Under consideration for publication in J. Functional Programming

The Kansas University Rewrite Engine

A Haskell-Embedded Strategic Programming Language with Custom Closed Universes

NEIL SCULTHORPE, NICOLAS FRISBY and ANDY GILL∗
Information and Telecommunication Technology Center
The University of Kansas
(*e-mail:* {neil,nfrisby,andygill}@ittc.ku.edu)

Abstract

When writing transformation systems, a significant amount of engineering effort goes into setting up the infrastructure needed to direct individual transformations to specific targets in the data being transformed. Strategic and generic programming languages provide general-purpose infrastructure for this task, which the author of a transformation system can use for any algebraic data structure.

The Kansas University Rewrite Engine (KURE) is a typed strategic programming language, implemented as a Haskell-embedded domain-specific language. Or, viewed from another perspective, KURE is a Haskell library for generic programming. KURE is designed to support *typed* transformations over *typed* data, and the main challenge is how to make such transformations compatible with generic traversal strategies that should operate over *any* type. Compared to other Haskell libraries for generic programming that provide such type support, the distinguishing feature of KURE's solution is that the user can configure the behavior of traversals based on the *location* of each datum in the tree, beyond their behavior being determined by the *type* of each datum.

This article describes KURE's approach to assigning types to generic traversals, and the implementation of that approach. We also compare KURE, its design choices, and their consequences, with other approaches to strategic and generic programming.

1 Introduction

This article describes an approach for building rewrite engines over strongly typed data structures. Rewriting rules are often expressed as local correctness-preserving transformations, sometimes with preconditions and contextual requirements, such as lexical scope. A rewrite engine takes these local rewriting rules and applies them in systematic ways to achieve a global effect. Strategic programming (Visser, 2005), a style of programming that explicitly supports *programmable strategies* for composing rewriting rules, is one approach to structuring rewrite engines. The principal design decision in any *typed* strategic-programming implementation is how to specialize generic traversal strategies to operate over a typed syntax. This design decision becomes more challenging when we wish to support complex traversal strategies, such as potentially failing traversals, selective traversals, and traversals over mutually recursive data types.

∗ This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

The Kansas University Rewrite Engine (KURE) is a domain-specific language for typed strategic programming, and is the principal subject of this article. The KURE implementation began as a component of the Haskell-embedded HERA system (Gill, 2006), but was later abstracted out to a standalone Haskell library. Since then KURE has gone through several stages of development, with various experimental designs implemented. Two prior publications discuss KURE at earlier development stages: Gill (2009) described the first standalone version of KURE; Farmer *et al.* (2012) used an interim version of KURE as part of a larger system. The current implementation of KURE has similarities to existing Haskell libraries for generic programming, but has different engineering compromises regarding data-structure traversal. KURE also builds on many years of research into strategic programming, especially the work of Stratego (Visser, 2004; Bravenboer *et al.*, 2008) and Strafunski (Lämmel & Visser, 2002).

Specifically, the contributions of this article are:

- We present an approach of using *closed universes* to type generic traversals over mutually recursive data types (§3).
- We demonstrate how these universes can be used to define *statically selective traversals* (§4.3.2), and traversals whose behavior can depend not just on the type of the values being traversed, but also on their *location* in the data structure (§4.3.3).
- We describe a Haskell implementation of strategic programming that uses these universes to index generic traversals (§4).
- We augment the implementation with a user-defined *context*, and show how it can be maintained automatically within a generic traversal (§4.4).
- We demonstrate the viability of using KURE for realistic applications by presenting a case study of KURE's usage in the HERMIT project (§5).
- We compare the KURE approach to generic traversals with that of Stratego, and with those of the SYB and Uniplate generic-programming Haskell libraries (§6).
- We compare KURE's performance with SYB, TYB, and Uniplate, and explain the causes of the performance variations (§7).

2 Strategic and Generic Programming

Traditional *term rewriting* involves defining a set of rewriting rules over some object language, and applying those rules exhaustively to a term in that language. *Strategic programming* (Visser, 2005) builds on term rewriting by adding *programmable strategies* that allow the user to control when and where in a term rewriting rules are applied (Bravenboer *et al.*, 2008). Typically the object language is a programming or document-markup language, but the ideas generalize to any tree-structured data.

The term *generic programming* has multiple meanings (Gibbons, 2003), but in this article we always mean *datatype-generic programming* (also known as *polytypic programming*) (Hinze & Löh, 2007; Rodriguez Yakushev *et al.*, 2008; Hinze & Löh, 2009). This notion of generic programming involves defining functions that operate over typed data, but that operate based on the *shape* of the data rather than its *type*. Typically, *generic traversals* are used to navigate to a particular set of locations in a data type, and then a type-specific function is applied at those locations.

KURE is implemented as a Haskell library, providing rewriting strategies as Haskell combinators, and operating on Haskell data types. Hence, KURE can be viewed as either an embedded domain-specific language for strategic programming, or as a generic programming library.

In this section we will introduce the strategic programming paradigm by presenting Stratego (Visser, 2004; Bravenboer *et al.*, 2008), the most widely used strategic rewriting system, and then overview approaches to generic programming in Haskell. First however, we will introduce some terminology, and explain our notation.

2.1 Terminology and Notation

In §2.2 we will describe the Stratego language, and use Stratego syntax to do so. In §3 we will discuss assigning types to rewrites and strategies at a language-independent level, but use a Haskell-like notation for expressing type signatures. In §4 to §7 we present the KURE implementation, and give examples of KURE usage, and use Haskell code to do so. We make use of several (commonly used) language extensions that are not yet part of the Haskell 2010 standard, but are provided by the Glasgow Haskell Compiler. We also take the liberty of allowing type variables in top-level type signatures to scope over local type signatures, so that we can provide such type signatures as an aid to the reader (this could be achieved with a language extension and some additional syntactic clutter, but we believe that inhibits readability). In all sections, we typeset the program code $->$ and $=>$ as $\rightarrow$ and $\Rightarrow$, respectively, and in the Haskell code we typeset $\backslash$ as λ.

Throughout this paper we will be working with various tree-structured object languages. These object languages will be represented as typed algebraic data types, either in Stratego or in Haskell. In both cases, we typeset data-type constructors in sans-serif font. As the distinction between the semantics of different strategic/generic languages can be quite subtle, we need to introduce some precise terminology for discussing algebraic data types. We recommend that the reader does not try to internalize all these definitions immediately, but refers back to this section as needed.

- The *proper components* of a value x are the arguments to the constructor of x.
- The *components* of a value x are its proper components, as well as x itself.
- The *proper substructures* of x are the proper components of x, and those components' proper substructures.
- The *substructures* of x are its proper substructures, as well x itself.
- Two substructures are *independent* if neither is a substructure of the other.
- The *similarly typed* proper components/substructures of x are those that have the same type as x.
- The *maximal* elements of a set of substructures are those that are not a proper substructure of any other element.
- A *node* is a substructure that can be targeted by a rewriting rule.
- *Child nodes* are the maximal independent proper substructures of a node that are themselves also nodes.

Which substructures can be targets for a rewriting rule, and hence are considered to be nodes, varies between individual strategic/generic languages, so is not defined in general.

2.2 Stratego

Stratego is a strategic programming language designed to operate on an arbitrary object language. The Stratego language is part of the larger Stratego/XT toolkit (Visser, 2004; Bravenboer *et al.*, 2008), which provides additional tools such as parsers and pretty printers to convert between the object language and its representation as an algebraic data type within Stratego. Recently, Stratego has been integrated with the Eclipse integrated development environment, as part of the Spoofax Language Workbench (Kalleberg & Visser, 2007; Kats & Visser, 2010).

As an example (adapted from Bravenboer *et al.* (2008)), a simple object language of integer addition could be defined in Stratego as follows:

```
constructors
  Add : Exp * Exp  →  Exp
  Int  : INT          →  Exp
```

That is, an expression is either the addition of two expressions, or an integer. Integers, floats and strings are built-in types in Stratego. Although the constructors are assigned type signatures, Stratego is not a typed language — the Stratego compiler only checks that the arities of these constructor definitions match their usage (Bravenboer *et al.*, 2008).

Rewriting rules over this object language can be defined in the form $name : pat_1 \rightarrow pat_2$, where *name* is the name of the rule, pat_1 is a constructor pattern defining which nodes the rule can target, and pat_2 is the result of the rule. For example, the commutativity and left-unit laws of addition can be expressed as rewriting rules as follows:

```
rules
  commutativity : Add (e1, e2)  →  Add (e2, e1)
  leftUnit       : Add (Int (0), e)  →  e
```

Note that e, e_1 and e_2 are meta-language variables that can match any node.

Rewriting rules can either succeed or fail. The typical cause of failure is when pat_1 does not match the node that the rule is being applied to. Stratego also provides two built-in rewriting rules:

- *id* is the identity rewriting rule that always succeeds;
- *fail* is the always-failing rewriting rule.

As well as rewriting rules, Stratego also provides some built-in *strategies*, which provide a means of combining rewriting rules. Some representative strategies are as follows:

- $r_1\ ;\ r_2$ is *sequential composition*: apply r_1, then apply r_2, requiring both to succeed;
- $r_1 <\!+\ r_2$ is *deterministic choice* (or "catch"): apply r_1, but if it fails then apply r_2;
- $all(r)$ is a *shallow traversal*: apply r to all child nodes, requiring all to succeed.

Note that strategies (and rewriting rules) have a notion of a "current node", and that the $all\ (r)$ strategy shifts the current node to its children when applying r. In Stratego, a node's children are its proper components, excluding the built-in string, integer and float types.

Using these basic strategies, more complex strategies can be defined. For example, a strategy that catches a failed rewrite with the identity rewrite (and thus always succeeds) can be defined as follows:

strategies
  $try\ (r) = r <\!\!+ id$

More interestingly, we can define strategies that traverse the entire tree. For example, the following strategies apply a rewriting rule to all nodes in the tree, as either a top-down (pre-order) or bottom-up (post-order) traversal:

strategies
  $alltd\ (r)\ = r\ ;\ all\ (alltd\ (r))$
  $allbu\ (r) = all\ (allbu\ (r))\ ;\ r$

We call these *deep* traversal strategies as they recursively descend through the nodes of the tree. In contrast, we call *all* a *shallow* traversal strategy because it descends only to a node's children, and no further.

Strategies can be invoked from within a rule by enclosing them in angled brackets. For example, the right-unit law of addition can be defined by sequencing *commutativity* and *leftUnit*, and applying them to a node:

rules
  $rightUnit : e\ \rightarrow\ <commutativity\ ;\ leftUnit> e$

Using this programming idiom, several rewriting systems have been implemented on top of Stratego and related tools, including Stratego itself, optimizers, pretty printers, and COBOL engineering tools (Westra, 2001). We have only given a cursory overview of the essence of Stratego here, and the interested reader is referred to the Stratego literature (Visser, 2004; Bravenboer *et al.*, 2008; Kalleberg & Visser, 2007).

2.3 Selective Traversals

Stratego's shallow traversal *all* descends into every child node, and the deep traversals *alltd* and *allbu* descend into every node in the tree. While this is often the required behavior, it is sometimes desirable to exclude some nodes from a traversal. This could be for semantic reasons: for example, when performing single-variable substitution on a representation of the lambda calculus, a traversal should not descend past a rebinding of the variable being substituted (shadowing). Alternatively, it may be for performance reasons: excluding nodes that are known not to contain any substructures that will be modified by the traversal. This can significantly improve the efficiency of traversals, so much so that entire libraries (e.g. Alloy (Brown & Sampson, 2009); Uniplate (Mitchell & Runciman, 2007)) have been designed around enabling it.

A traversal that descends into some nodes but not others is called a *selective traversal*. Selectivity comes in two forms: *static selectivity*, where it is known at compile-time that certain nodes are never descended into, and *dynamic selectivity*, where whether to descend into a particular node is determined by a run-time predicate. Static selectivity is more efficient than dynamic selectivity, as it allows compiler optimizations to be applied, and avoids the need for the run-time checks.

2.4 Strategic and Generic Programming in Haskell

KURE was inspired by Strafunski (Lämmel & Visser, 2002), a previous attempt to host strategic programming in Haskell. Strafunski follows in the tradition of Stratego, but adds types to rewriting rules and strategies, and uses Scrap-Your-Boilerplate (SYB) (Lämmel & Peyton Jones, 2003) generic programming to implement generic traversals over arbitrary data types. We will not discuss the specific features and limitations of Strafunski in this article; instead we will discuss more generally the SYB approach to generic traversals.

As KURE developed as a Haskell library for strategic programming, it came to resemble some existing Haskell libraries for generic programming, to varying degrees. In this article we compare only to the most similar libraries: those that focus on traversing data rather than on reflecting the structure of types.

We extensively compare KURE to the conventional formulation of the SYB library (Lämmel & Peyton Jones, 2003), and to the Uniplate library (Mitchell & Runciman, 2007) in §6. These two are the best-maintained and well-documented of the traversal-centric libraries. Other similar libraries include Smash (Kiselyov, 2006), Alloy (Brown & Sampson, 2009), Compos (Bringert & Ranta, 2008), and Multiplate (O'Connor, 2011). We already mentioned Alloy for its emphasis on selective traversal. Multiplate has the most similar representation of traversal to KURE: its central value is a record of rewrites, which is nearly isomorphic to a KURE rewrite over a universe (see §3.2). The distinguishing features of Smash and Compos are not relevant to this article, so we will not discuss them here.

Our focus is on traversal-centric libraries for generic programming, but there are also alternative approaches that support functions beyond traversals. The primary line of research explores alternative ways to reflect the structure of types and thereby define extensible and generic functions. We refer to the interested reader to the surveys (Hinze & Löh, 2007; Rodriguez Yakushev *et al.*, 2008) for discussions of most such libraries before 2009. The subsequent libraries culminate in GHC Generics (Magalhães *et al.*, 2010), which situates the type reflection and extensibility in two corresponding GHC Haskell language features: type families and type classes, respectively.

Finally, motivated by educational goals, van Noort *et al.* (2008) have applied the modern generic foundations specifically to rewriting. Their key innovation is a lightweight intensional representation that enables, for example, pretty-printing of rules and also unification of rules' left-hand sides with input terms.

2.5 Related Work in Other Languages

Stratego-style systems have been implemented as extensions to mainstream languages such as Java (Balland *et al.*, 2008), and there have also been implementations of generic programming in Scala (Moors *et al.*, 2006; Oliveira & Gibbons, 2008). There have also been many systems for supporting the translation of internal representations of programs using strategic programming concepts. A number of such systems improve the ability to provide core language constructions, and support for user-visible extensibility. SugarJ (Erdweg *et al.*, 2011) allows libraries to augment Java with syntactical extensions, and includes support for using Stratego rules directly. SugarHaskell (Erdweg *et al.*, 2012) pushes these ideas into Haskell, and supports the declarative authoring of what are cur-

ZU064-05-FPR kure 22 November 2013 11:13

rently hand-written extensions, such as *Arrow* syntax (Paterson, 2001). With Fortified Macros (Culpepper & Felleisen, 2010; Culpepper, 2012), new syntax can be supported with improved support for error checking and better error messages, using pattern-match–style rewriting rules. KURE as a library has a fundamentally orthogonal concern: KURE is designed for rewriting *internal* syntax that is represented as an algebraic data structure.

Transformations systems have also been used as a basis for impressive language endeavors in the space of mechanized meta theory. For example, the K framework (Lazar *et al.*, 2012) has been used to provide a best-in-class operational semantics for C (Ellison & Roşu, 2012), and PLT Redex (Felleisen *et al.*, 2009) uses transformations as a basis for authoring operational semantics (Klein *et al.*, 2012). KURE has a significantly more modest goal: supporting the rewriting of tree-structured data using techniques from generic programming and design patterns from strategic programming.

3 Strongly Typed Strategic Programming

KURE is a strongly typed strategic programming language. In this section we consider the challenge of how to assign types to Stratego-style rewriting rules and strategies. The main challenge is how to assign types to generic traversal strategies that operate over an object language containing nodes of more than one type.

3.1 Typing Transformations and Rewrites

First, we make a distinction between rewriting rules that preserve the type of a node, and rewriting rules that can change the type of a node. We will refer to the former as *rewrites*, and the latter as *transformations*. In a strongly typed setting, a rewrite can be used to modify a node in the tree, whereas a transformation cannot because the resulting tree may not be type correct. However, a transformation can be used to project information from a node, or multiple transformations can be composed to form a rewrite.

Let the type $\mathbf{T}\ \sigma\ \tau$ denote a transformation from a node of type σ to a node of type τ. Then let $\mathbf{R}\ \tau$ denote a rewrite over a node of type τ, and let us define $\mathbf{R}$ to be a synonym for the special case of $\mathbf{T}$ where both parameters are the same:

$$\mathbf{R}\ \tau = \mathbf{T}\ \tau\ \tau$$

Using $\mathbf{R}$ and $\mathbf{T}$, it is straightforward to assign types to the following Stratego strategies:

id	$:: \mathbf{R}\ \tau$	-- identity rewrite
$fail$	$:: \mathbf{T}\ \sigma\ \tau$	-- failing transformation
$(;)$	$:: \mathbf{T}\ \rho\ \sigma \to \mathbf{T}\ \sigma\ \tau \to \mathbf{T}\ \rho\ \tau$	-- sequential composition
$(<\!+)$	$:: \mathbf{T}\ \sigma\ \tau \to \mathbf{T}\ \sigma\ \tau \to \mathbf{T}\ \sigma\ \tau$	-- deterministic choice

3.2 Typing Generic Traversals

Now consider assigning a type to Stratego's *all* strategy. If all nodes in the object language have the same type τ, then this is straightforward:

$$all :: \mathbf{R}\ \tau \to \mathbf{R}\ \tau$$

However, if the object language is represented by multiple algebraic data types, then this is inadequate. Consider the situation where the types of some (or all) of the children of a node differ from their parent. In that case, a rewrite of the above type could only apply its argument rewrite to children of the same type as the parent — a significant shortcoming.

An alternative would be to give *all* a more general type:

$$\mathit{all} :: \mathbf{R}\ \sigma \to \mathbf{R}\ \tau$$

Unfortunately, this does not relate σ and τ at all, and so *all* cannot use the argument rewrite in any meaningful way — *without run-time type comparisons*. This is the approach taken by Dolstra & Visser (2001); Dolstra (2001), who invent a language with a type system that includes run-time type-case internally inside traversal combinators. This is also essentially the approach taken by Strafunski using SYB, as we will discuss in §6.

Ideally, we would like generic traversal strategies to accept a *set* of distinctly typed rewrites as an argument, one for each child. However, this approach would require us to introduce sets of rewrites as a new first-class notion, and then define additional strategies to operate on these sets, and to combine them with our existing **R**s and **T**s. This is possible, indeed it is the approach taken by the Multiplate library (O'Connor, 2011), but it is not the approach that KURE takes. Instead, we reuse the **R** type as the single argument to *all*. The challenge is finding a type υ such that $\mathbf{R}\ \upsilon$ encodes a set of rewrites.

One such type is an algebraic (disjoint) *sum type*, also known as a *variant type* (Pierce, 2002, §11.9 & §11.10). If a node of type τ has n possible children with types $\tau_1, \tau_2, \ldots, \tau_n$, then we can define the sum type:

$$\upsilon = \tau_1 + \tau_2 + \ldots + \tau_n$$

and the type of *all* could then be:

$$\mathit{all} :: \mathbf{R}\ \upsilon \to \mathbf{R}\ \tau$$

A disadvantage of this type is that it admits non-type-preserving rewrites. This can be seen if we expand the **R** synonym:

$$\mathbf{R}\ \upsilon = \mathbf{T}\ \upsilon\ \upsilon = \mathbf{T}\ (\tau_1 + \tau_2 + \ldots + \tau_n)\ (\tau_1 + \tau_2 + \ldots + \tau_n)$$

That is, a rewrite could transform a child of one type into a child of another type. KURE then performs run-time checks during generic traversals, converting any non-type-preserving uses of *all* into failing rewrites. By comparison, Multiplate's set approach has more type precision, but leads to less composable transformations.

That said, the above type for *all* is not yet very composable — for example, the Stratego definitions of *alltd* and *allbu* (§2.2) would not type check, as υ and τ are not the same type. To address this, we generalize from a sum of child types to a sum of the types of all nodes in the tree. We call this generalized sum type a *universe type*. We then modify the type of *all* such that its argument and return types are rewrites over the same universe. Thus, if $\tau_1, \tau_2, \ldots, \tau_n$ are the types of all nodes in the tree, and letting $\upsilon = \tau_1 + \tau_2 + \ldots + \tau_n$, then we assign *all* the following type:

$$\mathit{all} :: \mathbf{R}\ \upsilon \to \mathbf{R}\ \upsilon$$

This type is composable, and leads to deep traversals of the same type. For example:

$alltd :: \mathbf{R}\ \upsilon \rightarrow \mathbf{R}\ \upsilon$
$alltd\ (r) = r\ ;\ all\ (alltd\ (r))$

A universe need not contain all node types in the tree; it could contain only a subset of the node types. In this case, *all r* could only traverse nodes whose types inhabit that universe, and could only target children whose types inhabit that universe. That is, the universe acts as an index defining a *statically selective traversal* (§2.3). Consequently, *all* is an *ad-hoc polymorphic* (Strachey, 2000) function: it exhibits non-uniform behavior when instantiated to different universe types.

Actually, KURE traversals are even more configurable than this. A traversal need not treat all nodes of the same type uniformly; it can treat them distinctly based on their location in the tree. This is achieved by using a universe containing multiple summands of the same type, one for each distinct location. We will demonstrate this in §4.3.3.

4 KURE Implementation

In this section we describe how the ideas in §3 are implemented as a Haskell-embedded domain-specific language. We begin by describing the main aspects of KURE that are required by most use cases: side-effecting rewrites (§4.1); strategies (§4.2); and generic traversals using a universe (§4.3.1). We then discuss more advanced aspects of KURE that may only be needed for some use cases: traversals over multiple universes (§4.3.2); traversals that distinguish nodes based on their location rather than their type (§4.3.3); support for maintaining a context during generic traversals (§4.4); and strategies and traversals involving change detection (§4.5).

4.1 Transformations, Rewrites and Side-Effects

The central decision of the KURE implementation is how to represent the **T** type. Haskell is a purely functional language, so simply defining **T** as a function,

type $T\ a\ b = a \rightarrow b$

would be inadequate, as it does not allow a transformation to perform any side effects. Side effects are an inherent part of strategic programming, most notably the side effects of *failure*, and *catching* failure. Additionally, specific use cases may need to support arbitrary side-effecting operations, such as fresh name generation.

In Haskell, the standard way of encoding side effects is to use a *monadic* (Wadler, 1992) structure. Furthermore, the KURE library needs to be applicable to different use cases, with arbitrary side effects, so the library design should not commit to any specific set of effects. Therefore, rather than using a concrete monad, transformations are parameterized over an arbitrary monad. This leads to the following implementation of **T**, where *m* is the monad parameter[1]:

type $T\ m\ a\ b = a \rightarrow m\ b$

[1] Actually, *T* needs to be a **newtype**, rather than a type synonym, because type synonyms cannot be partially applied, and to avoid ambiguous type class instances. However, we elide that detail in this article to avoid the syntactic clutter introduced by the data constructor.

```
class Functor m where
  (<$>) :: (a → b) → m a → m b

class Functor m ⇒ Applicative m where
  pure  :: a → m a
  (<*>) :: m (a → b) → m a → m b

class Applicative m ⇒ Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  fail   :: String → m a
```

Fig. 1. Monads in the Haskell standard library. As *fmap* will only be used in an infix position in this article, we present the *Functor* class using its infix synonym <$>.

As before, we will define rewrites as a synonym for a special case of transformations:

```
type R m a = T m a a
```

As is standard Haskell practice, the constraint that the parameter *m* be a monad will be specified on the operations over *T*, rather than on the *T* type itself.

However, an arbitrary monad does not support catching failure, and indeed it is somewhat of a historical accident that Haskell's *Monad* type class (Fig. 1) even supports throwing failures. As catching failure is necessary to encode the deterministic-choice strategy, KURE provides a subclass of *Monad* that provides a catching operation:

```
class Monad m ⇒ MonadCatch m where
  catchM :: m a → (String → m a) → m a
```

The purpose of the *String* argument in *fail* and *catchM* is to pass error messages. Using a *String* to record failure information is fairly crude, and we intend to introduce a dedicated exception data type in the next version of KURE. The KURE library uses *catchM*, and derived functions, extensively to provide informative error messages for failing transformations. These derived functions are also exported, so that KURE users can also insert error messages specific to their domain. However, to avoid cluttering this article, we will omit all uses of error-message–handling combinators from the code we present.

Finally, we note that while our implementation of *T* as a (partial) function may appear to restrict us to deterministic transformations, the monadic structure allows non-deterministic transformations to be encoded, either by using a monad that supports pseudo-random choice, or a list-like monad that computes multiple results.

4.2 Strategies

We now need to implement strategies over the *T* type. The structure of *T* fits several standard Haskell type classes, including *Monad* and *Arrow*, and thus by making *T* an instance of these classes, many of our desired strategies are then provided by established combinators on those classes. Using these type classes brings three main benefits. First, a substantial amount of implementation effort is saved by reusing existing combinators. Second, this provides familiarity to a Haskell programmer new to the strategic programming paradigm. Third, this generates many new strategic combinators, as any existing Haskell combinator over these type classes gives rise to a strategy. However, we note that use of these classes is not *necessary* — the strategy combinators could also be implemented directly without making *T* an instance of any type class.

The main type-class instances provided by KURE are presented in Fig. 2. These definitions are fairly standard, and contain two key ideas. First, *T m a* forms a monad that

```
instance Functor m ⇒ Functor (T m a) where
  (<$>) :: (b → d) → T m a b → T m a d
  f <$> t = λa → f <$> t a

instance Applicative m ⇒ Applicative (T m a) where
  pure :: b → T m a b
  pure b = λ_ → pure b

  (<*>) :: T m a (b → d) → T m a b → T m a d
  t1 <*> t2 = λa → t1 a <*> t2 a

instance Monad m ⇒ Monad (T m a) where
  return :: b → T m a b
  return b = λ_ → return b

  fail :: String → T m a b
  fail s = λ_ → fail s

  (>>=) :: T m a b → (b → T m a d) → T m a d
  t >>= f = λa → (t a >>= λb → (f b) a)

instance MonadCatch m ⇒ MonadCatch (T m a) where
  catchM :: T m a b → (String → T m a b) → T m a b
  catchM t1 t2 = λa → catchM (t1 a) (λs → (t2 s) a)

instance Monad m ⇒ Category (T m) where
  id :: T m a a
  id = return

  ( ∘ ) :: T m b d → T m a b → T m a d
  t2 ∘ t1 = λa → t1 a >>= t2

instance Monad m ⇒ Arrow (T m) where
  arr :: (a → b) → T m a b
  arr f = λa → return (f a)

  first :: T m a b → T m (a,z) (b,z)
  first t = λ(a,z) → (λb → (b,z)) <$> t a
```

Fig. 2. Main type class instances for *T*.

corresponds to applying the Reader monad transformer (Liang *et al.*, 1995) to the underlying monad *m*, where the environment comprises the argument node *a*. Second, *T m* forms the *Kleisli arrow* (Hughes, 2000) induced by the underlying monad *m*.

The identity and sequential-composition strategies are then provided by the *id* and ∘ methods of the *Category* class. However, to increase readability, this article uses *idR* and >>> as synonyms for specializations of those methods:

```
idR :: Monad m ⇒ R m a
idR = id

(>>>) :: Monad m ⇒ T m a b → T m b d → T m a d
t1 >>> t2 = t2 ∘ t1
```

Failure and catching are provided by the *fail* and *catchM* methods of the *Monad* and *MonadCatch* classes. These two methods do not match Stratego's *fail* and <+ strategies exactly because they also handle error messages, but it is straightforward to define variants that ignore the error messages, and hence do match Stratego's strategies exactly:

```
failT :: Monad m ⇒ T m a b
failT = fail "failT"

(<+) :: MonadCatch m ⇒ T m a b → T m a b → T m a b
t1 <+ t2 = catchM t1 (λ_ → t2)
```

New strategies can then be constructed in the same manner as in Stratego, for example:

```
tryR :: MonadCatch m ⇒ R m a → R m a
tryR r = r <+ idR
```

Note that the type signatures can be entirely inferred by a Haskell compiler, and so can be omitted by a KURE user if desired.

In practice, we have found that the *Monad* instance for transformations is extremely useful, as the monadic structure allows multiple transformations to be applied to the same node, with the choice of subsequent transformations depending on the results of prior transformations. We will give some examples of this in §5.4.

4.3 Universe-indexed Traversals

We now turn to the main implementation challenge: implementing generic traversals using universe types. This subsection is divided into three parts:

- §4.3.1 demonstrates the main idea, and gives an example of standard KURE usage using a single universe that treats all nodes of the same type equally.
- §4.3.2 explains how to support statically selective traversals by using multiple universes.
- §4.3.3 describes how to define traversals that distinguish between nodes of the same type, based on their location in the tree.

4.3.1 Walking the Tree

Recall that the key idea from §3.2 is for *all* to be ad-hoc polymorphic in its universe type. KURE implements this using the following type class, where *u* is the universe type:

```
class Walker u where
  allR :: MonadCatch m ⇒ R m u → R m u
```

Notice that the monad *m* is not a class parameter — instead the *allR* method is *parametrically polymorphic* (Strachey, 2000) in any monad *m*. This choice brings expressiveness benefits, as we will explain in §4.5.

The KURE library provides a suite of traversal strategies defined using *allR*. For example, Stratego's *alltd* strategy is defined as follows:

```
alltdR :: (Walker u, MonadCatch m) ⇒ R m u → R m u
alltdR r = r >>> allR (alltdR r)
```

Providing the *Walker* instance for each universe is the responsibility of the KURE user, and this is how the user customizes the traversal behavior for each universe. To demonstrate how a KURE user would define a universe and *Walker* instance, we will use the following representation of an object language as a running example. To avoid confusion between KURE implementation code and user code, we enclose the latter in boxes.

```
data Prog = PrgCons Decl Prog | PrgNil
data Expr = Lit Literal | Var Name | Append Expr Expr | Let Decl Expr
data Decl = Decl Name Expr

type Name    = String
type Literal = String
```

This models a small expression language with non-recursive let-bindings for building strings. We deliberately designed this example to use strings in two semantically different ways (as variable names and as literals), so that we can later demonstrate how KURE can

```
class Injection a b where
  inject  :: a → b
  project :: b → Maybe a

injectT :: (Monad m, Injection a u) ⇒ T m a u
injectT = arr inject

projectT :: (Monad m, Injection a u) ⇒ T m u a
projectT = λu → case project u of
                  Just a   → return a
                  Nothing → fail "projectT"

promoteT :: (Monad m, Injection a u) ⇒ T m a b → T m u b
promoteT t = projectT >>> t

promoteR :: (Monad m, Injection a u) ⇒ R m a → R m u
promoteR r = projectT >>> r >>> injectT

extractR :: (Monad m, Injection a u) ⇒ R m u → R m a
extractR r = injectT >>> r >>> projectT
```

Fig. 3. Promotion and extraction combinators.

distinguish between the two during traversals. However, we recommend that a KURE user tries to avoid using the same data type in semantically different ways.

The first task for the KURE user is to define a universe. Let us assume she wants to traverse programs, declarations and expressions, but not variable names or string literals. That is, she wants to define a statically selective traversal that omits strings. (In Haskell, *String* is implemented as a synonym for a singly linked list of characters.) She begins by defining a universe consisting of the three node types she wishes to traverse:

```
data U = UP Prog | UD Decl | UE Expr
```

A common need when working with universe types is to inject nodes into a universe, or project them out of a universe. To aid this, KURE provides a type class *Injection*, along with several conversion functions for converting transformations and rewrites to and from universe types (Fig. 3). The important point about these conversion functions is that a failed projection will result in a failing transformation. Consequently, a rewrite over a universe type that is applied to a node will fail if it does not preserve the node type (this provides the run-time check mentioned in §3.2). Injection instances are straightforward to define, as exemplified by the following instance for *Expr*:

```
instance Injection Expr U where
  inject = UE

  project (UE expr) = Just expr
  project _         = Nothing
```

Having defined a universe and *Injection* instances, the user can now declare a corresponding *Walker* instance. This requires the user to decide which substructures should be considered child nodes, but the definition of *allR* is otherwise systematic. The essence of the definition is as follows. First, deconstruct the node by pattern matching. Then, for each child node, extract a specialized rewrite from the argument rewrite (which operates over the universe type), and apply that rewrite to child. Finally, reconstruct the node by reapplying the constructor(s). This is easiest to understand by example:

```
instance Walker U where
  allR :: MonadCatch m ⇒ R m U → R m U
  allR r = λu → case u of
                  UP p → UP <$> allProg p
                  UD d → UD <$> allDecl d
                  UE e → UE <$> allExpr e
    where
      allProg :: Prog → m Prog
      allProg PrgNil            = pure PrgNil
      allProg (PrgCons d p)     = PrgCons <$> (extractR r) d <*> (extractR r) p

      allDecl :: Decl → m Decl
      allDecl (Decl n e)        = Decl <$> pure n <*> (extractR r) e

      allExpr :: Expr → m Expr
      allExpr (Var n)           = Var    <$> pure n
      allExpr (Lit l)           = Lit    <$> pure l
      allExpr (Append e0 e1)    = Append <$> (extractR r) e0 <*> (extractR r) e1
      allExpr (Let d e)         = Let    <$> (extractR r) d  <*> (extractR r) e
```

Observe that the traversal does not descend below variable or literal nodes. These are the non-systematic parts of the definition that had to be decided by the user. These user decisions prevent this definition of *allR* from being generated automatically. Additionally, we will later (§4.4) augment the *Walker* class to support maintaining a context, which will also interfere with automatic generation of *allR* definitions.

However, it is possible to generate a *Walker* instance for a particular formulaic traversal. Indeed, an earlier version of KURE (Gill, 2009) used Template Haskell (Sheard & Peyton Jones, 2002) to generate both universe types and corresponding *Walker* instances. Those instances treated every proper component of a node as a child, and performed no context updates. In the case of this example string language, that would mean treating the *String* representation of variables and literals as the sole child of a variable or literal node, and the head and tail of every *String* would be children of that *String*. Thus, for example, deep traversals built from such a definition would visit every character of every variable name and string literal.

4.3.2 Alternative Universes

We now consider using multiple universes to provide multiple statically selective traversals over the same object language. Imagine that the user decides she wants a traversal that descends into strings, but wants to keep the existing capability to perform traversals that do not descend into strings. She begins by defining a new universe,

```
data U1 = UP1 Prog | UD1 Decl | UE1 Expr | US1 String
```

and declaring corresponding *Injection* instances. She then defines a *Walker* instance for U_1. This is mostly the same as the instance for U, but with three additions to *allR*. First, there is an additional local function *allString*:

```
allString :: String → m String
allString []       = pure []
allString (x : xs) = (:) <$> pure x <*> (extractR r) xs
```

Second, the Var and Lit clauses of *allExpr* are modified such that the traversal descends into the string:

```
allExpr (Var n) = Var <$> (extractR r) n
allExpr (Lit l)  = Lit  <$> (extractR r) l
```

Third, there is an extra case alternative when pattern matching on the universe:

```
US₁ s → US₁ <$> allString s
```

This may seem like a lot of code duplication, but in practice *allR* definitions operating over the same object language can be built out of a set of reusable *congruence combinators* (Visser, 2004), which factor out the duplication. We will demonstrate this in §5.2.

When performing a traversal, the user now has the option of which universe to use; that is, she can choose between statically selective traversals. We will give some examples of this in §4.4 and §5.4.

4.3.3 Locations not Types

Thus far we have treated all nodes of the same type in the same manner. However, a KURE traversal can distinguish between nodes based on their location, even if their types are the same. For example, imagine the user desired the U_1 traversal to descend into string literals but not into variables names, despite them both having type *String*. This could be achieved by changing the Var clause of *allExpr* to the following:

```
allExpr (Var n) = Var <$> pure n
```

Alternatively, if the user wished to support both behaviors, she could define a new universe with a *Walker* instance defined in this way. She could also define a universe for traversals that descend into variables but not literals.

More challenging is defining a traversal that descends into multiple nodes of the same type, yet treats them distinctly based on their location. For example, imagine the (contrived) situation where the user wants to apply a rewrite that operates on substrings of literals, and another that operates on substrings of variable names, during the same traversal.

The first step is to define **newtype**s to represent names and literals, thereby providing a type distinction between them:

```
newtype StringLit    = StringLit Literal
newtype StringName = StringName Name
```

Any rewrites that should succeed for only one of *Literal* or *Name* should then be defined over these types, rather than over *String*.

The user then defines a universe, with separate summands for names and literals:

```
data U₂ = UP₂ Prog | UD₂ Decl | UE₂ Expr | UL₂ Literal | UN₂ Name
```

When defining *Injection* instances for U_2, we use the *StringLit* and *StringName* types, rather than *String*. This is necessary, as otherwise there would be two overlapping *Injection* instances for *String*, which would be ambiguous. The instance for *StringName* is as follows (a similar instance is defined for *StringLit*):

```
instance Injection StringName U₂ where
  inject (StringName name) = UN₂ name

  project (UN₂ name) = Just (StringName name)
  project _          = Nothing
```

As we have not defined an *Injection* instance for *String*, we are not able to use *extractR* as directly as before in the definition of *allR*. Instead, we define two variants of *extractR*, one for names and one for literals, which add and remove the **newtype** wrappers:

```
extractRname :: Monad m ⇒ R m U₂ → R m Name
extractRname r = λn → unName <$> (extractR r) (StringName n)
  where
    unName (StringName n) = n

extractRlit :: Monad m ⇒ R m U₂ → R m Literal
extractRlit r = λl → unLit <$> (extractR r) (StringLit l)
  where
    unLit (StringLit l) = l
```

We then use these functions to define the Var and Lit clauses of *allExpr*, as well as two new local functions, *allName* and *allLit*:

```
allExpr (Var n) = Var <$> (extractRname r) n
allExpr (Lit l)  = Lit  <$> (extractRlit r) l

allName :: Name → m Name
allName []       = pure []
allName (x : xs) = (:) <$> pure x <*> (extractRname r) xs

allLit :: Literal → m Literal
allLit []       = pure []
allLit (x : xs) = (:) <$> pure x <*> (extractRlit r) xs
```

A traversal using the U_2 universe will now descend into both names and literals, but, for example, will only apply a rewrite of type *R m StringLit* to substrings of literals, not to substrings of names.

4.4 Maintaining a Context

It is often useful for transformations to have access to a *context*, and for this context to be updated automatically during generic traversals. The motivating example is storing the definitions of variable bindings in scope (when working with an object language that has variable bindings), to support local *fold/unfold* (Burstall & Darlington, 1977) transformations. One way to encode a context would be to constrain m to be a Reader monad (Liang *et al.*, 1995), but, for subtle reasons that we will discuss in §4.5, it is beneficial to keep the context separate from the monad. Instead, we make the context type (c) an additional parameter of T, and make the context available as an argument to each transformation:

```
type T c m a b = c → a → m b
```

The type-class instances from Fig. 2 (§4.2) need to be updated accordingly, but these changes are fairly minor. The *Monad* family of instances are adapted by applying another Reader monad transformation, where the new environment is the context c. Likewise, the *Arrow* family of instance are adapted by applying the Reader arrow transformation for the environment c. Note that the monadic instances already had an environment (the argument node), and so now the environment is both the context *and the argument node*. This contrasts with the arrow instances, where the environment is *only the context*.

As an example, we will consider adding a context to our string expression language. Let us assume the desired context is a list of all declarations currently in scope:

```
type Context = [Decl]
```

Having these declarations available allows, for example, variables to be inlined locally. A rewrite to perform such inlining can be defined as follows[2]:

```
inline :: Monad m ⇒ R Context m Expr
inline = λc e → case e of
                        Var n → lookupName n c
                        _     → fail "not a Var"
  where
    lookupName :: Monad m ⇒ Name → Context → m Expr
    lookupName _ []              = fail "Name not found"
    lookupName n (Decl n′ e : c) = if n == n′
                                     then return e
                                     else lookupName n c
```

We now need to consider how to correctly maintain the context during generic traversals. The key idea is to define all generic traversals in terms of *allR*. If we can ensure that *allR* performs all desired context updates, then all generic traversals will update the context. Furthermore, the definition of *allR* will be the *only* place that the context is modified — all other strategy definitions will only read the context.

To enable this, we add the context to the *Walker* class as an additional class parameter:

```
class Walker c u where
  allR :: MonadCatch m ⇒ R c m u → R c m u
```

[2] We assume no variable shadowing for simplicity; more realistic object languages would require a check for variable capture.

It is important that, unlike the monad *m*, the context be a class parameter rather than parameter of the *allR* method. This is because we need to instantiate the context to a concrete data type so that we can modify it during the traversal.

As an example, consider the following representative fragment of the modified *Walker* instance for the *U* universe (from §4.3.1):

```
instance Walker Context U where
  allR :: MonadCatch m ⇒ R Context m U → R Context m U
  ...
    allExpr c (Append e0 e1) = Append <$> (extractR r) c e0 <*> (extractR r) c e1
    allExpr c (Let d e)      = Let    <$> (extractR r) c d  <*> (extractR r) (d : c) e
  ...
```

Observe that the context is passed as an argument to the rewrites being applied to the children, and that in the case of the let-body, the context is updated by adding the let-bound declaration. We reiterate that because this choice of context update is made by the user, it is not possible to mechanically derive this *Walker* instance.

It is now possible to define deep traversals that make use of the context. For example, the following traversal will inline all variable occurrences in an expression:

```
inlineAll = alltdR (tryR (promoteR inline))
```

(The use of *promoteR* is necessary because *inline* is a rewrite over *Expr*, whereas *alltdR* expects a rewrite over a universe as its argument.)

As *inlineAll* has not been ascribed a type signature, its type is inferred to be polymorphic in its universe type. The traversal can thus be instantiated to whichever universe is desired; that is, the user can choose between statically selective traversals. Variables only occur as expressions, not within literals or names, so it will be more efficient to instantiate *inlineAll* to the *U* universe than the U_1 universe. This instantiation can be achieved by either at the use site of *inlineAll*, or by ascribing a type signature to its definition:

```
inlineAll :: MonadCatch m ⇒ R Context m U
```

4.5 Change Detection and Monad Transformers

When defining a strategy, it can be useful to be able to determine not just whether a rewrite succeeded, but whether it actually modified the node it was applied to. An early implementation of KURE included an identity-detection mechanism that allowed identity rewrites to be caught using various combinators (Gill, 2009). However, this was a rather fragile approach, and relied on the KURE user to manually invoke a function called *transparently* on rewrites that were known not to modify a node (such as *idR*). It also led to transformations no longer satisfying various expected properties, interfering with the usage of standard Haskell structures. For example, an *Arrow* instance for the *T* type would have violated the arrow laws, and so *Arrow* was not used.

The current version of KURE opts for a simpler, more robust approach. It supports an (optional) convention whereby rewrites that do not modify the node should fail, and that repeated application of a rewrite should fail after a finite number of iterations. When following this convention, a rewrite that can result in an identity rewrite (such as *tryR*) may be used, but only as a sub-component of a rewrite that is guaranteed to either make a change or fail. For example, the following *repeatR* strategy recursively applies a rewrite until it fails, returning the result before the failure:

$repeatR :: MonadCatch\ m \Rightarrow R\ c\ m\ a \rightarrow R\ c\ m\ a$
$repeatR\ r = r >\!\!>\!\!> tryR\ (repeatR\ r)$

To support this convention further, KURE provides a monad transformer *AnyR*, which can be used to convert a rewrite defined for an arbitrary monad into a rewrite that succeeds if *at least one* sub-rewrite succeeds, converting any failures into identity rewrites. We omit the implementation details, but the main idea is that *AnyR* is a writer monad transformer over the monoid of Boolean disjunction, with the Boolean value representing whether at least one sub-rewrite has succeeded. The transformer is used by applying a lifting function to the sub-rewrites that are allowed to fail, and a lowering function to the composite rewrite that is required to contain at least one internal success:

$liftAnyR\quad :: MonadCatch\ m \Rightarrow R\ c\ m\ a \rightarrow R\ c\ (AnyR\ m)\ a$
$lowerAnyR :: Monad\ m \Rightarrow R\ c\ (AnyR\ m)\ a \rightarrow R\ c\ m\ a$

For example, a variant of $>\!\!>\!\!>$ that succeeds if *either* rewrite succeeds can be defined using this transformer as follows:

$(>\!+\!>) :: MonadCatch\ m \Rightarrow R\ c\ m\ a \rightarrow R\ c\ m\ a \rightarrow R\ c\ m\ a$
$r_1 >\!+\!> r_2 = lowerAnyR\ (liftAnyR\ r_1 >\!\!>\!\!> liftAnyR\ r_2)$

Note that this differs from $tryR\ r_1 >\!\!>\!\!> tryR\ r_2$, which will always succeed.

As another example, we can define a variant of the top-down traversal *alltdR* that succeeds if the rewrite succeeds at *any* node in the tree:

$anytdR :: (Walker\ c\ u, MonadCatch\ m) \Rightarrow R\ c\ m\ u \rightarrow R\ c\ m\ u$
$anytdR = lowerAnyR \circ alltdR \circ liftAnyR$

For this monad-transformer technique to work in combination with generic traversal strategies such as *alltdR*, it is essential that the *allR* class method be parametrically polymorphic in its monad parameter. If the monad was a class parameter, then the *Walker* instance would be specialized to that particular monad, and could not be used with other monads, including monads generated by the *AnyR* transformer. This is the principal reason why the monad is not a class parameter, and, consequently, why the monad and context are kept separate.

This monad-transformer technique is taken from the SYB library, and KURE uses it extensively to define a suite of generic traversal combinators. To give a flavor of the other monad transformers that KURE uses, we list the *shallow* traversal combinators that KURE provides, each defined using a different transformer in terms of *allR*:

- $anyR :: (Walker\ c\ u, MonadCatch\ m) \Rightarrow R\ c\ m\ u \rightarrow R\ c\ m\ u$
  Apply a rewrite to all children, succeeding if any succeed.
- $oneR :: (Walker\ c\ u, MonadCatch\ m) \Rightarrow R\ c\ m\ u \rightarrow R\ c\ m\ u$
  Apply a rewrite to the first child for which it can succeed.

- *allT* :: (*Walker c u*, *MonadCatch m*, *Monoid b*) ⇒ *T c m u b* → *T c m u b*
  Apply a transformation to all children, succeeding if all succeed, and combine the results in a monoid.
- *oneT* :: (*Walker c u*, *MonadCatch m*) ⇒ *T c m u b* → *T c m u b*
  Apply a transformation to the first child for which it can succeed, returning the result.

The KURE library defines a number of deep generic traversal along similar lines. We will present some examples of these in Fig. 6 in §5.4.

5 Case Study: HERMIT

The principal use-case of KURE to date has been the HERMIT package (Farmer *et al.*, 2012; Sculthorpe *et al.*, 2013), a toolkit for the interactive transformation of Haskell programs inside the Glasgow Haskell Compiler (GHC). Indeed, much of the development of KURE has been driven by the needs of the HERMIT project. As this is a "realistic" use of KURE, we will now present HERMIT as a case study of KURE usage.

HERMIT and GHC are systems undergoing active development, and the details of their implementation, and the use of KURE, continue to fluctuate. Consequently, we will not attempt to present the exact HERMIT source code in this section. Instead, we will give a simplified presentation that demonstrates the key points of KURE usage in this project. As previously, we will enclose in boxes any code that is not provided by the KURE library.

5.1 GHC Core Universes

HERMIT uses KURE to manipulate GHC's internal intermediate language, GHC Core. GHC Core is an implementation of System F_C, which is System F (Girard, 1972; Reynolds, 1974) extended with let-binding, constructors, and first-class type equalities (Sulzmann *et al.*, 2007). KURE is used for rewriting nodes in the GHC Core abstract syntax tree, and for projecting information from nodes using transformations.

A representative subset of GHC Core is shown in Fig. 4. The majority of HERMIT's transformations and rewrites act on *Program*, *Bind*, *Expr* and *Alt* nodes, while only a few act on *Coercion* and *Type* nodes. Observe that *Bind*, *Expr* and *Alt* nodes are mutually recursive, but that neither they nor *Program* nodes appear as substructures of *Coercion* or *Type* nodes. Consequently, a statically selective traversal that omits *Coercion* and *Type* nodes will not change the semantics of a traversal that only acts on the former set of nodes, but will yield significant efficiency gains by avoiding unnecessarily traversing *Coercion* and *Type* nodes. To support this, HERMIT defines two universes:

```
data Core   = CProg Program | CBind Bind | CDef Def | CExpr Expr | CAlt Alt
data CoreTC = CCoercion Coercion | CType Type | CCore Core
```

The *CoreTC* universe is defined slightly differently to our previous universes, as it includes the *Core* universe as a summand. This definition is isomorphic to the usual enumeration of nodes, but allows for more code reuse.

There is one other point of interest about the *Core* universe. The type *Def* is not part of GHC Core, but is an auxiliary type that HERMIT introduces:

```
type Program   = [Bind]
data Bind      = NonRec Var Expr | Rec [(Var, Expr)]
data Expr      = Var Var | Lit Literal
               | App Expr Expr | Lam Var Expr
               | Let Bind Expr | Case Expr Var Type [Alt]
               | Cast Expr Coercion | Type Type
type Alt       = (AltCon, [Var], Expr)
data Coercion  = CoVar Var | Refl Type | Sym Coercion | Trans Coercion Coercion | ...
data Type      = TyVar Var | Fun Type Type | ForAll Var Type | ...
```

Fig. 4. A representative subset of GHC Core.

```
data Def = Def Var Expr
```

This is isomorphic to the type $(Var, Expr)$ that actually appears inside the Rec constructor of the *Bind* data type. The idea is that the Rec constructor should be considered to have n children of type *Def*, where n is the length of the list. HERMIT introduces the *Def* type rather than using the actual type $(Var, Expr)$ for clarity, rather than out of necessity. However, if there were other semantically distinct occurrences of $(Var, Expr)$ within the data structure, then introducing this new node type *Def* would be necessary to allow KURE to distinguish between them during generic traversals (in the same way that we needed to introduce *StringName* and *StringLit* in §4.3.3). In general, a summand of a KURE universe need not match the node type exactly, but must only be a *retraction* of that type.

5.2 Congruence Combinators

Rather than define *Walker* instances for our two universes directly, we shall first define *congruence combinators* (Visser, 2004) for the nodes in our tree. Congruence combinators provide an abstract way of traversing a node that factors out the commonality of different statically selective traversals. We will then define the *Walker* instances in terms of the congruence combinators, thereby avoiding code duplication.

Each congruence combinator is specialized to a single node constructor, and takes as arguments transformations to apply to each of the node's children, and a function to combine the results. For example, consider the following congruence combinator for App nodes:

```
appT :: Monad m ⇒ T c m Expr a1 → T c m Expr a2 → (a1 → a2 → b) → T c m Expr b
appT t1 t2 f = λc expr → case expr of
                              App e1 e2 → f <$> t1 c e1 <*> t2 c e2
                              _         → fail "not an App"
```

That is, an App node should be traversed by applying the argument transformations t_1 and t_2 to the child nodes e_1 and e_2, and then combining the results by mapping the function f over them. If the node is not an App, then the transformation should fail.

A useful specialization of a congruence combinator is when the argument transformations are rewrites, and the function to combine the results is the node constructor:

```
appR :: Monad m ⇒ R c m Expr → R c m Expr → R c m Expr
appR r1 r2 = appT r1 r2 App
```

This is essentially a version of *allR* specialized to App nodes, a point that will be important later. Note however that because the number and type of its child nodes are known, *appR* can take one argument rewrite for each child, with each rewrite matching the type of the child, rather than taking a single rewrite over the universe type as is done by *allR*.

The *appT* congruence combinator did not modify the context, and hence was polymorphic in its context parameter *c*. However, in general, if traversing the node should cause a context update, then the congruence combinator for that node should update the context accordingly. For example, the HERMIT context (*HermitC*) stores the set of variable bindings in scope (among other things). Thus, a congruence combinator for Let nodes needs to add the bindings to the context when traversing into the let-body:

```
letT :: Monad m ⇒
     T HermitC m Bind a1 → T HermitC m Expr a2 → (a1 → a2 → b) → T HermitC m Expr b
letT t1 t2 f = λc expr → case expr of
                           Let bds e → f <$> t1 c bds <*> t2 (addBindings bds c) e
                           _         → fail "not a Let"
addBindings :: Bind → HermitC → HermitC
  -- definition not given
```

As a more interesting example, consider the Rec node. There are two points of interest here. First, note that the substructures of Rec are a list of $(Var, Expr)$ pairs, but we have decided to consider the children of Rec to be *Def* nodes. Therefore we have to convert between the two representations within the congruence combinator. Second, the semantics of recursive binding groups are such that each binding is in scope for all other bindings in the group, and so all bindings need to be added to the context when we descend into any individual *Def*. This leads to the following congruence combinator:

```
recT :: Monad m ⇒ T HermitC m Def a → ([a] → b) → T HermitC m Bind b
recT t f = λc bind → case bind of
  Rec bds → f <$> mapM (λ(v,e) → t (addBindings (Rec bds) c) (Def v e)) bds
  _       → fail "not a Rec"
```

While congruence combinator definitions are fairly systematic, they do require some semantic knowledge of the object language. For example, consider the top-level *Program* type, which is a list of bindings. Each binding in the list could be considered a child of a *Program* node, much like Rec nodes. However, another interpretation could be that a (non-empty) list has exactly two children: the binding group at the head of the list, and another program making up the tail of the list. The latter interpretation is the one HERMIT uses, as that matches the scoping behavior of binding groups in GHC Core.

We conclude this subsection by recommending that the KURE user define congruence combinators for all nodes that she may wish to traverse. This may seem like a lot of work, but our experience with HERMIT has been that there is a significant pay-off in the simplicity with which subsequent transformations can be defined. Furthermore, this allows the context updates to be localized to one place. Any *Walker* instances that require the same context updates can be defined using these congruence combinators (see §5.3), and any node-specific transformations can also use the congruence combinators to apply transformations to descendant nodes — in both cases the context will be correctly updated while doing so. This localization makes it less likely that any contextual updates will be accidentally omitted by the programmer, and also aids code maintenance.

5.3 Walker Instances

Once a complete set of congruence combinators and *Injection* instances have been defined for the nodes of the object language, then defining *Walker* instances is straightforward and systematic. For example, the instances for the *Core* and *CoreTC* universes are presented in Fig. 5. The key point is that each congruence combinator is given *extractR r* as an argument for each child node that should be targeted, and *idR* as an argument for each child node that should not be targeted. The main difference between the two instances is in *allRexpr*, as several of the *Expr* nodes have *Type* or *Coercion* children (you may find it helpful to refer back to Fig. 4). Additionally, the *CoreTC* instance has functions for traversing *Type* and *Coercion* nodes, which were not needed in the *Core* case. In both cases, *Var*, *Literal* and *AltCon* nodes are not traversed.

Notice that having defined the context updates within the congruence combinators, we do not need to update it within each *Walker* instance. Furthermore, if HERMIT later adds more universes, it will be possible to define the corresponding *Walker* instances using the existing congruence combinators.

5.4 Example Rewrites and Transformations

We will now give some examples of HERMIT rewrites and transformations. We begin with a rewrite that converts the application of a lambda to a non-recursive let binding (a form of β-reduction that preserves sharing). This can be defined succinctly as follows:

```
appLamToLet :: Monad m ⇒ R c m Expr
appLamToLet = do App (Lam v e₂) e₁ ← idR
                 return (Let (NonRec v e₁) e₂)
```

This definition exploits Haskell's monadic **do**-notation to implicitly handle failure: if the argument node does not match the pattern App (Lam v e_2 e_1) then this rewrite will fail, invoking the *fail* method of the underlying monad *m*. This provides a concise means of expressing rewriting rules while remaining within Haskell's type system.

To apply this rewrite throughout the tree, we promote it to operate on the *Core* universe, and apply the *anytdR* traversal strategy:

```
appLamsToLets :: (Walker c Core, MonadCatch m) ⇒ R c m Core
appLamsToLets = anytdR (promoteR appLamToLet)
```

That is, this rewrite will convert any applications of lambdas in the tree to let expressions, and will succeed if there is at least one such conversion. Using *alltdR* rather than *anytdR*

```
instance Walker HermitC Core where
  allR :: MonadCatch m ⇒ R HermitC m Core → R HermitC m Core
  allR r  =  promoteR allRprog <+ promoteR allRbind <+ promoteR allRdef
          <+ promoteR allRalt    <+ promoteR allRexpr
    where
      allRprog  ::  R HermitC m Program
      allRprog  =  progNilR <+ progConsR (extractR r) (extractR r)

      allRbind  ::  R HermitC m Bind
      allRbind  =  nonRecR (extractR r) <+ recR (extractR r)

      allRdef   ::  R HermitC m Def
      allRdef   =  defR idR (extractR r)

      allRalt   ::  R HermitC m Alt
      allRalt   =  altR idR idR (extractR r)

      allRexpr  ::  R HermitC m Expr
      allRexpr  =  varR idR <+ litR idR <+ appR (extractR r) (extractR r)
                <+ lamR idR (extractR r) <+ letR (extractR r) (extractR r)
                <+ caseR (extractR r) idR idR (extractR r)
                <+ castR (extractR r) idR <+ typeR idR


instance Walker HermitC CoreTC where
  allR :: MonadCatch m ⇒ R HermitC m CoreTC → R HermitC m CoreTC
  allR r  =  promoteR allRprog       <+ promoteR allRbind <+ promoteR allRdef
          <+ promoteR allRalt        <+ promoteR allRexpr
          <+ promoteR allRcoercion <+ promoteR allRtype
    where
      allRprog      ::  R HermitC m Program
      allRprog      =  progNilR <+ progConsR (extractR r) (extractR r)

      allRbind      ::  R HermitC m Bind
      allRbind      =  nonRecR (extractR r) <+ recR (extractR r)

      allRdef       ::  R HermitC m Def
      allRdef       =  defR idR (extractR r)

      allRalt       ::  R HermitC m Alt
      allRalt       =  altR idR idR (extractR r)

      allRexpr      ::  R HermitC m Expr
      allRexpr      =  varR idR <+ litR idR <+ appR (extractR r) (extractR r)
                    <+ lamR idR (extractR r) <+ letR (extractR r) (extractR r)
                    <+ caseR (extractR r) idR (extractR r) (extractR r)
                    <+ castR (extractR r) (extractR r) <+ typeR (extractR r)

      allRcoercion  ::  R HermitC m Coercion
      allRcoercion  =  coVarR idR <+ reflR (extractR r) <+ symR (extractR r)
                    <+ transR (extractR r) (extractR r)

      allRtype      ::  R HermitC m Type
      allRtype      =  tyVarR idR <+ FunR (extractR r) (extractR r)
                    <+ forAllR idR (extractR r)
```

Fig. 5. *Walker* instances for the *Core* and *CoreTC* universes.

would not have the desired affect, as that would fail if *appLamtoLet* fails at any node in the tree (and the majority of nodes in the tree will not be applications of lambdas, and so will be failures). We use the *Core* universe rather than the *CoreTC* universe for efficiency, as there are no lambdas within *Type* or *Coercion* nodes.

As a second example, a rewrite that eliminates an unused let-binding can be defined as follows (where *freeVars* :: *Expr* → [*Var*] returns the list of free variables in an expression):

```
letElim :: Monad m ⇒ R c m Expr
letElim = do Let (NonRec v _) e ← idR
             if notElem v (freeVars e)
               then return e
               else failT
```

Here we again make use of the monadic structure of rewrites, in this case deciding whether to succeed or fail based on the result of the free variable check.

Thus far, the only monadic side effects used in this article have been failure, and catching failure. As an example of an application-specific effect, some HERMIT rewrites need to be able to generate globally fresh variables. HERMIT defines its own concrete monad (*HermitM*), which provides an operation *newVar* :: *HermitM Var*, among others. This is useful when, for example, defining a rewrite that introduces a let binding:

```
letIntro :: R c HermitM Expr
letIntro = λ_ e → do v ← newVar
                     return (Let (NonRec v e) (Var v))
```

As a final example, consider the task of collecting all variable occurrences in the tree. The general pattern of traversing a tree collecting values is a common one, and KURE provides several library traversals for this purpose. A selection of these are presented in Fig. 6. Briefly:

- *foldT* performs a fold over the tree, applying the argument transformation to every node, combining the results in a monoid.
- *crushT* is similar to *foldT*, except it replaces failures with the unit of the monoid.
- *collectT* is a specialization of *crushT* that collects successes in a list.

Using *collectT*, we can define a transformation that collects all variable occurrences:

```
collectVarOccurrences :: MonadCatch m ⇒ T HermitC m CoreTC [Var]
collectVarOccurrences = collectT projectVar
  where
    projectVar :: T HermitC m CoreTC Var
    projectVar = promoteT (varT idR) <+ promoteT (coVarT idR) <+ promoteT (tyVarT idR)
  -- definitions not given
varT    :: Monad m ⇒ T c m Var a → T c m Expr a
coVarT :: Monad m ⇒ T c m Var a → T c m Coercion a
tyVarT  :: Monad m ⇒ T c m Var a → T c m Type a
```

$foldT :: (Walker\ c\ u, MonadCatch\ m, Monoid\ b) \Rightarrow T\ c\ m\ u\ b \to T\ c\ m\ u\ b$
$foldT\ t = mappend <\$> allT\ (foldT\ t) <*> t$

$crushT :: (Walker\ c\ u, MonadCatch\ m, Monoid\ b) \Rightarrow T\ c\ m\ u\ b \to T\ c\ m\ u\ b$
$crushT\ t = foldT\ (t <+ return\ mempty)$

$collectT :: (Walker\ c\ u, MonadCatch\ m) \Rightarrow T\ c\ m\ u\ b \to T\ c\ m\ u\ [b]$
$collectT\ t = crushT\ ((\lambda b \to [b]) <\$> t)$

Fig. 6. A selection of fold-like traversals provided by the KURE library.

Notice the use of congruence combinators (*varT*, *coVarT* and *tyVarT*) to define the local transformation *projectVar*. We only use the congruence combinators for the Var, CoVar and TyVar constructors, and so *projectVar* will fail for any other constructor.

6 Comparison of KURE to SYB, Uniplate and Stratego

In this section we compare and contrast KURE's support for generic traversals with two other Haskell libraries that provide generic traversal infrastructure, Scrap Your Boilerplate (SYB) (Lämmel & Peyton Jones, 2003) and Uniplate (Mitchell & Runciman, 2007), as well as with Stratego. The focus of our comparison is the differing choices the libraries make as to which substructures of a node are considered "children", and how, given that choice, the generic traversal combinators of those libraries are typed. We also consider the consequences of these choices regarding modularity and statically selective traversals.

6.1 Identifying Children

In Stratego, the children of a node are its proper components, excluding the built-in string, integer and float types. The SYB library similarly takes the children of a node to be its proper components, but it does not exclude any types. Instead, primitive types are considered to be nodes with zero children.

Conversely, in KURE the user specifies which substructures of a node should be considered children (via the *allR* method of the *Walker* class for a specific universe). Typically, these children are zero or more independent proper substructures of the node. However, it is also possible for the children to be abstract *retractions* of the substructures, which can either be for clarity or to introduce a semantic difference between substructures of the same type. For example, the *StringName* and *StringLit* nodes in §4.3.3 are retractions of the *String* substructure, and the *Def* node in §5.1 is a retraction of the $(Var, Expr)$ substructure.

In Uniplate, the children of a node are determined by type. Specifically, the children of a node of type τ are the maximal proper substructures of that same type τ. Uniplate also provides an extension called Biplate, in which an additional type index σ is used. The children of a Biplate node of type τ are the maximal substructures of that type σ. This use of an ad-hoc polymorphic type to index a family of traversals has similarities with KURE's use of a universe type to index a family of traversals.

Thus, in Stratego and SYB children are determined by *structure*, in Uniplate children are determined by *type*, and in KURE children are determined by *location*, as specified by the user.

6.2 Typing the Shallow Traversal

We will now compare the types of the generic traversals in each library. As Stratego is an untyped language, we omit it from the discussion. The focus of our comparison will be the shallow traversal that requires success at every child node: KURE's *allR* strategy. The analogue in SYB is *gmapM*, while Uniplate has two analogues: *descendM* and *descendBiM*. The SYB library implements all of its other shallow traversals in terms of *gmapM* using monad transformers in a similar way to KURE (see §4.5). Uniplate does not provide such other shallow traversals, but a user could add them in the same way.

To allow us to compare the essence of each library's approach to traversal, we simplify the SYB and Uniplate library definitions of these shallow traversals to the following:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Typeable}\ \tau \Rightarrow \mathit{Data}\ \tau\ \textbf{where} \\
\quad \mathit{gmapM} :: \mathit{Applicative}\ m \Rightarrow (\textbf{forall}\ \sigma.\ \mathit{Data}\ \sigma \Rightarrow \sigma \to m\ \sigma) \to (\tau \to m\ \tau)
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Uniplate}\ \tau\ \textbf{where} \\
\quad \mathit{descendM} :: \mathit{Applicative}\ m \Rightarrow (\tau \to m\ \tau) \to (\tau \to m\ \tau) \\
\textbf{class}\ \mathit{Uniplate}\ \sigma \Rightarrow \mathit{Biplate}\ \tau\ \sigma\ \textbf{where} \\
\quad \mathit{descendBiM} :: \mathit{Applicative}\ m \Rightarrow (\sigma \to m\ \sigma) \to (\tau \to m\ \tau)
\end{array}
$$

These classes are analogous to KURE's *Walker* class, with the type $\tau \to m\ \tau$ corresponding to a rewrite.

The traversals of SYB and Uniplate do not support a *context* in the way that KURE does. Neither library's traversal technique is inherently incompatible with it; but they do not currently support it. Consequently, generic traversals in these libraries cannot automatically update a context.

The type of each shallow traversal is a consequence of each library's choice as to which substructures should be considered children. In SYB the children are all proper components of the node being traversed, and so the argument rewrite to *gmapM* must be applicable at almost any type. SYB achieves this near universality via *rank-2 polymorphism* (Peyton Jones *et al.*, 2007) bounded by a *Typeable* constraint, because every monomorphic type can have a *Typeable* instance derived by GHC. To enable reusable definitions of recursive traversals, SYB strengthens *Typeable* to *Data*. While not as universal as *Typeable*, GHC is able to derive a *Data* instance for most algebraic data types (though not GADTs).

The Uniplate shallow traversals have simpler types. As *descendM* treats only similarly typed proper substructures as children, its rewrite argument need only be applicable at that one type. The type of *descendBiM* is more general, allowing the type of the children to differ from their parent, but still requiring all children to have the same type.

Note that for both the SYB and Uniplate libraries, an *Applicative* constraint suffices[3], whereas a *Monad* constraint is required for KURE's *allR* traversal. This is because *allR* takes a rewrite that operates on a universe as an argument, and KURE needs the ability to fail in the (error) case that the argument rewrite changes the child to a different summand (which is handled by the *extractR* combinator (Fig. 3 in §4.3.1)). This is a disadvantage of the rewrite-over-a-sum-type approach.

[3] Actually, the SYB and Uniplate libraries are currently implemented using a *Monad* constraint, even though *Applicative* would be sufficient.

6.3 Example Instances

We will now present some example *Data*, *Uniplate* and *Biplate* instances. We will use the same object language of string expressions that we introduced in §4.3.1. To keep the examples concise, we will only give instances for traversing *Expr* and *Decl* nodes.

The intended SYB semantics entirely determine the *Data* instances: they must target every proper component. A major advantage of this inflexibility is that the definitions can be (and typically are) derived mechanically. The *Data* instances for *Decl* and *Expr* are:

```
instance Data Expr where
  gmapM :: Applicative m ⇒ (forall σ. Data σ ⇒ σ → m σ) → (Expr → m Expr)
  gmapM r (Var n)          = Var    <$> r n
  gmapM r (Lit s)          = Lit    <$> r s
  gmapM r (Append e0 e1)   = Append <$> r e0 <*> r e1
  gmapM r (Let d e)        = Let    <$> r d  <*> r e
instance Data Decl where
  gmapM :: Applicative m ⇒ (forall σ. Data σ ⇒ σ → m σ) → (Decl → m Decl)
  gmapM r (Decl n e) = Decl <$> r n <*> r e
```

Because all proper components are targeted, the types of all proper components require *Typeable* and *Data* instances to be declared for them, even if the user would prefer a statically selective traversal that does not descend into them. However, as these instances can be derived mechanically, defining them is not a burden on the user. In this case, these instances rely on existing *Typeable* and *Data* instances for lists and characters, as well as on *Typeable* instances for *Expr* and *Decl*.

The intended Uniplate semantics entirely determine the *Uniplate* and *Biplate* instances: in both cases the targets are the maximal substructures of a specific type. These targets can be below several constructors, which leads to more complex instances. To gain some code reuse, the instances are usually defined mutually recursively; but this still leads to a quadratic number of instances. For example, the instances for traversing *Expr* and *Decl* nodes are as follows:

```
instance Uniplate Expr where
  descendM :: Monad m ⇒ (Expr → m Expr) → (Expr → m Expr)
  descendM _ (Var n)          = Var    <$> pure n
  descendM _ (Lit l)          = Lit    <$> pure l
  descendM r (Append e0 e1)   = Append <$> r e0             <*> r e1
  descendM r (Let d e)        = Let    <$> descendBiM r d <*> r e
instance Uniplate Decl where
  descendM :: Monad m ⇒ (Decl → m Decl) → (Decl → m Decl)
  descendM r (Decl n e) = Decl <$> pure n <*> descendBiM r e
instance Biplate Decl Expr where
  descendBiM :: Monad m ⇒ (Expr → m Expr) → (Decl → m Decl)
  descendBiM r (Decl n e) = Decl <$> pure n <*> r e
instance Biplate Expr Decl where
  descendBiM :: Monad m ⇒ (Decl → m Decl) → (Expr → m Expr)
  descendBiM _ (Var n)          = Var    <$> pure n
  descendBiM _ (Lit l)          = Lit    <$> pure l
  descendBiM r (Append e0 e1)   = Append <$> descendBiM r e0 <*> descendBiM r e1
  descendBiM r (Let d e)        = Let    <$> r d             <*> descendBiM r e
```

To avoid the quadratic code explosion, the Uniplate library offers alternative means of defining instances by making them polymorphic, using SYB to various degrees.

6.4 Statically Selective Traversals and Data Abstraction

As discussed in §2.3 there are two forms of selective traversal: static and dynamic. Static selectivity is more efficient than dynamic, but static selectivity can also limit the expressiveness of a traversal.

The SYB semantics mandate exactly one way to traverse a node, and that involves descending into all proper components of a node. That is, the user is not intended to define statically selective traversals. This is analogous to being limited to a single KURE universe that contains all substructures of the tree as summands. Consequently, dynamic selectivity is virtually required for efficient use of SYB, else a deep traversal will descend into all substructures of the tree.

Stratego is essentially the same as SYB in this regard, in that it provides exactly one way to traverse a node. Furthermore, Stratego is an untyped language, so there is no static information available with which to define statically selective traversals.

Like SYB, the Uniplate semantics mandate how to traverse a node. Unlike SYB, Uniplate is fundamentally characterized by static selectivity: each traversal only ever has one target type, and it only descends into a substructure if it could contain that type. Essentially, Uniplate provides a separate statically selective traversal for each type of node, each of which corresponds to a KURE traversal indexed by a singleton universe.

However, the expressiveness cost of Uniplate's static selectivity is that if the user wishes to modify nodes of different types, then she must perform multiple traversals, one for each node type that she wishes to target. Decomposing a traversal into multiple sequential traversals in this way is inefficient, and could be problematic if the traversal uses monadic effects in such a way that they cannot be reordered.

The SYB and Uniplate libraries are the two endpoints of a spectrum regarding static selectivity, and a KURE traversal can lie anywhere in between. The universe type determines which locations are to be visited by a traversal, thereby allowing the user to customize the static selectivity. A large universe tends towards the SYB end of the spectrum; in which case dynamic selectivity may offer significant efficiency gains. Note that, in contrast to Uniplate, the static selectivity of KURE is location-directed, not type-directed.

While the intended SYB *semantics* mandate exactly one way to traverse a node, the SYB *implementation* does not prevent the user declaring semantically invalid *Data* instances that omit some proper components of a node during a traversal. However, this may cause third-party code to behave unexpectedly when used with such semantically invalid *Data* instances. Furthermore, as there can only be one *Data* instance in scope, this would not allow other statically selective traversals to be defined: in SYB the *Data* instance defines the one sole way to traverse all nodes of a given type.

There is a workaround for these problems though: the user can create **newtype** wrappers around the node types to be traversed, with a manually defined *Data* instance for each **newtype** that defines a statically selective traversal. The *Data* instance would wrap and unwrap the substructures in the **newtype**, in essentially the same way as KURE does to treat similarly typed nodes distinctly (as demonstrated in §4.3.3). This overcomes the limitation of there only being one *Data* instance in scope, as each abstract type has its own *Data* instance. The disadvantages of this approach are that it violates the intended semantics of SYB, the *Data* instances can no longer be mechanically generated, and it requires a set of **newtype** wrappers to be defined for each statically selective traversal.

Although the Uniplate semantics specify how to traverse a node based on the type of the targeted children, it is possible (as with SYB) to define semantically invalid instances that omit some children of the targeted type. Furthermore, if different (semantically invalid) statically selective traversals are required, then the user can use the **newtype** wrapper approach, and define a separate *Uniplate* (and/or *Biplate*) instance for each such **newtype**.

Ultimately, if the user is willing to disregard the intended semantics of SYB and Uniplate, then extensive use of **newtype** and manual instance declarations allows either library to simulate the traversal behavior of the other, or even of KURE. Meanwhile, KURE is designed to be configurable to different statically selective traversals, so simulating the behavior of SYB or Uniplate, or behaving somewhere in between, is within its intended semantics, and does not require inventive uses of **newtype**. Furthermore, as KURE uses the universe type to index its traversals, **newtype** wrappers are only required in the (rare) case that a traversal needs to traverse nodes of the same type in different locations, while treating them distinctly (see §4.3.3).

6.5 Modularity

We now consider the modularity of each library, by contrasting how easy it is to add new (statically selective) traversals, or to modify existing traversals. Stratego is the simplest case: there is exactly one way to traverse a node, and this is built-in to the Stratego language. Thus it is perfectly modular as it can traverse any object language with nothing required from the user, but that traversal cannot be modified in any way.

Of the Haskell libraries, SYB is the most modular because it defines a single statically selective traversal using an open type class. As the set of traversable node types is open, the user can add new instances without modifying any existing code, extending the SYB traversal to operate over types that were not previously traversable.

Uniplate is founded on the premise that each (statically selective) traversal targets children of exactly one type. Thus, for each traversal, the set of child types is a closed singleton set. The *Biplate* class allows for parent nodes of multiple types to be traversed, sharing code, but the child nodes being targeted must still be of that singleton type. Thus, the set of traversable node types is open, and there is no need to modify existing code when adding *Biplate* instances for new parent node types (though doing so may allow for more code reuse). That is, a Uniplate traversal is modular in the sense that it can be extended to operate over new traversable nodes, but it is not modular in the sense that rewrites targeting children of different types cannot be combined into a single traversal. To target children of a different type, an entirely new traversal must be defined.

In KURE, each universe is a closed sum type, and is used to define exactly one statically selective traversal. As the universe is closed, it is not possible to add new nodes without modifying existing code. However, new traversals can be defined by adding new universes (and their corresponding instances). Because KURE traversals are more configurable than those of Stratego, SYB and Uniplate, it is more likely that several similar traversals will be defined (e.g. over the *Core* and *CoreTC* universes of HERMIT). For these cases, some code reuse can be enabled by defining instances in terms of congruence combinators (§5.2).

6.6 Summary

If used as intended, SYB, Uniplate and KURE provide fairly distinct approaches to traversal, with SYB being the most similar to Stratego. SYB provides a single traversal based on an open set of node types; KURE allows the user to specify statically selective traversals, each with a closed set of node types; and Uniplate allows multiple statically selective traversals, each with an open set of traversable node types but only a single target node type. In SYB everything is a target; in Uniplate all substructures of a specific type are targets, and in KURE the user specifies which locations are to be targets. They each take different approaches to typing: SYB uses rank-2 polymorphism with *Typeable* and *Data* constraints; KURE uses a universe type; and Uniplate is able to give the type directly because it only targets one type.

However, by extending the semantics of SYB and Uniplate for abstract data types, and by using **newtype** wrappers, it is possible for each library to emulate the behavior of the others. We do not claim that this emulation is painless though, nor do we recommend it. Even though **newtype** wrappers are eliminated during compilation, their presence in a program can place an additional boilerplate burden on the user, and can interfere with optimizations (Weirich *et al.*, 2011).

Returning to the intended usage of each library, KURE can be considered more expressive because the user specifies the children to be targeted, whereas the targets are fixed by the semantics of SYB and Uniplate. However, this expressiveness comes at the cost of modularity: the universe type is closed and thus existing code has to be recompiled whenever a universe is extended.

KURE and Stratego differ from SYB and Uniplate in that they were designed with strategic programming in mind, and thus they provide an extensive suite of strategic traversals, with domain-specific error messages. KURE also provides support for allowing generic traversals to automatically update a context. These are pragmatic benefits: it would be possible to implement a strategic programming library in the style of SYB or Uniplate that contained similar infrastructure. However, the existence of this infrastructure in KURE and Stratego certainly makes it easier to start using them for strategic programming.

Finally, we note that the Stratego, SYB and Uniplate traversal semantics are specific enough that the traversals can be mechanically derived (except when using data abstraction). This is not possible in KURE, because children are user-specified locations, and because of the need to update the context.

7 Performance

This section compares the performance of KURE, SYB and Uniplate using two simple strategic programs. We also include the Template Your Boilerplate (TYB) (Adams & DuBuisson, 2012) Haskell library in our performance measurements. We include TYB because it emulates the conventional SYB interface, while using Template Haskell (Sheard & Peyton Jones, 2002) to eliminate almost all performance overhead.

Our focus is on the shallow traversal operator of each library, so we have selected small benchmarks that are just sufficient to demonstrate the differences in traversal performance. We try to identify the causes of the different performance results, and analyze the impact of selective traversals and the most relevant GHC optimizations. This analysis is quite in depth, and is aimed at a reader interested in the efficient implementation of strategic or generic programming libraries in Haskell.

We do not provide a performance comparison with Stratego. The reason for this is that recent Stratego versions are implemented as an Eclipse plugin, with Stratego code being compiled to Java. The priority of this implementation effort is portability rather than efficiency, and the slow loading time of Java classes is intended to be amortized over a lengthy Eclipse session (Visser, 2013).

Conversely, KURE, SYB, Uniplate and TYB are all implemented as Haskell libraries, and are significantly optimized by GHC. Although GHC offers many optimization parameters, we only use the standard level-`O2` optimization for our benchmarks, as we are interested in the performance of common usage. We measure using the `criterion` (O'Sullivan, 2012) benchmarking library, on an otherwise quiescent laptop; our measurements have less than 1% error with more than 95% confidence. By using the same compiler, compiler options and benchmarking tools, our results are a meaningful comparison of the relative performance of the differing traversal techniques.

7.1 Type Class Dictionaries and the Static Argument Transformation

In GHC, type class constraints are implemented as functions that take a *class dictionary* as an argument (Wadler & Blott, 1989). A class dictionary is essentially a record containing a definition for each method of the type class, specialized to a concrete class parameter (or parameters). Thus each class instance generates a single dictionary. The key to performance in our two benchmarks is to eliminate higher-order functions, of which a major source is the passing of these class dictionaries. We will explain how various aspects of each library enable or inhibit this objective.

Inlining directly eliminates higher-order functions. In GHC, inlining also enables many other optimizations, including type specialization. In particular, the *static argument transformation* (SAT) (Santos, 1995) makes it possible to specialize recursive definitions. The SAT pulls arguments — including dictionaries — that never change in recursive calls outside of the recursion, thereby creating an outer definition that can be inlined in order to specialize the inner recursive definition. However, in a mutually recursive set of definitions, GHC heuristically chooses one definition to be a loop-breaker, which prohibits it from being inlined. Thus, none of the arguments bound by the loop-breaker can be specialized via inlining. GHC's implementation of the SAT is quite conservative, because the transformation by itself is often detrimental unless it enables further optimizations. However, as in our context the transformation is beneficial, we write our definitions with explicitly static arguments whenever possible.

The SYB architecture precludes the *gmapM* method of the *Data* class from having explicitly static arguments when the class parameter is a recursive data type. In the *gmapM* method, any application of its rewrite argument must be supplied a corresponding *Data* dictionary, which creates mutual recursion between the *Data* dictionary and the traversal.

By preventing inlining and hence subsequent optimizations, this is a major contributor to the poor performance of SYB.

Mutual recursion between dictionaries also arises in Uniplate, but the consequences are less severe as the rewrite and dictionary arguments of *descendBiM* are explicitly static arguments. The KURE universe type enables the SAT for the same reason that it is detrimental to modularity: all of the traversals are defined in a single *Walker* instance. Thus the rewrite and the dictionary are explicitly static arguments, even in the presence of mutually recursive node types.

7.2 Benchmark: Fibonacci

This benchmark evaluates the Fibonacci function by rewriting the terms of a simple singly recursive expression language. This exercises the bottom-up deep-traversal strategy. The rewrite rules are a pedantic transcription of the Fibonacci recurrence relation:

```
type R m a = a → m a

data Fib = Lit Int | Plus Fib Fib | Fib Fib

plusRule :: Monad m ⇒ R m Fib
plusRule (Plus (Lit x) (Lit y)) = return (Lit (x + y))
plusRule _                      = fail "plusRule"

fibBaseRule :: Monad m ⇒ R m Fib
fibBaseRule (Fib (Lit 0)) = return (Lit 0)
fibBaseRule (Fib (Lit 1)) = return (Lit 1)
fibBaseRule _             = fail "fibBaseRule"

fibStepRule :: Monad m ⇒ R m Fib
fibStepRule (Fib n) = return (Plus (Fib (Plus n (Lit (−2)))) (Fib (Plus n (Lit (−1)))))
fibStepRule _       = fail "fibStepRule"
```

For each variant, we define a function to reduce a *Fib* term to an integer. The following hand-written statically selective traversal (`Hand`) is our baseline variant:

```
reduce :: MonadCatch m ⇒ R m Fib
reduce = eval
  where
    eval    = allbu (try (plusRule <+ evalFib))
    evalFib = fibBaseRule <+ (fibStepRule >>> eval)

allbu :: Monad m ⇒ R m Fib → R m Fib
allbu r = go
  where
    go = all go >>> r

all :: Applicative m ⇒ R m Fib → R m Fib
all r (Plus a b) = Plus <$> r a <*> r b
all r (Fib a)    = Fib  <$> r a
all _ x          = pure x
```

We omit the definitions of >>>, <+ , and *try*, which are defined in the usual manner.

The performance results are shown in Table 1. For each variant, we measure the time to compute *reduce* (Fib (Lit *n*)), where *n* ranges from 20 to 34. For the SYB variants we stop at 31 because they become slow. We list the geometric mean and geometric standard deviation of its slowdown with respect to the baseline. We also list the goodness of fit of

code variant	geometric mean	geometric standard deviation	goodness of fit (r^2)
`SYB-gmapM`	4.96	1.099	0.99997
`SYB-sat`	3.49	1.090	0.99996
`SYB-sel`	5.25	1.097	0.99997
`KURE`	1.03	1.005	0.99940
`Uni`	0.77	1.041	0.99896
`TYB`	1.00	1.015	0.99934

Table 1. *Fibonacci benchmark, slowdown with respect to* `Hand`*.*

each variant's measurements' logarithmic regression, since the expected complexity of the Fibonacci computation is exponential.

We used the following variants in our experiment:

- `SYB-gmapM` uses the SYB deep traversal *everywhereM* along with a hand-written *gmapM* definition, which outperforms the (omitted) GHC-derived *gmapM* definition by approximately 15%.
- `SYB-sat` uses the same *gmapM* definition from `SYB-gmapM`, but uses a SAT of *everywhereM*. This allows the traversal to be inlined and hence specialized to a concrete *Monad* dictionary.
- `SYB-sel` uses a **newtype** wrapper (as discussed in §6.4) to define a statically selective traversal that omits *Int* nodes.
- `KURE` simply uses *Fib* as a singleton universe.
- `Uni` defines the conventional *Uniplate* instance for *Fib*.
- `TYB` generates a traversal semantically equivalent to `Hand`.

For this dense bottom-up traversal of a singly recursive type, it is evident that Uniplate is well-tuned for its intended purpose, outperforming even the hand-written shallow traversal by nearly 25%. We suspect this is due to *constructor specialization* (Peyton Jones, 2007), because the resulting GHC Core is relatively large with many similar loops. KURE and TYB match the hand-written baseline. The SYB variants are at least three times slower than `Hand`. For this particular traversal, the statically selective SYB variant (`SYB-sel`) is actually slower, as the speedup of static selectivity is outweighed by the cost of the **newtype** wrapper interfering with other optimizations. To achieve these performance results, the user need only use level-`O2` optimization, and ensure that inlining and specialization are possible by defining functions with explicitly static arguments.

7.3 Benchmark: Paradise

Our second benchmark is the *increase* function from the Paradise benchmark (Lämmel & Peyton Jones, 2003). This differs from the Fibonacci benchmark by traversing several data types, some of which are mutually recursive. The performance results vary primarily because of the large potential for static selectivity — only 17% of the non-trivial substructures are the actual targets. The data types are as follows:

```
newtype Company = C [Dept]
data Dept       = D Name Employee [Unit]
data Unit       = PU Employee | DU Dept
data Employee   = E Person Salary
data Person     = P Name Address
newtype Salary  = S Integer

type Name    = String
type Address = String
```

The objective is to increase all *Salary* values in a *Company* by a given increment. The following hand-written statically selective traversal (`Hand`) is our baseline variant:

```
increase :: Monad m ⇒ Integer → R m Company
increase k = allbuC (inc k)

inc :: Monad m ⇒ Integer → R m Salary
inc k = λ(S x) → if x < 0 then fail "inc" else return (S (x + k))


allbuC f (C ds)       = C   <$> mapM (allbuD f) ds
allbuD f (D n m us)   = D n <$> allbuE f m <*> mapM (allbuU f) us
allbuE f (E p s)      = E p <$> f s
allbuU f (PU e)       = PU  <$> allbuE f e
allbuU f (DU d)       = DU  <$> allbuD f d
```

The performance results are shown in Table 2. Each traversal was applied to the same 100 arbitrary test inputs, which were roughly evenly distributed in sizes from 1 to 18000 constructors. We list the goodness of fit for a linear model, since the traversal complexity is approximately linear with respect to the number of constructors. We measured the following variants of the traversal:

- `Hand-sat` is a manual application of the SAT to `Hand`; it enables specialization with respect to both the monad and the rewrite argument.
- `SYB` uses dynamic selectivity to descend only into nodes that could contain a *Salary*, using a *Data* instance derived by SYB.
- `SYB-sel` uses a **newtype** wrapper with a manually defined *Data* instance to encode a statically selective traversal that only visits *Salary* substructures. As the *Data* dictionaries for *Dept* and *Unit* are mutually recursive via the *gmapM* definitions, one of those definitions will be a loop-breaker.
- `SYB-sel-sat` is a variant of `SYB-sel` that eliminates the mutual recursion by including a distinct copy of the *gmapM* definition for *Unit* within the *gmapM* definition for *Dept*. The copy has the SAT applied. This variant enables specialization with respect to both the monad and the rewrite argument.
- `KURE` uses a conventional universe for the data types, and uses the *allbuR* strategy to reach all of the *Salary* substructures.
- `KURE-sel` uses an alternative universe that only descends into *Salary* substructures.
- `Uni` defines the conventional *Uniplate* and *Biplate* instances for the *Salary* target type. As the *Biplate* dictionaries for *Dept* and *Unit* are mutually recursive via the *descendBiM* definitions, one of the those definitions will be a loop-breaker.
- `TYB` generates a statically selective traversal that visits only *Salary* substructures.

code variant	geometric mean	geometric standard deviation	goodness of fit (r^2)
`Hand-sat`	0.50	1.108	0.97444
`SYB`	4.62	1.145	0.98708
`SYB-sel`	1.52	1.096	0.99276
`SYB-sel-sat`	0.50	1.109	0.98946
`KURE`	0.61	1.115	0.98640
`KURE-sel`	0.49	1.109	0.95647
`Uni`	0.90	1.091	0.96421
`TYB`	0.46	1.173	0.98576

Table 2. *Paradise benchmark, slowdown with respect to `Hand`, 100 arbitrary inputs.*

We use `Hand` as the baseline, because it is the conventional definition of the traversal. The effectiveness of the SAT can been seen by the two factor speedup of the `Hand-sat` variant.

`SYB` is 450% slower than `Hand`, even with dynamic selectivity. The static selectivity of `SYB-sel` reduces the overhead to 150%, demonstrating the advantage of static over dynamic selectivity in SYB. The `SYB-sel-sat` variant further enables the SAT by breaking the mutual recursion among dictionaries, at the cost of some minor code duplication. The result performs as well as the `Hand-sat` variant, and just slightly poorer than TYB. The systemic SYB issue that precluded static arguments is avoided in the two statically selective variants: the target type *Salary* is not recursive, so the traversal does not visit any substructures that result in mutual recursion with the *Data* dictionary. And in the `SYB-sel-sat` variant there is no recursion among dictionaries, so inlining can specialize the traversals as much as possible.

The Uniplate variant is about twice as slow as the best KURE and SYB variants. This is an unexpected result, and Neil Mitchell has communicated that it is a regression. We anticipate performance at least as fast as KURE.

The conventional `KURE` definition, visiting all substructures inhabiting one of the data types, achieves an efficiency just 16% behind the best variant. The `KURE-sel` variant increases the static selectivity by only visiting *Salary* substructures, matching the `Hand-sat` variant. The TYB variant is the fastest, with performance matching hand-optimized code.

This benchmark emphasizes the importance of explicitly static arguments for GHC optimization and the potential speedup of static selectivity. As a case in point, a traversal defined with SYB — the conventionally worst performing library — that was designed specifically for both of those concerns gives the second best performance.

7.4 Summary

A key optimization for generic traversals is selectivity, and static selectivity is more efficient than dynamic selectivity. The performance gain of selectivity depends on the proportion of the data structure that needs to be traversed. In the Paradise benchmark, where much of the structure could be avoided, the gain was significant. Conversely, in the Fibonacci benchmark, where most of the structure had to be traversed, then the overhead of encoding the selectivity outweighed the benefit.

Performing the SAT can bring significant benefits to the performance of generic traversals in Haskell, as it allows inlining and thence specialization. However, the design of SYB interferes with the SAT, which contributes to SYB's relatively poor performance. Only by manually duplicating code were we able to apply the SAT for the SYB variant and achieve performance comparable with the other libraries. Note that the variants with hyphenated names (`-gmapM`, `-sat`, `-sel`, etc.) required manual implementation of the corresponding optimization; the unhyphenated variants represent a straightforward use of the libraries.

In summary, on both benchmarks TYB performed slightly better than KURE, and KURE was significantly better than SYB. Uniplate was the best library for Fibonacci, but performed noticeably worse than KURE for Paradise. However, the latter result appears to be a regression in Uniplate, and we expect KURE-like performance once that is resolved. The two benchmarks were both fairly simple, and comparing the relative performance of the libraries on larger and more complex examples remains as future work.

8 Conclusion

In this article we have described our approach of using *universe types* to assign types to generic traversals, and to support statically selective traversals and traversals that can distinguish between data based on its *location*, not just its *type*. We then described the implementation of this idea as part of KURE, a Haskell-embedded strategic programming language. To demonstrate the viability of this approach, we presented its usage in the HERMIT system.

Many of the features of KURE were motivated by the needs of HERMIT, including support for: statically selective traversals; rewriting nodes of different types during the same traversal; automatic maintenance of a context during generic traversals; and the ability to detect when a successful rewrite actually modified its target. This combination of features is not provided by any other Haskell library for strategic or generic programming (and a Haskell library was desired for ease of interoperability with HERMIT).

After HERMIT, the largest example of KURE use is the `html-kure` package (Gill, 2013), which layers the KURE interface on top of the HXT HTML parser and pretty printer (Schmidt *et al.*, 2012). Rewriting HTML is a task for which KURE was not specifically designed, so that this was straightforward provides evidence that KURE is more generally useful (indeed, we currently use this package to preprocess our research group's website). However, KURE needs to be used for many more applications before we are in a position to fairly assess its ease of use compared to other generic rewriting systems.

The main novelty of KURE compared to other strategic and generic rewriting systems is the way it uses universe types to index generic traversals. We gave a detailed comparison of the KURE type system with the type systems of SYB and Uniplate, two other approaches to typed generic programming in Haskell. The crucial distinction is that the semantics of KURE traversals are not determined by the structure or types of the data structure: instead the KURE user can customize how traversals should treat each *location* in the data type being traversed. Consequently KURE is relatively configurable, and can be used to simulate either SYB or Uniplate semantics, or, more usually, can be given a semantics somewhere in between. However, this flexibility does impose a cost in modularity, with modifications to a traversal requiring more recompilation of existing code than in either of the other two libraries (though the necessary changes are fairly small and localized).

Finally, we also compared the performance of KURE with SYB, Uniplate and TYB, examining the main optimizations that each library supports or inhibits. For the small benchmarks we used, we found KURE's performance to be roughly comparable with Uniplate and TYB, and superior to SYB (which, as a consequence of its design, has known performance issues (Rodriguez Yakushev *et al.*, 2008)).

In closing, we think that KURE is a useful addition to the family of Haskell-based generic programming libraries. The emphasis on customizable strategic traversals has allowed KURE to serve as the rewrite engine of HERMIT, and we expect to use KURE as a foundation for building higher-level rewriting languages in the future.

Acknowledgments

We thank Andrew Farmer and Ed Komp for valuable discussions on their experience using KURE to implement the HERMIT system; Neil Mitchell for his help with Uniplate performance issues; and Andrew Farmer, Ed Komp, José Pedro Magalhães, Jeremy Gibbons, and the anonymous reviewers for feedback on draft versions of this article.

References

Adams, Michael D., & DuBuisson, Thomas M. (2012). Template your boilerplate: Using Template Haskell for efficient generic programming. *Pages 13–24 of: Haskell symposium*. ACM.

Balland, Emilie, Moreau, Pierre-Etienne, & Reilles, Antoine. (2008). Rewriting strategies in Java. *Electronic notes in theoretical computer science*, **219**, 97–111.

Bravenboer, Martin, Kalleberg, Karl Trygve, Vermaas, Rob, & Visser, Eelco. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of computer programming*, **72**(1–2), 52–70.

Bringert, Björn, & Ranta, Aarne. (2008). A pattern for almost compositional functions. *Journal of functional programming*, **18**(5–6), 567–598.

Brown, Neil C. C., & Sampson, Adam T. (2009). Alloy: Fast generic transformations for Haskell. *Pages 105–116 of: Haskell symposium*. ACM.

Burstall, Rod. M., & Darlington, John. (1977). A transformation system for developing recursive programs. *Journal of the ACM*, **24**(1), 44–67.

Culpepper, Ryan. (2012). Fortifying macros. *Journal of functional programming*, **22**(4–5), 439–476.

Culpepper, Ryan, & Felleisen, Matthias. (2010). Fortifying macros. *Pages 235–246 of: International conference on functional programming*. ACM.

Dolstra, Eelco. (2001). *First class rules and generic traversals for program transformation languages*. M.Phil. thesis, Utrecht University.

Dolstra, Eelco, & Visser, Eelco. (2001). *First-class rules and generic traversal*. Tech. rept. Utrecht University.

Ellison, Chucky, & Roşu, Grigore. (2012). An executable formal semantics of C with applications. *Pages 533–544 of: Principles of programming languages*. ACM.

Erdweg, Sebastian, Rendel, Tillmann, Kästner, Christian, & Ostermann, Klaus. (2011). SugarJ: Library-based syntactic language extensibility. *Pages 391–406 of: Object-oriented programming, systems, languages, and applications*. ACM.

Erdweg, Sebastian, Rieger, Felix, Rendel, Tillmann, & Ostermann, Klaus. (2012). Layout-sensitive language extensibility with SugarHaskell. *Pages 149–160 of: Haskell symposium*. ACM.

Farmer, Andrew, Gill, Andy, Komp, Ed, & Sculthorpe, Neil. (2012). The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. *Pages 1–12 of: Haskell symposium*. ACM.

Felleisen, Matthias, Findler, Robert Bruce, & Flatt, Matthew. (2009). *Semantics engineering with PLT Redex*. MIT Press.

Gibbons, Jeremy. (2003). Patterns in datatype-generic programming. *Pages 277–289 of: Declarative programming in the context of object-oriented languages*. Multiparadigm Programming, vol. 27. NIC.

Gill, Andy. (2006). Introducing the Haskell equational reasoning assistant. *Pages 108–109 of: Haskell workshop*. ACM.

Gill, Andy. (2009). A Haskell hosted DSL for writing transformation systems. *Pages 285–309 of: Working conference on domain-specific languages*. Springer.

Gill, Andy. (2013). `http://hackage.haskell.org/package/html-kure`.

Girard, Jean-Yves. (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.

Hinze, Ralf, & Löh, Andres. (2007). Generic programming, now! *Pages 150–208 of: International conference on datatype-generic programming*. Springer.

Hinze, Ralf, & Löh, Andres. (2009). Generic programming in 3D. *Science of computer programming*, **74**(8), 590–628.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1–3), 67–111.

Kalleberg, Karl T., & Visser, Eelco. (2007). Spoofax: An interactive development environment for program transformation with Stratego/XT. *Pages 47–50 of: Workshop on language descriptions, tools, and applications*. Elsevier.

Kats, Lennart C. L., & Visser, Eelco. (2010). The Spoofax language workbench. Rules for declarative specification of languages and IDEs. *Pages 444–463 of: Object-oriented programming, systems, languages, and applications*. ACM.

Kiselyov, Oleg. (2006). *Smash your boiler-plate without class and typeable*. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`.

Klein, Casey, Clements, John, Dimoulas, Christos, Eastlund, Carl, Felleisen, Matthias, Flatt, Matthew, McCarthy, Jay A., Rafkind, Jon, Tobin-Hochstadt, Sam, & Findler, Robert Bruce. (2012). Run your research: On the effectiveness of lightweight mechanization. *Pages 285–296 of: Principles of programming languages*. ACM.

Lämmel, Ralf, & Peyton Jones, Simon. (2003). Scrap your boilerplate: a practical design pattern for generic programming. *Pages 26–37 of: Types in languages design and implementation*. ACM.

Lämmel, Ralf, & Visser, Joost. (2002). Typed combinators for generic traversal. *Pages 137–154 of: Practical aspects of declarative programming*. Springer.

Lazar, David, Arusoaie, Andrei, Serbanuta, Traian Florin, Ellison, Chucky, Mereuta, Radu, Lucanu, Dorel, & Roşu, Grigore. (2012). Executing formal semantics with the K tool. *Pages 267–271 of: Formal methods*. Lecture Notes in Computer Science, vol. 7436. Springer.

Liang, Sheng, Hudak, Paul, & Jones, Mark. (1995). Monad transformers and modular interpreters. *Pages 333–343 of: Principles of programming languages*. ACM.

Magalhães, José Pedro, Dijkstra, Atze, Jeuring, Johan, & Löh, Andres. (2010). A generic deriving mechanism for Haskell. *Pages 37–48 of: Haskell symposium*. ACM.

Mitchell, Neil, & Runciman, Colin. (2007). Uniform boilerplate and list processing. *Pages 49–60 of: Haskell workshop*. ACM.

Moors, Adriaan, Piessens, Frank, & Joosen, Wouter. (2006). An object-oriented approach to datatype-generic programming. *Pages 96–106 of: Workshop on generic programming*. ACM.

O'Connor, Russell. (2011). Functor is to Lens as Applicative is to Biplate: Introducing Multiplate. *Workshop on generic programming*. Excluded from the printed proceedings, available at `http://arxiv.org/abs/1103.2841`.

Oliveira, Bruno, & Gibbons, Jeremy. (2008). Scala for generic programmers. *Pages 25–36 of: Workshop on generic programming*. ACM.

O'Sullivan, Bryan. (2012). `http://hackage.haskell.org/package/criterion`.

Paterson, Ross. (2001). A new notation for arrows. *Pages 229–240 of: International conference on functional programming*. ACM.

Peyton Jones, Simon. (2007). Call-pattern specialisation for Haskell programs. *Pages 327–337 of: International conference on functional programming*. ACM.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Shields, Mark. (2007). Practical type inference for arbitrary-rank types. *Journal of functional programming*, **17**(1), 1–82.

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press.

Reynolds, John C. (1974). Towards a theory of type structure. *Pages 408–423 of: Colloque sur la programmation*. Lecture Notes in Computer Science, vol. 19. Springer–Verlag.

Rodriguez Yakushev, Alexey, Jeuring, Johan, Jansson, Patrik, Gerdes, Alex, Kiselyov, Oleg, & Oliveira, Bruno. (2008). Comparing libraries for generic programming in Haskell. *Pages 111–122 of: Haskell symposium*. ACM.

Santos, André. (1995). *Compilation by transformation in non-strict functional languages*. Ph.D. thesis, University of Glasgow.

Schmidt, Uwe, Schmidt, Martin, & Kuseler, Torben. (2012). `http://hackage.haskell.org/package/hxt`.

Sculthorpe, Neil, Farmer, Andrew, & Gill, Andy. (2013). The HERMIT in the tree: Mechanizing program transformations in the GHC core language. *Pages 86–103 of: Implementation and application of functional languages 2012*. Lecture Notes in Computer Science, vol. 8241. Springer.

Sheard, Tim, & Peyton Jones, Simon. (2002). Template metaprogramming for Haskell. *Pages 1–16 of: Haskell workshop*. ACM.

Strachey, Christopher. (2000). Fundamental concepts in programming languages. *Higher-order and symbolic computation*, **13**(1–2), 11–49.

Sulzmann, Martin, Chakravarty, Manuel M. T., Peyton Jones, Simon, & Donnelly, Kevin. (2007). System F with type equality coercions. *Pages 53–66 of: Types in language design and implementaion*. ACM.

van Noort, Thomas, Rodriguez Yakushev, Alexey, Holdermans, Stefan, Jeuring, Johan, & Heeren, Bastiaan. (2008). A lightweight approach to datatype-generic rewriting. *Pages 13–24 of: Workshop on generic programming*. ACM.

Visser, Eelco. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Pages 216–238 of: Domain-specific program generation*. Spinger.

Visser, Eelco. (2005). A survey of strategies in rule-based program transformation systems. *Journal of symbolic computation*, **40**(1), 831–873.

Visser, Eelco. (2013). Personal communication.

Wadler, Philip. (1992). The essence of functional programming. *Pages 1–14 of: Principles of programming languages*. ACM.

Wadler, Philip, & Blott, Stephen. (1989). How to make ad-hoc polymorphism less ad hoc. *Pages 60–76 of: Principles of programming languages*. ACM.

Weirich, Stephanie, Vytiniotis, Dimitrios, Peyton Jones, Simon, & Zdancewic, Steve. (2011). Generative type abstraction and type-level computation. *Pages 227–240 of: Principles of programming languages*. ACM.

Westra, Hedzer. (2001). CobolX: Transformations for improving COBOL programs. *Pages 40–60 of: Stratego users day*.