Type-Changing Program Transformations with Pattern Matching

Joeri van Eekelen, Sean Leather and Johan Jeuring

Utrecht University, j.v.eekelen@gmail.com, leather@cs.uu.nl, johanj@cs.uu.nl

Abstract

We present a system for program transformation based on the type system of let-polymorphic lambda calculus with **fix** and **case**. Transformations are expressed as inference rules and are the input to the transformation system. Transformations are allowed to change the types of terms while the type and transform system ensures type-correctness of the target term. We define both a general transformation system for expressions and patterns that is based on the underlying type system, and give examples of user-defined transformations. Finally, we deal with transforming code to code that uses abstract datatypes, on which we can not pattern match.

1 Introduction

When writing code, occasionally the need comes up to change the data representation or underlying data structures. For example because a different structure is more efficient, or because another library is better maintained. Manually replacing instances of the old datatype to the new datatype is tedious and error-prone, so this is a good candidate for automation.

In this paper, we will look at transforming standard Haskell lists to the *Seq* α type as found in *Data.Sequence*. *Seq* α is an abstract datatype implemented using finger trees [1] with a more efficient implementation for several operations.

In Section 2, we will describe the object language used in this paper. Section 3 describes the transformation system for expression transformations. Section 4 the one for patterns. Section 5 discusses a way to deal with abstract datatypes, for which there are no constructors to pattern match on. Section 6 concludes this paper, and mentions some current related research.

2 Object Language

The language we are concerned with transforming is let-polymorphic lambda calculus, extended with **fix**, and **case**-expressions.

The term language of expressions and patterns is as follows:

$$\begin{array}{lcl} e & ::= & x \mid e_1\ e_2 \mid \lambda x \to e \mid \textbf{fix}\ e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{case}\ e_s\ \textbf{of}\ \{\overline{p \to e;}\} \\ p & ::= & x \mid C\ \overline{x} \end{array}$$

An expression is either a variable, a function application, a lambda abstraction, a fixed point, a **let**-expression, or a **case**-expression. A **case** expression consists of a scrutinee e_s and clauses $p \to e$. Each clause has a pattern p and a body e corresponding to that pattern. A pattern is either a pattern variable, or a datatype constructor saturated with pattern variables. Note that this means that patterns cannot be nested, unlike in programming languages such as Haskell. The reason for this will be discussed in Section 5. For readability of examples, we will borrow some of Haskell's syntax, like infix operators or list notation. However, these examples can be translated to the language defined above.

$$\boxed{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e : \tau}$$

$$\frac{x : \forall\, \overline{\alpha : \kappa}.\tau \in \Gamma \quad \tau' = [\tau_i \,/\, \alpha_i]\,\tau \quad \{\mathrm{K} \vdash \tau_i : \kappa_i\}}{\Gamma, \mathrm{K} \vdash x \rightsquigarrow x : \tau'}\ (\textsc{Var}) \qquad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau \to \tau}{\Gamma, \mathrm{K} \vdash \mathbf{fix}\ e \rightsquigarrow \mathbf{fix}\ e' : \tau}\ (\textsc{Fix})$$

$$\frac{\Gamma, \mathrm{K} \vdash e_1 \rightsquigarrow e_1' : \tau_1 \to \tau_2 \qquad \Gamma, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_1}{\Gamma, \mathrm{K} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_2}\ (\textsc{App})$$

$$\frac{\mathrm{K} \vdash \tau_1 : \star \qquad \Gamma \cup \{x : \tau_1\}, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_2}{\Gamma, \mathrm{K} \vdash \lambda x \to e \rightsquigarrow \lambda x \to e' : \tau_1 \to \tau_2}\ (\textsc{Lam})$$

$$\frac{\begin{array}{c}\Gamma, \mathrm{K} \cup \overline{\alpha : \kappa} \vdash e_1 \rightsquigarrow e_1' : \tau_1 \\ \Gamma \cup \{x : \forall\, \overline{\alpha : \kappa}.\tau_1\}, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_2 \\ \overline{\alpha} \not\subseteq FV\,(\Gamma)\end{array}}{\Gamma, \mathrm{K} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2' : \tau_2}\ (\textsc{Let})$$

$$\frac{\begin{array}{c}\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \\ \{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i \rightsquigarrow p_i' : \tau_1 \Rightarrow \Gamma_i \\ \Gamma \cup \Gamma_i, \mathrm{K} \vdash e_i \rightsquigarrow e_i' : \tau_2\}\end{array}}{\begin{array}{l}\Gamma, \mathrm{K} \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to e_i;}\} \rightsquigarrow \\ \quad \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{p_i' \to e_i';}\} : \tau_2\end{array}}\ (\textsc{Case})$$

Figure 1: Expression transformation.

$$\boxed{\Gamma, \mathrm{K} \vdash_{\text{pat}} p \rightsquigarrow p : \tau \Rightarrow \Gamma}$$

$$\frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\text{pat}} x \rightsquigarrow x : \tau \Rightarrow \{x : \tau\}}\ (\textsc{pVar}) \qquad \frac{\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau}{\begin{array}{l}\Gamma, \mathrm{K} \vdash_{\text{pat}} C\ x_1 \ldots x_n \rightsquigarrow \\ \quad C\ x_1 \ldots x_n : \tau \Rightarrow \{x\_i : \tau_i\}\end{array}}\ (\textsc{pCon})$$

Figure 2: Pattern transformation.

The syntax of types and kinds is as follows:

$$\begin{array}{lcl}\kappa & ::= & \star \mid \kappa_1 \to \kappa_2 \\ \tau & ::= & t \mid \tau_1 \to \tau_2 \mid \tau_1\ \tau_2 \\ t & ::= & \mathcal{T} \mid \alpha\end{array}$$

The type system is the standard Hindley-Milner[2] type system for let-polymorphic lambda calculus. The judgement $\Gamma, \mathrm{K} \vdash e : \tau$ means that, in type environment Γ and kind environment K, expression e has type τ.

3 Expression Transformations

The type of transformations we are most interested in are those that change expressions of one type to expressions of another type. For example, code that uses the standard Haskell $[\alpha]$ datatype to code that makes use of the *Seq* α datatype of the *Data.Sequence* module. Or there is some code using *Maybe* α that you want to add error messages to, and thus rewrite it to use *Either String* α.

Expression transformations are specified by inference rules similar to the typing rules. The expression transformation judgement $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau$ states that in type environment Γ and kind environment K, there is a valid transformation from e to e', where e' has type τ.

A transformation system can be split up into a propagation part and a transformation part. The transformation part consists of rules that actually change the source term. The propagation part recursively applies the transformations to subterms, and thus propagates the transformation part. Transformation propagation is described in Figure 1, and can be derived from the typing rules.

The transformation part consists of user-specified rules. Typical rules are rewriting a library function that works for one type to a similar function that works for another type,

or conversions between two similar types. A transformation $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau$ is valid when $\Gamma, \mathrm{K} \vdash e' : \tau$. That is, the specified type of the target term must match the type assigned to it by typechecking it. This ensures that transformed programs are always type-correct.

For example, we want to rewrite the following code that uses lists to code using sequences from *Data.Sequence*:

$$main = \lambda x \to [1,2,3] +\!\!+ x$$

The $Seq\ \alpha$ equivalent of $(+\!\!+)$ is $(\bowtie)$, so we would like all applications of $(+\!\!+)$ to be rewritten to $(\bowtie)$. This is expressed by rule (LS-APP). To convert $[1,2,3]$ to a sequence, we can use the *fromList* function. Rule (LS-FROM) is used to apply *fromList* to any expression which has a list type after rewriting.

$$\frac{\Gamma, \mathrm{K} \vdash (\bowtie) : Seq\ \tau \to Seq\ \tau \to Seq\ \tau}{\Gamma, \mathrm{K} \vdash (+\!\!+) \rightsquigarrow (\bowtie) : Seq\ \tau \to Seq\ \tau \to Seq\ \tau}\ (\textsc{ls-app})$$

$$\frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : [\tau] \qquad \Gamma, \mathrm{K} \vdash fromList : [\tau] \to Seq\ \tau}{\Gamma, \mathrm{K} \vdash e \rightsquigarrow fromList\ e' : Seq\ \tau}\ (\textsc{ls-from})$$

We see that both rules are valid, as $\Gamma, \mathrm{K} \vdash (\bowtie) : Seq\ \tau \to Seq\ \tau \to Seq\ \tau$ is a premise of (LS-APP), and $\Gamma, \mathrm{K} \vdash fromList\ e' : Seq\ \tau$ follows from $\Gamma, \mathrm{K} \vdash e' : [\tau]$ and $\Gamma, \mathrm{K} \vdash fromList : [\tau] \to Seq\ \tau$. With these two rules, the above code can be rewritten to:

$$main = \lambda x \to fromList\ [1,2,3] \bowtie x$$

Note that the variable x does not necessarily need *fromList* applied to it. Rule (VAR) of the propagation part allows variables to change types, as long as the result is type-correct.

Since these inference rules are rather verbose, we will use a shorthand notation for these transformations. The first is simply abbreviated as $(+\!\!+) \rightsquigarrow (\bowtie)$. The second is written as $M \rightsquigarrow fromList\ M$. M is a variable that matches any expression, which will be recursively transformed.

We have formalized this syntax in an upcoming thesis, but the intuition is that the left-hand side of the rewrite arrow $\rightsquigarrow$ is a pattern, possibly including variables, that match expressions. When the pattern matches, the variables are bound, and the entire expression is replaced by the right-hand side, where the metavariables get replaced by the recursively transformed subexpressions.

4 Pattern Transformations

With expression transformations, we can rewrite a lot of list functions to the corresponding function in the *Data.Sequence* module. Together with two conversion transformations $M \rightsquigarrow fromList\ M$ and $M \rightsquigarrow toList\ M$.[1] However, pattern matching can force the use of lists. The following code can only be made to work with sequences by rewriting the scrutinee x to $toList\ x$ such that its type matches the type of the patterns.

$$\begin{aligned} main = \lambda x \to\ & \mathbf{case}\ x\ \mathbf{of} \\ & (y : ys) \to y \\ & [\,] \quad\ \to 1 \end{aligned}$$

This section will deal with pattern transformations which will allow us to rewrite patterns to patterns of a different type. However, since $Seq\ \alpha$ is an abstract type, we do not

[1] *toList* is part of the *Foldable* typeclass. Since we do not have typeclasses, we use the type-restricted $toList : Seq\ \alpha \to [\alpha]$ instead.

have the constructors available. We will first establish pattern transformations with an example to transform *Maybe* α to *Either String* α.

Using the shorthand, we use the expression transformations *Just* $\rightsquigarrow$ *Right* and *Nothing* $\rightsquigarrow$ *Left* `"Nothing"` to transform *Maybe* α expressions to *Either String* α espressions. The string `"Nothing"` is arbitrary, and will have to be replaced on a case-by-case basis. These two transformations will allow functions that report errors to be succesfully transformed. However, when using pattern matching to handle the error, we are forced to convert back to *Maybe* α. This is where pattern transformations come in.

Pattern transformations judgements look like $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p \rightsquigarrow p' : \tau \Rightarrow \Gamma'$, which states that in type and kind environments Γ and K, pattern p can be rewritten to p' with type τ, binding variables with types in Γ'. Figure 2 has the propagation rules for pattern transformations. Note that since patterns are not nested, the propagation part only exists to make sure that there is at least one valid transformation, otherwise the (Case) rule would fail.

Like expression transformations, pattern transformations must also agree with the type system. A pattern transformation $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p \rightsquigarrow p' : \tau \Rightarrow \Gamma'$ is valid when $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p' : \tau \Rightarrow \Gamma'$.

There needs to be a pattern transformation for each constructor of *Maybe* α in order to fully transform every pattern match. The transformations (ME-Nothing) and (ME-Just) can be used to transform

$$
\begin{array}{l}
\mathit{main} = \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\quad \mathit{Just}\ y \quad \rightarrow y \\
\qquad\qquad\quad \mathit{Nothing} \rightarrow 1
\end{array}
$$

into

$$
\begin{array}{l}
\mathit{main} = \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\quad \mathit{Right}\ y \rightarrow y \\
\qquad\qquad\quad \mathit{Left}\ z \quad \rightarrow 1
\end{array}
$$

The pattern transformations can be written using a similar shorthand as the expression transformations. (ME-Nothing) is then written as *Nothing* $\rightsquigarrow$ *Left* M, and (ME-Just) can be written as *Just* $M \rightsquigarrow$ *Right* M. Metavariables in pattern transformations can also only appear on the right-hand side, which means a fresh pattern variable must be generated.

$$
\frac{\Gamma, \mathrm{K} \vdash \mathit{Left} : \mathit{String} \rightarrow \mathit{Either\ String}\ \tau \qquad x \text{ fresh}}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Nothing} \rightsquigarrow \mathit{Left}\ x : \mathit{Either\ String}\ \tau \Rightarrow \{x : \mathit{String}\}}\ (\text{ME-Nothing})
$$

$$
\frac{\Gamma, \mathrm{K} \vdash \mathit{Right} : \tau \rightarrow \mathit{Either\ String}\ \tau}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Just}\ x \rightsquigarrow \mathit{Right}\ x : \mathit{Either\ String}\ \tau \Rightarrow \{x : \tau\}}\ (\text{ME-Just})
$$

We check that transformations are valid. $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Left}\ x : \mathit{Either\ String}\ \tau \Rightarrow \{x : \mathit{String}\}$ follows from $\Gamma, \mathrm{K} \vdash \mathit{Left} : \mathit{String} \rightarrow \mathit{Either\ String}\ \tau$, as $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} x : \tau \Rightarrow \{x : \tau\}$ is an axiom for every type τ. Similarly, $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Right}\ x : \mathit{Either\ String}\ \tau \Rightarrow \{x : \tau\}$ follows from its premise.

5 Transforming to Abstract Types

The above pattern transformations work well when the target type, such as *Either String* α, has its constructors exposed. However, many useful datatypes such as *Map* k α, *Seq* α, and *Text* are abstract. This means we cannot pattern match on them. In order to still be able to perform meaningful transformations we instead transform to a datatype that lies between the concrete type in the source term and the abstract type which we want

to transform to. The *Data.Sequence* module provides the *ViewL* datatype, which has a structure similar to lists, but contains a *Seq* α in its recursive position, instead of a $[\alpha]$. Together with a view function $viewl :: Seq\ \alpha \to ViewL\ \alpha$, they make up a view.[4]

$$\textbf{data}\ ViewL\ \alpha = EmptyL \mid \alpha :< Seq\ \alpha$$

Instead of writing pattern transformations for *Seq* α, which we cannot do, we transform list patterns to pattens for *ViewL* α. The pattern transformations $[\,] \rightsquigarrow EmptyL$ and $(M : mn) \rightsquigarrow (M :< mn)$ together with expression transformation $M \rightsquigarrow viewl\ M$ enable us to transform the code at the end of Section 4 to:

$$\begin{aligned} main = \lambda x \to \textbf{case}\ viewl\ x\ \textbf{of}& \\ (y :< ys) &\to y \\ EmptyL\ &\to 1 \end{aligned}$$

The downside is that this approach does not compose well. If we would allow nested patterns, we would have no way to transform the pattern $(x : y : ys)$. We feel that GHC's `ViewPatterns` extension, which more closely resembles Wadler's views, can elegantly solve this by moving the view function into the pattern language, allowing patterns that look like $viewl \to (x :< (viewl \to (y :< ys)))$.

The view approach extends to other datatypes. For a regular[3] concrete type $\mathcal{C}$ and abstract type $\mathcal{A}$, the view type $\mathcal{A}@\mathcal{C}$ can be mechanically derived. It has a constructor for every constructor of $\mathcal{C}$, with fields of type $\mathcal{A}$ at the recursive positions, instead of $\mathcal{C}$. A view function of type $\mathcal{A} \to \mathcal{A}@\mathcal{C}$ can also be derived if conversion functions $from : \mathcal{C} \to \mathcal{A}$ and $to : \mathcal{A} \to \mathcal{C}$ are given.

6 Conclusions and Future Work

We have described a type and transform system that allows us to define program-wide type-changing transformations. The transformation system consists of a propagation part that is based on the underlying type system of the object language used, and a part consisting of user-defined transformation rules. We specifically focus on transformations from one datatype to another, such as list-based code to code using sequences from *Data.Sequence*. Special care must be taken when transforming to an abstract datatype, as pattern-matching on them is impossible.

Current research is going into adding support for datatype definitions and other object language features, and extending the type and transform system such that it provides not only type-correctness guarantees, but also static guarantees about preservation of semantics.

References

[1] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, March 2006.

[2] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[3] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM.

[4] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM.