ (C\ c\ (vector\ n\ ts)) = None$</p> <p>lemma <i>one_pos_sos</i>: $(\exists (i::nat). 1 \leq i \wedge i \leq n$ $\wedge s\ (ts\ i) = Some\ (ts' i)$ $\wedge (\forall (i'::nat). 0 < i' \wedge i' \leq n \wedge i' < i \implies s\ (ts\ i') = None)$ $\wedge (\forall (i'::nat). 0 < i' \wedge i' \leq n \wedge i' \neq i \implies ts\ i' = ts' i'))$ $\implies one\ s\ (C\ c\ (vector\ n\ ts)) = Some\ (C\ c\ (vector\ n\ ts')))$</p> <p>lemma <i>one_neg_sos</i>: $(\forall (i::nat). 1 \leq i \wedge i \leq n \implies s\ (ts\ i) = None)$ $\implies one\ s\ (C\ c\ (vector\ n\ ts)) = None$</p>
<p>Helper function</p> <p>consts $vector :: nat \rightarrow (nat \rightarrow 'a) \rightarrow 'a\ list$</p> <p>primrec $vector\ 0\ f = []$ $vector\ (Suc\ n)\ f = ((vector\ n\ f)@[f\ (Suc\ n)])$</p>

Figure 6. SOS as lemmas (part II/II)

ages *index-bounded* universal and existential quantification over terms in lists of immediate subterms. The use of the different idioms can be shown to be equivalent.

Before we can even state the SOS rules as lemmas, we need to discipline the SOS notation [36] which make use of the “...” notation for indexed lists of terms. Consider again one of the SOS rules:

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

We use place holders for *lists* of terms instead of the “...” notation — as in $c(ts)$. Further, we assume that such variables can be annotated by the length of the vector — as in $c(ts : n)$. Finally,

we may use explicit index-based access — as in $ts!i$. Hence, we end up with this variation:

$$\frac{\forall i \in \{1, \dots, n\}. s @ ts!i \rightsquigarrow ts'!i}{\Box(s) @ c(ts : n) \rightsquigarrow c(ts' : n)} \quad [\text{all}^+]$$

The model of Fig. 6 formalizes these conventions. We view lists of terms as maps from positions to terms so that index-based access becomes function application. Such maps are trivially converted into actual lists by the trivial helper function *vector* (defined in Figure 6). In the applications of *vector*, we constrain the length of lists of immediate subterms as announced above.

5. Laws of strategy primitives

Let us extend the basic formal model by a layer with laws about strategy primitives. All of the laws from this section are known if not obvious, but we claim credit for collecting them and adding them to the mechanized (proven) model. The proofs of the laws are straightforward — except for the fusion law that we present last.

5.1 Laws for “sequ” and “choice”

The combinators *sequ* and *choice* obey obvious zeros and units. They are both also associative. Neither of them is commutative (but we omit a proven counterexample.)

lemma <i>sequ_left_unit_law</i> :	$sequ\ id\ s$	$= s$
lemma <i>sequ_right_unit_law</i> :	$sequ\ s\ id$	$= s$
lemma <i>sequ_left_zero_law</i> :	$sequ\ fail\ s$	$= fail$
lemma <i>sequ_right_zero_law</i> :	$sequ\ s\ fail$	$= fail$
lemma <i>choice_left_zero_law</i> :	$choice\ id\ s$	$= id$
lemma <i>choice_left_unit_law</i> :	$choice\ fail\ s$	$= s$
lemma <i>choice_right_unit_law</i> :	$choice\ s\ fail$	$= s$

lemma <i>sequ_assoc_law</i> :	$sequ\ s\ (sequ\ s'\ s'')$	$= sequ\ (sequ\ s\ s')\ s''$
lemma <i>choice_assoc_law</i> :	$choice\ s\ (choice\ s'\ s'')$	$= choice\ (choice\ s\ s')\ s''$

Distributivity only works from the left side, i.e., the chosen semantics of choice only provides local backtracking — as opposed to general backtracking or angelic choice [23]. Thus:

lemma <i>distr_left_law</i> :	$sequ\ s\ (choice\ s'\ s'')$	$= choice\ (sequ\ s\ s')\ (sequ\ s'\ s'')$
NOT A lemma <i>distr_right_law</i> :	$sequ\ (choice\ s\ s')\ s''$	$\neq choice\ (sequ\ s\ s')\ (sequ\ s'\ s'')$

5.2 Laws for “all” and “one”

There are two laws about “trivial” applications of *all* and *one*:

lemma <i>all_id_law</i> :	$all\ id$	$= id$
lemma <i>one_fail_law</i> :	$one\ fail$	$= fail$

There are laws with a condition on the term to be a constant:

consts <i>constant</i> ::	$c\ term \rightarrow bool$
defs <i>constant_def</i> :	$constant\ t = children\ t = []$

lemma <i>all_constant_law</i> :	$constant\ t$	$\implies all\ s\ t = id\ t$
lemma <i>all_not_constant_law</i> :	$\neg(constant\ s)$	$\implies all\ fail\ t = fail\ t$
lemma <i>one_constant_law</i> :	$constant\ t$	$\implies one\ s\ t = fail\ t$
lemma <i>one_not_constant_law</i> :	$\neg(constant\ t)$	$\implies one\ id\ t = id\ t$

There are laws for the preservation of the outermost constructor:

lemma <i>all_con_law</i> :	$all\ s\ t = Some\ t'$	$\implies con_of\ t = con_of\ t'$
lemma <i>one_con_law</i> :	$one\ s\ t = Some\ t'$	$\implies con_of\ t = con_of\ t'$

5.3 Fusion law for “all”

As already claimed (without proof) in the first “Scrap your boilerplate” publication [18], the *all* combinator allows for fusion, i.e., for composition of two traversals into one:

lemma all_fusion_law: sequ (all s) (all s') = all (sequ s s')

This law is important for practical applications of traversal strategies. For instance, in [7], the law is used to optimize strategic programs by calculation.

In [31], a non-mechanized proof of the law is given within the framework of generic functional programming based on sums-of-products representations.

In our formal model, we have developed a different and mechanized proof. It uses a fundamental fusion law for monadic list maps [28]. Two maps of a possibly failing function over a list can be combined into one:

```
axioms
map'_rule: map' (sequ s s') xs = bind (map' s xs) (map' s');
consts
bind :: 'a option → ('a → 'z option) → 'z option
map' :: ('a → 'a option) → 'a list → 'a list option
defs
map'_def: map' s xs = postMapAll (map s xs)
primrec
bind None s = None
bind (Some x) s = s x
```

We state this law as an axiom because we have not mechanically proven it. Based on this law, we can prove *all_fusion_law* by a simple mechanized calculation only involving unfolding and trivial properties of *sequ* and *all*. Informally, one ends up showing that a sequence of two *alls* is little more than a sequence of two *maps* — except that the outermost constructor is preserved on the way.

6. Modeling partial strategies

We have only considered primitive strategy combinators so far, and all of them turn out to represent total functions — as witnessed by our ability to define them in terms of some functional definition form of Isabelle/HOL. In general though, strategies (in particular, traversal strategies) may be diverging quite easily. Consider, for example, the combinator *repeat* (c.f., Fig. 3). It may diverge quite obviously — even when it is applied to a total strategy argument. For instance, *repeat id* is patently diverging.

Divergence signifies a programming error in strategic programming [19]. Hence, our formal model should make a contribution to the understanding of divergence. In fact, our formal model *must* take into account partiality of strategies because Isabelle/HOL exclusively assumes total models for all functions. Hence, we must use some encoding for the partial functions. In this section, we motivate our choice of encoding and describe the recipe behind it. We use *repeat* as a running example, but more examples follow in the next section.

6.1 Dismissal of functional definition forms

The combinator *try* is total and indeed the function definition as of Fig. 3 can be directly transliterated as an Isabelle/HOL function definition:

```
consts
try :: strategy → strategy
defs
try_def: try s t = choice s id t
```

This transliteration cannot succeed for the combinator *repeat* since we know that the general recursive function definition as of Fig. 3 denotes a partial function, but here is our attempt anyway:

```
/* This definition is illegal in Isabelle/HOL. */
consts
repeat :: strategy → strategy
defs
repeat_def: repeat s = try (sequ s (repeat s))
```

The definition does not meet the constraints of an Isabelle/HOL-like total function definition. Accordingly, it is rejected by the theorem prover.

6.2 Dismissal of axiomatization

One may feel tempted to axiomatize *repeat*:

```
/* This definition is formally risky. */
consts
repeat :: strategy → strategy
axioms
repeat_def: repeat s = try (sequ s (repeat s))
```

Again, the axiom directly captures the recursive function definition as of Fig. 3. This attempt provides certain problems. While the “intended” *repeat* function is not total, the “axiomatized” *repeat* function has a total model by definition (due to the semantics of Isabelle/HOL). The axiom states that the model should be such that for each strategy *s* and for each term *t* there is a result *r* as follows:

```
repeat s t = try (sequ s (repeat s)) t = r
```

However, it is not obvious that such a total model even exists. More generally, an axiom like the one above, could make our logic inconsistent. Also, such a lax approach would make us miss altogether the conditions under which *repeat* terminates.

6.3 Outline of modeling approach

Isabelle/HOL actually requires from us to *identify* and *prove* termination conditions. This is not the case for the more implementation-oriented models of strategic programming (e.g., “Strafunski”) with their unconstrained use of general recursion.

We model each partial strategy (combinator) as a *relation between input terms and results*. The relation is meant to cover the non-diverging part of the strategy (combinator) in question. In the case of traversal schemes, this relation is modeled as an *inductive set* (which is Isabelle/HOL’s inductive definition form for sets).

In a next phase, the relation is embedded into a new function. We prefer to view strategies (strategy combinators) generally as functions. Because of Isabelle/HOL’s semantics, the resulting function is total by definition, but it is left unspecified for anything not covered by the relation.

The last step is to show that the resulting function agrees with the general recursive definition (as of Fig. 3) for the assumed non-diverging part. At this stage (if not earlier), we need to employ a sufficient *termination condition* for the strategy (combinator) in question. (Of course, there may be several alternative sufficient conditions.)

6.4 Identification of termination conditions

What is the termination condition of *repeat s*? If we look closely at the general recursive definition of *repeat s*, then we see that the (verifiable) intention of *repeat s* is to apply the argument strategy *s* until it fails.

Thus, for a terminating application of *repeat s*, there must exist an $n \in \mathbf{N}$ such that $\text{repeat } s \ t_1 \mapsto t_n$ where $s \ t_1 \mapsto t_2, \dots, s \ t_{n-1} \mapsto t_n, s \ t_n \mapsto \uparrow$. The computation of these t_{i+1} from t_i for $1 \leq i < n$ can be modeled as follows:

```
consts
chain :: strategy → nat → cterm → result
primrec
chain s 0 t = id t
chain s (Suc i) t = (case s t of None → None | Some t' → chain s i t')
```

Thus, the termination condition for $repeat\ s\ t$ is this:

```

consts
repeat_condition :: strategy → cterm → bool
defs
repeat_condition_def:
repeat_condition s t =
  ∃ (t'::cterm). (∃ (i::nat). chain s i t = Some t') ∧ s t' = None

```

6.5 Leverage of a relational definition

We set up a relation to model the non-diverging part of the strategy in question:

```

consts
repeat_set :: strategy → (cterm × result) set
defs
repeat_set_def:
repeat_set s = {(t,r). (∃ (t'::cterm). (∃ (i::nat).
  chain s i t = Some t') ∧ s t' = None ∧ r = id t')};

```

The $repeat_condition$ appears inlined.

6.6 The functional property of a relation

We should establish that the relation is in fact a (possibly partial) function, i.e., every term is associated with at most one result. We use the following property to this end:

```

consts
functional :: (cterm × result) set → bool
defs
functional_def: functional rel =
  (∀ (t::cterm). (∀ (r::result). (∀ (r'::result).
  (t,r):rel ∧ (t,r'):rel ⇒ r = r')));

```

The following lemma states that $repeat_set\ s$ is a function:

```

lemma repeat_functional: functional (repeat_set s)

```

Its proof follows from the uniqueness of $chain$'s result:

```

lemma chain_uniqueness:
(chain s i t = Some t' ∧ s t' = None ∧
  chain s i' t = Some t'' ∧ s t'' = None)
  ⇒ t' = t'';

```

6.7 The relation-to-function conversion

Let us turn the relational model into a functional one. We simply use an axiom to embed the relation (which we know represents a function) into a function:

```

consts
repeat_fun :: strategy → strategy
axioms
repeat_set2fun: (t,r):(repeat_set s) ⇒ repeat_fun s t = r

```

That is, we describe a function that fully respects the given relation, but is left unspecified otherwise. That is, $repeat_fun$ is a total extension of $repeat_set$. This sort of axiom cannot create any inconsistency.

6.8 Verifying the general recursive definition

At this point we have obtained a model that appropriately describes the input/output behavior of the strategy (combinator) in question including one or more sufficient conditions for its termination. However, this development is disconnected from the original, general recursive definition of the strategy (combinator) in question; c.f. Fig. 3. We would like to be sure that the new definition agrees

(in some formal sense) with the original definition. This can be achieved by turning the general recursive definition into a property subject to proof. Thus:

```

lemma repeat_rec:
repeat_condition s t
  ⇒ repeat_fun s t = try (sequ s (repeat_fun s)) t;

```

To summarize, Isabelle/HOL's total model of repeat agrees with the diverging repeat (defined by general recursion) for the non-diverging part of it. The proof uses definition unfolding, case distinctions and some simple properties of the involved primitives.

7. Modeling traversal schemes

In the following, we will formalize the most established traversal schemes for transformation: $bottomup$, $oncebu$, $oncetd$, $stoptd$, $topdown$ and $innermost$. In fact, we only provide details for $bottomup$, $topdown$ and $innermost$. We pick $bottomup$ as a representative for total traversal schemes, thereby covering $oncebu$, $oncetd$ and $stoptd$. The mechanics of the two remaining schemes $topdown$ and $innermost$ are quite different, and hence, we treat them separately.

7.1 Model of the bottom-up scheme

Consider the axiomatization of bottom-up traversal as of Fig. 3:

```

bottomup s = sequ (all (bottomup s)) s

```

We are interested in an Isabelle/HOL-compliant definition. To this end, we also need to capture the termination condition for bottom-up traversal. Our knowledgeable guess is that $bottomup\ s\ t$ terminates for any t as long as s is a terminating function.

In order to deal with the recursive nature of traversal schemes, we use Isabelle/HOL's definition form of an *inductive set* instead. That is, we design a set $bottomup_set\ s$ which comprises (subject to a proof) the set of all term-result pairs for bottom-up traversal while we derive "bigger" elements of this set (measured by the size of the input term) inductively from "smaller" elements.

Hence, the important question is how to derive a useful inductive formulation here. So consider again the axiom above and imagine that we "cut the recursive knot" so that we define a variation that does *not recurse* into children by itself, but rather receives the recursive results and the original constructor as arguments. Based on additional unfolding of primitives, we obtain this function:

```

bottomup_step s c rs =
  case (postMapAll rs) of
  None → None
  | Some ts' → s (C c ts')

```

Fig. 7 completes $bottomup_step$ into an inductive set (indicated by *inductive* with introduction rule *rule*), $bottomup_set$, for bottom-up traversal. The addition to the set is covered by the final part of the intro:

```

... ⇒ (C c ts, bottomup_step s c rs):(bottomup_set s)

```

Here, $C\ c\ ts$ takes apart the input term, and $bottomup_step\ s\ c\ rs$ constructs the result. We note that both c , ts and rs are (implicitly) universally quantified as usual. Children and results are associated with each other by zipping:

```

(∀ (t::cterm). in_list t ts ⇒
  (∃ (r::result).
  in_list (t,r) (zip ts rs) ...

```

Child-result pairs are retrieved from the inductive set as follows:

```

... ∧ (t,r):(bottomup_set s)) ⇒ ...

```

We would again need to establish that the set $bottomup_set\ s$ represents a (possibly partial) function (c.f., §6.6), but we skip this step throughout this section. Let us embed $bottomup_set\ s$ into a function $bottomup_fun\ s$:

```

consts
  bottomup_step :: strategy → con → result list → result
  bottomup_set  :: strategy → (cterm × result) set

defs
  bottomup_step s c rs =
    case (postMapAll rs) of
      None  → None
    | Some ts' → s (C c ts')

inductive bottomup_set s
  intros
  rule[intro!]:
    (∀ (t::cterm). in_list t ts ⇒
     (∃ (r::result).
      in_list (t,r) (zip ts rs)
      ∧ length ts = length rs
      ∧ (t,r):(bottomup_set s)))
    ⇒ (C c ts, bottomup_step s c rs):(bottomup_set s)

```

Figure 7. Inductive set for bottom-up traversal

```

consts
  bottomup_fun :: strategy → strategy;
axioms
  bottomup_set2fun:
    (t,r):(bottomup_set s) ⇒ (bottomup_fun s t = r);

```

We expect bottom-up traversal to be total (non-diverging) for any total argument strategy. Hence, we expect *bottomup_set s* to represent a total function (for any total *s*). Hence, *bottomup_fun* should be fully specified by the above axiom, and we should be able to show that the set and the function are equivalent without further side conditions. Indeed:

lemma *bottomup_total*: *bottomup_fun s t = r = (t,r):(bottomup_set s)*;

The proof relies on induction on the size of term *t* while leveraging the definition of *bottomup_fun s* and the introduction rule of the inductive set *bottomup_set s* as well as basic properties of strategy primitives. Intuitively, induction on size of terms works out in this case because *bottomup_set s* visits the input term layer by layer, and it is clear that the size of terms decreases with each layer, i.e.:

$$\forall (t::cterm). \text{in_list } t \text{ ts} \implies \text{size } t < \text{size } (C \text{ c ts})$$

Finally, we can validate the new definition of bottom-up traversal with regard to the original, general recursive definition.

lemma *bottomup_rec*:
bottomup_fun s t = sequ (all (bottomup_fun s) s) t;

The proof is straightforward; no induction is needed.

7.2 Models of other total schemes

All other total schemes from Fig. 3 (*oncebu*, *oncetd* and *stoptd*) can be modeled in a very similar manner. These schemes are particularly close to each other in that they all recurse into the input term layer by layer. We use the term “input-driven traversal” hence.

Fig. 8 generalizes the inductive set that we presented for bottom-up traversal in Fig. 7. It is parametrized by the “non-recursive part” of a traversal scheme (such as *bottomup_step* in Fig. 7). This parameter is also generalized in so far that it receives not only the constructor of the input term, but even the complete term (c.f., *i*). This is necessary for the schemes that traverse in top-down manner (i.e., *oncetd* and *stoptd*) because they invoke the argument strategy on the input term in order to decide whether or not to recurse at all.

```

consts
  generalized_set ::
    (strategy → cterm → result list → result)
    → strategy
    → (cterm × result) set

inductive generalized_set f s
  intros
  rule[intro!]:
    (∀ (t::cterm). in_list t (children i) ⇒
     (∃ (r::result).
      in_list (t,r) (zip (children i) rs)
      ∧ length (children i) = length rs
      ∧ (t,r):(generalized_set f s)))
    ⇒ (i, f s i rs):(generalized_set f s)

```

Figure 8. Generalized inductive set for input-driven traversal

```

consts
  topdown_step :: strategy → cterm → result list → result
  topdown_set  :: strategy → (cterm × result) set

defs
  topdown_step_def:
  topdown_step s t rs =
    case s t of
      None  → None
    | Some t' → (case (postMapAll rs) of
      None  → None
      | Some ts' → Some (C (con_of t') ts'))

inductive topdown_set s
  intros
  rule[intro!]:
    ( s i = None
    ∨ ( s i = Some t'
      ∧ (∀ (t::cterm). in_list t (children t') ⇒
        (∃ (r::result).
         in_list (t,r) (zip (children t') rs)
         ∧ length (children t') = length rs
         ∧ (t,r):(topdown_set s))))))
    ⇒ (i, topdown_step s t rs):(topdown_set s)

```

Figure 9. Inductive set for top-down traversal

7.3 Model of the top-down scheme

The scheme for top-down traversal is *not* terminating for arbitrary (non-diverging) arguments. For instance, it is easy to see that *topdown s* diverges if *s* always increases the size of the given term. In the terminology of the previous section, the scheme for top-down traversal is not input-driven. Instead, the scheme recurses into a term that is obtained by first applying the argument strategy to the input term.

An inductive set can be still defined quite similarly as before; c.f., Fig. 9. The essential difference can be spotted in the position where terms are looked up from the inductive set:

$$s \text{ i} = \text{Some } t' \\ \wedge (\forall (t::cterm). \text{in_list } t \text{ (children } t') \implies \\ (\exists (r::result). \dots \wedge (t,r):(topdown_set \text{ s})))$$

That is, terms *t* are not retrieved from *i* but from *s i* (assuming it is a term). It would be possible to generalize *generalized_set* of Fig. 8 in order to also cover non-input-driven sets. This shows that we are relatively close to a model of general recursion combinator for traversal schemes.

As before, we embed the set into a function as follows:

```

consts
  topdown_fun :: strategy → strategy;
axioms
  topdown_set2fun:
    (t,r):(topdown_set s t) ⇒ topdown_fun s t = r;

```

We know that *topdown_set* only represent a partial function. Hence, we cannot expect to prove that *topdown_set* is equivalent to *topdown_fun* which has a total model by definition. We must identify a sufficient termination condition for top-down traversal. As a first attempt, let us use the condition that the argument strategy never increases the size of the term. Such a property is formalized as follows:

```

consts
  nonincreasing :: strategy → bool
defs
  nonincreasing_def: nonincreasing s =
    (∀ (t::cterm). s t = None
     ∨ (∃ (t'::cterm). s t = Some t' ∧ size t' ≤ size t))

```

The following lemma clarifies that the non-diverging part of top-down traversal, as it is captured by *topdown_set*, *subsumes* the case of non-increasing argument strategies:

```

lemma topdown_partial:
  nonincreasing s
  ⇒ topdown_fun s t = r = (t,r):(topdown_set s);

```

The proof is very similar to the one for *bottomup_total* because the *nonincreasing* property allows us again to use induction on the size of term *t* with a special case for *s t = None*.

Finally, we can also validate the new *topdown* function with regard to the original, general recursive definition. Again, we should only expect to prove compliance if the argument strategy meets a sufficient condition such as the *nonincreasing* property:

```

lemma topdown_rec:
  nonincreasing s
  ⇒ topdown_fun s t = sequ s (all (topdown_fun s)) t;

```

In general, we cannot restrict applications of the top-down scheme to arguments with the *nonincreasing* property. For instance, a typical application scenario of top-down traversal is *de-sugaring* where one pattern is replaced most likely by a larger term; think of macro expansion.

A generalized termination condition for *topdown s t* can be based on a *measure for terms*, *m*, such that any application of the argument strategy *s* to a given term *t* results in a term *t'* (if any) such that all children of *t'* are smaller (w.r.t. *m*) than *t*. Thus:

```

consts
  topdown_condition :: strategy → cterm → (cterm → nat) → bool
defs
  topdown_condition_def: topdown_condition s t m =
    ∀ (t'::cterm). ∀ (t''::cterm).
      in_list t'' (children t') ∧ s t = Some t'
      ⇒ m t'' < m t

```

For instance, we cover a restricted form of de-sugaring (say, macro expansion) with regard to a single constructor *c* to be eliminated if we assume that the count for this constructor is decreased continuously:

```

{* Count the occurrences of a given constructor in a term *}
consts
  conCount :: con → cterm → nat

```

(Definition omitted for brevity.) For reasons of practicality, we should not instantiate the measure argument with *conCount* directly

because the argument strategy will typically preserve the constructor count when the constructor of interest is not the outermost constructor. Hence, we need to *compose a measure* from *conCount* and *size*.

7.4 Model of innermost normalization

We do not need a designated inductive set this time because *innermost s* is defined by applying *repeat* to *oncebu s*. Hence, we can use an Isabelle/HOL function definition as a model:

```

consts
  innermost_fun :: strategy → cterm → result;
defs
  innermost_fun_def: innermost_fun s = repeat_fun (oncebu_fun s);

```

Our analysis of *repeat* (as of §6) has already provided us with a general termination condition for any application of *repeat*. Hence, we know (in fact, we have proven) that *innermost s* terminates if the following condition is met:

```

repeat_condition (oncebu_fun s)

```

Alas, this general condition is not very insightful. Let us try to find a termination condition that is based on a measure again — as in the case of top-down traversal. While it was straightforward to parametrize in an arbitrary measure back then, this is not immediately possible for innermost normalization because we rely on an extra property. It is not sufficient that *s* is measure-decreasing, we actually need to establish that *oncebu s* is measure-decreasing, too. For concrete measures we can prove this additional property. For instance, the following predicate is a sufficient termination condition for *innermost_fun s t*:

```

consts
  innermost_conCount :: strategy → cterm → con → bool
defs
  innermost_conCount_def:
    innermost_conCount s t c =
      ∀ (t'::cterm). s t = Some t' ⇒ conCount c t' < conCount c t

```

The correctness of this termination condition follows from the fact that we can prove that decrease of constructor count implies the finiteness of the repeat chain. Thus:

```

lemma innermost_conCount_ok:
  innermost_conCount s t c
  ⇒ repeat_condition (oncebu_fun s) t

```

8. Success/failure behavior

Traversal schemes make different assumptions about the success/failure behavior for their arguments, and they face different chances and reasons to succeed or fail. For instance, the argument of one scheme (say, *topdown*) may be expected to *succeed* “for most if not all terms”, whereas the argument of another scheme (say, *oncebu*) may be expected to *fail* “for most but not all terms”. Understanding these properties is an important element of mastering strategic programming [19].

Below, we show that the presented formal model allows us to capture and verify properties related to the success/failure behavior of strategy primitives as well as traversal schemes.

8.1 In-/fallibility of strategies

We say that a strategy is *fallible* if it fails for some term; a strategy is *infallible* if it is not fallible. Thus:

```

consts
  fallible      :: strategy → bool
  infallible    :: strategy → bool

```

defs

fallible_def : *fallible s* = $\exists (t::c\text{term}). s\ t = \text{None}$
infallible_def : *infallible s* = $\neg(\text{fallible } s)$

8.2 In-/fallibility of the strategy primitives

The strategy primitives meet the following infallibility properties:

lemma *id_not_fail*:

infallible id

lemma *sequ_not_fail*:

infallible s \wedge *infallible s'* \implies *infallible (sequ s s')*

lemma *choice_not_fail*:

infallible s \vee *infallible s'* \implies *infallible (choice s s')*

lemma *all_not_fail*:

infallible s \implies *infallible (all s)*

Moreover, the following fallibility properties hold:

lemma *fail_fail* : *fallible fail*

lemma *sequ_fail* : *fallible s* \implies *fallible (sequ s s')*

lemma *all_fail* : *fallible s* \implies *fallible (all s)*

lemma *one_fail* : *fallible (one s)*

For instance, lemma *one_fail* says that *one s* is fallible no matter what (say, even for an infallible *s* — because *one s* may still be applied to a constant, in which cases it fails definitely). There is also a (relatively plausible) property that does *not* hold:

NOT A lemma *choice_fail*:

fallible s \wedge *fallible s'* \implies *fallible (choice s s')*

That is, fallibility of two strategies does not imply that their composition by choice is fallible (because they could be failing in a mutually exclusive manner). Likewise, fallibility of *s'* in *sequ s s'* does *not* imply fallibility of the composition. It is easy to document and prove counter-examples in our setup; we omit them here for brevity.

8.3 Infallibility of the bottom-up scheme

We can prove the following property:

lemma *bottomup_not_fail*:

infallible s \implies *infallible (bottomup_fun s)*

The proof of the above lemma requires induction on the size of terms. To provide some insight into the proof needed, we inline a lemma that models the *induction step*:

lemma *bottomup_not_fail_step*:

infallible s

$\wedge (\forall (t'::c\text{term}). \text{size } t' < \text{size } t \implies \text{bottomup_fun } s\ t' \neq \text{None})$
 $\implies \text{bottomup_fun } s\ t \neq \text{None}$

The induction hypothesis (see second operand of conjunction) establishes here that *bottomup_fun s* does not fail for terms *t'* smaller in size than terms *t* being considered in the induction step.

8.4 Infallibility of the top-down scheme

One may feel tempted to treat the top-down scheme the same way by proving a property like the following:

NOT A lemma *topdown_total*:

topdown_fun s t = r = (t,r):(topdown_set s);

Just as in the case of modeling top-down traversal (c.f., §7.3), we need to constrain the argument strategy *s* so that we are able to perform an induction proof. Again, we may use an arbitrary measure for a “non-increasing” property of *s*. For concreteness’ sake, we show the lemma here for the specific case of a strategy that is non-increasing in *size*:

lemma *topdown_not_fail*:

infallible s \wedge *nonincreasing s* \implies *infallible (topdown_fun s)*

8.5 Infallibility of the top-down scheme with stop

We can prove, that the topdown scheme with stop is infallible — no matter what the argument strategy (because the traversal will try as long as it succeeds (in every “branch”), or it may hit a constant term eventually, which implies success, too).

lemma *stoptd_not_fail*: *infallible (stoptd_fun s)*

Again, an induction proof, similar to those above, is needed. For the base case, we can show that *stoptd_fun s* cannot fail whenever it is applied to a constant term — as a consequence of **lemma** *all_constant_law* of Sec. 5.

8.6 Success/failure behavior of the once... schemes

Let us start with the following attempts:

lemma *oncetd_not_fail_USELESS*:

infallible s \implies *infallible (oncetd_fun s)*

lemma *oncebu_not_fail_USELESS*:

infallible s \implies *infallible (oncebu_fun s)*

The infallibility of the argument strategies are indeed sufficient conditions for the infallibility of the schemes, but these are too strong requirements for practical purposes.

The *once...* schemes are typically used with argument strategies that “fail most of the time”. Arguably, the following claims are more useful: *oncetd_fun s t* and *oncebu_fun s t* succeed if *s* succeeds for some subterm of *t*.

The following definition maps a given term to the set of its subterms; we use Isabelle/HOL’s form of recursive definitions:

consts

in_term :: *cterm* \rightarrow *cterm set*

recdef

in_term *measure* ($\lambda t. \text{size } t$)

in_term_def: *in_term* *t* =

$\{t\}$

$\cup (\bigcup (t'::c\text{term}).$

if in_list t' (children t) then in_term t' else \{ \})

The announced properties of the *once...* schemes are formalized as follows:

lemma *oncebu_not_fail*:

$(\exists (t'::c\text{term}). t'::(\text{in_term } t) \wedge s\ t' \neq \text{None})$

$\implies \text{oncebu_fun } s\ t \neq \text{None}$

lemma *oncetd_not_fail*:

$(\exists (t'::c\text{term}). t'::(\text{in_term } t) \wedge s\ t' \neq \text{None})$

$\implies \text{oncetd_fun } s\ t \neq \text{None}$

8.7 Infallibility of innermost normalization

Innermost normalization is infallible — no matter what the argument strategy. Intuitively, this is obvious from its definition because we can only exit the iteration of *repeat* with a non-failure result.

We should mention that we can customize the notion of in-/fallibility to apply to set-based definitions (for the non-diverging parts of strategies). In this case, we do not even have to constrain the in-/fallibility lemmas with a termination condition (as we were forced to do in the case of top-down traversal above). Thus:

lemma *repeat_set_not_fail*: *infallible (repeat_set s)*

lemma *innermost_set_not_fail*: *infallible (innermost_set s)*

Here is the infallibility property *with* the termination condition:

lemma *innermost_fun_not_fail*:

repeat_condition (oncebu_fun s) \implies *infallible (innermost_fun s)*

9. Related work

Properties of traversal programs Algebraic laws of strategic primitives appear in the literature on Stratego-like strategies [20, 13, 18, 17, 7] — with “paper and pencil” proofs (if any). In [31], the fusion law for “all” was proved. In [7], 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). Those authors call out the need to formally prove the involved laws. In [13], specialized laws of applications of traversal schemes are leveraged to enable fusion-like techniques for optimizing strategies. We have not yet started to formalize such more sophisticated applications of algebraic reasoning, but this is an important topic for future work.

Proof tactics vs. traversal strategies A strategy in term rewriting is comparable to a *tactic* in theorem proving: the former starts from rewrite rules and combines them into a “function” that performs a more complex operation on terms; the latter starts from sound proof rules and combines them into a “function” that performs a compound proof step on an initial goal. In this vein, strategy combinators are like *tacticals* [11]. There is a classic view on theorem proving as involving rewriting (of goals) [29, 24]. There exist various *tactic languages* (e.g., [23, 9, 27, 2]) that share primitives with Stratego-like setups — with the noteworthy exception of “all” and “one”. Some tactic languages (e.g., Angel [23]) include combinators to apply tactics to subgoals (subterms), but generic one-layer traversal and generic traversal schemes are not covered (even though [14] suggests to lift this restriction).

The PoplMark challenge [3] This challenge aims to turn formalization of programming language metatheory into a regular practice. We can relate to some of the associated critical issues. That is, the challenge emphasizes binding, complex induction, component reuse, and experimentation — to be discussed one by one. *Binding* is not relevant for our limited development because we left out recursion and abstraction from the semantics part of the formal model. If we were adding those, the issue of termination conditions would need to be implanted into the semantics presumably. *Complex induction* is clearly relevant for the central theme of our development: generic traversal. (This instance of the complex induction issue does not appear in work on the PoplMark challenge.) Isabelle/HOL’s capabilities provided a good fit. *Component reuse* is trivially addressed by a collection of laws about strategy primitives; these laws are helpful in proving more substantial properties about traversal schemes. *Experimentation*, i.e., testing language implementations against formalized definitions, is successful in our case: the initial Isabelle/HOL-based semantics encodes an interpreter (a combinator library) that corresponds to an actual implementation of traversal strategies in Haskell [21].

Related uses of Isabelle/HOL We have leveraged a theorem prover with a track record in the formalization of programming language metatheory; see, e.g., [25, 8, 38, 16]. Isabelle/HOL’s primitive data types, recursive functions and other forms of definitions provide a convenient programming language for formal models (up to the point that restricted Haskell programs can be systematically mapped to Isabelle/HOL [32]). Isabelle/HOL’s induction schemes and inductive sets are essential for the verification of metatheory due to the recursive or iterative nature of syntactic and semantic domains and concepts. For instance, in [25], reduction systems of lambda calculi are modeled as inductive sets so that one can reason about the reflexive and transitive closure of reduction. We also refer to [8] for a metatheory-related evaluation of Isabelle/HOL capabilities. In our development, we rely on induction on the size of (traversed) terms for many of our proofs. Also, we use inductive sets to model partiality of traversal schemes.

Theory	LOC	KB	All	Main	Other
Terms (§3)	82	3	16	0	16
Primitives (§3)	146	5	25	3	22
SOS (§4)	56	2	12	12	0
Laws (§5)	895	33	161	29	132
Model of (§6, §7)					
• <i>repeat</i>	576	25	105	2	103
• <i>bottomup</i>	163	10	23	2	21
• <i>topdown</i>	247	15	34	2	32
• <i>oncebu</i>	148	8	21	2	19
• <i>innermost</i>	23	1	3	2	1
(In)fallibility of (§8)					
• <i>bottomup</i>	36	2	5	1	4
• <i>topdown</i>	126	6	19	4	15
• <i>stoptd</i>	46	2	7	1	6
• <i>innermost</i>	7	1	1	1	1

Figure 10. Complexity metrics for Isabelle/HOL theories (LOC: Lines Of Code; KB: Kilobytes in ASCII; All: all properties; Main: main properties; Other: helper lemmas. In fact, the measures cover approx. 70% of our development, where we left out some part that we consider routine (having to do with list processing and optionals). The extent shown in the table covers slightly more properties and variations than those mentioned in the paper. We have derived these numbers on the grounds of a suitable modularization and tagging scheme for all theories and properties.)

10. Concluding remarks

We have initiated the first mechanized, formal model of traversal strategies à la Stratego. The model covers semantics, basic algebraic laws, and some aspects of termination and success/failure behavior. For the record, Fig. 10 lists some complexity indicators for the developed Isabelle/HOL theories. In our experience, Isabelle/HOL worked very well as a *modeling environment* for the notion of traversal strategies; it allowed us to gather and *verify* insights that were not explicitly present in any form prior to this effort.

We have developed the model only for type-preserving strategies (say, transformations), but we contend that type-unifying strategies (say, queries) can be covered similarly. We have assumed untyped strategies, but suggest typed strategies as a future-work topic to better match existing functional and object-oriented incarnations of strategic programming. Along the same line of generalization, other related strategy languages and their semantics should be investigated and possibly integrated into the formal development [5, 22].

More interestingly, we expect the developed model also to be useful in studying further properties of strategic programming, e.g., the correctness of non-trivial optimizations of traversals [13, 7].

Another challenging elaboration of formally modeling traversal strategies is to cover advanced term-rewriting theory and thereby improve the precision of termination claims (e.g., effective, sufficient properties for rewrite systems to be strongly normalizing). There is related work on strategic programming and termination or normalization of rewrite systems that could form a foundation for such an additional layer of formal modeling [1, 15, 10].

Finally, we suggest to complement our theorem proving-based development by abstract interpretation efforts such that properties of strategies (e.g., termination, or success and failure behavior) can be computed, within limits — even for arbitrary recursive strategies. Such an abstract interpretation raises the formal challenge of proving it correct with regard to the formal semantics.

References

- [1] S. Abramsky, D. M. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures*. Oxford Science, 1992.
- [2] D. Aspinall, E. Denney, and C. Lüth. A tactic language for hiproofs. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 339–354. Springer, 2008.
- [3] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proceedings, Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.
- [4] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In *Proceedings, Term Rewriting and Applications, 18th International Conference, RTA 2007*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
- [5] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with Strategies in ELAN: A Functional Semantics. *Int. J. Found. Comput. Sci.*, 12(1):69–95, 2001.
- [6] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: components for transformation systems. In *PEPM'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 95–99. ACM, 2006.
- [7] A. Cunha and J. Visser. Transformation of structure-shy programs: applied to XPath queries and strategic functions. In *PEPM'07: Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 11–20. ACM Press, 2007.
- [8] J. E. Dawson and R. Goré. Embedding Display Calculi into Logical Frameworks: Comparing Twelf and Isabelle. *ENTCS*, 42, 2001.
- [9] D. Delahaye. A Tactic Language for the System Coq. In *LPAR 00: Proceedings of Logic for Programming and Automated Reasoning, 7th International Conference*, volume 1955 of *LNCS*, pages 85–95. Springer, 2000.
- [10] I. Gnaedig and H. Kirchner. Termination of rewriting under strategies. *ACM Transactions on Computational Logic*, 10(2), 2009.
- [11] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A Metalanguage for interactive proof in LCF. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130. ACM, 1978.
- [12] R. Hinze, A. Löh, and B. C. D. S. Oliveira. "Scrap Your Boilerplate" Reloaded. In *FLOPS'06: Proceedings of Functional and Logic Programming, 8th International Symposium*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.
- [13] P. Johann and E. Visser. Strategies for Fusing Logic and Control via Local, Application-Specific Transformations. Technical Report UU-CS-2003-050, Department of Information and Computing Sciences, Utrecht University, 2003.
- [14] C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computations and deductions. Festschrift for Peter Andrews. Available online at <http://www.lix.polytechnique.fr/~fkirchner/data/festschrift2008.pdf>, 2008.
- [15] H. Kirchner and I. Gnaedig. Termination and normalisation under strategy Proofs in ELAN. *ENTCS*, 36, 2000.
- [16] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006.
- [17] R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal Logic and Algebraic Programming*, 54(1–2):1–64, 2003.
- [18] R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI'03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37. ACM Press, 2003.
- [19] R. Lämmel, S. Thompson, and M. Kaiser. Programming errors in traversal programs over structured data. *ENTCS*, 2008. Selected papers of LDTA 2008, Volume number not yet known.
- [20] R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming. Unpublished manuscript available online <http://www.program-transformation.org/Transform/TheEssenceOfStrategicProgramming>, 2002.
- [21] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *PADL'02: Proceedings of Practical Aspects of Declarative Programming*, volume 2257 of *LNCS*, pages 137–154. Springer, Jan. 2002.
- [22] N. Martí-Oliet, J. Meseguer, and A. Verdejo. A Rewriting Semantics for Maude Strategies. *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008*, 238(3):227–247, 2009.
- [23] A. P. Martin, P. H. B. Gardiner, and J. Woodcock. A tactic calculus — abridged version. *Formal Aspects of Computing*, 8(4):479–489, 1996.
- [24] T. Nipkow. Term Rewriting and Beyond – Theorem Proving in Isabelle. *Formal Aspects of Computing (1989) 1*, pages 320–338, 1989.
- [25] T. Nipkow. More Church-Rosser Proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001.
- [26] T. Nipkow, L. Paulson, and M. Wenzel. Isabelle/HOL – A Proof Assistant for Higher-Order Logic, volume 2283 of *LNCS*. Springer, 2002.
- [27] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a tactic language for refinement. *Formal Aspects of Computing*, 15(1):28–47, 2003.
- [28] A. Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260(1–2):165–207, 2001.
- [29] L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.
- [30] L. Paulson and T. Nipkow. Isabelle website, 2008. <http://isabelle.in.tum.de>.
- [31] F. Reig. Generic proofs for combinator-based generic programs. In *Trends in Functional Programming*, pages 17–32, 2004.
- [32] P. Torrini, C. Lueth, C. Maeder, and T. Mossakowski. Translating Haskell to Isabelle. In *Theorem Proving in Higher-Order Logic: Emerging Trends Proceedings, Uni Kaiserslautern - Computer Science TR 364/07*, pages 178–193, 2007.
- [33] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):152–190, 2003.
- [34] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In *Domain-Specific Program Generation, Dagstuhl Seminar, 2003, Revised Papers*, volume 3016 of *LNCS*, pages 216–238. Springer, 2004.
- [35] E. Visser and Z.-e.-A. Benaïssa. A Core Language for Rewriting. In *Second International Workshop on Rewriting Logic and its Applications (WRLA 1998)*, volume 15 of *ENTCS*. Elsevier Science Publishers, 1998.
- [36] E. Visser, Z. el Abidine Benaïssa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM, 1998.
- [37] J. Visser. *Generic Traversal over Typed Source Code Representations*. PhD thesis, University of Amsterdam, 2003.
- [38] D. Wasserrab, T. Nipkow, G. Snelling, and F. Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 345–362. ACM, 2006.

A. Solution of the exercise in Fig. 3

lemma stopbu_id.law: $stopbu\ s = id$