Under consideration for publication in J. Functional Programming

KURE

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 programming languages provide much of this infrastructure, with the aim of minimizing the amount of such boilerplate code the author of the transformation system has to write.

KURE is a typed strategic programming language in the tradition of Stratego, implemented as a domain-specific Haskell embedding. 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 strategies/traversals that should operate over *any* type. Compared to other generic programming libraries 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, as well as their behavior being determined by the *type* of each datum.

This article describes the implementation of KURE, and compares its design choices, and their consequences, with other approaches to strategic and generic programming. We also demonstrate KURE usage with examples taken from the HERMIT project, which uses KURE as its underlying rewrite engine.

1 Introduction

This article describes an approach for building rewrite engines over strongly typed data structures. Rewrite rules are often expressed as local correctness-preserving transformations, with well-understood preconditions and contextual requirements, such as lexical scope. A rewrite engine takes these local rules and applies them in systematic ways to achieve a global effect. Strategic programming (Visser, 2005), a style of programming that provides explicit support for composition of *strategies* for rewriting transformations, 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

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

wish to support complex traversal strategies, such as potentially failing traversals, selective traversals, and traversals over mutually recursive data types.

The Kansas University Rewrite Engine (KURE) is a Haskell-embedded domain-specific language (DSL) for typed strategic programming, and is the principal subject of this article. KURE began as a component of the 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 stages of its development: 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 is similar to Scrap Your Boilerplate (SYB) (Lämmel & Peyton Jones, 2003) and Uniplate (Mitchell & Runciman, 2007), but has different engineering compromises regarding data-structure traversal. KURE also builds on many years of research into strategic rewriting, especially the work of Stratego (Visser, 2004; Bravenboer *et al.*, 2008) and StrategyLib (Lämmel & Visser, 2002).

Specifically, the contributions of this article are:

- We define a typed strategic rewriting system as a Haskell-embedded DSL. (§4; §5)
- We abstract our combinators over a user-definable context and monad, allowing our strategies to be customized for different settings (§4).
- We demonstrate the viability of using KURE for realistic applications by presenting running examples of KURE's usage in the HERMIT project. (§3; §4; §5; §6)
- We give a detailed comparison of the KURE design to the SYB and Uniplate libraries, discussing the trade-offs involved. (§7)
- We compare KURE's performance with SYB, TYB, and Uniplate, and discuss the causes of the performance variations. (§8)

2 Related Work: Strategic and Generic Programming

KURE is a domain-specific language for *strategic programming* (Visser, 2005). Strategic programming is a paradigm for expressing transformations and traversals of tree-structured data. Two key concepts are *rewrites*, which modify a node in the tree, and *strategies*, which combine rewrites to make new rewrites. Rewrites can either succeed or fail when applied to a node. That is, in functional parlance, rewrites are partial functions and strategies are combinators. Strategic programming languages provide several rewrites and strategies as primitives, a means for the user to define data-type–specific rewrites, and a means of combining rewrites and strategies to make new rewrites and strategies. As an example, we will begin by introducing Stratego, the most widely used strategic rewrite system.

2.1 Stratego

Stratego (Visser, 2004; Bravenboer *et al.*, 2008) was designed as a term-rewriting tool, to support the application of rewrites to nodes in the abstract syntax tree of an arbitrary object language. Some representative Stratego primitives are given by the following grammar, where *r* denotes a rewrite and *t* denotes a term in the abstract syntax of the object language:

$$\begin{array}{lcl} r & ::= & \mathit{id} \mid \mathit{fail} \mid r \,;\, r \mid r <\!+\, r \mid \mathit{all}\,(r) \\ t & ::= & <\! r \!>\! t \end{array}$$

Briefly, the semantics of these primitives are:

- *id* is the identity rewrite that always succeeds;
- *fail* is the always failing rewrite;
- $r_1 \mathbin{;} r_2$ is a strategy that sequences two rewrites (requiring both to succeed);
- $r_1 \mathbin{<\!\!+} r_2$ is a "catch" strategy (attempt r_1, and if it fails then attempt r_2);
- $\mathit{all}(r)$ applies r to all immediate child nodes (requiring all to succeed);
- $<\!r\!>t$ applies the rewrite r to the term t, producing a new term.

There are others, but these capture the spirit of the programming paradigm.

A rich algebra of combinators can be built on top of the concept of strategies. For example, a strategy that catches a failed rewrite with the identity rewrite (and thus always succeeds) can be defined as follows:

$$\mathit{try}\ (r) = r \mathbin{<\!\!+} \mathit{id}$$

With the exception of *all*, these rewrites and strategies act on the current node, ignoring the tree-structure of the data. The *all* strategy requires a tree structure, as some notion of "children" is assumed. We call *all* a *shallow* traversal strategy, because while it does traverse the tree, it does so only to a fixed depth (in this case, depth 1). However, we can use shallow strategies to implement *deep* traversals: strategies that recursively descend the tree to an arbitrary depth. For example, the following strategies perform pre-order and post-order traversals (respectively), applying r to every node in the tree:

$$\begin{aligned}\mathit{topdown}\ (r) &= r \mathbin{;} \mathit{all}\ (\mathit{topdown}\ (r))\\ \mathit{bottomup}\ (r) &= \mathit{all}\ (\mathit{bottomup}\ (r)) \mathbin{;} r\end{aligned}$$

Stratego also allows rewrites to be defined that are specific to a particular data type. The syntax for such definitions is as follows, where *sourceExpr* and *targetExpr* are expressions in the object language that may also contain meta-language variables:

$$\mathit{rewriteName} : \mathit{sourceExpr} \ \rightarrow\ \mathit{targetExpr}$$

To give an example (taken from Bravenboer *et al.* (2008)), given the following object language of integer addition,

$$e \ ::= \ \mathsf{Int}\ (\mathit{int}) \ \mid\ \mathsf{Add}\ (e, e)$$

then a rewrite to apply the left-unit law of addition could be defined as follows:

$$\mathit{LeftUnit} : \mathsf{Add}\ (\mathsf{Int}\ (0))\ e \ \rightarrow\ e$$

This rewrite attempts to match a term by checking if the top node is Add and its left child is $\mathsf{Int}\ (0)$. If so, then the whole term is replaced with the right child of Add (the meta-variable e) and the rewrite terminates with success. If the term is not matched, nothing is rewritten and the rewrite terminates with failure.

Stratego also allows a strategy to be invoked from within a rule, by enclosing the strategy in angle brackets. For example:

$$\begin{aligned}&\mathit{Commutativity} : \mathsf{Add}\ e_1\ e_2 \ \rightarrow\ \mathsf{Add}\ e_2\ e_1\\ &\mathit{RightUnit} \qquad\ : e \ \rightarrow\ <\!\mathit{Commutativity} \mathbin{;} \mathit{LeftUnit}\!>e\end{aligned}$$

By using application inside rules, strategy based programming jumps between rules and strategies, allowing complex rewrite strategies to be implemented. Critically, locally applicable rules are given the opportunity to act over many sub-terms in a controlled matter.

Using this programming idiom, several rewrite 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 and strategic programming here, and the interested reader is referred to the Stratego literature (Visser, 2004; Bravenboer *et al.*, 2008). There are also systems that express Stratego-style rewrites as extensions to mainstream languages such as Java, see for example Balland *et al.* (2008).

2.2 Selective Traversals

Stratego's shallow traversal *all* descends into every child node, and the example deep traversals *topdown* and *bottomup* descend into every node in the tree. While this is often the required behavior, it is sometimes desirable to exclude some branches of the tree from a traversal. This could be for semantic reasons: for example, when performing single-variable substitution on a representation of the lambda calculus we should not descend past a rebinding of the variable being substituted (shadowing). Alternatively, it may be for performance reasons: excluding branches that are known not to contain any values 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)) have been designed around enabling it.

A traversal that selects which branches to descend into is called a *selective traversal*. Selectivity comes in two forms: *static selectivity*, where certain types or locations are never descended into, and *dynamic selectivity*, where whether to descend into a particular branch is determined by a run-time predicate. We discuss the impact of selectivity on performance in §8.1.

2.3 Strategic and Generic Programming in Haskell

KURE was originally inspired by a previous attempt to host strategic programming in Haskell: the StrategyLib library (Lämmel & Visser, 2002). StrategyLib follows in the tradition of Stratego, but adds typing to rewrites and strategies (Stratego is untyped), and uses SYB (Lämmel & Peyton Jones, 2003) to implement traversals over arbitrary data types. Rather than focusing on the specific features and limitations of the StrategyLib system, we will instead discuss more generally what is possible with the SYB approach to generic traversals, and how that compares to our KURE approach (§7).

The key challenge is providing complex traversal strategies over mutually recursive data types. There are many Haskell approaches that offer reusable solutions, many of which coevolved with data-type–generic programming over its fruitful history: Polytypic Programming (Hinze, 2000, 2002; Norell & Jansson, 2005), SYB (Lämmel & Peyton Jones, 2003, 2004, 2005), the Generic Haskell language extension (Löh, 2004), RepLib (Weirich, 2006), Uniplate (Mitchell & Runciman, 2007), the Compos pattern (Bringert & Ranta, 2008), Alloy (Brown & Sampson, 2009), MultiRec (Yakushev *et al.*, 2009), and GHC Generics (Magalhães *et al.*, 2010). Many of these same concepts can also be implemented via Template Haskell metaprogramming (Sheard & Peyton Jones, 2002), as exemplified by the `geniplate` library (Augustsson, 2012) and the Template Your Boilerplate (TYB) method (Adams & DuBuisson, 2012). There is also a detailed, albeit slightly dated, survey paper regarding the various approaches to generic programming (Rodriguez *et al.*, 2008).

The range of possible solutions to the complex traversal challenge cover a large spectrum of type-centric compromises, from making tree traversals dynamically typed, to completely statically determinable paths through trees. The SYB library is a direct instance of the strategic-programming paradigm in Haskell, with its universal type corresponding to the untypedness of strategic languages such as Stratego. The recent development of the `lens` library (Kmett, 2012) supports both more precise types and the rich and natural composition of traversals. However, the `lens` library itself offers no support for generic traversals; it instead interfaces with generic libraries to obtain Uniplate-style traversals. For comparative purposes, this article will discuss the traversal support of SYB and Uniplate in some detail (§7). The more recent generic programming libraries RepLib, MultiRec, and GHC Generics do not have novel traversal schemes of their own, but they can usually derive the definitions of the various traversals we consider. The Compos pattern, as embodied by MultiRec, was also a candidate for our comparison, but we chose to exclude it as SYB and Uniplate establish the major points in the design space. The design space of the SYB approach has been described in detail by Hinze & Löh (2009), characterizing it by support for overloading, type representation and generic data representation. However, KURE and Uniplate do not fall within the design space they consider, as they do not use a generic representation of data.

3 Running Example: HERMIT and GHC Core

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 use HERMIT as a running example throughout this article. After HERMIT, the largest example of KURE usage is the `html-kure` package (Gill, 2013), which layers the complete KURE interface on top of the HXT HTML parser and pretty printer (Schmidt *et al.*, 2012). There are also several toy examples bundled with the KURE package (Sculthorpe & Gill, 2013), including the example featured in the first KURE paper (Gill, 2009).

As this article contains a mixture of both KURE implementation code and KURE usage code, we enclose the latter in boxes to reduce confusion between the two. Library strategies that are included in the KURE package but are not part of the KURE implementation could arguably be grouped with either, but we choose to group them with the former. That is, the code in boxes exemplifies what a KURE user would write, everything else is provided by the KURE package.

HERMIT uses KURE to manipulate GHC's internal intermediate language, GHC Core. GHC Core is an implementation of System $F_C^{\uparrow}$ (Yorgey *et al.*, 2012), which is System F (Pierce, 2002) extended with let-binding, constructors, type coercions, and algebraic and polymorphic kinds. KURE is used for refactoring and querying GHC Core. The subset of GHC Core that HERMIT uses KURE to traverse is shown in Fig. 1, presented using some inlined type synonyms for clarity.

```
data ModGuts       = ModGuts { _ :: CoreProgram, ... }
type CoreProgram = [CoreBind]
data CoreBind      = NonRec Var CoreExpr
                   | Rec [CoreDef]
data CoreDef       = Def Var CoreExpr
data CoreExpr      = Var Var
                   | Lit Literal
                   | App CoreExpr CoreExpr
                   | Lam Var CoreExpr
                   | Let CoreBind CoreExpr
                   | Case CoreExpr Var Type [CoreAlt]
                   | Cast CoreExpr Coercion
                   | Tick CoreTickish CoreExpr
                   | Type Type
                   | Coercion Coercion
type CoreAlt       = (AltCon, [Var], CoreExpr)
```

Fig. 1. GHC Core.

4 KURE Architecture

KURE is a Haskell-hosted strategic programming language that provides the following:

- typed strategies and rewrites;
- a common traversal mechanism for both rewriting and querying;
- support for maintaining a context during traversals;
- support for working in an arbitrary monad;
- user-definable closed universes of traversable types.

KURE has two types of operations: *rewrites*, which modify the tree, and *translations*, which query the tree. The common traversal mechanism allows the same set of combinators to be used to direct rewrites and translations to locations in the tree.

This section, and the two following, present the KURE library. We begin by discussing the benefits of supporting a context (§4.1) and an arbitrary monad (§4.2) while performing a traversal, and how they are implemented in KURE. We then introduce KURE's *Rewrite* and *Translate* abstractions (§4.3), explain how they tie-in to Haskell's type-class infrastructure (§4.4), and give some examples of using that infrastructure to write strategies (§4.5). The final part of this section discusses how change detection is supported by KURE (§4.6).

In §5 we describe tree traversals in KURE, starting with combinators that traverse specific constructors (§5.1). We then consider the problem of how to assign a type to a *generic* traversal that is expected to operate over nodes of varying types (§5.2), before describing how such generic shallow (§5.3) and deep (§5.4) strategy combinators are implemented in KURE. The section concludes with a description of KURE's *Lens* abstraction (§5.5), an alternative approach to directing translations and rewrites to locations in the tree.

In §6 we conclude the presentation of KURE with a variety of examples taken from the HERMIT system.

4.1 Contexts

Maintaining a context allows transformations to depend on information from elsewhere in the tree, and supporting it explicitly in KURE allows generic traversal strategies to automatically update the context as they traverse the tree. For now we will just assume the existence of a context, and postpone considering how to update it until §5.1.

As an example, the HERMIT context contains the set of bindings in scope, the depth of the lowest binding, and the path from the root to the current node:

```
data HermitC = HermitC { hermitBindings :: Map Var HermitBinding,
                         hermitDepth    :: Int,
                         hermitPath     :: AbsolutePath }
```

The key component is the set of bindings, which allows a variety of transformations (inlining, for example) to be performed locally.

The *AbsolutePath* data type stores the path from the root of the tree to the current position as an (abstract) list of integers. As this information is useful for applications of KURE beyond HERMIT, this data type is provided by KURE, along with several strategies and pure operations that use it. In particular, KURE provides a type class for contexts that contain an *AbsolutePath*, with methods to retrieve and extend the path:

```
class PathContext c where
  absPath :: c → AbsolutePath        -- retrieve the current absolute path
  (@@)    :: c → Int → c             -- extend the current absolute path
```

AbsolutePath is itself an instance of this class, and thus the instance for *HermitC* is trivial:

```
instance PathContext HermitC where
  absPath = hermitPath
  c @@ n  = c { hermitPath = hermitPath c @@ n }
```

Note that this class instance is not required by most of KURE's functionality; indeed many applications of strategic programming do not require any contextual information.

4.2 Monads

As well as a context, KURE supports an arbitrary monad so that transformations can access any required external operations, for example fresh name generation. The context is deliberately kept distinct from the monad, for reasons we will discuss in §7.8.2.

As catching failure is a key concept in strategic programming, it should be supported by the monad. To encode this, KURE provides the following subclass of *Monad*:

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

For failure, KURE uses the *fail* method of the *Monad* class, with the following law expected to hold for *MonadCatch* instances:

```
catchM (fail msg) f ≡ f msg
```

```
data KureM a = Failure String | Success a
instance Monad KureM where
  return               :: a → KureM a
  return               = Success
  fail                 :: String → KureM a
  fail                 = Failure
  (>>=)                :: KureM a → (a → KureM b) → KureM b
  (Success a)   >>= f  = f a
  (Failure msg) >>= _  = Failure msg
instance MonadCatch KureM where
  catchM                      :: KureM a → (String → KureM a) → KureM a
  catchM (Success a)     _    = Success a
  catchM (Failure msg) f      = f msg
```

Fig. 2. The *KureM* monad.

The use of *String* to record failure information is fairly crude, and we intend to introduce a dedicated exception data type in the next version of KURE.

Using *catchM*, KURE defines combinators that allow error messages to be modified while preserving failure or success, for example:

```
modFailMsg        :: MonadCatch m ⇒ (String → String) → m a → m a
modFailMsg f ma = catchM ma (fail ∘ f)
```

This is similar to the <?> error-labeling combinator used in the Parsec library (Leijen & Meijer, 2001). KURE uses these combinators extensively to make failing transformations produce more informative error messages. However, for clarity of presentation, this article omits all uses of these combinators from strategy definitions. Other than changing the error message produced, these omissions to not affect the semantics of the strategies.

For convenience, KURE provides a data type *KureM* with a *MonadCatch* instance (Fig. 2). A KURE user can either use *KureM*, or define a *MonadCatch* instance for her own monad. The monad used by HERMIT is a combination of *KureM* and GHC's *CoreM* monad:

```
data HermitM a = HermitM (CoreM (KureM a))
```

4.3 Translations and Rewrites

The central component of KURE is the following data type:

```
newtype Translate c m a b = Translate {apply :: c → a → m b}
```

The four type parameters are: the context (c), the underlying monad (m), the node to be transformed (a), and the result (b). Rewriting a node is just the special case when the result is a node of the same type, and KURE exploits this by defining rewrites as a type synonym:

```
type Rewrite c m a = Translate c m a a
```

The KURE library exposes several functions for constructing translations and rewrites. Those that will appear in this article are:

```
translate      :: (c → a → m b) → Translate c m a b
translate      = Translate

rewrite        :: (c → a → m a) → Rewrite c m a
rewrite        = translate

contextfreeT   :: (a → m b) → Translate c m a b
contextfreeT f = translate (λ _ a → f a)

constT         :: m b → Translate c m a b
constT mb      = contextfreeT (λ _ → mb)
```

Using these functions, the KURE user is able to define data-type–specific transformations.

4.4 Strategies

In KURE, transformation strategies are combinators operating over the *Translate* data type. KURE's approach to providing strategies is to make *Translate* an instance of well-known type classes such as *Monad* and *Arrow*, and then any monadic or arrow combinator gives rise to a strategy.

This section will discuss strategies that *do not* exploit the tree structure of the data being transformed; tree-traversal strategies will be considered in §5. As a reminder of this, we will refer to the data being transformed as values, rather than nodes. Furthermore, these strategies will not update the context, although they may *depend* on the context.

Our experience has been that there are three basic ways to combine translations:

1. Apply several translations to the same value (in the same context), and then combine the results of those translations.
2. Attempt to apply a translation to a value, and, if it fails, apply another translation to that value instead (in the same context).
3. Apply a translation to the result of another translation (in the same context).

The first situation has a *monadic* structure. Specifically, translations form a monad that corresponds to applying the Reader monad transformer to the underlying monad m, where the environment comprises the context c and the value a. As well as combining results, the monadic structure allows subsequent translations to *depend* on the results of previous translations. The details of the *Monad* instance declaration for *Translate c m a* are shown in Fig. 3, as well as for the related classes *Functor*, *Applicative* and *MonadCatch* (though note that these are all standard).

By making translations instances of *MonadCatch*, KURE provides the second way of combining translations. For example, we can define Stratego's <+ strategy:

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

The third way of combining translations has an *arrow* (Hughes, 2000) structure. Specifically, translations form an arrow that corresponds to applying the Reader arrow transformer to the Kleisli arrow induced by the underlying monad, where the environment is the context c. Notice the crucial difference: in the monadic case the environment is the context *and the value*, whereas in the arrow case the environment is *only the context*. Thus monadic composition (>>=) applies each translation *to the original value*, whereas arrow composition passes the result of each translation as the argument value to the next. The details

```
instance Functor m ⇒ Functor (Translate c m a) where
  fmap     :: (b → d) → Translate c m a b → Translate c m a d
  fmap f t = translate (λc a → fmap f (apply t c a))

instance Applicative m ⇒ Applicative (Translate c m a) where
  pure       :: b → Translate c m a b
  pure b     = constT (pure b)
  (<∗>)      :: Translate c m a (b → d) → Translate c m a b → Translate c m a d
  tf <∗> tb = translate (λc a → apply tf c a <∗> apply tb c a)

instance Monad m ⇒ Monad (Translate c m a) where
  return   :: b → Translate c m a b
  return b = constT (return b)
  fail     :: String → Translate c m a b
  fail msg = constT (fail msg)
  (≫=)     :: Translate c m a b → (b → Translate c m a d) → Translate c m a d
  t ≫= f   = translate $ λc a → do b ← apply t c a
                                   apply (f b) c a

instance MonadCatch m ⇒ MonadCatch (Translate c m a) where
  catchM       :: Translate c m a b → (String → Translate c m a b) → Translate c m a b
  catchM t1 t2 = translate (λc a → catchM (apply t1 c a) (λmsg → apply (t2 msg) c a))

instance Monad m ⇒ Category (Translate c m) where
  id      :: Translate c m a a
  id      = contextfreeT return
  ( ◦ )   :: Translate c m b d → Translate c m a b → Translate c m a d
  t2 ◦ t1 = translate (λc a → apply t1 c a ≫= apply t2 c)

instance Monad m ⇒ Arrow (Translate c m) where
  arr     :: (a → b) → Translate c m a b
  arr f   = contextfreeT (λa → return (f a))
  first   :: Translate c m a b → Translate c m (a,z) (b,z)
  first t = translate (λc (a,z) → liftM (λb → (b,z)) (apply t c a))

instance Monad m ⇒ ArrowApply (Translate c m) where
  app :: Translate c m (Translate c m a b,a) b
  app = translate (λc (t,a) → apply t c a)
```

Fig. 3. Type class instances for *Translate*.

of the *Arrow* instance declaration for *Translate c m* are shown in Fig. 3, as well as for the related classes *Category* and *ArrowApply* (again, these are standard). As the *Category* class methods *id* and $\circ$ clash with the Prelude functions of the same name, KURE uses the following synonyms when working with translations:

```
idR      :: Monad m ⇒ Rewrite c m a
idR      = id
(⋙)     :: Monad m ⇒ Translate c m a b → Translate c m b d → Translate c m a d
t₁ ⋙ t₂ = t₂ ∘ t₁
```

4.5 Example Strategies

Defining new strategies is as straightforward as in Stratego, for example:

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

For ease of reading, strategy combinators in this article are presented with their types specialized to *Translate* (or *Rewrite*), but in the KURE library many combinators are generalized to arbitrary monads. Experience with HERMIT has shown this generalization to be useful in practice, as it is sometimes more convenient to work at the level of the *Translate* monad, and sometimes more convenient to work at the level of the underlying monad. Additionally, having access to Haskell's **do**-notation at the *Translate* level has proved very useful; in particular, the invocation of *fail* on a pattern-matching failure enables a concise coding style. For example, rewrites that convert between a non-recursive let binding and the application of a lambda,

$$\textbf{let}\ v = e_1\ \textbf{in}\ e_2 \qquad \Longleftrightarrow \qquad (\lambda\ v \rightarrow e_2)\ e_1$$

can be defined succinctly as follows:

```
letToApp :: Monad m ⇒ Rewrite c m CoreExpr
letToApp = do Let (NonRec v e₁) e₂ ← idR
              return (App (Lam v e₂) e₁)

appToLet :: Monad m ⇒ Rewrite c m CoreExpr
appToLet = do App (Lam v e₂) e₁ ← idR
              return (Let (NonRec v e₁) e₂)
```

4.6 Detecting Change

The use of existing Haskell structures is one of the key improvements over the first standalone version of KURE, which did not make *Translate* an instance of these classes. One reason for this was that that it had an identity-detection mechanism that allowed identity rewrites to be caught using various combinators (Gill, 2009). This was useful when building larger rewrites, as we often want to detect not only whether a rewrite succeeded, but whether it actually changed anything. However, the mechanism would have caused a *Translate* instance of *Arrow* to violate the arrow laws. It was also rather fragile, relying on the programmer to correctly use a combinator called *transparently* to mark identity-preserving uses of *rewrite* and *translate*.

```
newtype AnyR m a = AnyR {unAnyR :: m (Bool, a)}
instance Monad m ⇒ Monad (AnyR m) where
  return     :: a → AnyR m a
  return a   = AnyR (return (False, a))
  fail       :: String → AnyR m a
  fail msg   = AnyR (fail msg)
  (≫=)       :: AnyR m a → (a → AnyR m d) → AnyR m d
  ma ≫= f    = AnyR $ do (b₁, a) ← unAnyR ma
                         (b₂, d) ← unAnyR (f a)
                         return ((b₁ ∨ b₂), d)
instance MonadCatch m ⇒ MonadCatch (AnyR m) where
  catchM        :: AnyR m a → (String → AnyR m a) → AnyR m a
  catchM ma f   = AnyR (catchM (unAnyR ma) (unAnyR ∘ f))
wrapAnyR   :: MonadCatch m ⇒ Rewrite c m a → Rewrite c (AnyR m) a
wrapAnyR r = rewrite $ λc a → AnyR (liftM (λa′ → (True, a′)) (apply r c a) <+ return (False, a))
unwrapAnyR   :: Monad m ⇒ Rewrite c (AnyR m) a → Rewrite c m a
unwrapAnyR r = rewrite $ λc a → do (b, a) ← unAnyR (apply r c a)
                                  if b then return a else fail "anyR failed"
```

Fig. 4. The *AnyR* monad transformer.

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

```
repeatR   :: MonadCatch m ⇒ Rewrite c m a → Rewrite c m a
repeatR r = r ⋙ tryR (repeatR r)
```

To support this convention further, KURE provides a monad transformer that converts a *Rewrite* defined for an arbitrary *MonadCatch* into a *Rewrite* that succeeds if *at least one* sub-rewrite succeeds, converting any failures into identity rewrites. Using this transformer requires applying a *wrapper* to the sub-rewrites that are allowed to fail, and an *unwrapper* on the outside of the composite rewrite. For example, a variant of ⋙ that succeeds if *either* rewrite succeeds can be defined using this transformer as follows:

```
(>+>)      :: MonadCatch m ⇒ Rewrite c m a → Rewrite c m a → Rewrite c m a
r₁ >+> r₂ = unwrapAnyR (wrapAnyR r₁ ⋙ wrapAnyR r₂)
```

Note that this differs from *tryR* r_1 ⋙ *tryR* r_2, which will always succeed.

KURE uses this monad transformer approach extensively to define its suite of shallow traversal combinators (see §5.3). The technique is taken from the SYB library, and we will not discuss the details of its implementation here. The interested reader can view the code for the *AnyR* transformer in Fig. 4.

5 Tree Traversals

Notably absent from the discussion of KURE thus far have been tree-traversal strategies. In this section we first consider traversal combinators that are specialized to known constructors, before addressing the problem of encoding Stratego's *generic* traversal strategies. The latter is more challenging, as assigning useful types to generic strategies is non-trivial.

To aid the forthcoming discussion, we will introduce some informal terminology (we will move to more formal terminology in §7.2). In KURE, we consider tree-structured data to be made up of two types of value: *nodes* and *leaves*. A node is a sub-tree, and may contain zero or more sub-nodes, which we call its *children*, as well as zero or more leaves. A leaf does not contain any nodes or sub-leaves. The children and leaves within a node may be encapsulated in arbitrarily nested data constructors. Children of a node *n* may themselves contain child nodes, but these "grandchild" nodes are not considered to be children of *n*. Instead, we use the term *descendant* node to include all children, and their own descendants. The distinction between nodes and leaves is that nodes can be targeted by generic traversal strategies, whereas leaves cannot (though primitive rewrites that modify a node can modify its leaves). For example, the *all* strategy should apply its rewrite to all children of its target node, but not its leaves. Consequently, a traversal will never descend into a leaf: essentially leaves are treated as atomic values by traversals. Which values KURE considers to be nodes and which to be leaves is dependent on their location and is entirely specified by the user. It is not determined by considerations such as the type of the value, the size of the value, or whether the value is a recursive data type (see §7.2 for a comparison with other approaches). Thus, while a leaf may contain values of the same type as nodes, those values would not be considered to *be* nodes.

5.1 Congruence Combinators

One of the distinguishing features of KURE is that translations may depend on a *context*. For example, in HERMIT we could define a traversal to collect all out-of-scope variables in an expression. Which variables are out-of-scope depends on the context (which stores all bindings in scope), and also on the bindings in the expression under consideration. During traversal of the expression, any bindings encountered need to be added to the context.

This is a recurring need. To take a related example, the HERMIT rewrite that inlines the definition of a named variable (by looking it up in the context) also needs to update the context with any new bindings as it traverses the expression.

Having each combinator explicitly update the context in this manner is repetitive and error prone. A better approach is to define for each node constructor a general-purpose traversal combinator that handles the context update, and then define all other combinators that manipulate that node constructor in terms of that combinator. These are called *congruence combinators* (Visser, 2004), and we recommend that the KURE user defines one for each data constructor that she wishes to traverse. This may seem like a lot of work, but our experience has been that there is a significant pay-off in the simplicity with which subsequent transformations can be defined.

More concretely, a congruence combinator constructs a translation over a node by combining translations to apply to that node's children with a function to combine the results of those translations with the leaves of the node. For example, the congruence combinator for the Lam constructor of *CoreExpr*, which has one child node *e* and one leaf *v*, is as follows[1]:

```
lamT :: Monad m ⇒ Translate HermitC m CoreExpr a → (Var → a → b)
                → Translate HermitC m CoreExpr b
lamT t f = translate $ λc expr → case expr of
                                   Lam v e → f v <$> apply t (addLamBinding v c@@0) e
                                   _       → fail "not a lambda node."
```

The important point is that the translation is applied to the child expression *in an updated context*. Specifically, the absolute path is updated with a number identifying the child descended into (in this case 0, as there is only one child), and the auxiliary function

```
addLamBinding :: Var → HermitC → HermitC
```

adds the variable *v* to the set of bindings in scope.

Some additional examples may make this clearer. The congruence combinators for App, which has two child nodes and zero leaves, and Var, which has zero child nodes and one leaf, are as follows:

```
appT :: Monad m ⇒ Translate HermitC m CoreExpr a1 → Translate HermitC m CoreExpr a2
                → (a1 → a2 → b) → Translate HermitC m CoreExpr b
appT t1 t2 f = translate $ λc expr → case expr of
                App e1 e2 → f <$> apply t1 (c@@0) e1 <*> apply t2 (c@@1) e2
                _         → fail "not an application node."
varT :: Monad m ⇒ (Var → b) → Translate HermitC m CoreExpr b
varT f = contextfreeT $ λexpr → case expr of
                Var v → return (f v)
                _     → fail "not a variable node."
```

In practice, we have also found it worthwhile to define a specialization of each congruence combinator that fixes the argument translations to be rewrites, and the combiner function to be the constructor of the node being transformed. For example:

```
appR :: Monad m ⇒ Rewrite HermitC m CoreExpr → Rewrite HermitC m CoreExpr
                → Rewrite HermitC m CoreExpr
appR r1 r2 = appT r1 r2 App
```

[1] Many components of KURE, including congruence combinators, only really require *m* to be an applicative functor (McBride & Paterson, 2008) augmented with a *fail* method, not a monad. Unfortunately, for historical reasons *Applicative* is not a superclass of *Monad* in the Haskell standard libraries, and the *fail* method is part of the monad class. However, as a monad *is* required for most top-level tasks in KURE (notably sequencing of rewrites), we made the design decision to keep the interface simple by always using a *Monad* constraint. For presentational purposes in this article, we assume an idealized setting where *Applicative* is a superclass of *Monad*, so that we can use applicative operators instead of the corresponding monadic combinators.

Such combinators exhibit behavior similar to the *all* strategy, except that they only succeed for a specific constructor, and take a separate rewrite argument for each child node. Indeed, in HERMIT we use such combinators to *define* the *all* strategy (see Fig. 6 in §5.3).

Finally, notice that the congruence combinators are specialized to the HERMIT *context*, but polymorphic in the *monad*. This is a deliberate design decision: congruence combinators need to be able to update the context, but should not require any non-proper morphisms of the monad. This polymorphism makes congruence combinators more reusable; for example, it allows them to be used in conjunction with monad transformers.

5.2 Typing Generic Traversals

We will now step back from the KURE implementation briefly, and discuss translations and rewrites at a more abstract, language-independent, level. While doing so, we will use **T** and **R** to denote translation and rewrite types, and will omit the *c* and *m* type parameters.

Consider the problem of assigning a type to the *all* strategy. The argument rewrite to *all* should operate on all child nodes, and the result should be a rewrite that operates on the parent node. However, the types of the children may differ from each other and their parent. Thus, a combinator of type

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

could only apply its argument to children *of the same type* — a significant shortcoming. An alternative would be to give *all* a more general type:

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

Unfortunately, this does not relate σ and τ at all, and so *all* cannot use the argument rewrite in any meaningful way—*without runtime type comparisons*. This is the approach taken by Dolstra & Visser (2001); Dolstra (2001), who invent a language with a type system that includes runtime type-case internally inside traversal combinators. This is also essentially the approach taken by StrategyLib using SYB, as we will discuss in §7.

Ideally, we would like traversal strategies to accept a *set* of potentially distinctly typed rewrites as an argument. Then, for example, *all* could take as its argument a set of rewrites, one for each child. In principal this idea works, but there are a number of shortcomings. First, the argument to *all* is difficult to generalize, given that it depends on the number of possible children of each node type. Second, and more importantly, the argument is now a set, not a **T**, and we are building a framework for manipulating **T**s. It would be *possible* to construct a new sequencing operation over sets, but this would not make good use of KURE's existing machinery.

A preferable approach is reuse the **R** type as the single argument to *all*. The challenge is finding a type *g* such that **R** *g* encodes a set of rewrites. In a language with algebraic data types such as Haskell, one such type is a sum type. That is, if a node of type τ has *n* possible children with types $\tau_1, \tau_2, \ldots, \tau_n$, then we can define:

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

and the type of *all* would therefore be:

$all :: \mathbf{R}\ (\tau_1 + \tau_2 + \ldots + \tau_n) \to \mathbf{R}\ \tau$

To enable composibility of traversals over different nodes, KURE generalizes this idea to a sum type containing all possible nodes in the tree we wish to traverse. The *all* strategy then also returns an **R** over this more general sum type:

$$all :: \mathbf{R}\ (\tau_1 + \tau_2 + \ldots + \tau_n) \to \mathbf{R}\ (\tau_1 + \tau_2 + \ldots + \tau_n)$$

$\tau_1, \tau_2, \ldots, \tau_n$ now represent all node types in the tree. That is, the argument of *all* is a rewrite over any node in the tree, and the result is a rewrite over any node in the tree.

Furthermore, the sum type now *determines* which children are considered to be nodes, and hence which should be rewritten by *all*. That is, at a first approximation, if a type appears as a summand then it should be considered a node for the purposes of this traversal, if not then it should be considered a leaf. Thus, parameterizing *all* over the sum type allows the user to *statically select* (§2.2) which types should be traversed. More precisely, each *location* in the tree that we wish to treat as a node should have an associated summand. Thus, if we wished to treat nodes of the same type but different location distinctly, then there would be multiple summands containing (a retraction of) the same type. We return to this point in more detail in §7.6.

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

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

That is, a rewrite could transform a child of one type into a child of another type. KURE addresses this problem by providing correct-by-construction promotion functions that convert non-type-preserving rewrites into failures, and using those promotion functions wherever type preservation is required (see §5.3).

5.3 Generic Shallow Traversals in KURE

We will now describe how KURE implements the ideas in §5.2, using the GHC-Core abstract syntax tree (Fig. 1) as a running example. First, the KURE user is required to define a sum type containing all values that the user wishes to treat as nodes. For some applications it may be desirable to use multiple sum types, but HERMIT just has one:

```
data Core = GutsCore ModGuts
          | ProgCore CoreProgram
          | BindCore CoreBind
          | DefCore  CoreDef
          | ExprCore CoreExpr
          | AltCore  CoreAlt
```

In strategic programming parlance, this sum type defines a *closed universe* of traversable types.

To allow KURE to automatically inject values into the sum type, and project them out, the user is required to declare an instance of the following class for each summand type:

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

```
injectT    :: (Monad m, Injection a g) ⇒ Translate c m a g
injectT    = arr inject

projectT   :: (Monad m, Injection a g) ⇒ Translate c m g a
projectT   = contextfreeT (maybe (fail "...") return ∘ project)

extractT   :: (Monad m, Injection a g) ⇒ Translate c m g b → Translate c m a b
extractT t = injectT ⋙ t

promoteT   :: (Monad m, Injection a g) ⇒ Translate c m a b → Translate c m g b
promoteT t = projectT ⋙ t

extractR   :: (Monad m, Injection a g) ⇒ Rewrite c m g → Rewrite c m a
extractR r = injectT ⋙ r ⋙ projectT

promoteR   :: (Monad m, Injection a g) ⇒ Rewrite c m a → Rewrite c m g
promoteR r = projectT ⋙ r ⋙ injectT
```

Fig. 5. Lifting and lowering combinators.

These instances are straightforward to define, as exemplified by the following instance for *CoreExpr*:

```
instance Injection CoreExpr Core where
  inject                    = ExprCore

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

Using *inject* and *project*, KURE provides a number of combinators that lift and lower translations and rewrites to and from a sum type (Fig. 5). Notice that a failing projection causes the entire translation to fail, and, consequently, an extracted rewrite over a sum type will fail if it does not preserve the summand type.

KURE's ability to traverse the tree is provided by the following type class:

```
class Walker c g where
  allR :: MonadCatch m ⇒ Rewrite c m g → Rewrite c m g
```

The two parameters are the context (c) and the sum type (g). Note that in the simple case where the tree only has one type of node, it is unnecessary to define a sum type and the node type can be used directly as the g parameter.

Defining *allR* is the main requirement of the KURE user. However, if the user has defined congruence combinators (§5.1) for all her nodes, then defining *allR* is just a systematic use of those combinators. Briefly, the idea is that *allR* takes a rewrite on the sum type as its argument, and extracts from that argument a rewrite for each possible child. These extracted rewrites are then applied to the children using congruence combinators, thereby ensuring that they are applied in the correct context. For each node type, the congruence combinators are combined using <+ to construct a rewrite that can handle any constructor of that type. These rewrites are then promoted to operate on the sum type, and then further combined using <+ to construct a rewrite that can handle any node type. This is easier to understand by example: see the *Walker* instance for *HermitC* and *Core* in Fig. 6.

The *Walker* class also provides a number of other shallow-traversal combinators. The definitions of these combinators are derived automatically from *allR*, but they are provided

```
instance Walker HermitC Core where
  allR   :: MonadCatch m ⇒ Rewrite HermitC m Core → Rewrite HermitC m Core
  allR r  =  promoteR allRmodguts <+ promoteR allRprog <+ promoteR allRbind
          <+ promoteR allRdef       <+ promoteR allRalt    <+ promoteR allRexpr
    where
      allRmodguts  :: MonadCatch m ⇒ Rewrite HermitC m ModGuts
      allRmodguts  =  modGutsR (extractR r)

      allRprog     :: MonadCatch m ⇒ Rewrite HermitC m CoreProg
      allRprog     =  progNilR <+ progConsR (extractR r) (extractR r)

      allRbind     :: MonadCatch m ⇒ Rewrite HermitC m CoreBind
      allRbind     =  nonRecR (extractR r) <+ recR (extractR r)

      allRdef      :: MonadCatch m ⇒ Rewrite HermitC m CoreDef
      allRdef      =  defR (extractR r)

      allRalt      :: MonadCatch m ⇒ Rewrite HermitC m CoreAlt
      allRalt      =  altR (extractR r)

      allRexpr     :: MonadCatch m ⇒ Rewrite HermitC m CoreExpr
      allRexpr     =  varR <+ litR <+ appR (extractR r) (extractR r) <+ lamR (extractR r)
                   <+ letR (extractR r) (extractR r) <+ caseR (extractR r) (extractR r)
                   <+ castR (extractR r) <+ tickR (extractR r) <+ typeR <+ CoercionR
```

Fig. 6. The *Walker* instance used by HERMIT.

as class methods with default definitions so that they may be overwritten if that would be beneficial for efficiency. For example, *anyR* is a variant of *allR* that applies its argument to all children, but succeeds if any of those rewrites succeed:

```
anyR :: MonadCatch m ⇒ Rewrite c m g → Rewrite c m g
anyR = unwrapAnyR ∘ allR ∘ wrapAnyR
```

Note that this would not be possible if the monad was a parameter to the *Walker* class, rather than being polymorphic, as *anyR* invokes *allR* with a transformed monad (using the *AnyR* monad transformer from Fig. 4).

Using a similar monad-transformer technique, the *Walker* class additionally provides the following combinators:

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

5.4 Generic Deep Traversals in KURE

Using the shallow-traversal infrastructure from §5.3, it is now straightforward to define deep traversals. For example, the *bottomup* and *topdown* traversals of Stratego can be defined in the same way in KURE:

```
allbuR, alltdR :: (Walker c g, MonadCatch m) ⇒ Rewrite c m g → Rewrite c m g
allbuR r = allR (allbuR r) ⋙ r
alltdR r  = r ⋙ allR (alltdR r)
```

The strategies have been renamed to *allbuR* and *alltdR* because there are other possible "top-down" or "bottom-up" traversal strategies. Whereas *allbuR* and *alltdR* require the argument rewrite to succeed at all nodes in the tree, KURE also provides *any* variants that succeed if the argument rewrite succeeds anywhere, and *one* variants that rewrite only the first descendent node at which the argument rewrite succeeds:

```
anytdR, onetdR, anybuR, onebuR :: (Walker c g, MonadCatch m) ⇒ Rewrite c m g → Rewrite c m g
anytdR r  = r >+> anyR (anytdR r)
onetdR r  = r  <+  oneR (onetdR r)
anybuR r = anyR (anybuR r) >+> r
onebuR r = oneR (onebuR r)  <+  r
```

As an example of a more complex traversal strategy, *innermostR* repeatedly applies a rewrite until a fixed point is reached, starting with the innermost (bottom) node and working outwards:

```
innermostR   :: (Walker c g, MonadCatch m) ⇒ Rewrite c m g → Rewrite c m g
innermostR r = anybuR (r ⋙ tryR (innermostR r))
```

KURE also provides deep-traversal translations, based on the *allT* and *oneT* shallow traversals. For example, the following traversals perform a fold on the tree, in either a top-down or bottom-up manner:

```
foldtdT, foldbuT, crushtdT, crushbuT :: (Walker c g, MonadCatch m, Monoid b)
                                      ⇒ Translate c m g b → Translate c m g b
foldtdT  t   = mappend t (allT (foldtdT t))
foldbuT t    = mappend (allT (foldbuT t)) t
crushtdT  t = foldtdT  (t <+ mempty)
crushbuT t = foldbuT (t <+ mempty)
instance (Monad m, Monoid b) ⇒ Monoid (Translate c m a b) where
   mempty   :: Translate c m a b
   mempty   = return mempty
   mappend :: Translate c m a b → Translate c m a b → Translate c m a b
   mappend = liftM2 mappend
```

The difference between the *foldT* and *crushT* variants is that the former require the argument translate to succeed everywhere, whereas the latter do not require any successes, replacing failures with the unit of the monoid.

A specialization of *crushtdT* that we have found useful is a traversal that applies a translation everywhere, collecting the results in a list. For more efficient accumulation of results, we work in the *difference-list* (Hughes, 1986) monoid rather than the list monoid:

```
collectT   :: (Walker c g, MonadCatch m) ⇒ Translate c m g b → Translate c m g [b]
collectT t = toList <$> crushtdT (singleton <$> t)
   -- operations provided by the dlist library (Stewart, 2009)
singleton :: a → DList a
toList      :: DList a → [a]
```

Finally, the following strategies traverse the tree, returning the result of applying the argument translation to the first node for which it can succeed:

$onetdT, onebuT :: (Walker\ c\ g, MonadCatch\ m) \Rightarrow Translate\ c\ m\ g\ b \rightarrow Translate\ c\ m\ g\ b$

$onetdT\ \ t = t <\!\!+ oneT\ (onetdT\ t)$
$onebuT\ t = oneT\ (onebuT\ t) <\!\!+ t$

Notice that there is no monoid constraint for these strategies, as only one result is returned.

5.5 Lenses

KURE also provides a *lens* abstraction (Foster *et al.*, 2007). In our (specialized) setting, a lens is a tool for directing a transformation to a specific descendant node in the tree[2]. For example, given a lens from a node of type *a* to a descendant node of type *b*, KURE provides combinators that allow a rewrite or translation to be applied to that descendant:

$focusR :: Monad\ m \Rightarrow Lens\ c\ m\ a\ b \rightarrow Rewrite\ c\ m\ b \rightarrow Rewrite\ c\ m\ a$
$focusT :: Monad\ m \Rightarrow Lens\ c\ m\ a\ b \rightarrow Translate\ c\ m\ b\ r \rightarrow Translate\ c\ m\ a\ r$

Note that *focusR* is *not* a special case of *focusT*, as the two have different semantics: *focusR* rewrites the descendant node, maintaining the structure of the rest of the tree; whereas *focusT* transforms the descendant node into a result value, discarding the rest of the tree.

Lenses are implemented as a specialized type of translation, using a newtype wrapper so that a separate *Category* instance can be defined for *Lens*:

$\mathbf{newtype}\ Lens\ c\ m\ a\ b = \mathsf{Lens}\ (Translate\ c\ m\ a\ ((c,b), b \rightarrow m\ a))$

That is, as a translation from a node of type *a* to a pair: a descendant node of type *b* with an accompanying context, and a monadic function that reconstructs the node of type *a* by replacing the original descendant node with its argument. We call this function that returns to the original node "the kick", a general term sometimes used for the mechanism for returning to an encapsulating environment (e.g. Nolan (2010)).

The desired *focus* combinators can now be defined as follows:

```
focusR              :: Monad m ⇒ Lens c m a b → Rewrite c m b → Rewrite c m a
focusR (Lens l) r = do ((c,b),k) ← l
                       constT (apply r c b >>= k)

focusT              :: Monad m ⇒ Lens c m a b → Translate c m b r → Translate c m a r
focusT (Lens l) t = do ((c,b),_) ← l
                       constT (apply t c b)
```

Finally, a function that constructs a lens to the *n*th child of a node is an additional method of the *Walker* class:

$childL :: (Walker\ c\ g, MonadCatch\ m) \Rightarrow Int \rightarrow Lens\ c\ m\ g\ g$

A default definition of *childL* is derived from *allR*, albeit somewhat inefficiently compared to typical user definitions. Thus a KURE user concerned with efficiency and making heavy use of *childL* is advised to overwrite the default definition.

[2] The *b* parameter is not restricted to being a descendant node: *b* could be any type such that there exists a section of type $a \rightarrow b$ and corresponding retraction of type $b \rightarrow a$. However, here we refer to *b* as a descendant node, as that special case is the principal use of KURE's *Lens* type.

6 Additional HERMIT Examples

To further demonstrate how KURE can be used in practice, this section presents some additional transformations from the HERMIT package. Note that some of these examples have been slightly simplified, in cases where we felt that the minutiae of HERMIT were causing obfuscation.

To simplify the forthcoming type signatures, we begin by defining synonyms for translations and rewrites specialized to the HERMIT context and monad:

```
type TranslateH a b = Translate HermitC HermitM a b
type RewriteH a     = TranslateH a a
```

Now, consider a transformation that floats a local let-binding to the top level:

$$\begin{array}{lcl} v_1 = (\textbf{let}\ v_2 = e_2\ \textbf{in}\ e_1) & & v_2 = e_2 \\ \ldots & \Longrightarrow & v_1 = e_1 \\ & & \ldots \end{array}$$

As with the *appToLet* and *letToApp* rewrites (§4.5), this can be defined concisely by pattern matching,

```
letFloatLetTop :: RewriteH CoreProgram
letFloatLetTop = do NonRec v1 (Let (NonRec v2 e2) e1) : bds ← idR
                    return (NonRec v2 e2 : NonRec v1 e1 : bds)
```

with the rewrite failing if the pattern match fails.

However, transformations often have constraints that cannot be expressed purely by pattern matching. For example, a translation to flatten a program (which is a list of binding groups) into a single recursive binding group should fail if the same name is bound in more than one group. This can be achieved by checking if there are any duplicated names, and invoking *fail* if so:

```
flattenProgram :: TranslateH CoreProg CoreBind
flattenProgram = do bds ← arr (concatMap binds)
                    if nodups (map fst bds)
                      then return (Rec bds)
                      else fail "name bound multiple times"
  where
    binds              :: CoreBind → [(Var, CoreExpr)]
    binds (NonRec v e) = [(v, e)]
    binds (Rec bds)    = bds

    nodups             :: Eq a ⇒ [a] → Bool
    nodups as          = length as ≡ length (nub as)
```

As another example, consider eliminating a let-expression with a single unused non-recursive binding:

$$\textbf{let}\ v = e_1\ \textbf{in}\ e_2 \quad\Longrightarrow\quad e_2 \qquad (\text{if } v \text{ does not occur free in } e_1 \text{ or } e_2)$$

Here there is both the syntactic constraint (the expression must be a let with a single non-recursive binding), and the requirement that the variable does not occur freely in e_2.

Expressing the two constraints is just a matter of combining the two previous approaches, which we do here using a (GHC provided) function that computes the free variables in an expression:

```
letElim :: RewriteH CoreExpr
letElim = do Let (NonRec v _) e ← idR
             if v ∉ coreExprFreeVars e
               then return e
               else fail "let-bound variable appears in expression"
```

In §4.5 we claimed that it is sometimes convenient to work at the level of the underlying monad, rather than at the level of the *Translate* monad. As an example of this, consider a transformation that introduces a let-expression:

$e \quad \Longrightarrow \quad \textbf{let}\ v = e\ \textbf{in}\ v$

This transformation introduces a new variable (v), and so a monadic operation that generates a globally fresh variable is required:

```
letIntro :: RewriteH CoreExpr
letIntro = contextfreeT $ λ e → do v ← newVar
                                  return (Let (NonRec v e) (Var v))
newVar :: HermitM Var   -- defined using GHC-provided functions
```

HERMIT also makes use of KURE's library of strategies to build new transformations. For example, the *simplify* rewrite used by Sculthorpe *et al.* (2013) is defined using *innermostR* to repeat a set of simplification rewrites until no more changes can be made:

```
simplify :: RewriteH Core
simplify = innermostR (promoteR (appToLet <+ safeLetSubst <+ caseReduce <+ letElim))
```

Finally, we note that congruence combinators (§5.1) are often useful in combination with library strategies. For example, HERMIT has a `consider` command that focuses on the binding site of a named variable (Farmer *et al.*, 2012). To provide tab-completion to the user when typing this command, HERMIT uses the following translation to collect a list of all visible bindings:

```
considerTargets :: TranslateH Core [Var]
considerTargets = allT (collectT (promoteT nonRec <+ promoteT def))
  where
    nonRec :: TranslateH CoreBind Var
    nonRec = nonRecT (return ()) (λv () → v)
    def    :: TranslateH CoreDef Var
    def    = defT (return ()) (λv () → v)
```

Notice that we provide a constant translation that returns () to the congruence combinator in both cases; this is because we are only interested in retrieving the bound variable; the expression itself can be ignored. The *allT* strategy is used to exclude the top node from the traversal, as that is the desired behaviour of HERMIT's tab completion.

7 Comparison of KURE to SYB and Uniplate

In this section we compare and contrast KURE's support for traversals with two other libraries with similar traversal infrastructure: Scrap Your Boilerplate (SYB) (Lämmel & Peyton Jones, 2003) and Uniplate (Mitchell & Runciman, 2007). We focus on the libraries' shallow traversal combinator (§7.1); in particular, its semantics (§7.2) and type (§7.4). We also discuss the libraries' support for modularity (§7.5), abstraction (§7.6), and strategic programming (§7.7), as well as the features that are unique to KURE (§7.8). We postpone considering the relative performance of the libraries until §8.

7.1 The Shallow Traversal

The only shallow traversal we shall compare is that which requires success at every targeted node: KURE's *allR*. The analogue in SYB is *gmapM*, while Uniplate has two analogues: *descendM* and *descendBiM*. The KURE library implements all of its other shallow traversals, such as *anyR* and *oneR*, in terms of *allR*. Similar default definitions are given in SYB. Uniplate provides no such definitions, but there is no reason why it could not do so. This shallow traversal is therefore a complete representation of a library's support for traversal.

For comparative purposes, we simplify the actual definition of this shallow traversal in each library to a conceptual kernel:

```
  -- SYB kernel
class Typeable τ ⇒ Data τ where
  gmapM :: Applicative m ⇒ (∀ σ. Data σ ⇒ σ → m σ) → τ → m τ
  -- Uniplate kernel
class Uniplate τ where
  descendM :: Applicative m ⇒ (τ → m τ) → τ → m τ
class Uniplate τ ⇒ Biplate σ τ where
  descendBiM :: Applicative m ⇒ (τ → m τ) → σ → m σ
  -- KURE kernel
class Walker g where
  allR :: Monad m ⇒ (g → m g) → g → m g
```

For SYB and Uniplate, we have simplified the classes in two distinct ways. First, we take the shallow traversal operator as the only class method, even though the actual classes in the libraries define this method by default in terms of another (*gfoldl* and *uniplate*/*biplate*, respectively). Second, we have changed the *Monad* constraints to *Applicative* constraints, because the monad interface is strictly more expressive than necessary for the shallow traversal's semantics (as we discuss in §7.8.1). We suspect that the reason SYB and Uniplate use *Monad* instead of *Applicative* is because they originated before McBride & Paterson (2008) popularized the *Applicative* abstraction. In contrast to the other libraries, the KURE kernel actually requires a *Monad* constraint, as we explain in §7.4. We also omit KURE's context parameter, discussion of which we relegate to §7.8.2.

7.2 The Semantics of the Shallow Traversal

We will now characterize each library in terms of what the user must provide in order to use the library within their strategic program. First let us introduce some terminology.

A *traversable type* is a type that the shallow traversal operator can be applied at; that is, a type that a traversal can descend into. A *target* is a value that should be re-written during a shallow traversal by the application of the shallow traversal's argument rewrite. In all the libraries we consider, the types of all possible targets are a subset of the traversable types. In KURE, targets are called child nodes (§5).

A *rewrite system* is the set of user declarations that cumulatively determine the targets, the traversable types, and how each type is to be traversed.

- There is always exactly one SYB rewrite system in scope. That rewrite system is the set of *Data* instances and the requisite *Typeable* instances. The traversable types are those with a *Data* instance.
- Multiple Uniplate rewrite systems can be in scope simultaneously, each indexed by a target type τ. Each rewrite system is the *Uniplate* instance for a type τ and the set of *Biplate* instances that have τ as the second parameter (the target type). The traversable types are τ and those types that appear as the first parameter of a *Biplate* instance that has τ as the second parameter.
- Multiple KURE rewrite systems can be in scope simultaneously, each indexed by a sum type g. Each rewrite system consists of an instance of *Walker* for g, and *Injection* instances embedding each summand type into g. The traversable types are those with an *Injection* instance for g.

Thus there is exactly one way to traverse a type with SYB in a program, whereas a type can be traversed in multiple ways in Uniplate and KURE. This is because Uniplate's *Biplate* class and KURE's *Injection* class have an additional index (the target type (τ) and the sum type (g), respectively), and how a type is traversed depends on this index. Consequently, for SYB to achieve traversals as precise as those in KURE and Uniplate, it is necessary to create newtype-based abstract types (see §7.6).

To explain each library's traversal semantics, we must establish some additional terminology. We say that 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 as the whole of x. Thus a proper component is one constructor deep, a component is zero or one constructors deep, a proper substructure is one or more constructors deep, and a substructure is zero or more constructors deep. Two substructures are *independent* if neither is a substructure of the other. The *similarly typed* (proper) components/substructures of x are those whose value inhabits the type of x. The *maximal* elements of a set of locations are those that are not a proper substructure of any other element.

We can now describe the semantics of each library's shallow traversal:

- The SYB shallow traversal operator, *gmapM*, targets every proper component.
- For a given target type τ, the Uniplate library has two primary shallow traversal operators. The first, *descendM*, is applied to a value of type τ, and targets the maximal proper substructures of type τ. The second, *descendBiM*, is applied to a value of some traversable type σ, and targets the maximal substructures of type τ. Note that a Uniplate shallow traversal can target locations that are under a long chain of constructors, even navigating through other types that are not themselves traversable.

- For a given sum type *g*, the KURE shallow traversal, *allR*, projects to a summand type, targets zero or more independent proper substructures whose values inhabit one of the summand types, and then injects back into *g*. For a conventional sum type, the projection is a case expression revealing a sum-type constructor and a summand value, and the injection reapplies that same constructor. As in Uniplate, the shallow traversal operator can descend under multiple constructors and through other types.

7.3 Example Rewrite Systems

Throughout this section we will use the following data types as a running example:

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

They model a small expression language with non-recursive let-bindings for building strings. The mutual recursion among the data types will help contrast the libraries.

Following the semantics in §7.2, we declare SYB, Uniplate, and KURE rewrite systems for an initial example: targeting *Expr* locations and traversing *Expr* values. In §7.5 we will consider rewrites that also target *Decl* and that traverse both *Prog* and *Decl*.

The SYB rewrite system is the *Typeable* and *Data* instances for *Expr* and *Decl*. In practice, these instances are all equivalently derived by GHC, but for our comparison we give explicit *Data* instances and treat *gmapM* as the only method of *Data*:

```
deriving instance Typeable Expr
deriving instance Typeable Decl
instance Data Expr where
  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 r (Decl n e) = Decl <$> r n <*> r e
```

These instances rely on the standard *Typeable* and *Data* instances for [ ] and *Char*. The library user is forced to make *Decl* a targetable type: some *Expr* values have proper components of type *Decl*, and the semantics of *gmapM* require all proper components to be targeted. However, nothing prevents the user from declaring semantically invalid instances that, for example, do not target variable names or string literals. The consequences, though, would be that any third-party code may 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 different traversals to make different decisions as to whether to target a type. We identify one workaround based on lightweight data abstraction in §7.6.

By contrast, the semantics of *descendM* are that it should target only similarly typed proper substructures, and the semantics of *allR* allow the targets to be whichever proper substructures the user desires. Thus it is possible to have *Expr* as the only targetable type. Moreover, in this case, the two libraries' traversals are effectively the same because there is only one target type:

```
wExpr                    :: Applicative m ⇒ (Expr → m Expr) → Expr → m Expr
wExpr _ (Var n)          = Var     <$> pure n
wExpr _ (Lit s)          = Lit     <$> pure s
wExpr r (Append e0 e1)   = Append  <$> r e0        <*> r e1
wExpr r (Let d e)        = Let     <$> wDecl r d <*> r e

wDecl                    :: Applicative m ⇒ (Expr → m Expr) → Decl → m Decl
wDecl r (Decl n e)       = Decl n <$> r e

  -- Uniplate
instance Uniplate Expr where
  descendM = wExpr

instance Biplate Expr Expr where
  descendBiM = id

  -- KURE
data G = GE Expr    -- degenerate singleton sum

instance Injection Expr G where
  inject          = GE
  project (GE e)  = Just e
  project _       = Nothing    -- robust against extensions of G

instance Walker G where
  allR r (GE e) = GE <$> wExpr (extractR r) e
```

Indeed, the *Expr* type itself could be used instead of *G* as the KURE sum type, in which case *allR* could also be defined directly as *wExpr*.

7.4 The Type of the Shallow Traversal

The type of the shallow traversal operator in each library is intertwined with its semantics. In SYB the *gmapM* method must target every proper component of every value traversed, and thus the rewrite argument 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 (but not GADTs). As demonstrated in §7.5, the run-time type case enabled by *Typeable* provides the extreme modularity that is unique to SYB rewrite systems, but, as we discuss in §8.1, has a detrimental effect on efficiency.

The Uniplate shallow traversal operators have simpler types. The *descendM* traversal targets only similarly typed locations, and thus its rewrite argument need only be applicable at that type. The type of *descendBiM* is more general, allowing the target type and traversable type to differ.

KURE's traversal semantics is less reflected by the traversal's type than in the other two libraries because KURE does not use the type system to distinguish sum types from traversable types. A consequence of this is that the *Monad* interface is required instead of the less expressive *Applicative* interface that suffices for the other libraries. The argument rewrite to *allR* acts on a sum type, and thus it might convert one summand to another. After applying the rewrite, it must then project the result back to the type that was originally injected into the sum. Because that projection may fail, composing rewrites together during traversal requires monadic composition. In the instances in this section, this composition is hidden inside the use of *extractR* (from Fig. 5).

The type of a rewrite in each library provides different information to the user. In particular, the type of a rewrite in SYB gives no indication of what the rewrite does: because of the rank-2 polymorphism it can do interesting things at nearly any type. The situation is reversed in Uniplate: the type of a Uniplate rewrite specifies the target type. The information in the type of a KURE rewrite lies instead on a spectrum between the other two libraries, depending on the number of summands in the sum type. Even though it is the most informative of the three, the type of a Uniplate rewrite remains a coarse indication of what the rewrite actually does. For example, the type does not indicate at what depth changes can be made, nor whether the depth of the tree can be increased or reduced.

In summary, the SYB typing uses the richest type features, especially simulating dynamic typing via *Typeable*-bounded rank-2 polymorphism. The Uniplate typing is the simplest, involving at most a multiparameter type class. The KURE typing is essentially a stylized use of the *Uniplate* class, with the unenforced constraint that the rewrite argument acts on a sum type. The requisite projection back from the sum type requires a more expressive interface than the *Applicative* that suffices in SYB and Uniplate.

7.5 Modularity

To demonstrate the modularity aspects of each library, we now separately consider adding traversable and target types to our example language.

7.5.1 Adding Traversable Types

In §7.3, we only considered rewriting *Expr* values within another *Expr* value. Now imagine that we require the *Decl* and *Prog* types to also be traversable. For the SYB rewrite system, we need only declare a *Data* instance for *Prog*. As *Decl* is mutually recursive with an existing traversable type, the *gmapM* semantics already forced it to be traversable.

To add *Decl* and *Prog* as traversable types to the Uniplate rewrite system, we need only add *Biplate* instances:

```
instance Biplate Decl Expr where
  descendBiM = wDecl

instance Biplate Prog Expr where
  descendBiM = wProg

wProg                   :: Applicative m ⇒ (Expr → m Expr) → Prog → m Prog
wProg r (PrgCons d p) = PrgCons <$> wDecl r d <*> wProg r p
wProg r PrgNil          = pure PrgNil
```

The *Uniplate Expr* instance does not need to be changed. However, note that the definitions of *wExpr* and *wDecl* (defined in §7.3) are now both equivalent to instance methods. Indeed, it is common to define *Uniplate* instances that rely on *Biplate* instances in the first place.

To add *Decl* and *Prog* as traversable types to the KURE rewrite system, the user must change *G* to additionally have both types as summands, provide the necessary *Injection* instances, and extend the *Walker* instance:

```
data G = GE Expr | GD Decl | GP Prog
```

```
instance Injection Decl G where
  inject          = GD
  project (GD d)  = Just d
  project _       = Nothing
instance Injection Prog G where
  inject          = GP
  project (GP p)  = Just p
  project _       = Nothing
instance Walker G where
  allR r (GE e)  = GE <$> wExpr (extractR r) e
  allR r (GD d)  = GD <$> wDecl (extractR r) d
  allR r (GP p)  = GP <$> wProg (extractR r) p
```

Adding a traversable type just requires adding an instance in SYB and Uniplate, whereas in KURE, because of the reliance on the sum type, it is necessary to modify existing code.

7.5.2 Adding Targets

Instead of adding traversable types, we now add target locations that inhabit *Decl*. The SYB rewrite system can be used without any changes, because *Decl* is already an instance of *Data*, whereas the Uniplate and KURE rewrite systems require very different changes.

Uniplate is founded on the premise that there is only one target type per rewrite system. To have two target types, we would need two Uniplate rewrite systems and hence two traversals, one per target type. In our example, the existing *Uniplate* instance would stay and a new one would be declared for *Decl*. There would also need to be *Biplate* instances if *Expr* values were to be rewritten within *Decl* values, and vice-versa. We omit the code.

This can lead to a quadratic number of *Biplate* instances if done naïvely, which is known as the "direct" method of defining instances (Mitchell & Runciman, 2007). To address this, the Uniplate library offers alternative means of defining instances that rely on SYB to various degrees. However, note that these alternatives merely reduce the number of required instances by making each polymorphic: they do not permit a traversal to have multiple target types. This is the only major disadvantage of Uniplate: to rewrite targets of different types, the user must use a sequence of traversals. This is particularly problematic if a desired traversal with multiple target types uses monadic effects in a way that precludes the traversal from being decomposed into sequential traversals with one target type each.

To add some *Decl* targets in the KURE rewrite system, the user would again add *Decl* as a summand but also update the *Walker G* instance to additionally target locations of type *Decl*. As KURE is not as type-directed as Uniplate, we may choose that some locations of type *Decl* are not targets. However, here we choose that all *Decl* locations are targets:

```
data G = GE Expr | GD Decl
instance Walker G where
  allR r (GE e)  = GE <$> wE e
                 where
                   wE (Var n)           = Var    <$> pure n
                   wE (Lit s)           = Lit    <$> pure s
                   wE (Append e0 e1)    = Append <$> extractR r e0 <*> extractR r e1
                   wE (Let d e)         = Let    <$> extractR r d  <*> extractR r e
```

```
allR r (GD d) = GD <$> wD d
                    where
                      wD (Decl n e) = Decl n <$> extractR r e
```

Here we see that KURE's use of a sum type supports traversals with multiple types, but inhibits reuse: the sum type must be extended with the types of all target locations. If a non-summand type is required to be a target, the user has two options: change existing code or define a new rewrite system indexed by a distinct sum type.

7.5.3 Summary

The modularity of KURE rewrite systems is hindered by the (closed) sum type, which has to be modified whenever a new traversable type is added to a rewrite system. Adding a traversable type requires adding a constructor to the sum type and declaring a corresponding *Injection* instance. The *Walker* instance also has to be updated, but some simple conventions allow existing *Injection* instances to remain unchanged. When adding target locations, the *Walker* instance must be changed, and the types of the new locations (if not already traversable) must be added as traversable types.

Each Uniplate rewrite system is modular with respect to new traversable types, requiring only a *Biplate* instance. However, as there can be only one target type in a Uniplate rewrite system, adding a new target type requires defining a distinct rewrite system. While modular in the sense that this does not require modifying existing code, this is immodular in that rewrites with different target types cannot be combined into a single traversal.

In contrast to both KURE and Uniplate, each SYB rewrite system is wholly modular: adding a new target or traversable type simply requires a *Data* instance for that type without any changes to existing code. Ultimately, the set of traversable types is open in SYB and Uniplate rewrite systems, but closed in a KURE rewrite system, and the set of target types is open in SYB rewrite systems, but closed in Uniplate and KURE rewrite systems.

7.6 Data Abstraction

The semantics of the SYB and Uniplate shallow traversals were originally defined (Lämmel & Peyton Jones, 2003; Mitchell & Runciman, 2007) as we have defined them: in terms of the traversed value's concrete shape (its constructors and arguments). However, in practice these traversals have transcended their original semantics in order to accommodate abstract data types. For example, the standard library `containers` defines *gmapM* for the abstract *Map* type as if a *Map* contained the corresponding association list as its only proper component, by using the conversion functions *toList* and *fromList*. The Uniplate library does not itself define traversals for abstract types, but it does provide combinators to help users define such traversals. Thus, while both libraries' traversal semantics involve only concrete substructures, the actual Haskell definitions allow the use of *section-retract pairs*[3]. A traversal over an abstract data type first concretizes via the section (e.g. *toList*), traverses the concrete value, and then re-abstracts via the retract (e.g. *fromList*).

[3] Two functions *section* and *retract* form a section-retract pair if $retract \circ section \equiv id$, with isomorphism pairs being a special case.

Abstraction enables a unifying interpretation of the three libraries' semantics. While traversals over abstract data types use section-retract pairs to respect the data abstraction by instead traversing a concretization, traversals over concrete data types can also be understood as a trivial traversal of a corresponding abstraction. In this way, every traversal retracts the traversable value to an *n*-ary tuple of the target locations, targets each field of the *n*-tuple, and then re-concretizes by applying the corresponding section. In SYB, *gmapM* uses the constructor arguments as the fields of the *n*-tuple and the section is the fully uncurried constructor. In KURE, the target locations that constitute the *n*-tuple are determined by the *Walker* instance, and the type of each location must be a summand type. In Uniplate, the actual user-defined operation (*uniplate*) explicitly performs a retraction to a list of target locations (a list is used rather than an *n*-tuple because in Uniplate all targets have the same type), and provides a section to rebuild the original value from a list of the same length. The default implementation of *descendM* calls *uniplate*, targets each element of the list, and then applies the section.

The SYB approach is conventionally at a significant efficiency disadvantage because its semantics require the traversal to visit all proper components of a value, even if they are not intended targets. We discuss this further in §8.1, but an alternative is made possible by abstraction: the user can create a simple newtype with a manually defined *Data* instance that uses a statically selective traversal just like the other libraries. This sort of abstraction blurs the distinction between the libraries, allowing each to emulate the others' expressiveness.

For example, there are two distinct occurrences of *String* within our *Expr* data type: as variable names and as string literals. While the user may sometimes want to target both in the same traversal, it is likely that sometimes she will want to target just one or the other. Similarly, she may wish to treat either names or literals atomically, and not have a deep traversal descend into them. Yet the intended semantics of SYB and Uniplate (but not KURE, which is more flexible) mandate that any traversal targeting *String* must target both names and literals, and that a deep traversal must recursively target the tail of each string. Both libraries can use abstract data types to avoid these problems in multiple ways.

In Uniplate, this can be achieved by defining a new rewrite system over an abstract target type, which can be as simple as a newtype wrapper. This instance could, for example, treat *VarName*s atomically, and justify it semantically by considering the corresponding *String* to be a concretization of an abstract variable name.

A similar technique can be used in SYB. However, because the target type is not an index for *Data* instances, it is also necessary to provide abstract traversable types, whose abstract semantics determine the abstract target locations of any traversal of that type. This overcomes the limitation of there only being one *Data* instance in scope, as each abstract type has its own *Data* instance. Even though SYB requires the additional use of abstract traversable types, both SYB and Uniplate ultimately have equal support for abstraction: it is simply a matter of where to place the newtype wrappers.

KURE supports abstract types in a similar manner, but it also provides an alternative approach. As the sum type is (usually) distinct from the traversable types, the sum type itself can index the section-retract pairs used by traversals. For example, the *Walker* instance for one sum type could target variable names but not string literals, whereas the *Walker* instance for another sum type could treat both as atomic values. Newtypes are only needed to support *Injection* instances where multiple summands would share the same type.

7.7 Support for Strategic Programming

We now switch focus from only traversal to support for strategic programming. KURE offers the most comprehensive set of strategy combinators, but in principle similar combinators could be added to the other libraries, because all of the libraries support first-class rewrites with arbitrary monadic effects.

All three libraries exhibit four qualities crucial for first-class support of rewrites:

1. Translations can be composed in sequence.
2. Rewrites can be used as translations without an explicit coercion.
3. Polymorphic deep traversals are recursively defined in terms of shallow traversals.
4. The argument to the shallow traversal is compatible with rewrites.

The first quality is satisfied because all three libraries use Kleisli arrows as translations. The second follows because the libraries use endomorphic translations as rewrites. The third and fourth qualities deserve more discussion.

All three libraries allow shallow traversals to be used recursively. In Uniplate and KURE this is straightforward because the involved typing is simple. In SYB the rank-2 polymorphism complicates matters slightly; specifically, *gmapM*'s argument rewrite has a *Data* constraint. Even though only a *Typeable* constraint would suffice for shallow traversals, the *Data* constraint is required for the rewrite to be usable within the definitions of datatype–generic deep traversals. The use of *Data* is usually not a burden on the user, because GHC can derive a *Data* instance for most algebraic data types.

All three libraries support the fourth quality, but KURE's support is less complete. There are implicit coercions in SYB and Uniplate between the traversal arguments and rewrites. The rank-2 polymorphism of SYB causes no issue because the GHC support for higher-rank polymorphism allows the argument to be used directly as a monomorphic rewrite and also allows a suitably ad-hoc polymorphic rewrite to be used directly as an argument. The Uniplate case is again trivial, because the argument is simply a rewrite. In KURE though, there are effectively two sorts of translate: those over a summand type and those over the sum type. The *allR* domain and range are rewrites on the sum type. The user must therefore convert between the two sorts with *extractR* and *promoteR*, except in the degenerate case where there is only one summand type, which can be used directly rather than introducing a singleton sum type.

7.8 Distinguishing KURE Features

In this section we discuss the features of KURE that are not present in SYB and Uniplate: the sum type and the context.

7.8.1 The KURE Sum Type

The previous version of KURE parameterized the *Walker* class on a traversable type rather than a sum type, and included an associated type function to assign a sum type to the traversable type (Farmer *et al.*, 2012). Each traversable type required a *Walker* instance (although the overall code burden on the programmer was only moderately larger than the current design). The main consequence of this was that each traversable type could only appear in a single sum type (without newtype wrappers). The (minor) benefit of this was that it aided type inference, but it made defining multiple sum types awkward. Additionally, the presence of the type function complicated the types of many operators throughout the KURE library. Consequently, we decided to adopt the current design, adding type signatures where necessary.

Having made this change, the current KURE design can be considered to be a stylized instantiation of the Uniplate architecture in which the shallow traversal targets locations with distinct types by using the *Injection* class to coerce back-and-forth to the sum type. We summarize the benefits and detriments.

KURE improves upon SYB in the exact same way that Uniplate does: more direct support for static selectivity, fewer complicated types, and a rewrite's type better indicates the types on which it can succeed. Thus KURE has the major expressive benefits of Uniplate, but also lifts its major restriction: KURE supports multiple target types. Furthermore, the static selectivity allows for most of the same performance benefits.

However, the use of the sum type, though an economical means to support multiple target types, is immodular because it requires that the set of traversable types be closed, unlike in SYB and Uniplate (§7.5.1). Additionally, the sum type prevents the type system from being able to enforce two intended properties of *allR* (though we believe that KURE users are unlikely to inadvertently violate these properties):

- The first property is that each target location in a traversal is visited exactly once. Using the *Applicative* interface for traversals allows parametricity to enforce this intended constraint on *allR*. This has recently been used to impressive effect within the `lens` library (Kmett, 2012). However, in KURE, the explicit partiality of projections requires that the type of *allR* involves something more expressive than *Applicative*. While the full power of a monad is not strictly required, we chose *Monad* (including *fail*) in order to keep the overall API simple and coherent.
- The second unenforced property is that the argument to *allR* must be a type-preserving rewrite. While the rewrite is superficially an endofunction, the domain and range are the sum type, which means the function could switch between summands. The intended semantics preclude such switching, since it will always cause the traversal to fail. In contrast, the rewrite parameters in SYB and Uniplate are type-preserving in a meaningful way. However, while undesirable, these ill-behaved summand-switching rewrites are merely one source of failure among many in strategic programming; KURE has the infrastructure to handle them.

Even though they cause little harm in a strategic program, this unintended artifact of deriving KURE from Uniplate suggests an alternative implementation of KURE as an enrichment of SYB. Instead of adding support for multiple targets to Uniplate, we could add support for static selectivity to SYB that does not require the use of an abstract data type. This remains as future work, but our initial design seems promising: it has good modularity (*i.e.* closed set of target types, open set of traversable types), similar support for data abstraction, and enforces the two intended properties. However, it seems to require some optimizations involving both generalized algebraic data types (GADTs) (Peyton Jones *et al.*, 2006) and constructor specialization (Peyton Jones, 2007) that GHC does not currently perform. The design is based on the `MultiRec` implementation (Yakushev *et al.*, 2009) of the Compos technique (Bringert & Ranta, 2008), and the essential idea is to replace the sum type with a GADT such that the possible indices serve as the summands. In the degenerate case of a single index, the library degrades to Uniplate. If the GADT instead just captures a *Data* dictionary, the library degrades to SYB.

7.8.2 The KURE Context

In the actual KURE library (as opposed to the kernel discussed in this section), traversals maintain a context. As the context is unique to KURE we omitted it from the comparison with SYB and Uniplate. Neither library is inherently incompatible with it; they merely do not currently support it.

The advantage of the context is that generic traversal strategies can automatically maintain contextual information as the tree is traversed. Whereas the monad is polymorphic in the *Walker* class methods, the context is a class parameter so that the shallow traversal can be defined using operations specific to the context being constructed. This would not be possible if the context was also polymorphic. An alternative design would be to merge the context with the monad (using, say, a reader monad transformer) and have the monad be a parameter to the *Walker* class (this approach was taken by the first standalone version of KURE (Gill, 2009)). However, this fixes the monad, and thus limits the usage of the shallow traversal strategies. For example, we would be unable to use the *AnyR* transformer (§4.6), as that changes the monad. While not a fatal restriction, the workarounds needed overcome this limitation are annoying, and place an extra burden on the KURE user.

The SYB and Uniplate traversal semantics are specific enough that the traversals can be mechanically derived for data types not involving data abstraction. This is not possible in KURE, because the targets are user-defined locations, and because of the need to update the context. However, in the case that the targets are exactly the locations inhabiting summand types, and where no context updates are required, *allR* can be mechanically derived (Gill, 2009). Ideally, we would like to allow the user to appeal to a generic definition of *allR* in the cases of constructors that do not modify the context, while directly specifying the cases that do. Support for this sort of partial derivation is being explored by the recent `yoko` library (Frisby *et al.*, 2012).

7.9 Conclusion

If used as intended, SYB, Uniplate and KURE appear to provide fairly distinct approaches to traversals. SYB provides a single rewrite system containing an open set of target and traversable types; KURE allows the user to specify multiple rewrite systems, each with a closed set of target and traversable types; and Uniplate allows multiple rewrite systems, each with an open set of traversable types but only a single target type. In SYB everything is a target; in Uniplate all values 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 sum 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 pain-free though. 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 targets, whereas the targets are fixed by the semantics of SYB and Uniplate. However, this expressiveness comes at the cost of modularity: the sum type is closed and thus existing code has to be recompiled whenever it is extended.

KURE is distinguished from the other libraries in that it was designed with strategic programming in mind. Thus it provides an extensive suite of traversals, including refined error messages, as well as support for maintaining a context. These are pragmatic benefits: there is nothing preventing the other libraries being augmented with similar infrastructure. However, the presence of this infrastructure certainly makes it easier to start using KURE for strategic programming than the other two libraries. Furthermore, the successful application of KURE in the HERMIT project (Farmer *et al.*, 2012) has shown that KURE provides an expressive and useful interface in practice.

8 Performance

We now compare the performance of KURE, SYB and Uniplate using two strategic programs. Both benchmark programs are relatively simple, but exercise the shallow traversal in distinct ways. We also include the Template Your Boilerplate (TYB) (Adams & DuBuisson, 2012) library in our performance measurements. TYB is distinct from the other libraries as it is based entirely on metaprogramming with Template Haskell (Sheard & Peyton Jones, 2002), though its interface is modeled after SYB.

We first discuss the impact of two optimization techniques: *statically selective traversal* (§8.1); and *inlining* in GHC, in particular the *static argument transformation* (Santos, 1995) (§8.2). We then discuss each benchmark in turn (§8.3 and §8.4), describing the problem, its solution in each library, the performance results, and the reasons for those results.

8.1 Selective Traversal

As discussed in §2.2 there are two forms of selective traversal: static and dynamic. Static selectivity is more efficient. The libraries' support for static selectivity follows from the multiplicity of their rewrite systems' target types. SYB tends to have no support for static selectivity, because its set of target types is open. Uniplate has the best support for static selectivity, since there is exactly one target type. KURE can lie anywhere in between, since it has a closed set of target types.

Selectivity is virtually required for efficient use of SYB; without it, for example, an SYB traversal will visit every cons and character inside a *String* even if the rewrite can only succeed at the type *Int*. This is a consequence of using a run-time typecast: SYB only supports dynamic selectivity. This restriction follows from the semantics that the *gmapM* traversal will visit every proper component.

The Uniplate library can also accommodate dynamic selectivity, but does not rely on it for efficiency to the degree that SYB does. In contrast to SYB, Uniplate is fundamentally characterized by static selectivity: the shallow traversals only descend into a substructure if its value could contain the target type.

The SYB and Uniplate libraries are the two endpoints of a spectrum regarding the accommodation of static selectivity, and a KURE rewrite system can lie anywhere in between. The sum type determines which locations are to be visited by a traversal, thereby providing static selectivity. A large sum type 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.

Static selectivity is more efficient than dynamic. However, too much static selectivity makes the rewrite system less expressive. For example, as Mitchell & Runciman (2007) say, "*If you wish to visit values of different type in a single traversal, Uniplate is unsuitable*". Instead, one has to apply multiple Uniplate traversals in sequence, each with a different target type. The repeated traversals cannot be fused and are thus likely significantly inefficient. In this sort of situation the more flexible KURE is an attractive alternative.

8.2 GHC Optimization Overview

The key to performance in our two benchmarks is to eliminate higher-order functions, of which a major source is the passing of class dictionaries. We explain how various aspects of each library make this objective easier or harder to attain.

Inlining directly eliminates higher-order functions. In GHC however, inlining is doubly paramount because it 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 each mutually recursive set of definitions, GHC heuristically chooses one value 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. Note that 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 *Data* methods 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. Thus, visiting locations whose value inhabits a mutually recursive type creates mutual recursion between the *Data* dictionary and the traversal. Furthermore, the static argument cannot be extricated from the recursion. Since the recursion includes the dictionary, there is no possible "outer" definition which could bind the rewrite as an explicitly static argument. 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. The *Biplate* dictionaries for mutually recursive data types are mutually recursive when defined conventionally with the "direct" combinators. However, the consequence is less severe: as the *biplate* method will be the loop-breaker and it takes no argument, the rewrite and dictionary arguments of the *descendBiM* method remain explicitly static. The KURE sum type is similarly conducive to 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 traversable types. The *Injection* instances do not depend on the *Walker* instance, so there are no recursive dictionaries.

Although GHC offers many optimization parameters, we only use level-`O2` optimization for our benchmarks as we are interested in the performance of common usage. We measure using the `criterion` library (O'Sullivan, 2012) on an otherwise quiescent laptop; our measurements have less than 1% error with more than 95% confidence.

8.3 Benchmark: Fibonacci

The Fibonacci benchmark evaluates the Fibonacci function by rewriting the terms of a toy expression language. It exercises the bottom-up deep-traversal strategy. The data type is simple: it is singly recursive and most constructor arguments are recursive occurrences. The rewrite rules are a pedantic transcription of the Fibonacci recurrence relation:

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

plusRule                          :: Fib → KureM Fib
plusRule (Plus (Lit x) (Lit y)) = return (Lit (x + y))
plusRule _                        = fail "plusRule"

fibBaseRule                 :: Fib → KureM Fib
fibBaseRule (Fib (Lit 0)) = return (Lit 0)
fibBaseRule (Fib (Lit 1)) = return (Lit 1)
fibBaseRule _               = fail "fibBaseRule"

fibStepRule          :: Fib → KureM Fib
fibStepRule (Fib n) = return (Plus (Fib (Plus n (Lit (−2)))) (Fib (Plus n (Lit (−1)))))
fibStepRule _        = fail "fibStepRule"

unLitRule          :: Fib → KureM Int
unLitRule (Lit n) = return n
unLitRule _        = fail "unLitRule"
```

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

```
reduce   :: Fib → KureM Int
reduce e = eval e ≫= unLitRule
         where eval     = allbu ((plusRule <+ evalFib) <+ return)
               evalFib = fibBaseRule <+ (λx → fibStepRule x ≫= eval)
               f <+ g  = λx → catchM (f x) (λ_ → g x)

allbu    :: (Fib → KureM Fib) → Fib → KureM Fib
allbu f  = go
         where go = λe → all go e ≫= f

all                :: (Fib → KureM Fib) → Fib → KureM Fib
all f (Plus a b) = Plus <$> f a <*> f b
all f (Fib a)     = Fib  <$> f a
all _ x            = return x
```

The definition of *all* changes between variants, whereas *reduce* is mostly preserved.

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`*.*

8.3.1 Results

The performance results for the Fibonacci benchmark 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. The expected complexity of the Fibonacci computation is exponential. We list the goodness of fit of each variant's measurements' logarithmic regression and the geometric mean and geometric standard deviation of its slowdown with respect to the baseline.

We used the following variants in our experiment:

- `SYB-gmapM` uses the SYB deep traversal *everywhereM* along with a hand-written *gmapM* definition. It outperforms the (omitted) GHC-derived *Data* instance by approximately 15%. The derived instance defines *gmapM* as an instantiation of *gfoldl*, but mutual recursion amongst the methods and dictionaries and a lack of *INLINE* pragmas prevents specialization from removing the overhead of the instantiation.
- `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 to allow static selectivity by appeal to data abstraction. The traversal only visits the similarly typed proper components (i.e. those of type *Fib*), which merely avoids visiting the argument to Lit.
  In this case, the speedup of static selectivity is outweighed by an unfortunately related overhead. The newtype wrapper necessary for semantically sound static selectivity in SYB requires some uses of monadic operations in the *gmapM* definition merely to apply the retract and section. However, because of SYB's rank-2 polymorphism, any *gmapM* definition that visits a similarly typed component is necessarily mutually recursive with the *Data* dictionary. Consequently, the *gmapM* becomes a loop-breaker, which prevents inlining and hence specialization. The *Monad* dictionary is not specialized, so even though the newtype wrapping and unwrapping is itself erased, the monadic operations allowing it will not be. These monadic remnants of the newtype then interfere with further optimizations.
- `KURE` simply uses *Fib* as the sum type.
- `Uni` uses the "direct" Uniplate interface to define the conventional *Uniplate* instance for the *Fib* data type.
- `TYB` generates a traversal semantically equivalent to `Hand`.

ZU064-05-FPR kure 29 April 2013 17:1

For dense bottom-up traversals 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 variant is not so helpful and instead introduces some overhead. Applying the static argument transform to *everywhereM* gives the biggest speedup for SYB, about 33%.

8.3.2 Summary

In summary, the Fibonacci benchmark behaves as expected. For such a simple traversal over a singly recursive type, Uniplate does better than the hand-written function, SYB does poorly, while KURE and TYB introduce no overhead. Cursory inspection of the generated GHC Core for the SYB variants reveals that *Data* and *Monad* dictionaries are still being explicitly passed, which is likely the principal source of inefficiency and also likely partially mitigated by the SAT of *everywhereM*. To achieve these performance results, the user need only use the level-`O2` optimization and ensure that enough inlining and specialization are possible, via techniques such as explicitly static arguments.

8.4 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 and primitive rewrite are as follows:

$$\begin{array}{ll}
\textbf{newtype}\ \mathit{Company} & = \mathsf{C}\ [\mathit{Dept}] \\
\textbf{data}\ \mathit{Dept} & = \mathsf{D}\ \mathit{Name}\ \mathit{Manager}\ [\mathit{Unit}] \\
\textbf{data}\ \mathit{Unit} & = \mathsf{PU}\ \mathit{Employee} \mid \mathsf{DU}\ \mathit{Dept} \\
\textbf{data}\ \mathit{Employee} & = \mathsf{E}\ \mathit{Person}\ \mathit{Salary} \\
\textbf{data}\ \mathit{Person} & = \mathsf{P}\ \mathit{Name}\ \mathit{Address} \\
\textbf{newtype}\ \mathit{Salary} & = \mathsf{S}\ \mathit{Integer} \\
\textbf{type}\ \mathit{Manager} & = \mathit{Employee}
\end{array}$$

$$\begin{array}{l}
\mathit{inc} \quad :: \mathit{Monad}\ m \Rightarrow \mathit{Integer} \to \mathit{Salary} \to m\ \mathit{Salary} \\
\mathit{inc}\ k = \lambda(\mathsf{S}\ x) \to \textbf{if}\ x < 0\ \textbf{then}\ \mathit{fail}\ \texttt{"..."}\ \textbf{else}\ \mathit{return}\ (\mathsf{S}\ (x+k))
\end{array}$$

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:

$$\begin{array}{lll}
\mathit{allbuC}\ f\ (\mathsf{C}\ \mathit{ds}) & = \mathsf{C} & <\$>\ \mathit{mapM}\ (\mathit{allbuD}\ f)\ \mathit{ds} \\
\mathit{allbuD}\ f\ (\mathsf{D}\ n\ m\ \mathit{us}) & = \mathsf{D}\ n & <\$>\ \mathit{allbuE}\ f\ m <\!*\!>\ \mathit{mapM}\ (\mathit{allbuU}\ f)\ \mathit{us} \\
\mathit{allbuE}\ f\ (\mathsf{E}\ p\ s) & = \mathsf{E}\ p & <\$>\ f\ s \\
\mathit{allbuU}\ f\ (\mathsf{PU}\ e) & = \mathsf{PU} & <\$>\ \mathit{allbuE}\ f\ e \\
\mathit{allbuU}\ f\ (\mathsf{DU}\ d) & = \mathsf{DU} & <\$>\ \mathit{allbuD}\ f\ d
\end{array}$$

$$\begin{array}{l}
\mathit{increase} \quad :: \mathit{Integer} \to \mathit{Company} \to \mathit{KureM}\ \mathit{Company} \\
\mathit{increase}\ k = \mathit{allbuC}\ (\mathit{inc}\ k)
\end{array}$$

It is primarily the traversal *allbuC* that differs between the variants. Furthermore, for those variants that visit only *Salary* substructures, the shallow traversal suffices because *Salary* values do not have *Salary* substructures. Note that as the functions for *Dept* and *Unit* are mutually recursive, one of them will be a loop-breaker.

8.4.1 Results

The performance results for the Paradise benchmark are shown in Table 2. We include the following variants in our measurements:

- `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 subcomponents that could contain a *Salary*. The entire *Data* instance is derived by GHC, so it includes the overhead of defining *gmapM* in terms of *gfoldl*.
- `SYB-sel` uses a newtype wrapper to allow static selectivity by appealing to data abstraction. The traversal only visits *Salary* substructures, and *gmapM* is manually defined. As the *Data* dictionaries for *Dept* and *Unit* are mutually recursive via the *gmapM* definitions, one of the *gmapM* 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 sum type for the family of data types, and uses the *allbuR* strategy to reach all of the *Salary* substructures.
- `KURE-sel` uses a newtype wrapper around the sum type of `KURE` to visit only the *Salary* substructures.
- `KURE-dyn` uses an optimized variation of the *Dynamic* type as the sum, but is otherwise the same as `KURE`.
- `Uni` uses the “direct” Uniplate interface to define the conventional *Uniplate* and *Biplate* instances for the *Salary* target type. As the *Data* dictionaries for *Dept* and *Unit* are mutually recursive via the corresponding *descendBiM* definitions, one of the *descendBiM* definitions will be a loop-breaker.
- `TYB` directly generates the statically selective traversal of *Company* values that visits all *Salary* substructures.

We use `Hand` as the baseline, because it is the conventional definition of the traversal. However, it does not take the rewrite as an explicitly static argument. The `Hand-sat` instead defines a record of mutually recursive rewrites where the entire record takes the rewrite parameter. Selecting the necessary traversal from the resulting records gives a two factor speedup; the SAT is effective in GHC.

`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 static argument transform by breaking a mutual recursion amongst dictionaries at the cost of some minor code duplication. The result performs as well as the `Hand-sat` variant and just slightly

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
`KURE-dyn`	0.77	1.101	0.98825
`Uni`	0.90	1.091	0.96421
`TYB`	0.46	1.173	0.98576

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

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. In the `SYB-sel-sat` variant there is no recursion among dictionaries, so inlining can specialize the traversals as much as possible.

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. We included the `KURE-dyn` variant to emphasize that run-time typing is not the primary cost of the SYB approach. Using a dynamic sum type is only 30% slower than the conventional sum. It is a viable trade-off in the exploratory development phases, when the summands may be in flux; the library user might appreciate the modularity of the dynamic sum's single *Injection* instance based on *Typeable*.

We did not benchmark a function that requires multiple target types of a single traversal, because Uniplate would have no viable alternative. If, however, we alter the KURE variants of *increase* to also, say, nullify all department names in the same traversal, we observe no significant difference in the execution time. If the primitive rewrite does simple things at multiple types, we expect a KURE traversal with multiple target types will have similar efficiency to one with a single target type.

8.4.2 Summary

In summary, the Uniplate variant is about twice as slow as the best KURE and SYB variants. Cursory inspection of the GHC Core reveals some dictionary passing, due to some of the "direct" combinators for defining *biplate* not being inlined. 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 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.

8.5 Conclusion

A key optimization for generic traversals is selectivity: only descending into the substructures of the data type that need to be modified can be a significant gain. Static selectivity, which is known at compile time, is much more efficient than dynamic selectivity, which requires run-time checks. 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, 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 our code 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. In particular, traversals that modify values of different types deserve some attention, as we expect Uniplate to struggle with such tasks.

9 Closing

In this article we have described KURE, a Haskell-embedded DSL for strategic programming. We described the KURE implementation, and demonstrated its usage by defining a variety of general-purpose traversal strategies, as well as some specialized rewrites on the GHC Core syntax tree taken from the HERMIT package.

The creation of KURE was motivated by the needs of HERMIT, and it is therefore unsurprising that it has successfully served the needs of the HERMIT project. Nevertheless, we note that the feedback from KURE users has been positive, with the main criticisms being lack of support for traversing two trees in lock-step, and the lack of documentation demonstrating "typical" KURE usage. The former remains a subject for future investigation, and the latter was a partial motivation for this article. Our other main use of KURE has been for rewriting HTML, a task for which KURE was not specifically designed. That this proved straightforward provides evidence that KURE is more generally useful (indeed, we currently use the `html-kure` package (Gill, 2013) to pre-process 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 rewriting systems is the particular way the type system is used to support generic traversals over typed syntax trees.

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 types in the tree: instead the KURE user can specify 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. The cost of this flexibility is paid in modularity, with changes to the set of traversable or target types requiring more recompilation of existing code than is the case in either of the other two libraries.

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 *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 term-rewriting support allows KURE to be the "system language" of HERMIT, and we expect to use KURE as a foundation for building higher-level term-rewriting DSLs in the future.

Acknowledgements

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, and José Pedro Magalhães 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.

Augustsson, Lennart. (2012). `http://hackage.haskell.org/package/geniplate`.

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.

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.

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.

Foster, Nathan J., Greenwald, Michael B., Moore, Jonathan T., Pierce, Benjamin C., & Schmitt, Alan. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *Transactions on programming languages and systems*, **29**(3).

Frisby, Nicolas, Gill, Andy, & Alexander, Perry. (2012). A pattern for almost homomorphic functions. *Pages 1–12 of: Workshop on generic programming*. ACM.

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`.

Hinze, Ralf. (2000). A new approach to generic functional programming. *Pages 119–132 of: Principles of programming languages*. ACM.

Hinze, Ralf. (2002). Polytypic values possess polykinded types. *Science of computer programming*, **43**(2–3), 129–159.

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.

Hughes, R. John Muir. (1986). A novel representation of lists and its application to the function “reverse”. *Information processing letters*, **22**(3), 141–144.

Kmett, Edward A. (2012). `http://hackage.haskell.org/package/lens`.

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, & Peyton Jones, Simon. (2004). Scrap more boilerplate: reflection, zips, and generalised casts. *Pages 244–255 of: International conference on functional programming*. ACM.

Lämmel, Ralf, & Peyton Jones, Simon. (2005). Scrap your boilerplate with class: extensible generic functions. *Pages 204–215 of: International conference on functional programming*. ACM.

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

Leijen, Daan, & Meijer, Erik. (2001). *Parsec: Direct style monadic parser combinators for the real world*. Tech. rept. Utrecht University.

Löh, Andres. (2004). *Exploring Generic Haskell*. Ph.D. thesis, Utrecht University.

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

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *Journal of functional programming*, **18**(1), 1–13.

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

Nolan, Christopher. (2010). *Inception*. Warner Bros. Pictures.

Norell, Ulf, & Jansson, Patrik. (2005). Polytypic programming in Haskell. *Pages 168–184 of: Implementation of functional languages*. Springer.

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

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, & Washburn, Geoffrey. (2006). Simple unification-based type inference for GADTs. *Pages 50–61 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.

Rodriguez, Alexey, Jeuring, Johan, Jansson, Patrik, Gerdes, Alex, Kiselyov, Oleg, & d. S. Oliveira, Bruno C. (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, & Gill, Andy. (2013). `http://hackage.haskell.org/package/kure`.

Sculthorpe, Neil, Farmer, Andrew, & Gill, Andy. (2013). The HERMIT in the tree: Mechanizing program transformations in the GHC core language. *Implementation and application of functional languages 2012*.

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

Stewart, Don. (2009). `http://hackage.haskell.org/package/dlist`.

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.

Weirich, Stephanie. (2006). RepLib: A library for derivable type classes. *Pages 1–12 of: Haskell workshop*. 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*.

Yakushev, Alexey Rodriguez, Holdermans, Stefan, Löh, Andres, & Jeuring, Johan. (2009). Generic programming with fixed points for mutually recursive datatypes. *Pages 233–244 of: International conference on functional programming*. ACM.

Yorgey, Brent A, Weirich, Stephanie, Cretin, Julien, Peyton Jones, Simon, Vytiniotis, Dimitrios, & Magalhães, José Pedro. (2012). Giving Haskell a promotion. *Pages 53–66 of: Types in language design and implementation*. ACM.