Type-Changing Program Transformations with Pattern Matching

Joeri van Eekelen

July 9, 2013

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 thesis, we will investigate a type system-based transformation system for rewriting expressions to make use of different datatypes. Specifically, we deal with the issues presented by pattern matching and custom datatype definitions. In this, we build on previous work done by Leather et al.[5].

To achieve this, we take the following steps:

1. Give a formal description of the object language and its type system
2. Introduce transformations for the expression language
3. Introduce transformations for the pattern language
4. Introduce transformations for datatype declarations
5. Give a notation and formal interpretation for these transformations

The rest of the thesis is structured as follows: In Section 2, we describe the object language and the structure of the transformation system, together with transformation rules for the expression language. Sections 3 describes the changes in the transformation system necessary to handle transformations of patterns, and Section 4 shows a way to use pattern transformations to deal with abstract datatypes, whose constructors are not exposed. In Section 5, we will go further into datatype transformations, and constructor transformations. Section 6 considers new object language features that can be found in Haskell, specifically extensions to our relatively simplistic pattern language. We then look at the consequences these features have on the pattern transformation system. Section 7 describes the implementation, and in what ways it is different from the declarative specification in the earlier sections. As new constructs of the transformation system are gradually being introduced to the system described in Section 2, an overview of the full system is included in Appendix A.

2 Basic system

In this section, we will first describe the object language and its type system. Then we introduce a set of type-based inference rules that make up a transformation system for expressions in our object language. An example transformation that is possible within this system, is rewriting the following piece of code to use *Set Int* instead of *Int*:

$$f :: [Int] \to [Int]$$
$$f\ xs\ ys = 1 : xs +\!\!+ ys$$

This will then look like:

$$f' :: Set\ Int \to Set\ Int$$
$$f'\ xs\ ys = insert\ 1\ (\cup\ xs\ ys)$$

Similarly, we can rewrite the *reverse* function to make use of Hughes' lists [3]:

$$reverse :: [a] \to [a]$$
$$reverse\ [\,] = [\,]$$
$$reverse\ (x : xs) = reverse\ xs +\!\!+ [x]$$

can be transformed into

$$\textbf{type}\ DL\ a = [a] \to [a]$$
$$reverse' :: [a] \to DL\ a$$
$$reverse'\ [\,] = id$$
$$reverse'\ (x : xs) = reverse'\ xs \circ ([x] +\!\!+)$$

2.1 Object language and type system

A program *P* consists of zero or more datatype declarations, followed by a single *main*= expression. The expression syntax is the lambda calculus, extended with **case**. Additionally, integer and string literals are interpreted as constructors *C*. Of note is that the patterns used in a **case** expression cannot be nested. A pattern is either a single pattern variable, or a constructor followed by variables.

The syntax of datatype declarations is similar to that of Haskell. Datatypes are defined with a list of constructors, each of which have zero or more fields of a specified type. The main difference is that type variables in datatype declarations need to have a kind annotation, which is not allowed in Haskell programs[1]. The sole purpose of this is simplicity: we could infer the kinds of datatypes and set the kind of uninferrable type variables to $\star$ as Haskell does, but this would distract from the type system. On the other hand, we cannot leave out the kinds entirely, as they prevent a class of errors concerning parameterized datatypes. An example of such an error occurs in the following Haskell code:

[1] GHC allows kind annotations on datatype declarations when the *KindSignatures* language extension is enabled.

P	::=	$\overline{D}$; $main = e$	program
D	::=	**data** $T\ \overline{\alpha : \kappa} = \overline{C}$ |	datatype declaration
C	::=	$C\ \overline{\tau}$	
κ	::=	$\star$	kind of types of terms
	|	$\kappa \to \kappa$	kind of type constructors
t	::=	α	type variable
	|	T	type constructor
τ	::=	t	type name
	|	$\tau \to \tau$	function type
	|	$\tau\ \tau$	type application
σ	::=	$\forall\ \overline{\alpha : \kappa}.\tau$	type scheme
e	::=	v	term name
	|	$e\ e$	term application
	|	$\lambda x \to e$	lambda abstraction
	|	**let** $x = e$ **in** e	let binding
	|	*fix* e	fixpoint operator
	|	**case** e **of** $\{\overline{p \to e;}\}$	case
v	::=	x	term variable
	|	C	data constructor
p	::=	x	pattern variable
	|	$C\ \overline{x}$	constructor pattern

Figure 1: Object language syntax

```
extract :: f a → a
data Weird a = Weird (a Int)
break = extract (Weird [1])
```

A formal definition of the object language syntax is given in Figure 1.

Some examples will use syntactic sugar that is not valid according to the syntax. These deviations include Haskell-style list syntax such as $[1,2,3]$, infix operators, and the circumfix list type $[a]$. All of these can be easily desugared into valid syntax, at the cost of some clarity. Specifically, $[1,2,3]$ desugars into (:) 1 ((:) 2 ((:) 3 [])), an infix application a ++ b desugars into (++) $a\ b$, and the list type $[a]$ desugars into [] a.

The type system for expressions as seen in Figure 2 is similar to the let-polymorphic lambda calculus. Type judgements are of the form $\Gamma, \mathrm{K} \vdash e : \tau$. In these judgements, Γ is the type environment, which is a mapping from term names to type schemes $v : \sigma$. K is the kind environment, which is a mapping from type names to kinds $t : \kappa$. Finally, e is the expression and τ is its type.

Generalisation happens explicitly in the *TLet*-rule by putting type variables in scope of the definition and requiring that they are not free in the type environment Γ. Instantiation of type schemes happens in the *TVar* rule, where the quantified type variables are substituted by types with matching kinds.

Type checking **case**-expressions requires a second type of judgement: that of pattern typing. These judgements are of the form $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p : \tau \Rightarrow \Gamma'$. Γ and K are the type and kind environment, as before. p is the pattern to check, and τ is its type. Γ' is a new type environment, which contains types for the pattern variables in p. This extra type

$$\textit{TVar}\quad \frac{x : \forall\, \overline{\alpha : \kappa}.\tau \in \Gamma \qquad \tau' = [\tau_i \,/\, \alpha_i]\,\tau \qquad \{\mathrm{K} \vdash \tau_i : \kappa_i\}}{\Gamma, \mathrm{K} \vdash x : \tau'}$$

$$\textit{TApp}\quad \frac{\Gamma, \mathrm{K} \vdash e_1 : \tau_2 \to \tau \qquad \Gamma, \mathrm{K} \vdash e_2 : \tau_2}{\Gamma, \mathrm{K} \vdash e_1\ e_2 : \tau}$$

$$\textit{TAbs}\quad \frac{\mathrm{K} \vdash \tau_1 : \star \qquad \Gamma \cup \{x : \tau_1\}, \mathrm{K} \vdash e : \tau}{\Gamma, \mathrm{K} \vdash \lambda x \to e : \tau_1 \to \tau}$$

$$\textit{TLet}\quad \frac{\Gamma, \mathrm{K} \cup \overline{\alpha : \kappa} \vdash e_1 : \tau_1 \qquad \Gamma \cup \{x : \forall\, \overline{\alpha : \kappa}.\tau_1\}, \mathrm{K} \vdash e_2 : \tau_2 \qquad \overline{\alpha} \notin FV\,(\Gamma)}{\Gamma, \mathrm{K} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\textit{TFix}\quad \frac{\Gamma, \mathrm{K} \vdash e : \tau \to \tau}{\Gamma, \mathrm{K} \vdash \mathit{fix}\ e : \tau}$$

$$\textit{TCase}\quad \frac{\Gamma, \mathrm{K} \vdash e : \tau_1 \qquad \{\Gamma \vdash_{\mathrm{pat}} p_i : \tau_1 \Rightarrow \Gamma_i \qquad \Gamma \cup \Gamma_i, \mathrm{K} \vdash e_i : \tau_2\}}{\Gamma, \mathrm{K} \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to e_i;}\} : \tau_2}$$

Figure 2: Typing for our expression language

$$\textit{PatVar}\quad \frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} x : \tau \Rightarrow \{x : \tau\}}$$

$$\textit{PatCon}\quad \frac{\Gamma, \mathrm{K} \vdash C : \overline{\tau_i \to}\ \tau}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} C\ \overline{x_i} : \tau \Rightarrow \{x_i : \tau_i\}}$$

Figure 3: Typing rules for patterns

environment is used in the *TCase* rule to typecheck the body of each pattern-matching clause.

The typing rules for patterns can be seen in Figure 3.

Datatype declarations do not require type checking directly, but they will add tothe type and kind environments. Furthermore, the fields of the constructors of the datatypes must all have a type of kind $\star$. Their typing can be seen in the *TData* and *TCon* rules of Figure 4. The judgement for datatype declarations produces a new type environment Γ' as well as a new kind environment K'. The kind environment will contain only the kind of the type constructor, while the type environment will have the types of the constructors.

The *TProg* rule brings the new type and data constructors in scope of the *main* expression, which wraps up the type checking.

$$\textit{TData}\quad \frac{\mathrm{K}' = \{T : \overline{\kappa_i \to} \star\} \quad \{\mathrm{K} \cup \mathrm{K}' \cup \overline{\alpha : \kappa} \vdash C_i \Rightarrow \Gamma_i'\}}{\mathrm{K} \vdash \textbf{data}\ T\ \overline{\alpha : \kappa} = \overline{C} \mid \Rightarrow \bigcup \Gamma_i', \mathrm{K}'}$$

$$\textit{TCon}\quad \frac{\{\mathrm{K} \vdash \tau_i : \star\}}{\mathrm{K} \vdash C\ \overline{\tau} \Rightarrow \{C : \overline{\tau \to}\ T\ \overline{\alpha}\}}$$

$$\textit{TProg}\quad \frac{\{\mathrm{K} \vdash D_i \Rightarrow \Gamma_i, \mathrm{K}_i\} \quad \Gamma \cup \bigcup \Gamma_i, \mathrm{K} \cup \bigcup \mathrm{K}_i \vdash e : \tau}{\Gamma, \mathrm{K} \vdash \overline{D}; \mathit{main} = e : \tau}$$

Figure 4: Typing rules for datatype declarations and programs

$$\textit{KVar}\quad \frac{t : \kappa \in \mathrm{K}}{\mathrm{K} \vdash t : \kappa}$$

$$\textit{KArrow}\quad \frac{\mathrm{K} \vdash \tau_1 : \star \quad \mathrm{K} \vdash \tau_2 : \star}{\mathrm{K} \vdash \tau_1 \to \tau_2 : \star}$$

$$\textit{KApp}\quad \frac{\mathrm{K} \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \mathrm{K} \vdash \tau_2 : \kappa_1}{\mathrm{K} \vdash \tau_1\ \tau_2 : \kappa_2}$$

Figure 5: Kind checking rules

2.2 Transformation system

Now that we have a complete description of the object language and its typing rules, we will take a look at the type of transformations we would like to be able to perform.

First, we will take a look at transforming expressions. The following short program, which consists only of a *main* expression and no datatype declarations, shows several changes we want to be able to perform:

```
   -- Original program
main1 = λx → insert x [1,2,3]
   -- Preferred translation
main2 = λx → Data.Set.insert x (to [1,2,3])
```

Given the type and kind environments Γ and K as shown in Figure 6, we can assign the types $\mathit{Int} \to [\mathit{Int}]$ and $\mathit{Int} \to \mathit{Set\ Int}$ to the two respective expressions.

Going from main_1 to main_2, two transformations have occurred. *insert* has been changed to *Data.Set.insert*, and [1,2,3] has been changed to *to* [1,2,3]. In Section

$$
\begin{array}{lllll}
\Gamma & = & \{ & \mathit{insert} & : \forall\, a : \star . a \to [a] \to [a] \\
 & & , & \mathit{Data.Set.insert} & : \forall\, a : \star . a \to \mathit{Set}\ a \to \mathit{Set}\ a \\
 & & , & \mathit{to} & : \forall\, a : \star . [a] \to \mathit{Set}\ a \\
 & & , & (:) & : \forall\, a : \star . a \to [a] \to [a] \\
 & & , & [\,] & : \forall\, a : \star . [a] \\
 & & \} & & \\
 & & & & \\
\mathrm{K} & = & \{ & \mathit{Int} & : \star \\
 & & , & [\,] & : \star \to \star \\
 & & \} & &
\end{array}
$$

Figure 6: Type and kind environment for the example

2.3 about the transformation syntax, we will see how these two are instances of two important types of transformations.

In order to decide whether an expression e can be transformed into an expression e', we introduce a type and transform system, or a TTS in short. A TTS is a set of inference rules, similar to the typing rules, with judgements of the form $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau$. These state exactly what we want: Under a type environment Γ and a kind environment K, we can transform an expression e to expression e', where e' has type τ.

A condition that must always hold, is that a TTS must be consistent with the type system. That is, a TTS must give the same type to the result term as the type system would.

$$\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau \implies \Gamma, \mathrm{K} \vdash e' : \tau \tag{1}$$

We do not care about the typeability of the source term, however, if there is no τ such that $\Gamma, \mathrm{K} \vdash e : \tau$, then there may not be any valid transformations either. Otherwise, there is always at least one valid result term, as we will see shortly.

$$\Gamma, \mathrm{K} \vdash e : \tau \implies \exists e', \tau' \text{ such that } \Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau' \tag{2}$$

Type and transform systems consist of two parts: a set of inference rules that depend only on the type system and the object language on which it operates, and a set of rules that are specific to the actual transformation under consideration. The first set of rules, shown in Figure 7 and known as the propagation system, are based on the typing rules. They are syntax-driven, and are itself a TTS capable of transforming every input program. While it cannot be used to perform any non-trivial transformations, 1 holds, and it is enough to prove 2:

There is only one possible inference rule per syntactic construct, and by induction on the syntax tree, it can be shown that the identity transformation is always valid. Formally: $\Gamma, \mathrm{K} \vdash e : \tau$ implies $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e : \tau$. Since any TTS must always at least include the rules of the propagation system, it is always possible to derive the identity transformation for any well-typed term.

To understand how the rules of the propagation system work, it helps to compare them to the typing rules.

$$\textit{PVar}\quad \frac{x : \forall\, \overline{\alpha : \kappa}.\tau \in \Gamma \qquad \tau' = [\,\tau_i \,/\, \alpha_i\,]\,\tau \qquad \{\mathrm{K} \vdash \tau_i : \kappa_i\}}{\Gamma, \mathrm{K} \vdash x \rightsquigarrow x : \tau'}$$

$$\textit{PApp}\quad \frac{\Gamma, \mathrm{K} \vdash e_1 \rightsquigarrow e_1' : \tau_2 \to \tau \qquad \Gamma, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma, \mathrm{K} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau}$$

$$\textit{PAbs}\quad \frac{\mathrm{K} \vdash \tau_1 : \star \qquad \Gamma \cup \{x : \tau_1\}, \mathrm{K} \vdash e \rightsquigarrow e' : \tau}{\Gamma, \mathrm{K} \vdash \lambda x \to e \rightsquigarrow e' : \tau_1 \to \tau}$$

$$\textit{PLet}\quad \frac{\Gamma, \mathrm{K} \cup \overline{\alpha : \kappa} \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \cup \{x : \forall\, \overline{\alpha : \kappa}.\tau_1\}, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad \overline{\alpha} \notin FV\,(\Gamma)}{\Gamma, \mathrm{K} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2' : \tau_2}$$

$$\textit{PFix}\quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau \to \tau}{\Gamma, \mathrm{K} \vdash \mathit{fix}\ e \rightsquigarrow \mathit{fix}\ e' : \tau}$$

$$\textit{PCase}\quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \qquad \{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p_i : \tau_1 \Rightarrow \Gamma_i \qquad \Gamma \cup \Gamma_i, \mathrm{K} \vdash e_i \rightsquigarrow e_i' : \tau_2\}}{\Gamma, \mathrm{K} \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to e_i;}\} \rightsquigarrow \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{p_i \to e_i';}\} : \tau_2}$$

Figure 7: Propagation system for the let-polymorphic lambda calculus

$$\textit{PVar}\quad \frac{x : \forall\, \overline{\alpha : \kappa}.\tau \in \Gamma \qquad \tau' = [\,\tau_i \,/\, \alpha_i\,]\,\tau \qquad \{\mathrm{K} \vdash \tau_i : \kappa_i\}}{\Gamma, \mathrm{K} \vdash x \rightsquigarrow x : \tau'} \qquad \textit{TVar}\quad \frac{x : \forall\, \overline{\alpha : \kappa}.\tau \in \Gamma \qquad \tau' = [\,\tau_i \,/\, \alpha_i\,]\,\tau \qquad \{\mathrm{K} \vdash \tau_i : \kappa_i\}}{\Gamma, \mathrm{K} \vdash x : \tau'}$$

$$\textit{PApp}\quad \frac{\Gamma, \mathrm{K} \vdash e_1 \rightsquigarrow e_1' : \tau_2 \to \tau \qquad \Gamma, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma, \mathrm{K} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau} \qquad \textit{TApp}\quad \frac{\Gamma, \mathrm{K} \vdash e_1 : \tau_2 \to \tau \qquad \Gamma, \mathrm{K} \vdash e_2 : \tau_2}{\Gamma, \mathrm{K} \vdash e_1\ e_2 : \tau}$$

The above suggests that the propagation rules can be formed from the typing rules just by changing the typing judgements to propagation judgements, and this is indeed the case. This makes 1 hold by construction, though it must still be proved for every rule specific to the transformation.

Now we return to the example earlier in this section. In it, there were two specific transformations: *insert* has changed to *Data.Set.insert*, and $[1,2,3]$ has changed to *to* $[1,2,3]$. These can be described with the following rules:

$$\textit{Replace} - \textit{insert} \qquad \frac{\Gamma, \mathrm{K} \vdash \textit{Data.Set.insert} : \tau \to \textit{Set}\ \tau \to \textit{Set}\ \tau}{\Gamma, \mathrm{K} \vdash \textit{insert} \rightsquigarrow \textit{Data.Set.insert} : \tau \to \textit{Set}\ \tau \to \textit{Set}\ \tau}$$

$$\textit{Intro} - \textit{to} \qquad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : [\tau] \qquad \Gamma, \mathrm{K} \vdash \textit{to} : [\tau] \to \textit{Set}\ \tau}{\Gamma, \mathrm{K} \vdash e \rightsquigarrow \textit{to}\ e' : \textit{Set}\ \tau}$$

The first rule describes how *insert* can transform to *Data.Set.insert*. The premises state that the *insert* function must be in scope with the expected type. This rule is obviously consistent with the type system, since the typeability of the result term is part of the premises.

The second rule states that if it is possible to transform a term e to a term e' with a list type, it is possible to apply *to* to it to get a term with a *Set* type. To see if this rule is consistent with the type system, we look at the result type. It is a function application, so *TApp* must hold. This means that the argument type of *to* must match the type of e'. As this is implied by the premises, we know that the rule is consistent.

Earlier, we stated that for a well-type source type, there will always be *at least* one transformation, we have not mentioned how many transformations there will be, or if there is an upper bound. Indeed, $\Gamma, \mathrm{K} \vdash \lambda x \to \textit{insert}\ x\ [1,2,3] \rightsquigarrow \lambda x \to \textit{to}\ (\textit{insert}\ x\ [1,2,3])$: *Set a* holds as well. In general, there is no upper bound to the amount of valid transformations:

$$\textit{Intro} - \textit{from} \qquad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \textit{Set}\ \tau \qquad \Gamma, \mathrm{K} \vdash \textit{from} : \textit{Set}\ \tau \to [\tau]}{\Gamma, \mathrm{K} \vdash e \rightsquigarrow \textit{from}\ e' : [\tau]}$$

Adding a rule like *Intro* − *from*, it becomes possible to derive arbitrarily long sequences of *from* (*to* (...)). In the theoretical framework which only states that a certain transformation is valid, this is not a problem. However, we will need to handle this in the implementation.

2.3 Transformation syntax

The current way of introducing new rules is rather clunky. We need to write a completely new inference rule, and then prove its consistency with the type system. These are the two most important reasons why we have devised a special syntax for transformations.

The two transformations of the previous section can be written, using this syntax, as:

1. *insert* $\rightsquigarrow$ *Data.Set.insert*
2. *M* $\rightsquigarrow$ *to M*

Transformations in the syntax follow the form $\pi \rightsquigarrow \pi$. Both πs are expressions that can contain metavariables, which are differentiated from normal variables by writing them as a single uppercase character. The π on the left-hand sign of the arrow is the pattern

TRF	$::=$	$\pi \rightsquigarrow \pi$	Metavariable rule
π	$::=$	$\pi\ \pi$	Rewrite pattern
	$\mid$	e	
	$\mid$	M	

Figure 8: Transformation syntax

of the transformation, and the π on the right-hand side is the substitute. Figure 8 shows the syntax of π.

There are some additional restrictions on the use of metavariables in the transformation syntax. Specifically:

1. Metavariables used on the right-hand side must also occur on the left-hand side.
2. Metavariables used on the left-hand side may only appear there once.

To use rules described in this syntax in transformations, we need to interpret them as inference rules. Figure 9 shows how this is done.

When transforming an expression using a rule described by this syntax, the expression is first matched against the pattern on the left-hand side, producing an environment Σ, which maps metavariables to typed expressions. Pattern matching happens by induction on the structure of the pattern.

A pattern of the form $\pi\ \pi$ only matches expressions of the form $e\ e$, in which case the resulting environment is the union of the environments obtained by matching the two subexpressions against the subpatterns. When a pattern is of the form e, it matches the exact expression e, up to alpha-equivalence, and the resulting environment is empty.

Metavariable patterns require more attention. Not only do they add a mapping for the metavariable to the environment, but they also transform the expression they're matching using rules either from the propagation system, or from other rules that follow from the syntax. It corresponds to the first premise in the $Intro-to$ rule, which only has one metavariable. The resulting environment maps the metavariable to the transformed expression and its type.

Transformations resulting from the syntax are automatically consistent with the type system, due to the rules in Figure 9. To prove this, we start with induction on the substitute. In the case of an expression, there is a premise that states exactly the typability of the resulting expression. For function application, the typing rule is built-in, assuming that the individual subexpressions are well-typed. For the metavariable rule, it is necessary to prove that if $M \mapsto e' : \tau \in \Sigma$, then $\Gamma, \mathrm{K} \vdash e' : \tau$. The only rule in the $\vdash_{\text{mat}}$ relation that adds to Σ is *MMeta*, which has $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau$ as a premise. By induction, this means that $\Gamma, \mathrm{K} \vdash e' : \tau$, as required.

3 Pattern transformations

The system described in the previous section can already be used to describe many useful transformations, but, especially with pattern matching, the results may not be as effective as hoped. Consider the following example:

$$\textit{TrfRule}\quad \frac{\pi_1 \rightsquigarrow \pi_2 \text{ is a rule} \quad \Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_1 @ e \Rightarrow \Sigma \quad \Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_2 = e' : \tau'}{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau'}$$

$$\textit{MApp}\quad \frac{\Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_1 @ e_1 \Rightarrow \Sigma_1 \quad \Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_2 @ e_2 \Rightarrow \Sigma_2}{\Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_1\ \pi_2 @ e_1\ e_2 \Rightarrow \Sigma_1 \cup \Sigma_2}$$

$$\textit{MExp}\quad \frac{}{\Gamma, \mathrm{K} \vdash_{\text{mat}} e @ e \Rightarrow \epsilon}$$

$$\textit{MMeta}\quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau}{\Gamma, \mathrm{K} \vdash_{\text{mat}} M @ e \Rightarrow \{M \mapsto e' : \tau\}}$$

$$\textit{SApp}\quad \frac{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_1 = e_1' : \tau \to \tau_1 \quad \Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_2 = e_2' : \tau}{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_1\ \pi_2 = e_1'\ e_2' : \tau_1}$$

$$\textit{SExp}\quad \frac{\Gamma, \mathrm{K} \vdash e : \tau}{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} e = e : \tau}$$

$$\textit{SMeta}\quad \frac{\Sigma\,(M) = e' : \tau'}{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} M = e' : \tau'}$$

Figure 9: Transformation relation

```
data IntList = IntNil | IntCons Int IntList
data CharList = CharNil | CharCons Char CharList
main = let map = λg → fix (λf → λl →
          case l of
            IntNil → IntNil
            IntCons x xs → IntCons (g x) (f xs))
       in map
```

In this code, we want to replace *IntList*s by *CharList*s, and use a TTS to transform the definition of *map* to something of type $(\mathit{Char} \to \mathit{Char}) \to \mathit{CharList} \to \mathit{CharList}$.

To do this, we have the following transformation rules:

- $M \rightsquigarrow \mathit{il2cl}\ M$
- $M \rightsquigarrow \mathit{cl2il}\ M$
- $M \rightsquigarrow \mathit{i2c}\ M$
- $M \rightsquigarrow \mathit{c2i}\ M$
- $\mathit{IntNil} \rightsquigarrow \mathit{CharNil}$
- $\mathit{IntCons} \rightsquigarrow \mathit{CharCons}$

Where $il2cl : IntList \to CharList$ and $cl2il : CharList \to IntList$ are inverses. Similarly, $i2c : Int \to Char$ and $c2i : Char \to Int$ are also inverses. Because the **case** rule uses the same patterns in the target expression as in the source, we find that the result of applying these rules that comes closest to a direct implementation of *map* for *CharList*s is:

$$
\begin{aligned}
main = \textbf{let}\ & map = \lambda g \to fix\ (\lambda f \to \lambda l \to \\
& \quad \textbf{case}\ cl2il\ l\ \textbf{of} \\
& \qquad IntNil \to CharNil \\
& \qquad IntCons\ x\ xs \to CharCons\ (g\ (i2c\ x))\ (f\ (il2cl\ xs))) \\
& \textbf{in}\ map
\end{aligned}
$$

It has the right type, but there is still conversion between *IntList* and *CharList*. In the ideal case, we'd prefer not to have to use any embedding/projection functions, though there may be functions outside the transformation scope that have no suitable replacement.

As a second example, consider transforming values of type *Maybe a* to values of type *Either String a*. Both types are often used for error-handling in pure code. The main difference is that *Nothing* simply encodes an error, while *Left e* can provide some extra details.

The rules we have are few:

- $Just \rightsquigarrow Right$
- $Nothing \rightsquigarrow Left$ `"Nothing"`
- $M \rightsquigarrow maybeToEither\ M$
- $M \rightsquigarrow eitherToMaybe\ M$

Here $maybeToEither : Maybe\ a \to Either\ String\ a$ and $eitherToMaybe : Either\ e\ a \to Maybe\ a$ are an embedding/projection pair, though *eitherToMaybe* loses the information of the *Left* constructor. The choice of *String* is arbitrary, any choice of *e* works, but a value of that type needs to be provided in the transformation rule for *Nothing*.

The first two transformations work very well for transforming the actual error-reporting code itself, but when transforming a handler for *Maybe a* values to a handler for *Either String a* values, we have to resort to conversion back to *Maybe a*:

$$
\begin{aligned}
main = \lambda x \to\ & \textbf{case}\ x\ \textbf{of} \\
& \quad Nothing \to \texttt{"Error!"} \\
& \quad Just\ i \to \texttt{"No error: "} +\!\!+ showInt\ i \\
main' = \lambda x \to\ & \textbf{case}\ eitherToMaybe\ x\ \textbf{of} \\
& \quad Nothing \to \texttt{"Error!"} \\
& \quad Just\ i \to \texttt{"No error: "} +\!\!+ showInt\ i
\end{aligned}
$$

From these examples, it becomes apparent that pattern-matching ties us down to use the type of the patterns in the source program. In the expressions above, the fact that we match on *Maybe a* patterns forces the scrutinee to be of type *Maybe a* as well. To combat this, we will introduce pattern transformations later in this section, which will allow us to turn the above examples into:

$$\textit{PatVar} \qquad \frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\text{pat}} x \rightsquigarrow x : \tau \Rightarrow \{x : \tau\}}$$

$$\textit{PatCon} \qquad \frac{\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\ x_1 \ldots x_n \rightsquigarrow C\ x_1 \ldots x_n : \tau \Rightarrow \{x_i : \tau_i\}}$$

Figure 10: Identity transformation for patterns

$$\begin{aligned}
\mathit{main} = \textbf{let}\ \mathit{map} = \lambda g \to \mathit{fix}\ (\lambda f \to \lambda l \to \\
\quad \textbf{case}\ l\ \textbf{of} \\
\quad\quad \mathit{CharNil} \to \mathit{CharNil} \\
\quad\quad \mathit{CharCons}\ x\ xs \to \mathit{CharCons}\ (g\ x)\ (f\ xs)) \\
\quad \textbf{in}\ \mathit{map}
\end{aligned}$$

and

$$\begin{aligned}
\mathit{main}' = \lambda x \to \textbf{case}\ x\ \textbf{of} \\
\quad\quad \mathit{Left}\ e \to \texttt{"Error!"} \\
\quad\quad \mathit{Right}\ i \to \texttt{"No error: "} +\!\!+\ \mathit{showInt}\ i
\end{aligned}$$

respectively.

In the rest of this section, we will introduce transformation inference rules for patterns, and alter the existing propagation rules to make use of them. Finally, we introduce another specialized syntax to accurately denote such pattern transformations.

3.1 Inference rules

Like the transformation system for expressions, we will define a similar system for patterns. However, since our object language does not allow for nested patterns, there are no pattern transformation premises for the pattern rules. Only a single pattern rule will be used per pattern.

Similar to the transformation judgement for expressions, the transformation judgements for patterns will look like typing rules:

Typing:	$\Gamma, \mathrm{K} \vdash_{\text{pat}} p : \tau \Rightarrow \Gamma'$
Transformation:	$\Gamma, \mathrm{K} \vdash_{\text{pat}} p_1 \rightsquigarrow p_2 : \tau \Rightarrow \Gamma'$

Judgements of this form mean that, with type and kind environments Γ and K, pattern p_1 can be transformed to pattern p_2 with type τ. Moreover, pattern p_2 produces a type environment Γ' containing the variables bound in the pattern.

As a parallel to the propagation system, there are transformation rules that are based on the typing rule for patterns. These rules are listed in Figure 10.

To use these inference rules, we need to combine them with the rules for expressions. We will modify the propagation rule for **case** expressions to transform patterns, and

$$PCase \qquad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \quad \{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i \rightsquigarrow p_i' : \tau_1 \Rightarrow \Gamma_i \qquad \Gamma \cup \Gamma_i, \mathrm{K} \vdash_{\text{pat}} e_i \rightsquigarrow e_i' : \tau_2\}}{\Gamma, \mathrm{K} \vdash_{\text{pro}} \textbf{case}\ e\ \textbf{of}\ \{\overline{p_i \to e_i;}\} \rightsquigarrow \textbf{case}\ e'\ \textbf{of}\ \{\overline{p_i' \to e_i';}\} : \tau_2}$$

Figure 11: Modified case propagation rule

use the transformed pattern in the resulting expression. The modifications of the **case** rule are minor. Instead of premises $\{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i : \tau \Rightarrow \Gamma_i\}$, we have premises $\{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i \rightsquigarrow p_i' : \tau \Rightarrow \Gamma_i\}$. In the conclusion, the clauses of the **case** expression are $\{\overline{p_i' \to e_i';}\}$ instead of $\{\overline{p_i \to e_i';}\}$. The new **case** rule can be seen in Figure 11.

For patterns, we have a similar consistency constraint as we had for expressions:

$$\Gamma, \mathrm{K} \vdash_{\text{pat}} p \rightsquigarrow p' : \tau \Rightarrow \Gamma' \implies \Gamma, \mathrm{K} \vdash_{\text{pat}} p' : \tau \Rightarrow \Gamma' \tag{3}$$

This consistency constraint holds by definition for the transformation rules in Figure 10, and from it follows that the consistency constraint for expressions holds for the new **case** rule. Secondly, we have the guarantee that, if the source pattern has a valid type, then there is at least one possible pattern transformation. As with expressions, this is the identity transformation.

$$\Gamma, \mathrm{K} \vdash_{\text{pat}} p : \tau \Rightarrow \Gamma' \implies \exists p', \tau', \Gamma'' \text{ such that } \Gamma, \mathrm{K} \vdash_{\text{pat}} p \rightsquigarrow p' : \tau' \Rightarrow \Gamma'' \tag{4}$$

Now we return to the *IntList* example. The pattern rule for transforming *IntNil* to *CharNil* is fairly straightforward. There are no bound variables, so we only need to make sure that the rule is consistent with the type system. For transforming *IntCons* to *CharCons*, we need to make sure that the resulting type environment contains the right pattern variables, with the correct types.

$$Pattern - IntNil \qquad \frac{\Gamma, \mathrm{K} \vdash \mathit{CharNil} : \mathit{CharList}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} \mathit{IntNil} \rightsquigarrow \mathit{CharNil} : \mathit{CharList} \Rightarrow \epsilon}$$

$$Pattern - IntCons \qquad \frac{\Gamma, \mathrm{K} \vdash \mathit{CharCons} : \mathit{Char} \to \mathit{CharList} \to \mathit{CharList}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} \mathit{IntCons}\ x\ y \rightsquigarrow \mathit{CharCons}\ x\ y : \mathit{CharList} \Rightarrow \{x : \mathit{Char}, y : \mathit{CharList}\}}$$

To prove that the consistency constraint holds, we look at the typing rule for constructor patterns. Because *CharNil* has no arguments, the pattern does not have any bound variables, and the produced type environment is empty. *CharCons* has two arguments, one of type *Char*, and one of type *CharList*. The result pattern has two bound variables, and the produced type environment maps these variables to the types as specified by the type of *CharCons*. This is precisely what the typing rule states, so these transformations are consistent.

The transformations for the *Maybe* example look as follows:

$$TRF \quad ::= \quad C\ \overline{M} \rightsquigarrow C\ \overline{M} \quad \text{Pattern transformation}$$

Figure 12: Pattern transformation syntax

$$\textit{Pattern} - \textit{Nothing} \qquad \frac{\Gamma, \mathrm{K} \vdash \textit{Left} : \tau' \to \textit{Either}\ \tau'\ \tau}{\Gamma, \mathrm{K} \vdash_{\text{pat}} \textit{Nothing} \rightsquigarrow \textit{Left}\ e : \textit{Either}\ \tau'\ \tau \Rightarrow \{e : \tau'\}}$$

$$\textit{Pattern} - \textit{Just} \qquad \frac{\Gamma, \mathrm{K} \vdash \textit{Right} : \tau \to \textit{Either}\ \tau'\ \tau}{\Gamma, \mathrm{K} \vdash_{\text{pat}} \textit{Just}\ x \rightsquigarrow \textit{Right}\ x : \textit{Either}\ \tau'\ \tau \Rightarrow \{x : \tau\}}$$

Although *Nothing* has no pattern variables, *Left e* does, so we need to include it in the type environment of bound variables. While the source program has no way of using this fresh variable, its inclusion in the type environment is necessary to remain consistent with the type system. The proof that the consistency constraint holds for these two transformations is similar to the proof for the transformations of the *IntList* example.

3.2 Pattern transformation syntax

As with expressions, giving inference rules is a verbose way of specifying transformations, and the consistency with the type system is not guaranteed. To counter these two issues, we introduce special syntax for pattern transformations.

The transformations of the previous section can be written, using this syntas, as:

1. $\textit{IntNil} \rightsquigarrow \textit{CharNil}$
2. $\textit{IntCons}\ M_1\ M_2 \rightsquigarrow \textit{CharCons}\ M_1\ M_2$
3. $\textit{Nothing} \rightsquigarrow \textit{Left}\ M$
4. $\textit{Just}\ M \rightsquigarrow \textit{Right}\ M$

As shown in Figure 12, a pattern transformation written in the syntax looks like $C\ \overline{M} \rightsquigarrow C\ \overline{M}$. On left-hand side of the arrow is the pattern of the transformation. It takes the shape of a constructor followed by an amount of metavariables equal to the arity of the constructor. The right-hand side is the substitute.

The intuition is that the transformation pattern matches against a constructor pattern with the same constructor, and a substitution environment Σ is built, mapping metavariables to pattern variables. The resulting pattern consists of the constructor in the substitute and the pattern variables that are bound to the metavariables in the substitute. The type of the resulting pattern and the types of the pattern variables in the type environment are based on the type of constructor C'.

Figure 13 shows the formalized interpretation of the pattern transformation syntax. Of note are the judgements of the form $\Sigma \vdash M \mapsto x$. These are similar to a simple lookup $\Sigma\ (M)$, except that the result is a fresh pattern variable if M is not bound in Σ. This occurs when there are metavariables on the right-hand side that are not present on the left-hand side of the transformation, as in the $\textit{Nothing} \rightsquigarrow \textit{Left}\ M$ example.

$$\textit{PatRule}\qquad \frac{\begin{array}{c} C\ M_1 \ldots M_n \rightsquigarrow C'\ N_1 \ldots N_m \text{ is a rule} \\ \Gamma, \mathrm{K} \vdash C' : \tau_1 \to \ldots \to \tau_m \to \tau \\ \Sigma = \{ M_i \mapsto x_i \} \\ \{ \Sigma \vdash N_i \mapsto y_i \qquad \Gamma_i = \{ y_i : \tau_i \} \} \end{array}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\ x_1 \ldots x_n \rightsquigarrow C'\ y_1 \ldots y_m : \tau \Rightarrow \bigcup \Gamma_i}$$

$$\frac{M \in \Sigma}{\Sigma \vdash M \mapsto \Sigma\,(M)} \qquad \frac{M \notin \Sigma \qquad x \text{ fresh}}{\Sigma \mapsto M \mapsto x}$$

Figure 13: Interpretation of pattern transformation syntax

This interpretation makes transformations specified by the pattern transformation syntax automatically consistent with the type system. The sole premise of the *PatCon* rule, $\Gamma, \mathrm{K} \vdash C : \overline{\tau_i} \to \tau$ is also a premise of *PatRule*. It is written as $\Gamma, \mathrm{K} \vdash C' : \tau_1 \to \ldots \to \tau_m \to \tau$ to make clear that there must be as many metavariables on the right-hand side as the arity of the constructor.

4 View types

With the pattern transformations of the previous section, we can transform the scrutinee of **case** expressions to an expression of a different type. However, the constructors of this different type must be exposed, otherwise it is not possible to match on them. In this section, we will describe a technique to still do pattern matching on abstract types, which do not expose their constructors, without converting the entire expression back to its original type.

4.1 From $[a]$ to *Seq a*

Many of Haskell's more efficient data structures, such as *Map* for finite maps, *Text* as an alternative to *String*, or *Seq* for finite sequences, have no exposed constructors. Doing so would allow the programmer to break important invariants which cause these data structures to be efficient in the first place. The unavailability of constructors is therefore a good thing. On the other hand, we would still like to transform code from using an inefficient datatype to code using these better alternatives, including programs in which we use pattern matching. As an example, consider the following *safeTail* function with type $[a] \to [a]$, which we would like to transform to a function of type $\mathit{Seq}\ a \to \mathit{Seq}\ a$:

$$\begin{array}{l} \mathit{main} = \lambda l \to \textbf{case}\ l\ \textbf{of} \\ \quad [\,] \quad\ \to [\,] \\ \quad (x : xs) \to xs \end{array}$$

We also have the following expression transformations:

1. $M \rightsquigarrow Data.Sequence.toList\ M$[2]
2. $M \rightsquigarrow Data.Sequence.fromList\ M$
3. $[\,] \rightsquigarrow Data.Sequence.\epsilon$
4. $(:) \rightsquigarrow (Data.Sequence.\triangleleft)$

Without any other transformations, we arrive at the following expression:

$$
\begin{array}{l}
main = \lambda l \rightarrow \textbf{case}\ Data.Sequence.toList\ l\ \textbf{of} \\
\quad [\,] \quad\ \rightarrow Data.Sequence.\epsilon \\
\quad (x : xs) \rightarrow Data.Sequence.fromList\ xs
\end{array}
$$

It has the right type $Seq\ a \rightarrow Seq\ a$, but it requires a conversion to lists, and another conversion back to *Seq*. *Data.Sequence.fromList* is strict in the spine of its argument, which forces a full evaluation of *toList*. This means that this implementation of *safeTail* runs in linear time instead of constant time.

Luckily, the *Data.Sequence* module exports the *ViewL* type and accompanying *viewl* function:

$$
\begin{array}{l}
\textbf{data}\ ViewL\ (a : \star) = EmptyL \\
\qquad\qquad\qquad\qquad\ \mid\ a :< (Seq\ a) \\
viewl : Seq\ a \rightarrow ViewL\ a
\end{array}
$$

Instead of converting a $Seq\ a$ to a $[a]$, we can use *viewl* to convert it to the $ViewL\ a$ type of which the constructors are exposed, and then pattern match on them:

$$
\begin{array}{l}
main = \lambda l \rightarrow \textbf{case}\ Data.Sequence.viewl\ l\ \textbf{of} \\
\quad EmptyL \rightarrow Data.Sequence.\epsilon \\
\quad (x :< xs) \rightarrow xs
\end{array}
$$

The *viewl* function runs in constant time, so this implementation of *safeTail* has the desired complexity. For this transformation, we need the following extra transformation rules:

1. $M \rightsquigarrow Data.Sequence.viewl\ M$
2. $[\,] \rightsquigarrow EmptyL$
3. $(M : N) \rightsquigarrow (M :< N)$

The last two are pattern transformations that transform list patterns to patterns for the $ViewL\ a$ type. We do not introduce a transformation that converts a $ViewL\ a$ back to a $Seq\ a$, because we intend to use $ViewL\ a$ for pattern matching only.

[2] In reality, *Data.Sequence* does not have a *toList* function, but recommends using the *Foldable* typeclass. Since our system does not support typeclasses, we pretend $toList : Seq\ a \rightarrow [a]$ is exported by the *Data.Sequence* module.

4.2 From concrete to abstract types

When transforming expressions from type τ_1 to type τ_2, there are two straightforward methods of dealing with pattern matches: transform the scrutinee back to type τ_1, or transform the patterns to patterns for type τ_2. As seen in the previous section, transforming patterns to type τ_2 is not always possible, and transforming the scrutinee to τ_1 is not always desirable.

In the previous section, we used the *ViewL a* type provided by *Data.Sequence*, which is sufficiently list-like to pattern match on as if it actually is a list, but does not require a full conversion from *Seq a*. In the general case of transforming expressions of concrete type C to expressions of abstract type A, we will refer to a type that is similarly "in-between" C and A as a **view type** $A@C$.

Next to the view type $A@C$, we also need a general version of *viewl*. We will refer to this function as a **view function** *view*, which has type $A \rightarrow A@C$. View types and view functions are introduced by Wadler in [6]

To transform programs, we need transformation rules. The transformation rule for expressions is simple: $M \rightsquigarrow \mathit{view}\ M$. However, we will see that the definition of *view* will have to behave according to some specific rules. Finally, the transformation rules for patterns depend on the structure of both C and $A@C$.

4.2.1 Structure of $A@C$

Because the patterns in the target program will be patterns for $A@C$, we need to know its constructors. To successfully transform all patterns for C that the source program has, we need a pattern transformation that turns a pattern for a constructor of C into a pattern for a constructor of $A@C$. It follows that $A@C$ must have as many constructors as C has.

Additionally, for a constructor C_i of C, every field must map to a field of the corresponding constructor C_i' of $A@C$. This means that patterns for $A@C$ will differ from patterns for C only in constructor names. The difference between C and $A@C$ will be at the recursive positions. Where a constructor of C has a field of type C, the corresponding constructor of $A@C$ will have a field of type A.

When this definition is applied, the type *Seq@List* will look like:

$$\begin{aligned}&\textbf{data}\ \mathit{Seq@List}\ (a : \star) = \mathit{SeqNil}'\\ &\qquad\qquad\qquad\qquad\mid \mathit{SeqCons}'\ a\ (\mathit{Seq}\ a)\end{aligned}$$

This is isomorphic to the *ViewL* datatype. However, we can go one step further. There is nothing *Seq*-specific about the above datatype definition. In fact, the type *Set@List* looks like:

$$\begin{aligned}&\textbf{data}\ \mathit{Set@List}\ (a : \star) = \mathit{SetNil}'\\ &\qquad\qquad\qquad\qquad\mid \mathit{SetCons}'\ a\ (\mathit{Set}\ a)\end{aligned}$$

This suggests defining the view type as $A@C = @C\ A$, where $@C$ is the *base functor* of C. It takes an extra type parameter, which is used wherever the original datatype is used recursively. Now we have:

$$
\begin{aligned}
&\textbf{data}\ C\ \overline{\alpha : \kappa} = \overline{Con\ \overline{\tau}}\ \mid \\
&\quad \rightsquigarrow \\
&\textbf{data}@C\ \overline{\alpha : \kappa}\ (\rho : \star) = \overline{Con'\ \overline{\tau'}}\ \mid \\
&\tau'_{i,j}\ \mid\ \tau_{i,j} \equiv C\ \overline{\alpha} = \rho \\
&\qquad \mid\ otherwise = \tau_{i,j}
\end{aligned}
$$

Figure 14: Construction of $@C$

$$
\begin{aligned}
&\textbf{data}@List\ (a : \star)\ (b : \star) = Nil' \\
&\qquad\qquad\qquad\qquad \mid\ Cons'\ a\ b \\
&\textbf{type}\ Set@List\ (a : \star) = @List\ a\ (Set\ a) \\
&\textbf{type}\ Seq@List\ (a : \star) = @List\ a\ (Seq\ a)
\end{aligned}
$$

A schematic formulation of the construction of the view type can be found in Figure 14. In this figure, the type variables $\overline{\alpha}$ are those that are part of the original datatype, such as the a in $[a]$. For every constructor Con of C, there is a constructor Con' of $@C$.

For every constructor, the amount of fields stays the same. However, if a field $\tau_{i,j}$ is recursive, the corresponding field in $@C$ will have type ρ, which is the type parameter that will be instantiated by the type we wish to view as C.

4.2.2 Definition of *view*

While any definition of $view : A \to A@C$ would do, we want it to behave sensibly in combination with $to : A \to C$ and $from : C \to A$. First, we want $from \circ to : A \to A$ to be the identity. The other way, $to \circ from : C \to C$ is nice to have, but not necessary. For example, for *Set*, $toList \circ fromList$ may not preserve the element order.

Next we look at the view type, specifically at $C@C$. As per the previous section, this is equal to $@C\ C$, the type that is the same as C, except at the recursive positions, where it has the type of its parameter. In this case, it is C, so $@C\ C$ is isomorphic to C. This isomorphism is given by $out : C \to C@C$ and $in_{\_} : C@C \to C$. We require that $out \circ in_{\_} : C@C \to C@C$ and $in_{\_} \circ out : C \to C$ are the identity. Finally, let $fmap_{@C} : (a \to b) \to @C\ a \to @C\ b$. This function can be automatically generated based on the structure of $@C\ a$. For $fmap_{@C}$, we require that the functor laws hold: $fmap_{@C}\ f \circ fmap_{@C}\ g = fmap_{@C}\ (f \circ g)$, and $fmap_{@C}\ id = id$.

The relations between these functions and *view* are displayed in Figure 15. Keep in mind that $to \circ from$ is not necessarily the identity, and thus neither is $fmap_{@C}\ to \circ fmap_{@C}\ from$.

For the diagram to commute, *view* should be equivalent to $fmap_{@C}\ from \circ out \circ to$. As a motivation for this equivalence, we look back at the *safeTail* function from section 4.1, modified to use $toList' = out \circ toList$, which has type $Seq\ a \to @List\ a\ [a]$, instead of *toList* for the transformation without *view*. Furthermore, we consider the type $@List\ a\ b$ to be defined as:

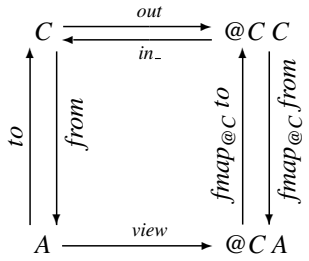

Figure 15: Diagram for *view*

$$\textbf{data}\,@List\ (a:\star)\ (b:\star) = EmptyL \mid (a :< b)$$

Since $@List\ \alpha\ [a]$ is isomorphic to $[a]$, and $@List\ a\ (Seq\ a)$ is isomorphic to the original $ViewL\ \alpha$ datatype, these modifications do not influence the type of the program.

$$\begin{aligned}
&\text{-- Original code. } main : [\alpha] \to [\alpha] \\
&main = \lambda l \to \textbf{case}\ l\ \textbf{of} \\
&\qquad\qquad [\,] \quad\ \to [\,] \\
&\qquad\qquad (x : xs) \to xs \\
&\text{-- Transformation without } viewl.\ main' : Seq\ a \to Seq\ a \\
&main' = \lambda l \to \textbf{case}\ toList'\ l\ \textbf{of} \\
&\qquad\qquad EmptyL \ \to \epsilon \\
&\qquad\qquad (x :< xs) \ \to fromList\ xs \\
&\text{-- Transformation with } viewl.\ main'' : Seq\ a \to Seq\ a \\
&main'' = \lambda l \to \ \textbf{case}\ viewl\ l\ \textbf{of} \\
&\qquad\qquad EmptyL \to \epsilon \\
&\qquad\qquad (x :< xs) \to xs
\end{aligned}$$

We want both transformation results, $main'$ and $main''$, to be the same function. Note in the transformation without *viewl*, *fromList* gets applied to the variable bound to the recursive field of (:<). This is exactly what $fmap_{@List}\ fromList$ accomplishes. This makes the resulting code of the transformation without *viewl* equivalent to:

$$\begin{aligned}
&main''' = \lambda l \to \textbf{case}\ fmap_{@List}\ fromList\ (toList'\ l)\ \textbf{of} \\
&\qquad\qquad EmptyL \to \epsilon \\
&\qquad\qquad (x :< xs) \to xs
\end{aligned}$$

From this, we find that $viewl\ l$ should be equivalent to $fmap_{@List}\ fromList\ (toList'\ l)$, i.e., that *viewl* should be equivalent to $fmap_{@List}\ fromList \circ out \circ toList$, exactly what the diagram suggests.

4.2.3 Pattern transformations

The previous section covered the *view* function used in the expression transformation rule $M \rightsquigarrow view\ M$. We still need to define the pattern transformations.

The pattern transformations will need to transform patterns for C into patterns for $A@C$. In the example that transforms lists to *Seq*, we need the rules $[\,] \rightsquigarrow \mathit{EmptyL}$ and $(M : N) \rightsquigarrow (M :< N)$. In the transformation rule for patterns with the (:) constructor, metavariables appear that have a different type on the left-hand side and the right-hand side. It corresponds to the following inference rule:

$$\frac{\Gamma, \mathrm{K} \vdash (:<) : \alpha \to \mathit{Seq}\ \alpha \to @\mathit{List}\ \alpha\ (\mathit{Seq}\ \alpha)}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} (x : y) \rightsquigarrow (x :< y) : @\mathit{List}\ \alpha\ (\mathit{Seq}\ \alpha) \Rightarrow \{x : \alpha, y : \mathit{Seq}\ \alpha\}}$$

It turns out that the construction of $A@C$ and the corresponding *view* function allow for a relatively simple construction of the pattern transformation rule. Since $A@C$ has a constructor for every constructor of C, and every field of constructor Con' of $A@C$ corresponds to a field of constructor Con of C, the transformation rules will be of the form: $\mathit{Con}\ \overline{M} \rightsquigarrow \mathit{Con}'\ \overline{M}$.

5 Datatype transformations

In the previous sections, we discussed the changes to the transformation system on the expression level, and through **case** expressions also the pattern level. However, the introduction of datatype declarations in addition to a *main* brings additional complexity. Consider the following program, in which a datatype *Bingo* is defined that contains a list of numbers drawn in a game of Bingo. The defined *main* function checks if all numbers of a given list are drawn during a given game. It has type $[\mathit{Int}] \to \mathit{Bingo} \to \mathit{Bool}$.

```
data Bingo = Bingo [Int]
main = λnums bingo →
  case bingo of
    Bingo bingo' →
      foldr (λn b → elem n bingo' ∧ b) True nums
```

We would like to change this code to use *Set Int* instead of $[\mathit{Int}]$. We will use the following transformation rules:[3]

- $M \rightsquigarrow \mathit{fromList}\ M$
- $M \rightsquigarrow \mathit{toList}\ M$
- $\mathit{foldr} \rightsquigarrow \mathit{Data.Set.foldr}$
- $\mathit{elem} \rightsquigarrow \mathit{member}$

Here, $\mathit{fromList} :: [\mathit{Int}] \to \mathit{Set\ Int}$, $\mathit{toList} :: \mathit{Set}\ a \to [a]$, $\mathit{Data.Set.foldr} :: (a \to b \to b) \to b \to \mathit{Set}\ a \to b$, and $\mathit{member} :: \mathit{Int} \to \mathit{Set\ Int} \to \mathit{Bool}$.

The following *main* functions are both valid transformations of the original function:

[3] Since *Set* has an *Ord* constraint, and our language does not support typeclasses, I've specialized the types to *Set Int*

```
data Bingo = Bingo [Int]
main = λnums bingo →
  case bingo of
    Bingo bingo′ →
      Data.Set.foldr (λn b → member n (fromList bingo′) ∧ b) True nums
main′ = λnums bingo →
  case bingo of
    Bingo bingo′ →
      Data.Set.foldr (λn b → elem n bingo′ ∧ b) True nums
```

These are not the only ones, but all valid transformations have one thing in common: *bingo′* has type [*Int*]. This results in using either *fromList* or *elem*, both linear in the length of the list, while *member* is only logarithmic.

The reason we cannot change the type of *bingo′* is because it is bound by a pattern match on the *Bingo* constructor, which has type [*Int*] → *Bingo*. Defining a second datatype **data** *Bingo′* = *Bingo′* (*Set Int*) and a pattern transformation *Bingo M* ⇝ *Bingo′ M* solves the problem, but if every transformation of a datatype requires the definition of an auxiliary datatype, that can get quite verbose. Instead, we will introduce yet another type of transformation: constructor transformations. These are used to transform the datatype definitions directly.

Apart from changing field types, there are other transformations for datatypes that are desirable, like adding or removing fields from a constructor. As usual, we will first show the inference rules for this transformation and how to fit these into the existing system. Then we will introduce a syntax for constructor transformations and its interpretation.

5.1 Constructor transformations

As before, the transformation judgements for constructors will be based on the typing rules (recall Figure 4):

Typing:	$\mathrm{K} \vdash C\ \overline{\tau} \Rightarrow \{C : \overline{\tau \to}\ T\ \overline{\alpha}\}$
Transformation:	$\mathrm{K} \vdash C\ \overline{\tau} \rightsquigarrow C\ \overline{\tau'} \Rightarrow \{C : \overline{\tau' \to}\ T\ \overline{\alpha}\}$

Having a judgement for constructor transformations necessitates a judgement for datatype declaration transformations, as wel as an adapted program transformation judgement. Figure 16 shows the required propagation rules that allow for both constructor and program transformations.

Because they are propagation rules, they only specify the identity transformation. As with expressions and patterns, actual transformations need to be specified separately. For example, the constructor transformation rule for *Bingo* in the previous example looks like the following:

$$\textit{PData}\quad \frac{\mathrm{K}' = \{T : \overline{\kappa_i \to} \star\} \quad \{\mathrm{K} \cup \mathrm{K}' \cup \overline{\alpha : \kappa} \vdash C_i \rightsquigarrow C_i' \Rightarrow \Gamma_i'\}}{\mathrm{K} \vdash \mathbf{data}\ T\ \overline{\alpha : \kappa} = \overline{C} \mid \rightsquigarrow \mathbf{data}\ T\ \overline{\alpha : \kappa} = \overline{C'} \mid \Rightarrow \bigcup \Gamma_i', \mathrm{K}'}$$

$$\textit{PCon}\quad \frac{\{\mathrm{K} \vdash \tau_i : \star\}}{\mathrm{K} \vdash C\ \overline{\tau} \rightsquigarrow C\ \overline{\tau} \Rightarrow \{C : \overline{\tau \to} T\ \overline{\alpha}\}}$$

$$\textit{PProg}\quad \frac{\{\mathrm{K} \vdash D_i \rightsquigarrow D_i' \Rightarrow \Gamma_i, \mathrm{K}_i\} \quad \Gamma \cup \bigcup \Gamma_i, \mathrm{K} \cup \bigcup \mathrm{K}_i \vdash e \rightsquigarrow e' : \tau}{\Gamma, \mathrm{K} \vdash \overline{D}; \mathit{main} = e \rightsquigarrow \overline{D'}; \mathit{main} = e' : \tau}$$

Figure 16: Propagation rules for datatype declarations and programs

$$\frac{\mathrm{K} \vdash \mathit{Set\ Int} : \star}{\mathrm{K} \vdash \mathit{Bingo}\ [\mathit{Int}] \rightsquigarrow \mathit{Bingo}\ (\mathit{Set\ Int}) \Rightarrow \{\mathit{Bingo} : \mathit{Set\ Int} \to \mathit{Bingo}\}}$$

A single constructor transformation suffices for transformations that change the type of the fields of a constructor. However, when adding, removing, or reordering fields, we will need additional transformations on the pattern and expression level.

When removing a field, the corresponding variable in patterns is removed as well, and is no longer in scope on the rhs of the pattern matching clause. Furthermore, any usage of the constructor as a function on the expression level will have one argument too many.

When adding a field, the pattern for the constructor will gain an extra pattern variable, which is unused. This does not make the program incorrect, but manual changes to actually make use of this variable are needed. However, when used as a function on the expression level, there will be one fewer argument than needed. A suitable default will need to be specified.

Reordering fields is not simply a retyping of the original fields. On the pattern level, the variables will need to be reordered, and on the expression level, arguments to the constructor need to be rearranged as well.

It is clear that constructor transformations have a lot of influence on the rest of the code. In the following sections, we will look at the other transformations that arise from a constructor transformation. First we will cover adding fields and reordering fields. This introduces pattern transformations and expression transformations induced by a constructor transformation. Then we will look at removing fields, which has the additional difficulty of removing bound variables in pattern matches.

5.1.1 Adding and reordering fields

First we will look at our original *Bingo* datatype:

data *Bingo* = *Bingo* [*Int*]

Say we want to add a field of type *Int* that holds the size of the list, in order to avoid recomputation. We want the new datatype to look like:

data *Bingo* = *Bingo* [*Int*] *Int*

A suitable constructor transformation would be:

$$\frac{\mathrm{K} \vdash [\mathit{Int}] : \star \qquad \mathrm{K} \vdash \mathit{Int} : \star}{\mathrm{K} \vdash \mathit{Bingo}\ [\mathit{Int}] \rightsquigarrow \mathit{Bingo}\ [\mathit{Int}]\ \mathit{Int} \Rightarrow \{\mathit{Bingo} : [\mathit{Int}] \to \mathit{Int} \to \mathit{Bingo}\}}$$

For the pattern transformation, we need to introduce a new pattern variable for the *Int* field:

$$\frac{\Gamma, \mathrm{K} \vdash \mathit{Bingo} : [\mathit{Int}] \to \mathit{Int} \to \mathit{Bingo}}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Bingo}\ x \rightsquigarrow \mathit{Bingo}\ x\ \_y : \mathit{Bingo} \Rightarrow \{x : [\mathit{Int}], \_y : \mathit{Int}\}}$$

Here *_y* needs to be a variable that does not occur in the rest of the program. In general, an added field in the constructor must have a corresponding pattern variable.

If we later decide that the order of the fields is not right, and we would like the *Int* to be first, the following two transformations (the first for the constructor, the second for patterns) can be used:

$$\frac{\mathrm{K} \vdash \mathit{Int} : \star \qquad \mathrm{K} \vdash [\mathit{Int}] : \star}{\mathrm{K} \vdash \mathit{Bingo}\ [\mathit{Int}]\ \mathit{Int} \rightsquigarrow \mathit{Bingo}\ \mathit{Int}\ [\mathit{Int}] \Rightarrow \{\mathit{Bingo} : \mathit{Int} \to [\mathit{Int}] \to \mathit{Bingo}\}}$$

$$\frac{\Gamma, \mathrm{K} \vdash \mathit{Bingo} : \mathit{Int} \to [\mathit{Int}] \to \mathit{Bingo}}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Bingo}\ x\ y \rightsquigarrow \mathit{Bingo}\ y\ x : \mathit{Bingo} \Rightarrow \{x : [\mathit{Int}], y : \mathit{Int}\}}$$

Expression transformations are more difficult to define. We could try *Bingo M* $\rightsquigarrow$ *Bingo M* (*length M*) and *Bingo M N* $\rightsquigarrow$ *Bingo N M*. However, these transformations will not occur if *Bingo* appears partially applied. It is thus necessary to define a smart constructor, and transform *Bingo* to that:

```
  -- For adding the integer field.
  -- Expression transformation: Bingo ⇝ smart_Bingo_1
smart_Bingo_1 :: [Int] → Bingo
smart_Bingo_1 lst = Bingo lst (length lst)
  -- For swapping the fields.
  -- Expression transformation: Bingo ⇝ smart_Bingo_2
smart_Bingo_2 :: [Int] → Int → Bingo
smart_Bingo_2 lst int = Bingo int lst
```

In the above code, *length lst* is used because the intention of the *Int* field was to store the length of the list. The definition of the smart constructor needs to be provided by

the user, and must be consistent with the actual constructor transformation. In general, the smart constructor will take as many parameters as the original constructor, and it will be an application of the new constructor to as many arguments as it has fields. The pattern transformation must be specified separately, and must also be consistent. When we introduce the syntax to specify constructor transformation rules in Section 5.2, we will also show how the pattern transformation can be derived from the constructor transformation.

5.1.2 Removing fields

When removing fields, we can still define a smart constructor as before. Suppose we decide that storing the size of the list as a separate field is not worth it after all, and we want to revert back to the version with just a plain list of *Int*s. The constructor transformation and smart constructor will look as follows:

$$\frac{\mathrm{K} \vdash [\mathit{Int}] : \star}{\mathrm{K} \vdash \mathit{Bingo}\ \mathit{Int}\ [\mathit{Int}] \rightsquigarrow \mathit{Bingo}\ [\mathit{Int}] \Rightarrow \{\mathit{Bingo} : [\mathit{Int}] \to \mathit{Bingo}\}}$$

$$\begin{aligned}&\mathit{smart\_Bingo}_3 :: \mathit{Int} \to \mathit{String} \to \mathit{Bingo}\\&\mathit{smart\_Bingo}_3\ \mathit{int}\ \mathit{lst} = \mathit{Bingo}\ \mathit{lst}\end{aligned}$$

The difficulty lies in pattern-matching. While the pattern transformation is easily defined, we run into problems if the removed field is bound to a variable which is used in the code:

$$\begin{aligned}&\textbf{data}\ \mathit{Bingo} = \mathit{Bingo}\ \mathit{Int}\ [\mathit{Int}]\\&\mathit{main} = \lambda \mathit{bingo} \to \textbf{case}\ \mathit{bingo}\ \textbf{of}\\&\quad \mathit{Bingo}\ \mathit{int}\ \mathit{lst} \to \mathit{int}\end{aligned}$$

Here we can not simply remove the *Int* field, as the variable *int* is bound to it. To solve this, we add a set of substitutions to the pattern transformation judgment. These substitutions list, for fields that get removed, what expression they must be replaced by. This expression can make use of the other pattern variables. The rule for this pattern transformation looks like:

$$\frac{\Gamma, \mathrm{K} \vdash \mathit{Bingo} : [\mathit{Int}] \to \mathit{Bingo}}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} \mathit{Bingo}\ x\ y \rightsquigarrow \mathit{Bingo}\ y : \mathit{Bingo} \Rightarrow \{y : [\mathit{Int}]\}, \{x \rightsquigarrow \mathit{length}\ y\}}$$

The set of substitutions here is $\{x \rightsquigarrow \mathit{length}\ y\}$. Since there is only one pattern variable absent in the pattern on the right-hand side, there is only one substitution. In general, the pattern transformation judgements now look like: $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p \rightsquigarrow p : \tau \Rightarrow \Gamma, \Delta$. In this judgement, Δ is a mapping from pattern variables x to expressions e. Pattern variables of the pattern on the left-hand side must either occur in the pattern on the right-hand side, or have a corresponding mapping in Δ.

$$PCase \quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \qquad \{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i \rightsquigarrow p_i' : \tau_1 \Rightarrow \Gamma_i, \Delta_i \qquad \Gamma \cup \Gamma_i, \mathrm{K} \vdash \Delta_i\,(e_i) \rightsquigarrow e_i' : \tau_2\}}{\Gamma, \mathrm{K} \vdash \textbf{case}\ e\ \textbf{of}\ \{\overline{p_i \to e_i;}\} \rightsquigarrow \textbf{case}\ e'\ \textbf{of}\ \{\overline{p_i' \to e_i';}\} : \tau_2}$$

Figure 17: Modified case propagation rule

CONTRF	::=	$C\ \overline{M} \rightsquigarrow C\ \overline{\hat{M}} \Rightarrow \{\overline{M \mapsto e_M}\}$	constructor transformation
$\hat{M}$	::=	M	
	|	$M : \tau$	Metavariable with type annotation

Figure 18: Constructor transformation syntax

Changing the pattern transformation judgement means we have to change the identity transformation for patterns, as well as the *PCase* inference rule. The new version of the latter needs to take the set of substitutions into account (Figure 17)

The most important change is that we no longer have $e_i \rightsquigarrow e_i'$, but $\Delta_i\,(e_i) \rightsquigarrow e_i'$. This performs the substitution of removed pattern variables, before propagating the transformation. The substitution is a straightforward bottom-up scope-aware substitution, replacing variables that occur in Δ with their associated expression.

The identity transformation remains largely the same, except that it needs to produce a set of substitutions. Since there are no removed pattern variables, this set will be empty.

Lastly, due to the inclusion of the mapping Δ, the pattern transformation syntax must provide a way for that, too. Pattern transformations are written as. $C\ \overline{M} \rightsquigarrow C\ \overline{M} \Rightarrow \{\overline{M \mapsto e_M}\}$. When interpreting the syntax, the metavariables in $\{\overline{M \mapsto e_M}\}$ get instantiated to object variables to form the mapping Δ. For a formal specification, see the Appendix A.

5.2 Constructor transformation syntax

In order to reduce the verbosity of the inference rules, we also introduce a special syntax for constructor transformations. Additionally, the syntax allows us to derive a pattern transformation that corresponds to the constructor transformation.

The syntax, described in Figure 18, is almost the same as the syntax for pattern transformations. The difference is that any new metavariables on the right-hand side need a type annotation for use in the constructor definition. In case of type-changing fields, metavariables on the right-hand side that also appear on the left-hand side need a type annotation as well. For example, the first transformation described in this chapter, changing a field of type $[Int]$ to a field of type *Set Int*, is described as $Bingo\ M \rightsquigarrow Bingo\ (M : Set\ Int) \Rightarrow \{\}$.

The derivation of a pattern transformation from a constructor transformation is done by simply untagging the metavariables on the right-hand side:

$$\textit{ConRule}\quad \frac{\begin{array}{c} C\ \overline{M} \rightsquigarrow C'\ \overline{\hat{N}} \Rightarrow \Delta \text{ is a constructor transformation} \\ \Sigma = \{M_i \mapsto \tau_i\} \\ \{\Sigma \vdash \hat{N}_i \mapsto \tau_i' \qquad \mathrm{K} \vdash \tau_i' : \star\} \end{array}}{\mathrm{K} \vdash C\ \overline{\tau} \rightsquigarrow C'\ \overline{\tau'} \Rightarrow \{C' : \overline{\tau \to}\ T\ \overline{\alpha}\}}$$

$$\frac{}{\Sigma \vdash M : \tau \mapsto \tau} \qquad \frac{M \in \Sigma}{\Sigma \vdash M \mapsto \Sigma\,(M)}$$

Figure 19: Interpretation of constructor transformation syntax

$$\frac{\begin{array}{c} C\ \overline{M} \rightsquigarrow C'\ \overline{\hat{N}} \Rightarrow \Delta \text{ is a constructor transformation} \\ \{M_i' = \mathit{untag}\ (M_i)\} \end{array}}{C\ \overline{M'} \rightsquigarrow C'\ \overline{N} \Rightarrow \Delta \text{ is a pattern transformation}}$$

$\mathit{untag}\ (M) = M$
$\mathit{untag}\ (M : \tau) = M$

The interpretation of the constructor transformation syntax is described in Figure 19. The metavariables on the left-hand side are mapped to the types in the constructor to result in a mapping Σ. Next, the metavariables on the right-hand side are looked up. However, when the metavariable has a type annotation, that annotation will be the result type, not even if Σ maps it to a different type.

6 Object language extensions

Now that we have explored the types of transformations applicable to our minimal language, we will look at some advanced features found in Haskell, that may change the way our transformations work. Specifically, we will lift the restriction that patterns cannot be nested, and look at two attempts at solving the problems that result from lifting this restriction. However, both attempts introduce complexities of their own.

6.1 Nested patterns

In the object language described in this thesis, patterns can not be nested: a constructor pattern can only contain pattern variables, not more constructor patterns. This keeps the language and transformations simple, but with it, we lose some expressive power. For example, the following piece of code will become more verbose, and require either code duplication or defining an auxiliary function to minimize the duplication to a function call:

```
  -- with nested patterns
main = λx → case x of
```

$$
\begin{array}{lcll}
p & ::= & x & \text{pattern variable} \\
 & | & C\,\overline{p} & \text{constructor pattern}
\end{array}
$$

$$
\textit{PatVar}\quad \frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\text{pat}} x : \tau \Rightarrow \{x : \tau\}} \qquad \frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\text{pat}} x \rightsquigarrow x : \tau \Rightarrow \{x : \tau\}, \epsilon}
$$

$$
\textit{PatCon}\quad \frac{\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau \quad \{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i : \tau_i \Rightarrow \Gamma_i\}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\,p_1 \ldots p_n : \tau \Rightarrow \bigcup \Gamma_i} \qquad \frac{\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau \quad \{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i \rightsquigarrow p_i' : \tau_i \Rightarrow \Gamma_i, \Delta_i\}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\,p_1 \ldots p_n \rightsquigarrow C\,p_1' \ldots p_n' : \tau \Rightarrow \bigcup \Gamma_i, \bigcup \Delta_i}
$$

Figure 20: Nested patterns. Syntax, typing, and propagation.

```
                (y : [ ]) → "One element"
                t         → "Not one element"
  -- without nested patterns, duplication
main = λx → case x of
  (x : xs) → case xs of
               [ ] → "One element"
               t   → "Not one element"
  t        → "Not one element"
```

To allow nested patterns, the syntax, typing, and propagation subtly change, compare Figure 20 to Figure 3 and Figure 10. Typing a constructor pattern first recursively types the sub-patterns, and then checks those types against the types of the fields of the constructor. For propagation, the sub-patterns are first transformed independently, and then typechecked against the constructor.

With the appropriate transformation rules, we can now perform the following *List* to *Seq* transformation:

```
main = λe → case e of
              (1 : xs) → xs
              x        → [ ]
main' = λe → case viewl e of
               (1 :< xs) → xs
               x         → ϵ
```

However, the following will still fail:

```
main = λe → case e of
              (1 : 2 : xs) → xs
              x            → [ ]
main' = λe → case viewl e of
               (1 :< ...) → ...
               x          → ϵ
```

The pattern on the … needs to be a pattern for *Seq Int*, which is abstract. In this particular case, we could write a two-layer view function:

$$
\begin{array}{l}
\mathit{view2}\ e = \textbf{case}\ \mathit{viewl}\ e\ \textbf{of} \\
\qquad \begin{array}{l}
\mathit{EmptyL} \rightarrow \mathit{Nothing} \\
(x :< xs) \rightarrow \mathit{Just}\ (x, \mathit{viewl}\ xs)
\end{array}
\end{array}
$$

This function would allow us to transform the program into:

$$
\begin{array}{l}
\mathit{main}' = \lambda e \rightarrow \textbf{case}\ \mathit{view2}\ e\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Just}\ (1, 2 :< xs) & \rightarrow xs \\
x & \rightarrow \epsilon
\end{array}
\end{array}
$$

However, this would mean that we have to define a different view function and view type for every used nesting of patterns. This becomes very cumbersome: we can go down the spine of the list as often as we want, and we could have lists of a more structured datatype than *Int*, and pattern match on those values. If we want to allow nested patterns, and also allow abstract datatypes, we need a better way than view functions to cover arbitrary nesting.

6.2 Guards

A feature in Haskell that is closely related to pattern matching is guards. A guard allows you to annotate a pattern with an expression of type *Bool*, possibly making use of the variables bound by the pattern, followed by the result. If the pattern matches and the boolean expression is *True*, the result of the **case**-expression is the body of the clause. If the boolean expression is *False*, the next clause is tried. For example[4]:

$$
\begin{array}{l}
\mathit{main} = \lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad \begin{array}{lll}
(x : xs) \mid \mathit{odd}\ x & \rightarrow & \texttt{"odd head"} \\
(x : xs) \mid \mathit{otherwise} & \rightarrow & \texttt{"even head"} \\
[\,] & \rightarrow & \texttt{"no head"}
\end{array}
\end{array}
$$

Guards introduce expressions to the pattern language, which complicates the transformations. When we specify a transformation for an unguarded pattern, we want this transformation to work on guarded patterns as well. Transforming the above code from *List* to *Seq* should turn it into:

$$
\begin{array}{l}
\mathit{main} = \lambda e \rightarrow \textbf{case}\ \mathit{viewl}\ e\ \textbf{of} \\
\qquad \begin{array}{lll}
(x :< xs) \mid \mathit{odd}\ x & \rightarrow & \texttt{"odd head"} \\
(x :< xs) \mid \mathit{otherwise} & \rightarrow & \texttt{"even head"} \\
\mathit{EmptyL} & \rightarrow & \texttt{"no head"}
\end{array}
\end{array}
$$

Furthermore, if any of the pattern variables changes type, any guards that use this variable must transform as well, otherwise the entire transformation fails. We want the following transformation to be valid:

$$
\begin{array}{l}
\mathit{main} = \lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad (x : xs) \mid \mathit{odd}\ (\mathit{length}\ xs) \rightarrow \texttt{"odd tail"}
\end{array}
$$

[4] In Haskell, multiple guards can be attached to one pattern. Having multiple guards on one pattern has the same semantics as repeating the pattern multiple times, with just one guard per copy.

$$
\begin{array}{l}
\qquad\qquad (x : xs) \mid otherwise \quad \rightarrow \texttt{"even tail"} \\
\qquad\qquad [\,] \qquad\qquad\qquad\quad \rightarrow \texttt{"no tail"} \\
\rightsquigarrow \\
main = \lambda e \rightarrow \textbf{case}\ viewl\ e\ \textbf{of} \\
\qquad\qquad (x :< xs) \mid odd\ (Seq.length\ xs) \rightarrow \texttt{"odd tail"} \\
\qquad\qquad (x :< xs) \mid otherwise \qquad\quad \rightarrow \texttt{"even tail"} \\
\qquad\qquad [\,] \qquad\qquad\qquad\qquad\quad \rightarrow \texttt{"no tail"}
\end{array}
$$

Thirdly, it should be possible to transform unguarded patterns to guarded patterns. This will help pattern matching on abstract datatypes, as we split the pattern into a guard that checks for an equivalent of the constructor being matched, and projection functions to get at the equivalent of the fields of the constructor. For example, as an alternative to using *viewl*, the following could be a valid transformation:

$$
\begin{array}{l}
main = \lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad (x : xs) \rightarrow Just\ (x, xs) \\
\qquad\qquad [\,] \quad\ \rightarrow Nothing \\
\rightsquigarrow \\
main = \lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad s \mid \neg\ (Seq.null\ s) \rightarrow Just\ (Seq.index\ s\ 0, Seq.drop\ 1\ s) \\
\qquad\qquad s \mid Seq.null\ s \quad\ \rightarrow Nothing
\end{array}
$$

And finally, this should also work for patterns that have guards already:

$$
\begin{array}{l}
main = \lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad (x : xs) \mid odd\ x \rightarrow Just\ xs \\
\qquad\qquad l \qquad\qquad\quad \rightarrow Nothing \\
\rightsquigarrow \\
main = \lambda e \rightarrow \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad s \mid \neg\ (Seq.null\ s) \wedge odd\ (Seq.index\ s\ 0) \rightarrow Just\ (Seq.drop\ 1\ s) \\
\qquad\qquad l \qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow Nothing
\end{array}
$$

To define these transformations, we first extend our object language:

gp	::=	p	pattern
	|	$p \mid e$	guarded pattern
e	::=	...	
	|	**case** e **of** $\{\overline{gp \rightarrow e;}\}$	case expression for patterns with guards

To allow all of the earlier mentioned transformations, we split up pattern transformation judgements in two similar judgement types. First there are judgements for transformations from guarded pattern to guarded pattern: $\Gamma, \mathrm{K} \vdash_{\mathrm{gpat}} gp \rightsquigarrow gp : \tau \Rightarrow \Gamma, \Delta$. The propagation rule for **case** expressions will use rules with that shape for its patterns.

The pattern transformations themselves look like $\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p \rightsquigarrow gp : \tau \Rightarrow \Gamma, \Delta$. Note the guarded pattern on the right-hand side, instead of an unguarded pattern. Figure 21 shows the propagation rules for **case** expressions with guarded patterns.

The *UnguardPat* and *GuardPat* rules are used to convert pattern transformations (with $\vdash_{\mathrm{pat}}$) into guarded pattern transformations (with $\vdash_{\mathrm{gpat}}$). *UnguardPat* states that a pattern

$$\textit{UnguardPat}\quad \frac{\Gamma, \mathrm{K} \vdash_{\text{pat}} p \rightsquigarrow gp' : \tau \Rightarrow \Gamma', \Delta}{\Gamma, \mathrm{K} \vdash_{\text{gpat}} p \rightsquigarrow gp' : \tau \Rightarrow \Gamma', \Delta}$$

$$\textit{GuardPat}\quad \frac{\Gamma, \mathrm{K} \vdash_{\text{pat}} p \rightsquigarrow gp' : \tau \Rightarrow \Gamma', \Delta \quad \Gamma \cup \Gamma', \mathrm{K} \vdash \Delta\,(e) \rightsquigarrow e' : \mathit{Bool}}{\Gamma, \mathrm{K} \vdash_{\text{gpat}} p \mid e \rightsquigarrow gp' \,\&\, e' : \tau \Rightarrow \Gamma', \Delta}$$

$$\textit{PCase}\quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \quad \{\Gamma, \mathrm{K} \vdash_{\text{gpat}} gp_i \rightsquigarrow gp_i' : \tau_1 \Rightarrow \Gamma_i, \Delta_i \qquad \Gamma \cup \Gamma_i, \mathrm{K} \vdash \Delta_i\,(e_i) \rightsquigarrow e_i' : \tau_2\}}{\Gamma, \mathrm{K} \vdash \textbf{case}\ e\ \textbf{of}\ \{\overline{gp_i \to e_i;}\} \rightsquigarrow \textbf{case}\ e'\ \textbf{of}\ \{\overline{gp_i' \to e_i';}\} : \tau_2}$$

$$\textit{PatVar}\quad \frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\text{pat}} x \rightsquigarrow x : \tau \Rightarrow \{x : \tau\}, \epsilon}$$

$$\textit{PatCon}\quad \frac{\Gamma, \mathrm{K} \vdash C' : \overline{\tau \to}\ \tau_r \quad \{\Gamma, \mathrm{K} \vdash_{\text{pat}} p_i \rightsquigarrow p_i' \mid g_i : \tau_i \Rightarrow \Gamma_i, \Delta_i\}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\ \overline{p} \rightsquigarrow C'\ \overline{p'} \,\&\, \overline{g_i \&} : \tau_r \Rightarrow \bigcup \Gamma', \bigcup \Delta'}$$

Figure 21: Inference rules for patterns with guards

transformation can directly be used on a pattern without guard. *GuardPat* takes care of the guard that may already be present. Like the body of the clause, the substitution Δ is first applied to it, as it may contain variables that have been removed by the pattern transformation. Then it is transformed according to the transformation rules for expressions, to account for possible type changes in the variables that are still bound. The resulting expression gets appended to the guarded pattern that is the result of the transformation. & is a helper function that appends a guard to a guarded pattern, and defined as follows:

$$\begin{aligned} (p \mid e_1) \,\&\, e_2 &= p \mid (e_1 \wedge e_2) \\ p \qquad \,\&\, e &= p \mid e \end{aligned}$$

The order here matters. e_2 may make use of projection functions that may not be total, but truth of e_1 may imply that these projection functions are safe to use. For example: *Just x* | *odd x* may be transformed into *y* | *isJust y* ∧ *odd* (*fromJust y*). Here *fromJust y* is undefined if *isJust y* is *False*.

PatVar and *PatCon* are the propagation rules for the patterns. Variable transformations are straightforward. Transformations for constructor patterns need to take into account that there are nested patterns that may be transformed to guarded patterns. When the subpatterns are transformed, their guards are collected and appended to the transformed pattern. If a transformed subpattern does not have a guard *True* is implied.

Having defined the propagation transformations, we will look at some list to *Seq* transformations.

$$\textit{PatNil}\quad \frac{y \text{ fresh} \qquad \Gamma' = \{y : \textit{Seq}\ a\} \qquad \Gamma \cup \Gamma', \mathrm{K} \vdash \textit{Seq.null}\ y : \textit{Bool}}{\Gamma, \mathrm{K} \vdash_{\text{gpat}} [\,] \rightsquigarrow y \mid \textit{Seq.null}\ y : \textit{Seq}\ a \Rightarrow \Gamma', \epsilon}$$

$$\textit{PatCons}\quad \frac{\begin{array}{c}\Gamma, \mathrm{K} \vdash_{\text{pat}} p_1 \rightsquigarrow x \mid e_1 \Rightarrow \Gamma_1, \Delta_1 \\ \Gamma, \mathrm{K} \vdash_{\text{pat}} p_2 \rightsquigarrow y \mid e_2 \Rightarrow \Gamma_2, \Delta_2 \\ y \text{ fresh} \qquad \Gamma' = \{z : \textit{Seq}\ a\} \\ \Delta = \{x \mapsto \textit{Seq.index}\ z\ 0, y \mapsto \textit{Seq.drop}\ z\ 1\} \\ \Gamma \cup \Gamma', \mathrm{K} \vdash \neg\ (\textit{Seq.null}\ z) : \textit{Bool}\end{array}}{\begin{array}{l}\Gamma, \mathrm{K} \vdash_{\text{pat}} (p_1 : p_2) \rightsquigarrow z \mid \neg\ (\textit{Seq.null}\ z) \wedge \Delta\ (e_1) \wedge \Delta\ (e_2) : \textit{Seq}\ a \Rightarrow \\ \qquad F\ \{z : \textit{Seq}\ a\}, \Delta \cup \Delta\ (\Delta_1) \cup \Delta\ (\Delta_2)\end{array}}$$

The *PatNil* rule rewrites a pattern for [ ] to a fresh variable of type *Seq a*, and adds a guard that checks if the sequence is empty. *PatCons* is more involved. The substitution environment Δ is applied not only to the guards of the subpatterns e_1 and e_2, but also to the substitution environments of the subpattern transformations Δ_1 and Δ_2, as they may use x and y, which are bound by patterns p_1 and p_2.

These transformations can be used to handle subpatterns on the tail side of a list. The transformation $\textit{main} \rightsquigarrow \textit{main}'$ is valid:

$$\begin{array}{l}\textit{main} = \lambda s \to \textbf{case}\ s\ \textbf{of} \\ \qquad\qquad (x : y : z) \mid \textit{odd}\ y \to 5 \\ \qquad\qquad x \qquad\qquad\qquad\quad \to 6 \\ \textit{main}' = \lambda s \to \textbf{case}\ s\ \textbf{of} \\ \qquad\qquad a \mid \neg\ (\textit{Seq.null}\ a) \\ \qquad\qquad\qquad \wedge \neg\ (\textit{Seq.null}\ (\textit{Seq.drop}\ 1\ a)) \\ \qquad\qquad\qquad \wedge \textit{odd}\ (\textit{Seq.index}\ (\textit{Seq.drop}\ 1\ a)\ 0) \to 5 \\ \qquad\qquad x \qquad\qquad\qquad\qquad\qquad\qquad\quad \to 6\end{array}$$

The result is code that not only looks ugly, but there is another downside: if either of the transformed subpatterns is a constructor pattern, the transformation becomes invalid. The *PatCons* rule relies on the subpatterns transforming into a variable pattern with a guard, as it completely ignores Γ_1 and Γ_2. As long as the result of the subpattern transformations are variables, they will be substituted away by Δ, and we do not need their typing information.

A solution would be to transform every pattern to a combination of a guard and projection functions. However, the size of the resulting guard increases at least quadratically in the amount of nestings of the pattern. Every pattern layer introduces another term to the guard that corresponds to a check for the right constructor. Furthermore, an additional projection function is inserted into all other terms. For example, the pattern [*Just x*] is transformed into the guarded pattern $z \mid \neg\ (\textit{null}\ z) \wedge \textit{isJust}\ (\textit{head}\ z) \wedge \textit{null}\ (\textit{tail}\ z)$, and the expression corresponding to x is *fromJust* (*head z*).

Not only is this code unpleasant to deal with for the programmer, we also lose any exhaustiveness checks that we may have been able to do with actual patterns. While guards certainly do have their place as alternative to nested if-then-else, but because they are a different syntactic category than patterns, they are not really suitable as replacement for patterns.

$$\textit{PatView}\quad \frac{\Gamma, \mathrm{K} \vdash e : \tau_1 \to \tau_2 \qquad \Gamma, \mathrm{K} \vdash_{\text{pat}} p : \tau_2 \Rightarrow \Gamma'}{\Gamma, \mathrm{K} \vdash_{\text{pat}} (e \to p) : \tau_1 \Rightarrow \Gamma'}$$

$$\frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \to \tau_2 \qquad \Gamma, \mathrm{K} \vdash_{\text{pat}} p \rightsquigarrow p' : \tau_2 \Rightarrow \Gamma', \Delta'}{\Gamma, \mathrm{K} \vdash_{\text{pat}} (e \to p) \rightsquigarrow (e' \to p') : \tau_1 \Rightarrow \Gamma', \Delta'}$$

$$\textit{PatCon}\quad \frac{\begin{array}{c}\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau \\ \Gamma_0 = \Gamma \\ \{\Gamma_{i-1}, \mathrm{K} \vdash_{\text{pat}} p_i : \tau_i \Rightarrow \Gamma_i' \qquad \Gamma_i = \Gamma_{i-1} \cup \Gamma_i'\}\end{array}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\, p_1 \ldots p_n : \tau \Rightarrow \bigcup \Gamma_i'}$$

$$\frac{\begin{array}{c}\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau \\ \Gamma_0 = \Gamma \qquad \Delta_0 = \epsilon \\ \{\Gamma_{i-1}, \mathrm{K} \vdash_{\text{pat}} \Delta_{i-1}\ (p_i) \rightsquigarrow p_i' : \tau_i \Rightarrow \Gamma_i', \Delta_i' \\ \Gamma_i = \Gamma_{i-1} \cup \Gamma_i' \qquad \Delta_i = \Delta_{i-1} \cup \Delta_i'\}\end{array}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\, p_1 \ldots p_n \rightsquigarrow C\, p_1' \ldots p_n' : \tau \Rightarrow \bigcup \Gamma_i', \bigcup \Delta_i'}$$

Figure 22: Typing and propagation for view patterns

6.3 View patterns

Since version 6.10.1 [1] GHC has the *ViewPatterns* extension. View patterns are based on the theory of views and view functions as introduced by Wadler in [6]. Instead of applying a view function to the scrutinee of a **case** expression as we did in Section 4, this language extension makes view functions part of the pattern language. As opposed to guards, this extension directly adds another form of pattern, which allows it to be nested. The new production is:

$$\begin{array}{lcll} p & ::= & \ldots & \\ & | & (e \to p) & \text{view pattern} \end{array}$$

Here, e is a view function, such as *viewl* of *Data.Sequence*. When a scrutinee e_1 is matched against a pattern $e_2 \to p$, $e_2\ e_1$ is matched against p. Typing and propagation rules can be found in Figure 22. Because view functions can make use of variables bound in "earlier" patterns, the *PatCon* rules have changed as well. A version of Γ needs to be threaded through the patterns when typechecking. In the propagation rules, Δ needs to be similarly threaded, and applied to subpatterns, as the view function may make use of variables in earlier patterns that may have been removed.

The increase in expressivity comes from the fact that $e \to p$ is itself a pattern, and can thus be used inside a constructor pattern. Instead of writing a view function for looking at the first element of a *Seq a*, and a different view function for looking at the first two elements, we can use a view pattern in the recursive position. Continuing with the example at the end of Section 6.1:

$$\begin{array}{l} \mathit{main} = \lambda e \to \textbf{case}\ e\ \textbf{of} \\ \qquad\qquad (\mathit{viewl} \to (1 :< (\mathit{viewl} \to 2 :< \mathit{xs}))) \to \mathit{xs} \\ \qquad\qquad x \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \epsilon \end{array}$$

With the addition of view patterns, the techniques of Section 4 are more widely applicable, because these can safely be nested. They also remove the need for expression transformations like $M \rightsquigarrow \mathit{view}\ M$ that are meant to be used with the scrutinee. Instead, all pattern transformations will result in a view pattern with said view function. Where pattern transformations took on the form of $\mathit{Con}\ \overline{M} \rightsquigarrow \mathit{Con}'\ \overline{M}$, with *Con* is a constructor of the original datatype and Con' the corresponding constructor of the view type, pattern transformations that use view patterns will look like $\mathit{Con}\ \overline{M} \rightsquigarrow (\mathit{view} \rightarrow \mathit{Con}'\ \overline{M})$. As an example, the pattern transformations of lists to *Seq*s look like $[\,] \rightsquigarrow (\mathit{viewl} \rightarrow \mathit{EmptyL})$ and $(M : N) \rightsquigarrow (\mathit{viewl} \rightarrow (M :< N))$.

These two transformations are the only two needed to transform the code in Section 6.1 to the code shown above. The pattern $[\mathit{Just}\ x]$, which resulted in barely readable code when using guards and projection functions, will be transformed into $\mathit{viewl} \rightarrow (\mathit{Just}\ x :< (\mathit{viewl} \rightarrow \mathit{EmptyL}))$.

7 Algorithm

The previous sections dealt with the theoretical foundations of our type and transform system. However, when putting it in practice, there is one glaring issue briefly mentioned at the end of Section 2.2. The combination of rules $M \rightsquigarrow \mathit{fromList}\ M$ and $M \rightsquigarrow \mathit{toList}\ M$ allows for potentially infinite chains of *fromList* (*toList* (*fromList*…)).

We restrict the system by allowing only one transformation rule to be applied to an expression. Rules are applied bottom-up. To enforce this, we split up the transformation judgement $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e : \tau$ into $\Gamma, \mathrm{K} \vdash_{\mathrm{pro}} e \rightsquigarrow e : \tau$, $\Gamma, \mathrm{K} \vdash_{\mathrm{tra}} e \rightsquigarrow e : \tau$, and $\Gamma, \mathrm{K} \vdash_{\mathrm{rule}} e \rightsquigarrow e : \tau$.

Judgements using $\vdash_{\mathrm{pro}}$ are used for the propagation relation as described in Figure 7. $\vdash_{\mathrm{rule}}$ judgements are the result of transformation rules as described in 2.3. $\vdash_{\mathrm{tra}}$ is the result expression, after both propagating and applying a rule. It has only one inference rule:

$$\frac{\begin{array}{c}\Gamma, \mathrm{K} \vdash_{\mathrm{pro}} e \rightsquigarrow e' : \tau \\ \Gamma, \mathrm{K} \vdash_{\mathrm{rule}} e' \rightsquigarrow e'' : \tau'\end{array}}{\Gamma, \mathrm{K} \vdash_{\mathrm{tra}} e \rightsquigarrow e'' : \tau'}$$

All premises of the propagation relation of shape $\Gamma, \mathrm{K} \vdash e \rightsquigarrow e : \tau$ are to be changed to $\Gamma, \mathrm{K} \vdash_{\mathrm{tra}} e \rightsquigarrow e : \tau$, while those premises of the transformation rules, which correspond to metavariables in the syntax, are removed entirely. The subexpressions they are about have been transformed by the $\vdash_{\mathrm{pro}}$ premise of $\vdash_{\mathrm{tra}}$.

As a result of this layering, and requiring that partially transformed expressions are type-correct per $\vdash_{\mathrm{pro}}$, transformation rules other than $M \rightsquigarrow \pi$ or $x \rightsquigarrow e$ become less effective. To see why, consider the transformation rule $(+\!+)\ M\ N \rightsquigarrow (\bowtie)\ M\ N$ in combination with $M \rightsquigarrow \mathit{toList}\ M$ and $M \rightsquigarrow \mathit{fromList}\ M$. Here, $(+\!+) : \forall\, \alpha.[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$ is standard list concatenation, and $(\bowtie) : \forall\, \alpha.\mathit{Seq}\ \alpha \rightarrow \mathit{Seq}\ \alpha \rightarrow \mathit{Seq}\ \alpha$ is sequence concatenation.

Now, for an expression such as $(+\!+)\ [1,2,3]\ [4,5,6]$, we cannot derive the expected $(\bowtie)$ (*fromList* $[1,2,3]$) (*fromList* $[4,5,6]$) from the $\vdash_{\text{tra}}$ rule and the transformation rules listed above. The $\vdash_{\text{pro}}$ premise of the $\vdash_{\text{tra}}$ rule requires its result to be well-typed before it the transformation rule is applied. While propagation causes $[1,2,3]$ and $[4,5,6]$ to be transformed to *fromList* $[1,2,3]$ and *fromList* $[4,5,6]$, applying $(+\!+)$ to those results in a type error. The only valid result of the propagation step is $(+\!+)\ [1,2,3]\ [4,5,6]$, after which applying the transformation rule results in a type error.

8 Conclusion

In this thesis, we have extended the type and transform system of Leather et al. [5]. Before this project, the system could transform expressions from the lambda calculus with **let**-bindings and a built-in *fix* operator. To that, we have added the following functionality:

1. Handling object languages with pattern matching and **case**-expressions
2. Transformations of patterns
3. Handling object languages with datatype declarations
4. A syntax to succinctly describe transformations

Pattern matching with **case** expressions is a feature commonly seen in functional programming languages such as Haskell or Scala. It allows breaking down of data while providing better static guarantees about totality than projection functions and checks for constructors.

While it is possible to transform the scrutinee of a **case**-expression to an expression whose type matches the type of the patterns by using appropriate *from* and *to* functions, we can sometimes improve on this by transforming the patterns themselves. This can result in a faster runtime than with conversion functions around the scrutinee and alternatives.

In functional programming languages, datatypes are used to model the part of the world that is relevant to the program. This allows the programmer to define functions on programmer-defined types (e.g. Students or Sets). Because datatypes are so essential to functional programming languages it is important that transformations of functional programs can handle them. Enabling programmer-defined datatypes in program transformations is therefore one of the most important contributions of this research.

Transformations are normally described using inference rules that directly fit in with the inference rules of the type and transform system. Because these are verbose, we invented a shorter notation that, with an initial type environment, can be automatically translated to these inference rules. The syntax has been kept minimal to minimize type-incorrect transformations.

With these improvements on the original system, we can express transformations of programs to use datatypes that are better suited to the algorithm, such as *Seq*s instead of lists in Haskell.

8.1 Related work

In this section we compare some existing work on program transformation to our type and transform systems.

Like type and transform systems, Strafunski [4] is based on rewrite rules, and has a library of strategies for combining rewrite rules and traversing a datatype. However, the transformations that are allowed either do not change types, or change all types to one specified type, as opposed to the more controlled type changes of type and transform systems.

Erwig and Ren's update calculus [2] allows some form of type-changing transformations, but these type changes do not propagate through bound variables. On the other hand, the update calculus has more flexibility in dealing with scope.

One of the shortcomings in the system described above is that there are no guarantees of preservation of semantics. The intention is that the types that are transformed are in some way related, and that there are conversion functions both ways that are inverses. For every function that gets transformed, the target function should be "semantically equivalent" to the source function, but our system does not enforce this. Leather et al. [5] define a type and transform system that has such semantic guarantees.

References

[1] Ghc manual, section 7.3.5 `http://www.haskell.org/ghc/docs/latest/html/users_guide/syntax-extns.html#view-patterns`.

[2] Martin Erwig and Deling Ren. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67(2-3):199–222, 2007.

[3] R J M Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[4] Ralf Lämmel and Joost Visser. Typed combinators for generic traversal. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 137–154. Springer-Verlag, 2002.

[5] Sean Leather, Johan Jeuring, Andres Lh, and Bram Schuur. Type-and-transform systems. Technical Report UU-CS-2012-004, Department of Information and Computing Sciences, Utrecht University, 2012.

[6] 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.

A Complete type and transform system

A.1 Propagation rules

$$PVar \quad \frac{x : \forall\, \overline{\alpha : \kappa}.\tau \in \Gamma \qquad \tau' = [\tau_i \,/\, \alpha_i]\,\tau \qquad \{\mathrm{K} \vdash \tau_i : \kappa_i\}}{\Gamma, \mathrm{K} \vdash x \rightsquigarrow x : \tau'}$$

$$PApp \quad \frac{\Gamma, \mathrm{K} \vdash e_1 \rightsquigarrow e_1' : \tau_2 \to \tau \qquad \Gamma, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma, \mathrm{K} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau}$$

$$PAbs \quad \frac{\mathrm{K} \vdash \tau_1 : \star \qquad \Gamma \cup \{x : \tau_1\}, \mathrm{K} \vdash e \rightsquigarrow e' : \tau}{\Gamma, \mathrm{K} \vdash \lambda x \to e \rightsquigarrow e' : \tau_1 \to \tau}$$

$$PLet \quad \frac{\Gamma, \mathrm{K} \cup \overline{\alpha : \kappa} \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \cup \{x : \forall\, \overline{\alpha : \kappa}.\tau_1\}, \mathrm{K} \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad \overline{\alpha} \notin FV\,(\Gamma)}{\Gamma, \mathrm{K} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2' : \tau_2}$$

$$PFix \quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau \to \tau}{\Gamma, \mathrm{K} \vdash \mathit{fix}\ e \rightsquigarrow \mathit{fix}\ e' : \tau}$$

$$PCase \quad \frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau_1 \qquad \{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} p_i \rightsquigarrow p_i' : \tau_1 \Rightarrow \Gamma_i, \Delta_i \qquad \Gamma \cup \Gamma_i, \mathrm{K} \vdash \Delta_i\,(e_i) \rightsquigarrow e_i' : \tau_2\}}{\Gamma, \mathrm{K} \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\overline{p_i \to e_i;}\} \rightsquigarrow \mathbf{case}\ e'\ \mathbf{of}\ \{\overline{p_i' \to e_i';}\} : \tau_2}$$

Figure 23: Propagation system for the let-polymorphic lambda calculus

$$PatVar \quad \frac{\mathrm{K} \vdash \tau : \star}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} x \rightsquigarrow x : \tau \Rightarrow \{x : \tau\}, \epsilon}$$

$$PatCon \quad \frac{\Gamma, \mathrm{K} \vdash C : \tau_1 \to \ldots \to \tau_n \to \tau}{\Gamma, \mathrm{K} \vdash_{\mathrm{pat}} C\ x_1 \ldots x_n \rightsquigarrow C\ x_1 \ldots x_n : \tau \Rightarrow \{x_i : \tau_i\}, \epsilon}$$

Figure 24: Identity transformation for patterns

$$\textit{PData}\quad \frac{\mathrm{K}' = \{T : \overline{\kappa_i \rightarrow}\ \star\} \qquad \{\mathrm{K} \cup \mathrm{K}' \cup \overline{\alpha : \kappa} \vdash C_i \rightsquigarrow C_i' \Rightarrow \Gamma_i'\}}{\mathrm{K} \vdash \mathbf{data}\ T\ \overline{\alpha : \kappa} = \overline{C\ |}\ \rightsquigarrow \mathbf{data}\ T\ \overline{\alpha : \kappa} = \overline{C'\ |} \Rightarrow \bigcup \Gamma_i', \mathrm{K}'}$$

$$\textit{PCon}\quad \frac{\{\mathrm{K} \vdash \tau_i : \star\}}{\mathrm{K} \vdash C\ \overline{\tau} \rightsquigarrow C\ \overline{\tau} \Rightarrow \{C : \overline{\tau \rightarrow}\ T\ \overline{\alpha}\}}$$

$$\textit{PProg}\quad \frac{\{\mathrm{K} \vdash D_i \rightsquigarrow D_i' \Rightarrow \Gamma_i, \mathrm{K}_i\} \qquad \Gamma \cup \bigcup \Gamma_i, \mathrm{K} \cup \bigcup \mathrm{K}_i \vdash e \rightsquigarrow e' : \tau}{\Gamma, \mathrm{K} \vdash \overline{D}; \mathit{main} = e \rightsquigarrow \overline{D'}; \mathit{main} = e' : \tau}$$

Figure 25: Propagation rules for datatype declarations and programs

A.2 Syntax interpretation

TrfRule
$$\frac{\pi_1 \rightsquigarrow \pi_2 \text{ is a rule} \quad \Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_1 @ e \Rightarrow \Sigma \quad \Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_2 = e' : \tau'}{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau'}$$

MApp
$$\frac{\Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_1 @ e_1 \Rightarrow \Sigma_1 \quad \Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_2 @ e_2 \Rightarrow \Sigma_2}{\Gamma, \mathrm{K} \vdash_{\text{mat}} \pi_1\ \pi_2 @ e_1\ e_2 \Rightarrow \Sigma_1 \cup \Sigma_2}$$

MExp
$$\frac{}{\Gamma, \mathrm{K} \vdash_{\text{mat}} e @ e \Rightarrow \epsilon}$$

MMeta
$$\frac{\Gamma, \mathrm{K} \vdash e \rightsquigarrow e' : \tau}{\Gamma, \mathrm{K} \vdash_{\text{mat}} M @ e \Rightarrow \{M \mapsto e' : \tau\}}$$

SApp
$$\frac{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_1 = e_1' : \tau \to \tau_1 \quad \Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_2 = e_2' : \tau}{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} \pi_1\ \pi_2 = e_1'\ e_2' : \tau_1}$$

SExp
$$\frac{\Gamma, \mathrm{K} \vdash e : \tau}{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} e = e : \tau}$$

SMeta
$$\frac{\Sigma\,(M) = e' : \tau'}{\Gamma, \mathrm{K}, \Sigma \vdash_{\text{subst}} M = e' : \tau'}$$

Figure 26: Interpretation of expression transformation syntax

$$\textit{PatRule}\qquad \frac{\begin{array}{c} C\ M_1 \ldots M_n \rightsquigarrow C'\ N_1 \ldots N_m \Rightarrow \Delta \text{ is a rule} \\ \Gamma, \mathrm{K} \vdash C' : \tau_1 \to \ldots \to \tau_m \to \tau \\ \Sigma = \{M_i \mapsto x_i\} \\ \{\Sigma \vdash N_i \mapsto y_i \qquad \Gamma_i = \{y_i : \tau_i\}\} \end{array}}{\Gamma, \mathrm{K} \vdash_{\text{pat}} C\ x_1 \ldots x_n \rightsquigarrow C'\ y_1 \ldots y_m : \tau \Rightarrow \bigcup \Gamma_i, \Sigma\ (\Delta)}$$

$$\frac{M \in \Sigma}{\Sigma \vdash M \mapsto \Sigma\ (M)} \qquad \frac{M \notin \Sigma \qquad x \text{ fresh}}{\Sigma \mapsto M \mapsto x}$$

Figure 27: Interpretation of pattern transformation syntax

$$\textit{ConRule}\qquad \frac{\begin{array}{c} C\ \overline{M} \rightsquigarrow C'\ \overline{\hat{N}} \Rightarrow \Delta \text{ is a constructor transformation} \\ \Sigma = \{M_i \mapsto \tau_i\} \\ \{\Sigma \vdash \hat{N}_i \mapsto \tau_i' \qquad \mathrm{K} \vdash \tau_i' : \star\} \end{array}}{\mathrm{K} \vdash C\ \overline{\tau} \rightsquigarrow C'\ \overline{\tau'} \Rightarrow \{C' : \overline{\tau \to}\ T\ \overline{\alpha}\}}$$

$$\frac{}{\Sigma \vdash M : \tau \mapsto \tau} \qquad \frac{M \in \Sigma}{\Sigma \vdash M \mapsto \Sigma\ (M)}$$

Figure 28: Interpretation of constructor transformation syntax

$$\frac{\begin{array}{c} C\ \overline{M} \rightsquigarrow C'\ \overline{\hat{N}} \Rightarrow \Delta \text{ is a constructor transformation} \\ \{M_i' = \mathit{untag}\ (M_i)\} \end{array}}{C\ \overline{M'} \rightsquigarrow C'\ \overline{N} \Rightarrow \Delta \text{ is a pattern transformation}}$$

$$\mathit{untag}\ (M) = M$$
$$\mathit{untag}\ (M : \tau) = M$$

Figure 29: Induced pattern rule from constructor rule