A Haskell Hosted DSL for Writing Transformation Systems

Andy Gill

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045
andygill@ku.edu

Abstract. KURE is a Haskell hosted Domain Specific Language (DSL) for writing transformation systems based on rewrite strategies. When writing transformation systems, a significant amount of engineering effort goes into setting up plumbing to make sure that specific rewrite rules can fire. Systems like Stratego and Strafunski provide most of this plumbing as infrastructure, allowing the DSL user to focus on the rewrite rules. KURE is a strongly typed strategy control language in the tradition of Stratego and Strafunski. It is intended for writing reasonably efficient rewrite systems, makes use of type families to provide a delimited generic mechanism for tree rewriting, and provides support for efficient identity rewrite detection.

1 Introduction

Sometimes its easy to say what you want to do, but tedious to get the scaffolding in place. Such a situation is an ideal candidate for a Domain Specific Language (DSL) where the language provides the scaffolding as a pre-packaged service. For example, scaffolding is needed to promoting local, syntax directed rewrites into a global context. Take the first case rewriting rule from the Haskell 98 Report [1] which says:

(a) `case` e `of { alts }` = `(\v -> case v of { alts })` e
where v is a new variable

This rule is used to simplify a case over an arbitrary expression, and convert it to a case over a *variable*. We can express this rule directly in Haskell, using a rewrite in the Haskell Syntax provided by Template Haskell [2], using the function `rule_a`.

```
rule_a :: ExpE -> Q ExpE
rule_a (CaseE e matches) = do
         v <- newName "v"
         return $ AppE (mkLamE [VarP v] $ CaseE (VarE v) matches) e
rule_a _ = fail "rule_a not applicable"
```

This rule reflects our syntax rewrite rule almost directly, and cleanly utilizing a monadic name supply for a fresh variable. The rule also acts locally, and needs additional machinery to perform this rewrite on whole programs. Strategic programming is a paradigm which builds on rewrite primitives like `rule_a` by using combinators to construct complex and powerful rewrites and transformation systems.

This paper introduces KURE, a strategic programming DSL hosted in Haskell being developed at the University of Kansas. KURE provides a small set of combinators that can be used to build parameterized term rewriting and general user-defined rule application. In this paper, we discuss in detail the design of KURE, and how we used the tradition of type-centric DSL design to drive our implementation. This paper makes the following specific contributions.

- We move both rewrites and rewrite strategies into a strongly typed world. Throughout the paper we make detailed comparisons with previous attempts to providing typing to rewrite strategy systems.
- More specifically, we provide typing for term traversing strategies using a new capability, a lightweight and customized generic term traversal mechanism implemented using associated type synonyms [3].
- We introduce the ability to record equality over terms, which is traditionally a weakness of purely functional rewrite engines. Equality is important because the use-case for many transformations in practice is to iterate until nothing else can be achieved.
- We abstract our combinators explicitly over a user-defined context, allowing concerns like environment or location to be represented. We show the generality of this contribution by implementing path selection. This design choice allows a particularly clean relationship between the semantics being hosted by the DSL, and the implementation using our DSL.
- We lay out the methodology used in the design of our hosted DSL, with the intent that others can use our design template.

KURE builds on many years of research and development into rewrite strategies, especially the work of Stratego [4] and Strafunski [5], and our own experiences with HERA [6]. We compare KURE to these systems throughout this paper, as well as make different design decisions explicit, and summarize the differences in section 11. In recognition of the contributions made by these systems, we reuse where possible the naming conventions of Stratego and Strafunski.

2 Introducing Strategic Programming

Stratego and other strategic programming languages provide a mechanism to express rewrites as primitive strategies, and a small set of combinators that act on strategies, giving the ability to build complex custom strategies. In this section, we introduce the basic combinators of strategic programing used in Stratego. This will guide our design of KURE.

A basic strategy is a rewrite from one abstract syntax term to another abstract syntax term, for example:

```
NOT1 : Not(Not(e)) -> e
```

This strategy, called `NOT1`, attempts to match a term, and if the top node is `Not`, and the immediate child is also `Not`, then Stratego replaces this whole term with the immediate child of the second `Not` and terminates successfully. If the term is not matched, the strategy fails. In functional parlance, strategies take a term, and return a new term, or fail.

Table 1. Combinators in Stratego

Combinator	Purpose
`id`	identity strategy
`fail`	always failing strategy
$\mathcal{S}$ `<+` $\mathcal{S}$	local backtracking
$\mathcal{S}$ `;` $\mathcal{S}$	sequencing
`all(`$\mathcal{S}$`)`	apply $\mathcal{S}$ to each immediate child

As for parsing combinators [7], a rich algebra of combinators can be built on top of the concept of strategies. If $\mathcal{S}$ is a strategy, the key combinators in Stratego are listed in Table 1. `id` is the identity transformation, `fail` is a transformation that always fails, `<+` is a choice operator with local backtracking, '`;`' sequences two transformations, and `all` provides a shallow traversal, a traversal of only the immediate children of a node. There are others, but these capture the spirit of the programming paradigm.

We can use these primitive combinators to write other combinators, like `try`

$$\texttt{try}(s) = s \texttt{ <+ id}$$

which attempts a rewrite, and if it fails, performs the identity rewrite instead. `try` has the nice property, therefore, that it never fails.

All of the combinators introduced so far are shallow, and only act on at most a single level of a term. We can use these to implement deeper rewrites, which act over *every* sub-node in a tree.

$$\texttt{topdown}(s) = s \texttt{ ; all(topdown(}s\texttt{))}$$

Stratego also provides the ability to invoke a strategy from within a rule, by enclosing the strategy in angle brackets.

```
EvalAdd : Add(Int(i), Int(j)) -> Int(<addS>(i,j))
```

Here, `addS` is itself a rewrite strategy which takes a 2-tuple of integers, and generates the result of the addition.

By using application inside rules strategy-based programming jumps between rules and strategies, and back again. Basic user strategies are named rules, and rules can use strategies to express rewrites over terms, collectively forming a productive environment to implement complex rewrites. 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. The idiom has demonstrated the ability to have all the qualities of a great DSL, giving leverage to the rewrite author. We have only given a cursory overview of the essence of Stratego and strategic programming, and the interested reader is referred to the Stratego website, http://strategoxt.org/.

When considering a rewriting system using strategies hosted in Haskell, Strafunski is currently the most mature library. Strafunski is a strategic programming implementation which follows in the tradition of Stratego, adds typing to primitive transformations, and uses "scrap your boilerplate" (SYB) generics [8] to implement shallow (single-level) and deep (multi-level) traversals over arbitrary term types. Strafunski provides both type-preserving rewrites and unifying rewrites (rewrites that map to a single common type). However, there are shortcomings that merit revisiting the design decisions of Strafunski. KURE is an attempt to revisit these design decisions. Specifically, KURE replaces the powerful hammer of SYB generics provided in Strafunski with a more precise, user configurable and lightweight generics mechanism. KURE also provides a number of parameterization opportunities over Strafunski, as well as other differences, as discussed in section 9.

3 Design of the KURE Kernel

Our design and implementation of KURE follows the classical DSL hosted in Haskell approach, namely

- we propose specific functionality for the primitive combinators of our DSL,
- we unify the combinators around a small number of (typically abstract) types,
- we postulate the monad that is contained inside the computation of these primitives,
- we invent some structure around this monad, to provide the significant user-level types in our DSL,
- and at this point, our combinators are largely implemented using routine plumbing between monadic computations.

There are structures other than monads that can be used to model computations, but monads are certainly the tool of choice in the Haskell community for modeling such things.

The major primitive combinators in a DSL for strategic programming are well understood, and have been discussed in section 2. We want to add two

new facilities, both of which have analogues in Stratego, but are implemented in a functional and strongly typed way in KURE. First, we add the ability to understand the context of a rule. Rather than introduce dynamically scoped rules [9] we want our rules to execute in a readable environment that reflects the context within which the rule is being executed. We also add the ability to create new global bindings from within local rules.

Most importantly, we want our DSL to use types to reflect the transformations taking place. The next section reviews how other systems have added types to strategic programming.

4 Previous Uses of Types in Strategic Programming

There have been a number of attempts to add typing to strategy based programming. There are two independent issues to resolve. Firstly, giving a type to a strategy $\mathcal{S}$. Secondly, giving general types to functions that do both shallow and deep rewrites, like `all` and `topdown`. We address each of these in turn.

4.1 Giving a Type to $\mathcal{S}$

Giving a type to $\mathcal{S}$ is straightforward. Consider a typed strategy or transformer, called $\mathcal{T}$. We can give $\mathcal{T}$ two type parameters, the initial term's type, and the type of the term after rewriting. Thus, our strategies have the type:

$$\mathcal{T}\ t_1\ t_2$$

We can now give types to the depthless combinators in Stratego, which we do in Table 2.

Table 2. Types for Depthless Combinators

Combinator	**Type**
`id`	$\forall t_1.\quad \mathcal{T}\ t_1\ t_1$
`fail`	$\forall t_1, t_2.\quad \mathcal{T}\ t_1\ t_2$
$\mathcal{S}$ `<+` $\mathcal{S}$	$\forall t_1, t_2.\quad \mathcal{T}\ t_1\ t_2 \to \mathcal{T}\ t_1\ t_2 \to \mathcal{T}\ t_1\ t_2$
$\mathcal{S}$ `;` $\mathcal{S}$	$\forall t_1, t_2, t_3.\ \mathcal{T}\ t_1\ t_2 \to \mathcal{T}\ t_2\ t_3 \to \mathcal{T}\ t_1\ t_3$

For combinations of these combinators, the type system works unremarkably. For example, the function `try` can be given a straightforward type.

$$\begin{aligned} &\texttt{try} &&:: \forall t_1.\ \mathcal{T}\ t_1\ t_1 \to \mathcal{T}\ t_1\ t_1 \\ &\texttt{try}(s) &&= s\ \texttt{<+}\ \texttt{id} \end{aligned}$$

Transforms that are type-preserving are common in strategy based programming, so we give such transforms there own type name, $\mathcal{R}$.

$$\mathcal{R}\ t_1 = \mathcal{T}\ t_1\ t_1$$

$\mathcal{R}$ is simply an abbreviation for $\mathcal{T}$ for notational convenience. Using $\mathcal{R}$, `try` can be giving the equivalent type

$$\texttt{try} :: \forall t_1.\ \mathcal{R}\ t_1 \rightarrow \mathcal{R}\ t_1$$

Giving types to application and abstraction using $\mathcal{T}$ is straightforward once a monad for rules has been selected, so we defer giving the actual types until section 6 in the context of our KURE monad, and observe that there are no technical issues with their typing.

4.2 Types for Shallow and Deep Combinators

Adding types is more problematic for shallow and deep combinators. Consider the combinator `all`, which acts on immediate children, but leave the root node unmodified. This implies a type-preserving rewrite, and we would expect something of the form

$$\texttt{all} :: \forall t_1.\ \mathcal{R}\ t_1 \rightarrow \mathcal{R}\ t_1$$

This combinator can only apply its argument to immediate children *of the same type*, a significant shortcoming, but is the approach taken by Dolstra [10], when he attempts to host strategies directly in Haskell.

If we give `all` a more general type, other issues surface

$$\texttt{all} :: \forall t_1, t_2.\ \mathcal{R}\ t_1 \rightarrow \mathcal{R}\ t_2$$

Unfortunately, the type t_1 is completely unrelated to t_2, so it is well understood that `all` can never actually use its argument in any interesting or non-trivial way, *without runtime type comparisons*. This is the the approach taken by Dolstra [11]; Dolstra and Visser [11, 12], who invent a new language with its own type system which includes runtime type-case internally inside `all`.

This is also essentially the approach taken in Strafunski [5]. Rather than work on polymorphic rewrites directly, Strafunski implements `all` by wrapping $\mathcal{R}$ in an abstraction `TP`. Here, we use the notation from [13] for expressing type contexts.

$$\texttt{TP} = \forall t_1 \langle Term\ t_1 \rangle \Rightarrow\ \mathcal{R}\ t_1$$

This means that `TP` can be used to express any rewrite $\mathcal{R}$, as long as the type of the candidate syntax admits $Term$, which is the ability to decompose a data-structure into a universally typed constructor and arguments, and recompose the data-structure back again. Such an approach is not unique to Strafunski. Other typed rewrite systems have also taken the same approach using a universal representation, for example [14]. Using `TP`, we can give `all` the type

$$\texttt{all} :: \texttt{TP} \rightarrow \texttt{TP}$$

Now it is possible to pass rewrites to `all` with an arbitrary type, using SYB style generics [8]. Strafunski provides functions for getting into and out of the `TP` abstraction. Two of the functions for building `TP` are `adhocTP` and `failTP`.

$$\texttt{adhocTP} :: \forall t_1 \langle \mathit{Term}\ t_1 \rangle\ \texttt{TP} \rightarrow \mathcal{R}\ t_1 \rightarrow \texttt{TP}$$
$$\texttt{failTP} \ :: \texttt{TP}$$

Using these combinators, it is possible to use `all`. We can chain different types of rewrites through our `TP` abstraction,

```
all ((failTP `adhocTP` rr1) `adhocTP` rr2)
```

Here, we have built a `TP` that can perform two possible rewrites to the immediate children. These two rewrites can be at completely different types. For terms that are not of a matching type, the `failTP` combinator is applied, causing the rewrite to abort with failure.

The use of generics have significant ramifications for our mission of having a strongly typed DSL for rewrites. There are three shortcomings:

- Even though the generic mechanism itself can be implemented efficiently using a type-safe cast, deep traversals, using recursive invocations of `all`, become prohibitively expensive, because the universal-type nature of `TP` results in every single sub-node being considered as a rewrite candidate. For example, when implementing compiler passes, there is a considerable cost to examining every character of every string as a candidate for rewriting.
- The `TP` type is universal, and rewrites over two completely unrelated syntaxes use the same abstraction. We want the type of our traversal combinators to at least reflect something about the types they operate on. This issue could be at least partially addressed in Strafunski by creatively using a phantom type [15] on TP.
- `TP` is not a $\mathcal{R}$, therefore not a $\mathcal{T}$ either. We want to build a combinator library around operators on $\mathcal{T}$ (our design decision) and each new user-level type complicates the DSL and the abstractions it is attempting to capture.

5 Supporting Shallow and Deep Traversals in KURE

What we want to support in KURE is a version of `all` that accepts a set of potentially distinctly typed rewrites. Consider a rewrite function, `all`, to which every relevant correctly typed rewrite is passed explicitly as an element of a tuple.

$$\texttt{all} :: \forall t, t_1, \ldots, t_n \langle t_i \in \textbf{childrenOf}\ t \rangle \Rightarrow (\mathcal{R}\ t_1 \times \mathcal{R}\ t_2 \times \ldots \times \mathcal{R}\ t_n) \rightarrow \mathcal{R}\ t$$

where **childrenOf** is a function from a *type* to the *set of types* of possible children. Implementing `all` is now a plumbing problem, and straightforward for any specific t.

In principle, this idea for implementing `all` works; the rewriting inside `all` knows what constructors are being rewritten, so can select the rewrite of the appropriate type. But this version of `all` has a number of shortcomings. First, the argument to `all` is difficult to generalize, given that it depends on the number of distinct types contained in the constructors of type t. Second, and more importantly, the argument is no longer first class, in the sense that we are building a framework for manipulating rewrites and transforms, and this is a tuple. It would be *possible* to construct a new sequencing operation over tuples, but this would not make good use of the existing machinery in our DSL.

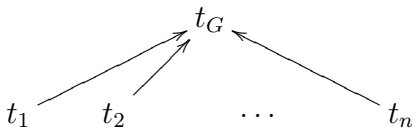
A cleaner and preferable approach is to reuse the $\mathcal{R}$ rewrite type as the single argument to `all`. As discussed in section 4.2, this argument to `all` can not be parametrically polymorphic, but we can consider how to use our (type) function **childrenOf** to give a useful and implementable type for `all`. The question to be considered is can we encode

$$(\mathcal{R}\ t_1 \times \mathcal{R}\ t_2 \times \ldots \times \mathcal{R}\ t_n)$$

inside

$$\mathcal{R}\ t_G$$

What type is t_G? If t_G was a common super-type of each of t_i, then this relationship would hold



and it would be possible to encode and decode each of the $\mathcal{R}\ t_i$ rewrites inside a single $\mathcal{R}\ t_G$, using this typing relationship. In Haskell, we can model such a typing relationship using sums, so

$$t_G = t_1 + t_2 + \ldots + t_n$$

The type of `all` would therefore be

$$\texttt{all} :: \forall t, t_1, \ldots, t_n \langle t_i \in \textbf{childrenOf}\ t \rangle \Rightarrow \mathcal{R}\ (t_1 + t_2 + \ldots + t_n) \rightarrow \mathcal{R}\ t$$

We can simplify the type of `all`, by inventing a type function, $\mathcal{G}$, which maps t_i to its super-type t_G.

$$\mathcal{G}\ t_i = t_G = t_1 + t_2 + \ldots + t_n \text{ where } 0 < i \leq n$$

That is, for each of the type t_i, we can use $\mathcal{G}$ to look up the generic type, t_G. Furthermore, if we make t one of the elements in our sum type, then

$$\mathcal{G}\ t = \mathcal{G}\ t_i = t_G = t + t_1 + t_2 + \ldots + t_n \text{ where } 0 < i \leq n$$

and we can also use the type t to look up the type of t_G.

This gives us a clean type for `all`

$$\texttt{all} :: \forall t \langle \mathcal{G}\ t \rangle \Rightarrow \mathcal{R}\ (\mathcal{G}\ t) \rightarrow \mathcal{R}\ t$$

where $\langle \mathcal{G}\ t \rangle$ means that $\mathcal{G}$ is defined for t. $\mathcal{R}\ (\mathcal{G}\ t)$ replaces `TP`, as used in Strafunski. Unlike Strafunski, the `all` used in KURE acts over a specific type, not the universal or generic type. In the same way as in Strafunski uses `TP` for deep traversals, we use $\mathcal{R}\ (\mathcal{G}\ t)$ to provide deep traversal combinators in KURE. We will examine the implementation of $\mathcal{G}$ and shallow and deep traversals in section 7.

$\mathcal{G}$ is the type function that provides our generic mapping to a local universal type, t_G. By design, we make available to the DSL user the ability to specify what types can map through $\mathcal{G}$ to t_G. In some cases, there might only be a single type that maps to t_G, and in larger syntaxes, there might be many. This design flexibility allows the KURE user to craft the type scope of the deep traversals, and for example, avoid considering rewriting strings.

The construction of the sum-type and $\mathcal{R}$ used as an argument to `all` results in a $\mathcal{T}$ that *can* represent non type-preserving translations, because

$$\mathcal{R}\ (t_1 + t_2 + \ldots + t_n) = \mathcal{T}\ (t_1 + t_2 + \ldots + t_n)\ (t_1 + t_2 + \ldots + t_n)$$

We mitigate against this by providing correct-by-construction promotion functions, and defining translations that are not type-preserving as failed transformations. So a rewrite of type $\mathcal{R}\ t$ promoted to a $\mathcal{R}\ (\mathcal{G}\ t)$ never fails because of incorrect typing, and can be safely combined with other promoted translations, at the $\mathcal{R}\ (\mathcal{G}\ t)$ type.

There are other ways of providing deeper traversals, for example, by using fold algebras [16]. Like passing in tuples of $\mathcal{R}$, we could provide combinators that take a representation of a (typed) algebra, and perform a fold-like rewrite. In fact, this could be coded up in KURE. We choose, however, to keep our design simple, where everything is a transform.

Finally, there is one `all` like combinator we adopt from Strafunski, the `crush` combinator, which crushes together the results of all the rewrites on the immediate children of a node into one common type. Reusing our type function $\mathcal{G}$, this function would have type.

$$\texttt{crush} :: \forall t. \langle \mathcal{G}\ t \text{ and } r \text{ is a Monoid} \rangle \Rightarrow \mathcal{T}\ (\mathcal{G}\ t)\ r \rightarrow \mathcal{R}\ r$$

6 Computations inside KURE

In KURE, we want to promote small, local rewrites into strategies than can be applied over large syntax terms. We provide a monad, called $\mathcal{M}$, for authoring small rewrites, which also provides some basic services, and a larger abstraction $\mathcal{T}$, which will contain the services of this monad. So, we propose that our rewrites are of the form

$$\mathcal{T}\ t_1\ t_2 = t_1 \rightarrow \mathcal{M}\ t_2$$

where $\mathcal{M}$ is our as yet undefined monad. We choose to put all our services into this monad, but there are other choices.

In order to understand the shape of $\mathcal{M}$, we enumerate the capabilities we want to provide in our DSL, towards a generic strategic programming engine in Haskell, and propose how to implement each facility using a specific monad.

- We provide the ability to represent failure, for denoting transformations that fail. This failure does not carry any environment or memory. We implement this using the failure monad.
- We provide the capability to detect an identity transformation. The Stratego family of rewrite systems offers equality for the cost of a pointer comparison, as provided by the underlying term representation. Knowing that a rewrite performed the identity transformation is critical to a number of important strategy idioms, for example finding a fix-point of a transformation. We implement this status using a count of the non-identity preserving translations performed, and we propagate this using the writer monad.
- By design, we allow the construction of rules that have the ability to generate new global bindings in our candidate syntax. We implement this using the writer monad.
- We provide the ability for a rule to understand and have visibility into its context in a parameterizable manner. We implement this using the reader monad.
- Finally, we provide the ability for KURE users to add other arbitrary capabilities, like unique name generation, and others that may be specific to the underlying grammar.

Combining these capabilities is straightforward using monad transformers [17]. After unfolding our monad transformers, we reach a flexible and parameterizable $\mathcal{M}$ monad

$$\mathcal{M}\ \alpha = env_{read} \rightarrow m((\alpha \times env_{write} \times count_{write}) + Fail)$$

where m is another monad. In KURE, we implement the $\mathcal{M}$ monad directly using this equation. We will return to the shape of $count_{write}$ shortly, but we require both env_{write} and $count_{write}$ to be monoids. We explicitly use the reader and writer monad rather than the state monad to allow the possibility of a concurrent implementation of KURE in the future.

In addition to the monadic operators, we provide two additional operations on $\mathcal{T}$ and $\mathcal{M}$, specifically the coercions between the two structures.

$$\texttt{translate} :: (t_1 \rightarrow \mathcal{M}\ t_2) \rightarrow \mathcal{T}\ t_1\ t_2$$
$$\texttt{apply} :: \mathcal{T}\ t_1\ t_2 \rightarrow t_1 \rightarrow \mathcal{M}\ t_2$$

`translate` promotes a monadic translation into a $\mathcal{T}$ rewrite, allowing rewrite rules expressed monadically to be constructed. `apply` allows this monadic code to itself make applications of $\mathcal{T}$ structures; it is the direct equivalent of the '`<...>`' syntax in Stratego.

6.1 Counting Translations and the Equality Flag in $\mathcal{M}$

An important question to ask after a term has been translated is "has something changed because of this translation?". Comparisons over terms should be cheap if this question is asked often. Checking for equality in Stratego requires a straightforward pointer equality because terms are represented using maximal sharing. In KURE, we want to carefully record an equality approximation, specifically the identity rewrites, and the congruence traversals that transitively only perform identity rewrites. Of course, it is possible for the user to write by hand an identity rewrite and generate a false negative, and we intend to perform future measurements to determine how effective our system is in practice. Providing this equality approximation tracking introduces an important challenge.

- The identity rewrite should be tagged as making no change to its argument.
- Also, every `translate` that contains user-defined code should, by default, be recorded as performing a non-identity translation. That is we assume by default that a function of the form $t_1 \rightarrow \mathcal{M}\ t_2$ is not simply returning its argument, but has altered the return value in some way.
- However, some translation rules are *transparent*, in that they perform identity rewrites if the applications of `apply` inside the translation rule are all transitively the identity rewrite. The conceptual model is that the translation is transparent, and non-identities show through the rule specification.

This is the dilemma. We use `translate` to promote for both *leaf* monadic actions, which are not identity translations, and also use `translate` for building the *nodes*, which in many cases are identity preserving if the internal calls to `apply` are also identity preserving. Rather than have two types of `translate`, we provide a new combinator which can mark a specific translation as transparent.

$$\texttt{transparently} :: \forall t_1, t_2.\ \mathcal{T}\ t_1\ t_2 \rightarrow \mathcal{T}\ t_1\ t_2$$

By default, `translate` marks the resulting translation as non-identity preserving by adding one to the count of non-identity sub-translations. `transparently` is an adjective which modifies `translate` to not add any additional count value to the non-identity sub-translation count.

Obviously, only an $\mathcal{R}$ can actually be an identity translation, but many combinators are implemented with their most general type (for example `<+` and '`;`'), so we want the ability to also mark these translations as transparent. When `transparently` is used in a non-type preserving way, the leaf rewrite that actually does the non-type preserving transformation is marked as non-identity, and therefore the whole transformation is also marked as non-identity.

Our use of this checking for equality has an unfortunate consequence; a `Translate` is not an arrow [18]. $\mathcal{T}$ do not hold for the arrow laws, specifically the law

$$\texttt{pure id >>>}\ arr = arr$$

does not hold where *arr* is a transformation that has been marked as identity preserving. This means that KURE can not use the arrow interface or laws.

6.2 Implementing Rewrites and Translations

We have two structures, $\mathcal{T}$, our transformer, and $\mathcal{M}$, our monad, which jointly form the basis of our rewrite system. We used our list of requirements, and the well understood technologies around monads to informally derive a basic implementation for our structures. We now address the question of how we implement $\mathcal{T}$ and $\mathcal{M}$ in Haskell to build a pragmatic DSL for defining rewrites by introducing our primary Haskell data-types, and various functions that act as combinators on these functions.

In general, KURE provides syntax rewrites by **providing** a library of depthless functions, like '`;`' and `try`, then **requiring** the user to write shallow traversal combinators for their candidate syntax, then **providing** a library of deep traversal combinator which use the supplied shallow traversal combinators.

Table 3. Principal Types in KURE

Name	Type	Implementation	Interface
$\mathcal{R}$ `e`	`Rewrite m dec e`	`Translate m dec e e`	Synonym
$\mathcal{T}$ `e1 e2`	`Translate m dec e1 e2`	`e1` → `RewriteM m dec e2`	Abstract
$\mathcal{M}$ `e`	`RewriteM m dec e`	`dec` → `m (RewriteStatusM dec e)`	Abstract

Table 3 maps our abstract names onto their type, implementation and interface. `Rewrite` is a synonym to allow combinators like '`;`' to be shared between `Rewrite` and `Translate`. In Table 3, `m` is a monad and `dec` is the (declaration based) environment. `RewriteStatusM` can represent success or failure, directly reflecting the internals of the type definition of $\mathcal{M}$.

Table 4 gives the depthless functions provided by KURE, split into functions that perform a monadic computation, functions that generate a `Translate`, and functions that generate a `Rewrite`, with brief descriptions. Most of these combinator directly inherite the abilities of their Stratego equivalent. There is also functionality for handling identity detection, provided by `changedR`, '`.+`' and '`!->`'. `changedR` turns an identity rewrite into a failed rewrite, mirroring `tryR` which turns a failing rewrite into an identity rewrite, '`.+`' which backtracks on identity, and '`!->`' which only performs the second rewrite after the first rewrite if the first rewrite was a non-identity rewrite.

7 Constructing Shallow Traversal Combinators

In this section, we employ the KURE DSL to build a rewrite DSL over a small syntax, adapted from [19, Grammar 3.1]. We distinguish between DSL code written using KURE and the implementation of KURE itself by placing code fragments that are uses of the KURE DSL (or similar) inside boxes.

Table 4. Principal Monadic, Translate and Rewrite Functions in KURE

Name	Type	Purpose
All functions in this table have context	`(Monad m,Monoid d) => ...`	
`apply`	`:: Translate m d e1 e2 -> e1 -> RewriteM m d e2`	`Translate` application.
`fail`	`:: String -> RewriteM m d a`	Failure.
`liftQ`	`:: m a -> RewriteM m d a`	Lift a monadic operation.
`translate`	`:: (e -> RewriteM m d e2) -> Translate m d e1 e2`	Build a `Translate` from a monadic translation.
`rewrite`	`:: (e -> RewriteM m d e) -> Rewrite m d e`	Build a `Rewrite` from a type-preserving monadic translation.
`<+`	`:: Translate m d e1 e2 -> Translate m d e1 e2 -> Translate m d e1 e2`	Local backtracking.
`>->`	`:: Translate m d e1 e2 -> Translate m d e2 e3 -> Translate m d e1 e3`	Sequencing.
`failT`	`:: Translate m d e1 e2`	Always failing `Translate`.
`transparently`	`:: Translate m d e1 e2 -> Translate m d e1 e2`	Mark a `translate` as transparent.
`idR`	`:: Rewrite m d e`	The identity `Rewrite`.
`failR`	`:: Rewrite m d e`	The failing `Rewrite`.
`tryR`	`:: Rewrite m d e -> Rewrite m d e`	Attempt a `Rewrite`, otherwise perform the identity `Rewrite`.
`changedR`	`:: Rewrite m d e -> Rewrite m d e`	Fail if the argument `Rewrite` has no effect.
`.+`	`:: Rewrite m d e -> Rewrite m d e -> Rewrite m d e`	Backtracking on identity.
`!->`	`:: Rewrite m d e -> Rewrite m d e -> Rewrite m d e`	Sequencing for non-identity.
`acceptR`	`:: (e -> Bool) -> Rewrite m d e`	Identity or failure, depending on the predicate.

```
data Stmt = Seq Stmt Stmt
          | Assign Name Expr

data Expr = Var Name
          | Lit Int
          | Add Expr Expr
          | ESeq Stmt Expr

type Name = String
```

We are going to build a set of combinators that act over `Stmt` and `Expr`, using `Translate` and `Rewrite`. We can use the predefined KURE combinators to write rewrites immediately.

```
make_left_assoc = rewrite $ \ e -> case e of
     (Add e1 (Add e2 e3)) -> return $ Add (Add e1 e2) e3
     _ -> fail "make_left_assoc"
```

This rewrite attempts to change right associative additions into left associative additions at a single level.

On top of these rewrites, we provide the opportunity to write congruence and shallow traversal combinators, and then take advantage of deep combinators if the DSL user provides the shallow combinators. In KURE we provide two separate styles of structural rewrites, following the conventions of Strafunski. We have *type-preserving* rewrites, suffixed with `R`, and *unifying via a common type* rewrites, suffixed with `U`. There are other possible translation patterns possible, for examples based on fold [16]. Our objective is to build a set of translations specific to this syntax, common-up these transformation inside a shallow tree traversal `all`-like function, and then host our deep generic tree traversals on top of our `all` combinator.

7.1 Supporting Congruence and Friends

It is possible to immediately define direct support for congruence and shallow translations in KURE. As an example, we will construct a congruence combinator and the unifying translation for the `ESeq` constructor. These will have the type.

$$\texttt{eseqR} :: \mathcal{R}\ \texttt{Expr} \rightarrow \mathcal{R}\ \texttt{Expr} \rightarrow \mathcal{R}\ \texttt{Expr}$$
$$\texttt{eseqU} :: \forall u \langle \text{Monoid } u \rangle \Rightarrow \mathcal{T}\ \texttt{Expr}\ u \rightarrow \mathcal{T}\ \texttt{Expr}\ u \rightarrow \mathcal{T}\ \texttt{Expr}\ u$$

Their implementation is tedious but straightforward. The `eseqR` combinator is marked as `transparently` rewriting its target, but the `eseqU` is never an identity transformation. Both combinators use `apply` to invoke the transformation on the arguments of the specific constructor.

```
eseqR :: (Monad m, Monoid dec)
      => Rewrite m dec Stmt
      -> Rewrite m dec Expr
      -> Rewrite m dec Expr
eseqR rr1 rr2 = transparently $ rewrite $ \ e -> case e of
        (ESeq s1 s2) -> liftM2 ESeq (apply rr1 s1)
                                    (apply rr2 s2)
        _ -> fail "eseqR"
eseqU :: (Monad m, Monoid dec,Monoid r)
      => Translate m dec Stmt r
      -> Translate m dec Expr r
      -> Translate m dec Expr r
eseqU rr1 rr2 = translate $ \ e -> case e of
        (ESeq s1 s2) -> liftM2 mappend (apply rr1 s1)
                                       (apply rr2 s2)
        _ -> fail "eseqU"
```

By jumping into the monadic expression world and using `apply`, writing such congruence combinators is mechanical and prone to cut-and-paste induced errors. The type of the constructors dictates the structure of the implementation of all the functions. The KURE user has the option of being supported in this task with a Template Haskell library, called "KURE your boilerplate" (KYB) which generates these congruence functions automatically from the data structures, performing the tedious task of writing these functions. There may also be good reasons why a library author might want to write these by hand, as we will discuss in section 9, but KYB certainly makes it easier to get KURE working for any specific grammar.

7.2 Supporting Generic Shallow Term Traversals

How do we support generic traversals over a syntax tree? In the previous section we implemented congruence, which was supported using multiple `Rewrite` or `Translate` arguments, of potentially different types, one per argument of the individual constructor. We want to build generic traversals that take a single `Rewrite` or `Translate`. We discussed `all` in section 5, where we gave it the type

$$\texttt{all} :: \forall t \langle \mathcal{G}\ t = t + t_1 + t_2 + \ldots \rangle \Rightarrow \mathcal{R}\ (\mathcal{G}\ t) \rightarrow \mathcal{R}\ t$$

The traditional way of capturing the type function $\mathcal{G}$ that is a mapping from t to the sum of possible sub-children type in Haskell is multi-parameter type classes and functional dependencies [20]. However, a more natural implementation route is now possible; we can directly use a type function, as implemented in experimental extension to Haskell called associated type synonyms [3]. We now introduce the relevant part of associated type synonyms, and use it to implement the generic rewrite scheme for KURE.

Type families allow the introduction of `data` and `type` declarations inside a `class` declaration.

```
class Term exp where
  type Generic *

  -- | 'select' selects into a 'Generic' exp,
  --    to get the exp inside, or fails with Nothing.
  select :: Generic exp -> Maybe exp

  -- | 'inject' injects an exp into a 'Generic' exp.
  inject  :: exp -> Generic exp
```

This `class` declaration creates a type function, `Generic` which looks like a type synonym inside the class – it does not introduce any constructors or abstraction – but actually provides a configurable function from a type to another type. Unlike traditional type synonyms `Generic` is only defined for specific instances, specifically instances of our class `Term`. We have two utility functions. `select` takes one of our `Generic` type, and selects a specific type `exp`, or fails. `inject` creates something of the `Generic` type, and can not fail.

It is now easy to find the Generic type for a supported type, simply using `Generic`. For our example, we choose to create a new data-type `OurGeneric`, though we could have chosen to use the Haskell type `Either`.

```
data OurGeneric = GStmt Stmt
                | GExpr Expr

instance Term Stmt where
  type Generic Stmt   = OurGeneric
  inject              = GStmt
  select (GStmt stmt) = Just stmt
  select _            = Nothing

instance Term Expr where
  type Generic Expr   = OurGeneric
  inject              = GExpr
  select (GExpr expr) = Just expr
  select _            = Nothing
```

This gives the type equalities

$$\texttt{Generic Stmt} = \texttt{Stmt} + \texttt{Expr} = \texttt{Generic Expr}$$

Following Strafunski, as well as providing an `all` combinator in KURE, we also provide a unify function, which requires all (interesting) children to be translatable to a single, unified, monoidal type. Following our naming conventions, we call these two variants `allR` and `crushU`. We can now give their interface using class overloading.

```
class (Monoid dec,Monad m,Term exp) => Walker m dec exp where
  allR :: Rewrite m dec (Generic exp)
       -> Rewrite m dec exp
  crushU :: (Monoid result)
         => Translate m dec (Generic exp) result
         -> Translate m dec exp result
```

`Walker` is a multi-parameter type class, which signifies the ability to walk over a specific type. It requires that the type be an instance of type `Term`, which will provide our generics machinery. `allR` applies the `Generic` rewrites to all the chosen children of this node. `crushU` applied a `Generic` `Translate` to a common, monoidal result, to all the interesting children of this node.

Table 5 lists the functions that use the `Term` typeclass to provide various promotion and extraction functions over the the `Translate` type, as well as `allR` and `crushU`. `Term` is implicit in the type of `allR` and `crushU`, because `Walker` is a subclass of `Term`.

Table 5. Shallow Translate Combinators in KURE

Name	Type	Purpose
All functions in this table have context `(Monad m,Monoid dec,...) =>` ...		
`allR` ::	`(..., Walker m dec e) -> Rewrite m dec (Generic e) -> Rewrite m dec e`	Apply the argument to each immediate child.
`crushU` ::	`(..., Walker m dec e, Monoid r) -> Translate m dec (Generic e) r -> Translate m dec e r`	Apply the argument to each immediate child, to build something of a unified type.
`extractR` ::	`(..., Term e) -> Rewrite m dec (Generic e) -> Rewrite m dec e`	Extract a specific `Rewrite` from our generic `Rewrite`.
`extractU` ::	`(..., Term e) -> Translate m dec (Generic e) r -> Translate m dec e r`	Extract a specific `Translate` from our generic `Translate`, where both share a common (unifying) target type.
`promoteR` ::	`(..., Term e) -> Rewrite m dec e -> Rewrite m dec (Generic e)`	Create a generic `Rewrite` out of a specific `Rewrite`, where all other types fail.
`promoteU` ::	`(..., Term e) -> Translate m dec e r -> Translate m dec (Generic e) r`	Create a generic `Translate` out of a specific `Translate`, where all other types fail.

If we have provided a complete set of congruence operators (section 7.1), then providing the implementation for `allR` and `crushU` becomes straightforward. For `Expr`, our instance can be written as

```
instance (Monad m,Monoid dec) => Walker m dec Expr where
   allR rr = varR
          <+ litR
          <+ addR (extractR rr) (extractR rr)
          <+ eseqR (extractR rr) (extractR rr)
   crushU rr = varU
            <+ litU
            <+ addU (extractU rr) (extractU rr)
            <+ eseqU (extractU rr) (extractU rr)
```

One caveat is we do need to make sure we successfully accept all constructors, including the ones with no interesting arguments, otherwise `allR` and `crushU` will unexpectedly fail on these constructors.

8 Providing Deep Generic Traversal Combinators

We reach a significant issue when we try to implement deep traversals. Our shallow depth rewrite combinators, like `allR` act on *specific* types, but we want to rewrite anything in our `Generic` type sibling set. If we consider writing a topdown rewriter that applies a rewrite at every node in a top-down manner, we appear to want a function of type

```
topdownR :: (Walker m dec e)                -- INCORRECT TYPE, ATTEMPT 1
         => Rewrite m dec (Generic e)
         -> Rewrite m dec e
topdownR rr = extractR rr >-> allR (promoteR (topdownR rr))
```

We are confident that `Generic e` contains all the possible interesting sub-children. However, attempts to compile this function lead to type failures, even though this appears to be a reasonable implementation. We extract the rewrite at the current type, apply it, then rewrite all the children, using `topdownR` recursively. However, this function is unresolvably ambiguous. The `promote` allows a *single* rewrite type to be promoted, and critically, this type is not determinable at compile time for associated type synonyms. This definition actually type-checks for associated *data-types*, but the use of associated type synonyms is central to our lightweight generic mechanism.

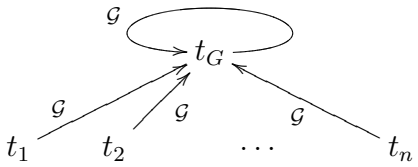
We need to take a different approach. We could take the type from `topdown` (and `all`) in Strafunski, which performs the whole traversal at the universal type. In KURE, using our typed 'universal' type, `topdown` would transliterate into

```
topdownR :: (Walker m dec e)                -- INCORRECT TYPE, ATTEMPT 2
         => Rewrite m dec (Generic e)
         -> Rewrite m dec (Generic e)
topdownR rr = rr >-> allR (topdownR rr)
```

We are now operating at the generic type, hiding any extraction inside `rr`. However, the type of `allR` takes a rewrite of type `Generic e` and returns a rewrite over `e`. This fails because `Generic e` does not have the ability to invoke `allR` directly.

> The key to getting `topdownR` working is empowering `Generic e` with the same traversal abilities as `e`, specifically the ability to use `Generic` to find the generic type. This means that for this function to work, `Generic e` and `e` must be of same type. We can achieve this by using the `~` operator provided by associated types, and augmenting the context of deep traversal with the aesthetically pleasing
>
> ```
> (...,Generic e ~ e) =>
> ```
>
> which means literally that `Generic e` and `e` must unify.

Diagrammatically, this means that we now have the following coercion relationship provided by our generic type function, $\mathcal{G}$.



Retrospectively this result is not surprising, in the sense that `Generic` is being used as a type coercion, and in a custom DSL implementation, the identity coercion would be trivial. Hosting KURE in Haskell has forced the creative use of the type system. Taking this into consideration, we can now give our type and implementation of `topdownR`, which is

```
topdownR :: (Generic e ~ e,Walker m dec e)
         => Rewrite m dec (Generic e)
         -> Rewrite m dec (Generic e)
topdownR rr = rr >-> allR (topdownR rr)
```

Like in Strafunski, we still return a `Rewrite` over our universal type, and we can safely use `extract` when invoking `topdownR` to build a topdown rewrite strategy at the required type. We can also write the type of `topdownR` as

```
topdownR :: (Generic e ~ e,Walker m dec e)
         => Rewrite m dec e
         -> Rewrite m dec e
topdownR rr = rr >-> allR (topdownR rr)
```

but prefer the former because it expresses explicitly that we are operating over our `Generic` type.

Table 6 gives a list of the provided deep traversal combinators, all of which use the ~ trick, and have straightforward implementations.

Table 6. Deep Translate Combinators in KURE

Name	Purpose
`topdownR`	Apply a rewrite in a top down manner.
`bottomupR`	Apply a rewrite in a bottom up manner.
`alltdR`	Apply a rewrite in a top down manner, pruning at successful rewrites.
`downupR`	Apply a rewrite twice, in a top down and bottom up way, using one single tree traversal.
`innermostR`	A fixed point traversal, starting with the innermost term.

	All these functions have type
	`(Generic e ~ e,Walker m dec e)`
	`=> Rewrite m dec (Generic e)`
	`-> Rewrite m dec (Generic e)`

In order to provide this reflective `Generic` ability, we need to provide an instance for `Generic e`, for each `Generic e`. To continue our running example, we can have to add an `OurGeneric` instance for both `Term` and `Walker`.

```
instance Term OurGeneric where
  -- OurGeneric is its own Generic root.
  type Generic OurGeneric = OurGeneric
  inject    = id
  select e  = Just e

instance (Walker m dec Stmt, Walker m dec Expr, Monad m,Monoid dec)
       => Walker m dec OurGeneric where
        allR rr = transparently $ rewrite $ \ e -> case e of
                GStmt s -> liftM GStmt $ apply (allR rr) s
                GExpr s -> liftM GExpr $ apply (allR rr) s
        crushU rr = translate $ \ e -> case e of
                GStmt s -> apply (crushU rr) s
                GExpr s -> apply (crushU rr) s
```

Again, this construction is tedious, but performed only once for each new type of syntax tree. KYB also generates the instance of `Walker` automatically for the Generic data-type, literally curing more of this boilerplate painfulness. Unfortunately, KYB does not generate the instance for `Term` because of limitations of the current implementation of Template Haskell.

Though this generic system have proven both useful and sufficient in practice, there are significant technical limitations with our generic mechanism. Specifically, we assume a completely monomorphic syntax tree without any parameterization, and KYB enforces this limitation when generating boilerplate code.

9 Using KURE Extensions

KURE is intended to help build rewrite engines, based on strategic programming technologies. KURE threads a user-defined local environment though all its transformations and rewrites. This environment must be a monoid. Often, this environment would be `()`, meaning that the rules are being evaluated in context free manner. Consider an operational semantics where the rules have the form

$$C \vdash E \rightarrow C \vdash E'$$

C itself does not change, and is read-only. In KURE, we can model C using our environment. For example, C might be the path through the rewrite tree.

$$C ::= root \mid \langle n \rangle C$$

We can construct C using the following Haskell code, and define a version of `apply` that records what edge number we are traversing when applying a `Translate`.

```
data C = C [Int] deriving (Monoid)

applyN :: (Monad m)
       => Int -> Translate m C e1 e2 -> e1 -> RewriteM m C e2
applyN n rr e = mapDecsM (\ (C xs) -> C (n:xs)) $ apply rr e
```

Using `applyN`, we can rework our congruence methods to use our context, `C`. For example `eseqR` would be rewritten

```
eseqR :: (Monad m)
      => Rewrite m C Stmt
      -> Rewrite m C Expr
      -> Rewrite m C Expr
eseqR rr1 rr2 = transparently $ translate $ \ e -> case e of
        (ESeq s1 s2) -> liftM2 ESeq (applyN 0 rr1 s1)
                                    (applyN 1 rr2 s2)
        _ -> fail "eseqR"
```

This `eseqR` performs all the services over `Expr` that the original `eseqR` did, but also updates the environment to record the path taken by any tree walker that uses `eseqR`. With this new location facility provided by the new `eseqR` (and siblings) we can write a combinator that rewrites at a specific, unique node, using the path through the term tree to find the node.

```
rewriteAtR :: (Walker m C e,Monad m, Generic e ~ e)
        => [Int] -> Rewrite m C (Generic e) -> Rewrite m C (Generic e)
rewriteAtR []     rr = rr
rewriteAtR (p:ps) rr = allR (getDecsT $ \ (C (x:xs)) ->
        if x == p then rewriteAtR ps rr else idR)
```

This performs a single rewrite, potentially deep inside the term tree, and gives a small flavor of the flexibility and power of the combinators provided by KURE.

10 Performance

In this section we perform some preliminary measurements to evaluate the cost of the flexibility that KURE provides. We know from experience that KURE is reasonably efficient on small and medium sized examples, but for a simple evaluation we implemented Fibonacci using both top-down and bottom-up rewrites in five different systems over a common term representation. In KURE, we implemented Fibonacci using the rewrite `fibR`.

```
fibR :: Rewrite Id () Exp
fibR = rewrite $ \ e -> case e of
        Fib (Val 0) -> return $ Val 1
        Fib (Val 1) -> return $ Val 1
        Fib (Val n) -> return $ Add (Fib (Dec (Val n)))
                                    (Fib (Dec (Dec (Val n))))
        _ -> fail "no match for fib"
```

We also implemented basic arithmetic reductions, as a separate rewrite, and repeatedly use a deep rewrite combinator until no more rewrites can be performed. For bottom-up rewriting, we expressed this using

```
evalFibR :: Rewrite Id () Exp
evalFibR = repeatR (bottomupR (tryR (fibR <+ arithR)) .+ failR "done")
```

Figure 7 gives the preliminary results of our five implementations, measuring wall-clock time in seconds on a 2.5GHz Intel Mac laptop. Our first two implementations use strategies that can never fail, implemented as simple endomorphic functions. The final three implementations use strategies that can encode failure.

All the implementations use the same basic term rewriting algorithm. The first implementation (Tree Rewrite) uses a straightforward and explicit tree traversal directly coded in Haskell. The second implementation (SYB) used SYB to provide the deep tree traversal. Both of these implementations do not allow

Table 7. Fibonacci Implemented Using Rewrites

		Top Down			**Bottom Up**		
Family	**Test**	`fib 20`	`25`	`30`	`fib 20`	`25`	`30`
No Failures	Tree Rewrite	0.018	0.157	2.024	0.010	0.075	0.987
	SYB	0.074	0.844	9.452	0.044	0.377	4.298
Can Fail	KURE	0.363	4.040	45.531	0.434	5.731	66.451
	SYB+`Maybe`	0.054	0.559	6.256	0.050	0.513	5.764
	StrategyLib	0.119	1.376	15.385	0.060	0.546	6.080

for the encoding of failure. The SYB-`Maybe` and the StrategyLib implementations are both transliterations of the KURE solution. SYB-`Maybe` encodes failure using the `Maybe` monad. StrategyLib is a recent version of Strafunski available on hackage.haskell.org. All implementations except KURE compare the result terms after each deep traversal for equality to find a fix-point, while KURE uses the built-in equality rewrite checker, via the '`.+`' combinator.

Clearly, KURE is the slowest implementation, and the DSL users pay a cost for the various extra capabilities and counters that KURE provides, up to an order of magnitude of wall clock time over Strafunski. Further investigation will reveal if the costs of KURE can be reduced by using new optimizations inside the Glasgow Haskell compiler. It would be nice if there was a way to completely eliminate the cost of the extra capabilities in KURE which are not being utilized. This brings KURE full circle; it was written to investigate exactly such a transformation [21], inside a more controlled setting than the Glasgow Haskell compiler.

11 Related Work

We have already surveyed in some detail two strategic programming systems, Stratego and Strafunski. The widest used and most mature strategy based rewrite system is Stratego [4], which grew out of the work on building a strategy language to translate RML [22], and drew inspiration from ELAN [14], a specification formalism. Stratego is a untyped language for expressing both rewrites and rewrite systems. Strafunski [5] is an implementation of strategic program in Haskell which gives types to strategies, and uses "scrap your boilerplate" generics to provide a universal type, and thus shallow and deep rewrites. Visser [23] is a useful resource as an overview of the discipline.

There have been many, many syntax translators written in Haskell. The Glasgow Haskell compiler itself is an example of a large rewriting optimizer. Another general framework, also being developed at the University of Kansas, is InterpreterLib [24], which uses Modular monadic semantics [25] as its basis for building algebra combinators over co-algebras. There are also systems that express rewrites as extensions to mainstream languages, including Java [26].

12 Conclusion and Further Work

We have hosted a strategy based rewrite system in Haskell, using the well understood principles of thinking in terms of types and computations, before implementation. Haskell as a host language worked remarkably well, especially the recent extension for associated type synonyms which were invaluable for finding types for our deeper traversal combinators. KURE as a DSL is clearly aimed at existing Haskell programmers, and requires a significant level of comfort with Haskell before being able to be productive in KURE. This same issue exists with KURE-enabled DSLs that export specific functionality for rewriting a specific syntax. In summary this experiment supports the long held believe that hosted

DSLs are great for the users of the host language, and for understanding the intrinsics of a specific DSL, but of questionable general purpose use, except to guide a future stand-alone language.

KURE as a system is in use at the University of Kansas, and forms a base tool for several exploration experiments into rewriting and optimizations. The new `Generic` mechanism works well in practice, and was a great application of associated type families, but has a number of significant limitations. We will look to lift these restrictions as issues arise in real grammars. We also we want explore the use Haskell's quasi-quoting facility [27] which will allow rewrite rules to be expressed directly in the target syntax as an alternative to working in abstract syntax.

Acknowledgments

I would like to thank the members of the Computer Systems Design Laboratory at the Information and Telecommunication Technology Center at the University of Kansas for providing a creative research environment. Specifically, I would like to thank Perry Alexander, Prasad Kulkarni, Vijayanand Manickam, Garrin Kimmell, Nicolas Frisby, as well as Adam Proctor of the University of Missouri. I would also like to thank the referees, for their many detailed and useful suggestions.

References

1. Peyton Jones, S., ed.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England (2003)
2. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In Chakravarty, M.M.T., ed.: ACM SIGPLAN Haskell Workshop 02, ACM Press (October 2002) 1–16
3. Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2005) 241–253
4. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C., et al., eds.: Domain-Specific Program Generation. Volume 3016 of Lecture Notes in Computer Science. Spinger-Verlag (June 2004) 216–238
5. Lämmel, R., Visser, J.: Typed Combinators for Generic Traversal. In: Proc. Practical Aspects of Declarative Programming PADL 2002. Volume 2257 of LNCS., Springer-Verlag (January 2002) 137–154
6. Gill, A.: Introducing the Haskell Equational Reasoning Assistant. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell, ACM Press (2006) 108–109
7. Hutton, G., Meijer, E.: Monadic parsing in Haskell. Journal of Functional Programing **8**(4) (1998) 437–444
8. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices **38**(3) (2003) 26–37 Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

9. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fundamenta Informaticae **69**(1–2) (2006) 123–178
10. Dolstra, E.: Functional stratego. In Visser, E., ed.: Proceedings of the Second Stratego Users Day (SUD'01). (2001) 10–17
11. Dolstra, E.: First-class rules and generic traversal for program transformation languages. Master's thesis, Utrecht University, Utrecht, The Netherlands (August 2001) INF/SCR-2001-15.
12. Dolstra, E., Visser, E.: First-class rules and generic traversal. Technical Report UU-CS-2001-38, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands (2001)
13. Hall, C., Hammond, K., Jones, S.P., Wadler, P.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems **18** (1996) 241–256
14. Borovansky, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with strategies in ELAN: A functional semantics. International Journal of Foundations of Computer Science **1** (2001) 69–95
15. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: 2nd USENIX Conference on Domain Specific Languages (DSL'99), Austin, Texas (October 1999) 109–122
16. Lämmel, R., Visser, J.: Type-safe functional strategies. In: In Scottish Functional Programming Workshop. (2000)
17. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In ACM, ed.: Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995, New York, NY, USA, ACM Press (1995) 333–343
18. Hughes, J.: Generalising monads to arrows. Science of Computer Programming **37** (May 2000) 67–111
19. Appel, A.W.: Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press (2002)
20. Jones, M.P., Diatchki, I.S.: Language and program design for functional dependencies. In: Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell, New York, NY, USA, ACM (2008) 87–98
21. Gill, A., Hutton, G.: The worker/wrapper transformation. Journal of Functional Programming **19**(2) (March 2009) 227–251
22. Visser, E., Benaissa, Z., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98), ACM Press (September 1998) 13–26
23. Visser, E.: A survey of rewriting strategies in program transformation systems. In Gramlich, B., Lucas, S., eds.: Workshop on Reduction Strategies in Rewriting and Programming (WRS'01). Volume 57 of Electronic Notes in Theoretical Computer Science., Utrecht, The Netherlands, Elsevier Science Publishers (May 2001)
24. Weaver, P., Kimmell, G., Frisby, N., Alexander, P.: Constructing language processors with algebra combinators. In: GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering, New York, NY, USA, ACM (2007) 155–164
25. Harrison, W.L., Kamin, S.N.: Metacomputation-based compiler architecture. In: Mathematics of Program Construction. (2000) 213–229
26. Balland, E., Moreau, P.E., Reilles, A.: Rewriting strategies in Java. Electron. Notes Theor. Comput. Sci. **219** (2008) 97–111
27. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM (2007) 73–82