Implementation of PMC

Implementation of Pattern Matching Calculus Using Type-indexed Expressions

By
Xiaoheng Ji, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (2005)
(Department of Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Implementation of Pattern Matching Calculus Using Type-indexed Expressions
AUTHOR: Xiaoheng Ji, B.Sc. (Wuhan University, China)
SUPERVISOR: Dr. Wolfram Kahl
NUMBER OF PAGES: viii, 109

Abstract

The pattern matching calculus introduced by Kahl provides a fine-grained mechanism of modelling non-strict pattern matching in modern functional programming languages. By changing the rule of interpretting the empty expression that results from matching failures, the pattern matching calculus can be transformed into another calculus that abstracts a "more successful" evaluation. Kahl also showed that the two calculi have both a confluent reduction system and a same normalising strategy, which constitute the operational semantics of the pattern matching calculi.

As a new technique based on Haskell's language extensions of type-saft cast, arbitrary-rank polymorphism and generalised algebraic data types, type-indexed expressions introduced by Kahl demonstrate a uniform way of defining all expressions as type-indexed to guarantee type safety.

In this thesis, we implemented the type-indexed syntax and operational semantics of the pattern matching calculi using type-indexed expressions. Our type-indexed syntax mirrors the definition of the pattern matching calculi. We implemented the operational semantics of the two calculi perfectly and provided reduction and normalisation examples that show that the pattern matching calculus can be a useful basis of modelling non-strict pattern matching.

We formalised and implemented the bimonadic semantics of the pattern matching calculi using categorical concepts and type-indexed expressions respectively. The bimonadic semantics employs two monads to reflect two kinds of computational effects, which correspond to the two major syntactic categories of the pattern matching calculi, i.e. expressons and matchings. Thus, the resulting implementation provides the detotational model of non-strict pattern matching with more accuracy.

Finally, from a practical programming viewpoint, our implementation is a good demonstration of how to program in the pure type-indexed setting by taking fully advantage of Haskell's language extensions of type-safe cast, arbitrary-rank polymorphism and generalised algebraic data types.

Acknowledgements

My deep gratitude and appreciation first and foremost go to my supervisor, Dr. Wolfram Kahl, for introducing me to the functional programming language Haskell, and for his thoughtful guidance and many productive discussions through out this thesis research, and for all invaluable support needed to finish this thesis, especially during the final few weeks of the genesis of this thesis. This thesis benefits greatly from his incredible enthusiasm in teaching and research as well as his deep insights, instructive suggestions, and patient revisions.

I would like to express my sincere thanks to Dr. Christopher Anand for his time to be the chair of my thesis defense committee. I am indebted to Dr. Jacques Carette for his time to be on my thesis defense committee and his programming languages graduate course, which enriched my understanding of programming languages. I sincerely thank Dr. Hitoshi Furusawa for discussions on my thesis research.

Finally, I am infinitely grateful to my wife Wei and my family for their love, support and patience during the genesis of this thesis.

Table of Contents

Abstract iii

Acknowledgements iv

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Background: The Pattern Matching Calculi 2

1.2.1 Abstract Syntax 3

1.2.2 Operational Semantics 5

1.3 Background: The Functional Programming Language Haskell 6

1.4 Background: Type-Indexed Expressions 7

1.4.1 Variables 7

1.4.2 Type-indexed λ-expressions 7

1.4.3 Type-Indexed Maps 8

1.4.4 Expression Evaluation 10

1.5 Contributions of the Thesis 11

1.6 Structure of the Thesis 12

Chapter 2 Type-Indexed Implementation of PMC **13**

2.1 Patterns and Expressions 13

2.1.1 Variables 14

2.1.2 Constructors 15

2.1.3 Patterns 16

2.1.4 Expressions 16

2.2 Matchings 17

2.3 PMC Auxiliary Function Library 17

2.4 Examples 19

2.4.1 Example 1 19
2.4.2 Example 2 21
2.4.3 Example 3 24
2.4.4 Example 4 26
2.4.5 Example 5 26
2.5 Summary 27

Chapter 3 Operational Semantics of PMC **28**
3.1 Substitutions 28
3.2 Reduction Rules 30
3.2.1 PMC Expressions Reduction Rules 30
3.2.2 First-order PMC Matchings Reduction Rules 31
3.2.3 Second-order PMC Matchings Rules or Rule Schemas 32
3.2.4 Fixed-point Reduction Rules 34
3.2.5 Unioning All Reduction Rules 34
3.2.6 Type-Lost Problem of Implementing Rules Using Rewriting 35
3.3 Reduction Examples 38
3.3.1 One-Step Reduction Example 38
3.3.2 Many-Step Reduction Example 39
3.3.3 Transformation Rule for Interpretting Operators 40
3.3.4 Difference Reduction Sequences of the Two Calculi 41
3.4 Normalisation 44
3.4.1 Leftmost-Outermost Strategy 44
3.4.2 Normalising Strategy 45
3.5 Normalisation Examples 47
3.5.1 Reduction to SHNF 47
3.5.2 Normalisation Examples of PMC Fixed-point Expressions 48
3.6 Summary 49

Chapter 4 Bimonadic Semantics of PMC **50**
4.1 Introduction 50
4.2 Categorical Notation 51
4.3 Formalisation of the Bimonadic Semantics of PMC 53
4.4 Monads 57

4.4.1 Matching Monad 57
4.4.2 Expression Monad for $\mathsf{PMC}_{\oslash}$ 58
4.4.3 Expression Monad for Resurrection of Matching Failure 58
4.5 Implementation of Type Semantics 59
4.6 Variable Semantics 61
4.6.1 Variable Assignments 61
4.6.2 Operator Semantics 62
4.7 Constructor Semantics 63
4.7.1 Constructor Assignments 63
4.7.2 Semantics of Pattern Constructors 65
4.8 Implementation of Bimonadic Semantics 67
4.8.1 Semantic Function for Patterns 68
4.8.2 Semantic Function for Expressions 69
4.8.3 Semantic Function for Matchings 70
4.9 Evaluation Examples 71
4.9.1 Four Simple Evaluation Examples 71
4.9.2 Evaluation Example of Variable Scope 73
4.9.3 Different Evaluation Results of the Two Calculi 74
4.10 Summary 75

Chapter 5 Conclusions and Future Work **76**
5.1 Summary of the Thesis 76
5.2 Related Work 77
5.3 Accomplishments 77
5.4 Future Work 78

Appendix A Syntax of PMC **79**
A.1 Variable 79
A.2 Constructors 82
A.3 Patterns 83
A.3.1 The Definition of Patterns 83
A.3.2 *HasVar* and *HasConstructorApp* classes and instances 84
A.4 Type-Indexed Syntax of Pattern Matching Calculi 84
A.4.1 Type-Indexed Implementation of Syntax of Pattern Matching Calculi 85

A.4.2 *HasVar* instance 86

Appendix B Text Representations of PMC Terms **87**

B.1 Text Representation of PMC Terms 87

B.2 Examples of Text Representations of PMC Terms 89

Appendix C Tool Modules from Kahl's work **91**

C.1 Type-Indexed Maps 91

C.2 Q-Combinators 94

C.3 Transformations and Transformation Combinators 95

C.4 Transformation Transformers 96

C.5 Prelude Extensions 98

C.5.1 Material Related to *Show* 98

C.5.2 Lists 99

C.5.3 Monads 99

C.5.4 Other Datatypes 99

Appendix D α-conversion **100**

D.1 α-conversion 100

D.1.1 α-conversion 100

D.1.2 Variable Suffixes 101

D.1.3 Renaming Variables 102

D.2 α-conversion Examples 103

D.2.1 Closure 106

Chapter 1

Introduction

"Computer languages that have a syntax for discriminating among data with different structures are said to perform pattern matching"[1]. Term rewriting languages employ pattern matching as a fundamental way of evaluating a program to a result. Functions in functional programming languages can also be defined and evaluated using pattern matching. Issues such as the order of matching against patterns and the mechanisms of attaching a computational condition to supplement the structural pattern are important topics in the field of pattern matching.

Haskell is a modern, purely functional programming language, where functions can be defined using pattern matching. In the Haskell 98 language report [9], for semantics of pattern matching, the only internalisation are case expressions. Pattern matching is translated into case expressions to interpret. In Kahl's seminal paper [11], he argued that case expressions mix too many different aspects of rewriting into a single syntactic construct, and proposed pattern matching calculi (PMC) as a more attractive alternative. Moreover, he presented operational semantics of PMC to demonstrate how to execute a program in PMC setting. Kahl also provided a mechanised confluence proof performed in Isabelle 2003 [12] and a *normalisation strategy* for PMC.

The Glasgow Haskell Compiler (GHC) is an industrial strength Haskell compiler. GHC provides a language extension of generalised algebraic data types (GADTs). GADTs, which are discussed in [20], are a modest generalisation of conventional data types. GADTs provides the mechanism of defining well typed programs in syntactical level. Based on Haskell's language extensions of type-safe cast, arbitrary-rank polymorphism, and GADTs, type-indexed expressions are introduced by Kahl in [14], which demonstrates a uniform way of defining all expressions as type-indexed to capture more program abstraction. The mechanism for using type-indexed expressions to model PMC data structures can offer both convenience in programming and clarity in code. With PMC syntax completely based on type-indexed expressions, we can model PMC's data structures with surprising accuracy by mirroring the original definition in [11]. Moreover, type-indexed expressions can express function properties through the families of index types and thus capture more program errors at compile time.

1.1 Motivation

The motivation of our research in this thesis is that, by taking full advantage of the power of type-indexed expressions, we can provide a more robust and efficient implementation of

PMC, which itself is a new calculus providing the two fine-grained interpretations of the empty expression that results from matching failures.

The rest of this chapter is organized as follows. We first introduce the background that our work is based on, which includes Section 1.2 pattern matching calculus and Section 1.4 type-indexed expressions. We then outline contributions of the thesis in Section 1.5. Finally, we give the structure of the thesis in Section 1.6.

1.2 Background: The Pattern Matching Calculi

The operational semantics of functional programming languages studies how to execute programs. It is usually explained by translating a function into a set of term rewriting rules in a certain kind of term rewriting system or a single expression in an appropriate λ-calculus.

Modern functional programming languages support function definitions based on *pattern matching.* In Haskell 98 report [9], the meaning of pattern matching in function definitions is specified in terms of *case* expressions.

In this section, the pattern matching calculi will be introduced. Most material of this section has been adapted slightly from [11].

In Haskell, we can use pattern matching to define a function that determine whether a list is empty or not as follows:

```
isEmtpyList (x : xs) = False
isEmtpyList ys = True
```

This function will be translated into case expressions to define its operational semantics:

```
isEmptyList zs = case zs of
  (x : xs) → False
  ys → True
```

However, seen as an internalisation of pattern matching, case expressions is not completely analogous to the internalisation of function abstraction in λ-calculus. Case expressions mix too many different aspects of rewriting into a single syntactic construct, not only including an addition application to an argument, but including such complicate mechanisms as *Boolean guards* and *pattern guards.*

Kahl presented a new calculus named pattern matching calculus (PMC) that cleanly internalises pattern matching via a modest abstraction in his seminal PMC paper [11].

Now we can use this new calculus to define the above function as follows:

$$\texttt{isEmptyList} = \{\!|\, (x : xs) \mapsto\!\!\!\Rightarrow \mathsf{False} \,|\, ys \mapsto\!\!\!\Rightarrow \mathsf{True} \,|\!\}$$

The new straightforward internalisation of pattern matching has advantages for expressivity: it saves additional variable names like *zs* when using *case* expressions.

PMC itself can be implemented in functional programming languages. Therefore, it also has advantages for reasoning about programs. Compared with priority rewriting systems, where unconditional equations have to be added to define priority systems, PMC allows direct transliteration of such priortised definitions without additional cost, and even its syntactical expressivity is so powerful that it is sufficient to express both Boolean guards and pattern guards.

Avoiding complicated unconditional priority equations and with powerful expressivity, PMC can be seen as a simple and uniform internalisation of pattern matching.

When treating matching against non-covered alternatives as a run-time error, this kind of PMC is called $\mathsf{PMC}_{\oslash}$, which mirrors exactly the definition of pattern matching in Haskell. By changing the single rule concerned with results of matching failure to "failure as exception", we have $\mathsf{PMC}_{\lightning}$, which is a promising foundation for further exploration of the "failure as exception" approach proposed by Erwig and Peyton Jones [5]. The two kinds of calculi are both confluent and equipped with the same normalising strategy.

1.2.1 Abstract Syntax

PMC has two major syntactic categories, namely *expressions* and *matchings*. These are defined by mutual recursion. *Expressions* can be seen as expressions of functional programming languages and *matchings* can be seen as a generalisation of case alternatives, or groups of case alternatives. Matchings that directly correspond to (groups of) case alternatives expose patterns to be matched against arguments; we say such matchings are *waiting for argument supply*, for example:

$$(x : xs) \mapsto \mathsf{False} \,|\, ys \mapsto \mathsf{True}$$

Complete case expressions correspond to expressions formed from matchings that already have an argument supplied to their outermost patterns; matchings that have arguemnts supplied to all their open patterns are called *saturated*, for example:

$$[5] \rhd (x : xs) \mapsto \mathsf{False} \,|\, [5] \rhd ys \mapsto \mathsf{True}$$

A *pattern* is an expression built only from variables and *constructors*. *Patterns* form a separate syntactic category that will be used to construct pattern matchings.

We use the following base sets:

- Var is the set of *variables*, and
- Constr is the set of *constructors*.

In our later implementations, all literals, like numbers and characters, are assumed to be elements of Constr and are used only in zero-ary constructions.

As known in functional programming languages, constructors will be used to build both patterns and expressions.

The following summarises the abstract syntax of PMC:

Pat	$::= \mathsf{Var}$	variable
	$\mid \mathsf{Constr}(\mathsf{Pat}, \ldots, \mathsf{Pat})$	constructor pattern
Expr	$::= \mathsf{Var}$	variable
	$\mid \mathsf{Constr}(\mathsf{Expr}, \ldots, \mathsf{Expr})$	constructor application
	$\mid \mathsf{Expr}\ \mathsf{Expr}$	function application
	$\mid \{\!\!\mid \mathsf{Match} \mid\!\!\}$	matching abstraction
	$\mid \oslash$	empty expression
	$\mid \mathsf{EFix}$	fixed-point combinator
Match	$::= \rceil \mathsf{Expr} \lceil$	expression matching
	$\mid \lightning$	failure
	$\mid \mathsf{Pat} \Mapsto \mathsf{Match}$	pattern matching
	$\mid \mathsf{Expr} \rhd \mathsf{Match}$	argument supply
	$\mid \mathsf{Match} \mathbin{\pmb{\mid}} \mathsf{Match}$	alternative

Patterns are built from variables and constructor applications.

Expressions correspond to expressions of functional programming languages. Besides variables, constructor application and function application, we also have the following special kinds of expressions:

- *Matching abstraction* $\{\!\!\mid m \mid\!\!\}$ is built from a matching m. It can be read "*match m*"..

  If the matching m is *unsaturated*, i.e., "waiting for arguments", then $\{\!\!\mid m \mid\!\!\}$ abstracts m into a function.

  If m is a saturated matching, then it can either succeed or fail; if it succeeds, then $\{\!\!\mid m \mid\!\!\}$ reduces to the value "returned" by m; otherwise matching failure happens, $\{\!\!\mid m \mid\!\!\}$ is considered ill-defined.

- $\oslash$ is called the *empty expression*, which results from matching failures. It could also be called the "ill-defined expression" as the matching abstraction built from a failed saturated matching.

  Two interpretations of $\oslash$ will be considered:

  – It can be a "manifestly undefined" expression equivalent to non-termination — following the common view that divergence is semantically equivalent to run-time errors.

  – It can be a special "error" value propagating matching failure considered as an "exception" through the syntactic category of expressions.

As known in functional programming languages, the result of matching constructor applications of the same constructor, but with different arities, will produce a matching failure.

Matchings are the syntactic category that embodies the pattern analysis aspects:

- For an expression $e : \mathsf{Expr}$, the *expression matching* $\rceil \mathsf{Expr} \lceil$ always succeeds and returns e. It can be read *"return e"*.
- $\lightning$ is the matching that always fails.
- The *pattern matching* $p \mapsto m$ waits for supply of one argument more than m; this pattern matching can be understood as succeeding on instances of the (linear) pattern $p : \mathsf{Pat}$ and then continuing to behave as the resulting instance of the matching $m : \mathsf{Match}$. It roughly corresponds to a single case alternative in languages with case expressions.
- *argument supply* $a \rhd m$ is the matching-level incarnation of function application, with the argument on the left and the matching it is supplied to on the right. It saturates the first argument m is waiting for.

  The inclusion of argument supply into PMC makes it feasible for the design of the reduction system to implement separation of the concerns of on the one hand traversing the boundary between expressions and matchings and on the other hand matching patterns against the right arguments.
- the *alternative* $m_1 \,|\, m_2$ is understood sequentially: it behaves like m_1 until this fails, and only then it behaves like m_2.

Note that there are no matching variables; variables can only occur as patterns or as expressions.

The parentheses in matchings of the shape $a \rhd (p \mapsto m)$ can be ommited since there is only one way to parse $a \rhd p \mapsto m$ in PMC.

1.2.2 Operational Semantics

Kahl presented a set of reduction rules for PMC in [11]. These will be presented in Section 3.2 together with their implementation. The reduction rules can be united to constitute a confluent rewriting system. The intuitive explanation and detailed proof of this confluence result can be found in [11] and [12], respectively.

PMC is equipped with a normalising strategy of the reduction rules, which reduces expressions and matchings to *strong head normal form* (SHNF). The definition of SHNF introduced in [21] has been translated into the PMC setting by Kahl in [11]. This deterministic strategy for reduction to SHNF induces a deterministic normalising strategy for PMC and will be presented in Section 3.4 together with an implementation.

The operational semantics of PMC consists of the set of confluent reduction rules and the normalisation strategy.

The pattern matching calculus $\mathsf{PMC}_{\oslash}$ mirros exactly the definition of pattern matching of current functional programming languages and can form a more appropriate basis than term rewriting by providing a confluent and normalising reduction system. By changing a single reduction rule concerned with results of matching failure to "failure as exception", we will have $\mathsf{PMC}_{\lightning}$, which results in "more successful" evaluation. $\mathsf{PMC}_{\lightning}$ can be turned into a basis for programming language implementations.

1.3 Background: The Functional Programming Language Haskell

Haskell is a generel purpose, non-strict, purely functinal programming language. Haskell is a *general purpose* language that means it can be used to develop almost all kinds of programs, from web browers to compilers. Hasekll is *non-strict* that means Haskell is a language with lazy evaluation. Lazy evaluation means that an expression is evaluated only when its value is needed. Haskell is *purely functional* that means function evaluations have no *side effects* in Haskell. A function is said to produce a side-effect if it modifies some state other than its return value. Haskell doesn't allow side-effects, which leads to less bugs.

As an experimental language for research goals, Haskll has evolved with many extensions, which include syntactic sugar, type system innovations, control extensions and etc. Syntactic sugar facilitates the construction of some complex syntactic structures. Type system innovations make Haskell more powerful in expressiveness. Control extensions provide a more fine-grained control capacity in organising control structures of programs.

There are three main Haskell compilers and interpreters, namely Hugs, the Glasgow Haskell Compiler (GHC) and nhc98. Hugs is evclusively a Haskell interpreter, meaning that you can test and debug programs in an interactive environment. GHC is both an interpreter and a compiler which will produce stand-alone programs. NHC is exclusively a compiler. GHC implements all the Haskell 98 language report and extensions, which is a definition of the Haskell language and its standard libraries.

Compared with many other programming languages, Haskell has many advantages: Haskell is strongly typed and doesn't allow "side effects", which makes Haskell program easier to write and maintain. Haskell is non-strict that frees the programmer from many concerns about evaluation order. If a value of a argument is not necessary for evaluating the result of a function, the argument will never be evaluted. Another advantage of the non-strict feature of Haskell is that its data constructors are also non-strict and therefore can be used to define *infinite* data structures. Finally, Haskell is close to its semantics so that it is amenable to formal techniques.

One of the disadvantages of Haskell is that it is difficult to analyze its intensional behavior, such as the time a program takes to run and the execution order of program statements.

1.4 Background: Type-Indexed Expressions

Most functional programming languages such as Haskell and ML allow to define functions using pattern matching. In general, these languages also support the concept of algebraic data types, which allows pattern matching over user-definable types. Over the decades, there have been many efforts on languages extensions to increase the expressiveness of the languages. GHC is extended with generalized algebraic data types (GADTs) [6] in its 6.4 version, which support some extensions of algebraic data types. Based on GADTs as well as some extensions like type-safe cast and arbitrary-rank polymorphism in Haskell, Kahl introduced the technique of type-indexed expressions to produce a type-safe data type of typed expressions in [14]. Type-indexed expressions demonstrate how to use GADTs as well as other Haskell language extensions of type-safe cast and arbitrary-rank polymorphism to structure their programs in a way that makes them type-safty. Our implementations of PMC syntax, operational semantics and bimonadic semantics are completely based on type-indexed expressions to guarantee type safety.

In this section, we first introduce definitions of type-indexed variables and λ-expressions and then introduce type-index maps as an environment of interpreting variable assignments. Finally, by using special cases of type-indexed maps to act as an environment, we introduct two simple evaluation examples.

Because this section is a brief introduction to type-indexed expressions, we do not cover all aspects of type-indexed expression for simplicity. For example, this section does not include the subsitution module, which encapsulates type-indexed maps and maps values of type-indexed variables to values of type-indexed expressions, and the rule module, which defines matching and rule applications. In addition, some underlying utility libraries are also not included in this section. For a detailed information about type-indexed expressions, readers can refer to Kahl's paper [14].

1.4.1 Variables

A type-indexed type is defined for variables.

```
newtype Var a = V String
```

An auxiliary function is defined to facilitate variable construction.

```
mkVar s = if all (λc → isAlphaNum c ∨ c ∈ "'_") s then V s
  else error $ "mkVar: illegal variable name ‘" ++ s ++ "’"
```

1.4.2 Type-indexed λ-expressions

The type of type-indexed λ-expressions is defined using a GADT as follows.

```
data Expr :: * → *where
```

```
Const :: ShowSPrec a → a → Expr a
Apply :: Typeable a ⇒ Expr (a → b) → Expr a → Expr b
Var :: Var a → Expr a
Lambda :: (Typeable a, Typeable b) ⇒ Var a → Expr b → Expr (a → b)
```

GHC's Typeable class reifies types to some extent by associating *comparable* type representations to types. Here the constraint *Typeable a* make Haskell's type inference system able to type expressions.

Due to that constrainted constructors are not supported, we cannot directly use *Const* :: *Show a* ⇒ *a* → *Expr a*. Currently, we use explicit argument of the class dictionary as a substitute. The type of *showsPrec* maximum the flexibility.

The following two auxilliary functions are defined to facilitate construction of expressions. The construction function *constant* is used to construct value of type *Expr a* when type *a* has an instance of class *Show*.

```
constant :: Show a ⇒ a → Expr a
constant = Const showsPrec
```

The construction function *named* is used when the corresponding type has not an instance of class *Show*. This function also provide non-standard *Show* instances without having to declare newtype.

```
named :: String → a → Expr a
named s = Const (λ_ _ → (s++))
```

1.4.3 Type-Indexed Maps

This subsection presents the central parts of Kahl's implementation of type-indexed maps, which can be used to implement β-reduction without subsitutions. A type-indexed map from typed variables to correspondingly typed values acts as environment to interpret variable assignments of PMC.

A type-indexed map *m* :: *TIMap k r* represents type-indexed families $m = (m_a)_{a::*}$ of maps $m_a :: \mathit{Map}\ (k\ a)\ (r\ a)$ where both the source and target types may depend on the index.

This is made possible by the type-safe casts from Data.*Typeable* and the arbitrary-rank polymorphism supported by *GHC* with `-fglasgow-exts`.

Part code of the module *TIMap* including the implementation of *type-indexed maps* is presented in this subsection.

The definition of type-indexed map need Data.*Map* module, which is intended to be imported qualified, to avoid name clashes with Prelude functions.

```
import qualified Data.Map as Map
```

We define a type-indexed map as a list of *Map*s, where each *Map* is the component map for a specific type.

For these *type-specific maps*, we need a newtype so that *gcast* can be applied to them directly:

```
newtype TSMap k r a = TSMap (Map.Map (k a) (r a))
```

A type-indexed map is then implemented essentially as a list of existentially quantified type-specific maps — we use GADT notation to define this in a single definition as a specialised list type (the *Typeable* instance has to be done manually again).

```
data TIMap :: (* → *) → (* → *) → *where
  Empty :: TIMap k r
  Cons :: (Typeable a, Ord (k a)) ⇒ TSMap k r a → TIMap k r → TIMap k r
tcTIMap = mkTyCon "TIMap.TIMap"
instance (Typeable1 k, Typeable1 r) ⇒ Typeable (TIMap k r) where
  typeOf (_ :: TIMap k r) = mkTyConApp tcTIMap
    [typeOf1 (⊥ :: k ())
    , typeOf1 (⊥ :: r ())
    ]
```

The constructors are not exported. The exported interface will guarantee the *invariant* that no two elements of such a list have the same type, and that no list element is an empty type-specific map.

A more efficient implementation could be implemented via a *Map TypeRep* (*ETSMap k r*) — this would need an *Ord* instance for *TypeRep* (currently not provided in Data.*Typeable*), and a wrapper type *ETSMap* for the existentially quantified version of *TSMap*.

For lookup, we use *gcast* on each list element to test whether it has the right type for the argument; if it has, then, according to the *TIMap k* invariant, it is the only list element of that type, and Map.*lookup* produces the result.

```
lookup :: (Typeable a, Ord (k a)) ⇒ k a → TIMap k r → Maybe (r a)
lookup v Empty = Nothing
lookup v (Cons tsm tim) = case gcast tsm of
  Nothing → lookup v tim
  Just (TSMap m) → case Map.lookup v m of
    Nothing → lookup v tim
    j → j
```

The functions *insert* and *delete* can be implemented in the same pattern.

Additionally, an empty *TIMap* value is implemented to be used as an initial environment value in evaluating closed expressions.

```
empty :: TIMap k r
empty = Empty
```

Some other functions has also been implemented in [14]. For simplicity, we will not introduced them here.

1.4.4 Expression Evaluation

In this subsection, two evaluation examples are introduced to demonstrate evaluations of type-indexed expressions.

The module *TIMap* is imported to build a type-indexed map that acts as environment to implement β-reduction rule without subsitutions, i.e., it is used to interpret variable assignments.

```
import qualified TIMap as VA
```

Since type-indexed maps require type constructor *applications* for key and value types, we have to use an explicit *Identity* type constructor for the value type.

```
type VarAssign = VA.TIMap Var Identity
```

All the type-safe casts are now hidden behind the *VA* interface; we only have to import Data.*Typeable* to be able to state the type signature explicitly:

```
eval :: Typeable a => VarAssign -> Expr a -> a
eval va (Var v) = case VA.lookup v va of
  Just r -> runIdentity r
  Nothing -> error $ "eval: free variable " ++ show v
eval va (Const _ c) = c
eval va (Apply f a) = eval va f (eval va a)
eval va (Lambda v e) = \r -> eval (VA.insert v (Identity r) va) e
```

An empty variable assignment is needed in evaluating closed expressions.

```
eval' :: Typeable a => Expr a -> a
eval' = eval VA.empty
```

We define two expressions as follows:

```
e1 :: Expr Int
e1 = Apply (Lambda v1 $ Apply (named "S" succ) (Var v1)) (constant (4 :: Int))
  where v1 = vVar 1
e2 :: Expr Int
e2 = Apply
  (Apply (Lambda x (Lambda y (Apply (Apply (named "add" (+)) (Var x)) (Var y))))
  (constant (4 :: Int))) (constant (5 :: Int))
  where x :: Var Int
    x = mkVar' "x"
    y :: Var Int
    y = mkVar' "y"
```

We then apply evaluation function *eval'* on them.

```
*ExprTest> e1
(\ v1 :: Int -> S v1) 4
```

```
*ExprTest> eval' e1
5

*ExprTest> e2
(\ x :: Int -> \ y :: Int -> add x y) 4 5
*ExprTest> eval' e2
9
```

1.5 Contributions of the Thesis

The thesis has three principal contributions. The first is that we implemented type-indexed syntax and operational semantics of the pattern matching calculi. The second is that we formalised and implemented bimonadic semantics of the pattern matching calculi. The last is that by implementing PMC completely based on type-indexed expressions, our implementation demonstrates how to use the new technique, which is based on GHC's new languages extensions, to guarantee type safety.

As new calculi modellinig non-strict pattern matching, PMCs introduced by Kahl refine traditional pattern matching by dividing PMC terms into two major syntactic categories, namely *expressions* and *matchings*, to provide two kinds of interpretations for the *empty expression* that results from matching failures when such an empty expression is matched against a constructor pattern. Our implementation of PMC's syntax and operational semantics as well as sophisticated evaluation examples show that PMC can be a useful basis for implementations of modern functional programming language.

In the thesis, we also formalise and implement the bimonadic semantics of PMC. Compared with traditional denotational semantics, our implementation take advantage of a bimonadic approach to structure denotational semantics, which achieves a high level of modularity and extensibility.

GHC's Typeable class uses *comparable* type representations as type encodings to reify types so that type-safe cast operations are definable. Based on the feature, GHC is extended with generalized algebraic data types (GADTs). Type-indexed expressions take full advantage of the GHC's new features. In this thesis, by using type-indexed expressions, we explore a new design space of programming, where the type-indexed syntax of PMC not only describe PMC construction forms of syntactical structures but also express type dependency relations of these construction forms. The obvious advantage of such an implementation is that the Haskell type system gives the validity of structures of our PMC expressions and matchings for free. However, some limitations have also been discovered that, as a tradeoff, for example, the type-lost problem in the Haskell type system have been exposed in syntactical level in the pure type-indexed setting. We discovered and described the type-lost problem in attempting to implement the PMC reduction rules using rewriting techniques.

1.6 Structure of the Thesis

This thesis consists of five chapters. The rest of this thesis is organized as follows.

Chapter 2 gives a complete type-indexed PMC definition as well as some examples of PMC matchings and expressions. The definition is a basis for later implementation of the operational semantics and the bimonadic semantics of PMC.

Chapter 3 implements the operational semantics of PMC, based on Kahl's paper [11, 13] and also provides some reduction and normalisation examples.

Chapter 4 formalises and implements the bimonadic semantics of PMC. Some evaluation examples are also provided.

Finally, In Chapter 5, we summarise our work in the thesis, describe related work, list accomplishements of this thesis, and discuss possible future work.

The appendices are provided in the end of the thesis.

Appendix A includes a complete code of definition of PMC syntax, which corresponds to the definition in Chapter 2.

Appendix B includes a complete code of text representations of PMC terms, which provide a mechanism to simply display PMC.

Appendix C includes some auxiliary tool modules from Kahl's work. We include them for completeness.

Appendix D includes a complete runnable code of implementing α-conversion in the PMC context.

The bibliography includes all references used in this work.

Chapter 2

Type-Indexed Implementation of Pattern Matching Calculi

This chapter includes our type-indexed implementation of the pattern matching calculi, which were introduced by Kahl in [11, 13].

The abstract syntax of the pattern matching calculi has been included in 1.2.1. The chapter will focus on the type-indexed implementation of the pattern matching calculi. We first implement variables and constructors in the type-indexed setting in the first two sections 2.1.1 and 2.1.2. Variables and constructors are two syntactic units of building *patterns* and *expressions*. We then implement the separate syntactic category *patterns* in section 2.1.3. In the subsequent section 2, we implement the two major syntactic categories *expressions* and *matchings*. Finally, we define some auxiliary functions to facilitate constructions and operations of PMC terms in section 2.3 and employ these functions to implement some examples of building sophisticated PMC terms in section 2.4. These example PMC terms are later used in reduction examples of the section 3.3, normalising examples of the section 3.5 and bimonadic semantics evaluation examples of the section 4.9.

All the code included in this chapter as well as in the subsequent chapters is excerpted from the implementation code, the rest of which has been included in whole in the appendices. Most of the code is written in the language of GHC-6.4 except some functions that are mutually recursively defined, which need at least current beta version 6.5 of GHC.

2.1 Patterns and Expressions

In order to be able to *match* patterns' *constructor functions* with expressions' *constructor functions*, we have to define the data type of expressions regarding *constructor functions* in the same way as we define the data type of patterns.

Although patterns form a separate syntactic category that will be used to construct pattern matchings, one might consider patterns as a subset of expressions.

Variables and constructors are two base sets, which are used to build both patterns and expressions.

According to abstract syntax of PMC, the syntax of patterns can be defined naturally and directly as follows:

```
data Pat :: * → * =
  VarPat :: Typeable a ⇒ Var a → Pat a
```

```
ConstrPat :: Typeable a ⇒ Constr a → Pat a
PatApply :: (Typeable a, Typeable b) ⇒ Pat (b → a) → Pat b → Pat a
```

However, such a definition can on the one hand obscure the distinction between full and partial constructor application and on the other hand produce ill-defined patterns. An partial constructor application can be as follows:

```
illDefPat1 :: Pat ([Int] → [Int])
IllDefPat1 = (ConstrPat (Constr (:))) ‘PatApply‘ (ConstrPat (Constr 5))
```

The corresponding partial constructor application in Haskell is:

```
illDefPat1' = (:) 5
```

However, the partial constructor application is already of type *Pat a* so that it can directly used in *Match a* to build the following pattern matching, which is obviously ill-defined in Haskell:

```
illDefMatch1 = case (:) 5 of
  (:) ys → Just ys
  _ → Nothing
```

Another source of defining ill-defined pattern is that this definition of patterns syntactically allows to build the following pattern:

```
illDefPat2 :: Pat ([Int] → [Int])
illDefPat2 = (VarPat (V "x" :: Var (Int → [Int] → [Int])))
  ‘PatApply‘ (VarPat (V "y" :: Var Int))
```

Obviously, such a pattern is also ill-defined.

In this section, by defining a special encoding of constructor types, we provide a more dedicate definition of constructor applications to enforce full application of constructors to all arguments. Thus, we use the Haskell type system to guarantee type safety for free and avoid the above-mentioned problems. In the subsequent subsections, we first define the two base sets of variables and constructors in 2.1.1 and 2.1.2 and then use the definitions of variables and constructors to define patterns and expressions respectively in 2.1.3 and 2.1.4.

2.1.1 Variables

Variables is one of two syntactic units of building *patterns* and *expressions* and can only occur as patterns or as expressions. Note that there are no matching variables.

In the type-indexed implementation of PMC, all syntactic elements are defined as type-indexed forms. Variables are defined as follows.

```
newtype Var a = V String
```

In the definition of variables, *String* is variable name's type and every type-indexed variable has of type *Var a*, which is a variable type with type *a* as index type.

Since the module *Variable*, which is excerpted in whole in the appendix A.1, exports *Var* as an abstract type, the constructor *V* is hidden and not exported. The following partial function *mkVar'* is provided to as the only interface to build a variable from a variable name of type *String*.

```
mkVar' :: forall a ∘ String → Var a
mkVar' = either error id ∘ mkVar
```

The function *mkVar* is used to facilitate defining the function *mkVar'*; it return a variable if the argument is a valid variable name or return an error message otherwise.

```
mkVar :: forall a ∘ String → Either String (Var a)
mkVar s = if isVarName s ∨ isOperator s then Right (V s)
  else Left $ "mkVar: illegal variable name or operator name ‘‘" ++ s ++ "’’"
```

Note that primitive operators are considered as variables in the implementation. For every primitive operator, a corresponding reduction rule has to be added in order to interpret it in the operational semantics and a correspondence between its variable in the implementation and real function in the source language has to be added into a semantic dictionary of type *TIMap* in the bimonadic semantics.

2.1.2 Constructors

In this subsection, we provide an abstract datatype for constructors that are type-indexed in a disciplined way, enabling syntactic distinction between full and partial constructor application.

We use the Haskell type system to enforce full application of constructors to all arguments by defining a special encoding of constructor types.

Constants expecting no arguments have a *CResult* type:

```
data CResult a = CResult String
```

Constructors expecting arguments have a *CArg* type:

For adding an additional first expected argument of type *a*, the constructor type is wrapped in *CArg c*

```
data CArg a c = CArg c
```

The following class relates constructor type encodings with the encoded types:

```
class CType c t | c → t where
instance CType (CResult a) a
instance CType c b ⇒ CType (CArg a c) (a → b)
```

2.1.3 Patterns

The abstract syntax of *patterns* is summarised as follows.

Pat ::=	Var	variable
	| Constr(Pat, ..., Pat)	constructor pattern

```
data Pat :: * → *where
  VarPat :: Typeable a ⇒ Var a → Pat a
  ConstrPat :: ConstrApp Pat (CResult a) → Pat a
```

Variables should be type-indexed. Therefore, we use *Var a* instead of *Var*.

We parameterise the type of fully applied constructor applications with the syntactic category *s* so that we can use this both for patterns and expressions.

```
data ConstrApp :: (* → *) → * → *where
  Constr :: c → ConstrApp s c
  ConstrApply :: Typeable a ⇒ ConstrApp s (CArg a c) → s a → ConstrApp s c
```

2.1.4 Expressions

The abstract syntax of *expressions* is summarised as follows.

Expr::=	Var	variable
	| Constr(Expr, ..., Expr)	constructor application
	| Expr Expr	function application
	| ⦃ Match ⦄	matching abstraction
	| ⊘	empty expression
	| EFix	fixed-point combinator

The application of the technique of type-indexed expressions in the definition of expressions can offer both convenience in programming and clarity in code. By using the technique of type-indexed expressions, we can translate directly the abstract syntax of expressions into the type-indexed setting. The type-indexed definition of expressions exactly mirrors the orignial definition of the type-indexed calculus in [11].

```
data Expr :: * → *where
  EVar   :: Typeable a ⇒ Var a → Expr a
  ConstrExpr :: Typeable a ⇒ ConstrApp Expr (CResult a) → Expr a
  Apply  :: (Typeable a, Typeable (a → b), Typeable b) ⇒
            Expr (a → b) → Expr a → Expr b
  MExpr :: Typeable a ⇒ Match a → Expr a
  Empty :: Typeable a ⇒ Expr a
```

$EFix :: Typeable\ a \Rightarrow Expr\ ((a \rightarrow a) \rightarrow a)$

2.2 Matchings

The abstract syntax of *matchings* is summarised as follows.

Match::= ⌉Expr⌈	expression matching
| ⇜	failure
| Pat ⊫> Match	pattern matching
| Expr ▷ Match	argument supply
| Match ❙ Match	alternative

By using the technique of type-indexed expressions, we can translate directly the abstract syntax of matchings into the type-indexed setting. The type-indexed definition of matchings exactly mirrors the orignial definition of the type-indexed calculus in [11].

```
data Match :: * → *where
  Return :: Typeable a ⇒ Expr a → Match a
  Fail   :: Typeable a ⇒ Match a
  PMatch :: (Typeable a, Typeable b) ⇒ Pat a → Match b → Match (a → b)
  Supply :: (Typeable a, Typeable b) ⇒ Expr a → Match (a → b) → Match b
  MAlt :: Typeable a ⇒ Match a → Match a → Match a
```

2.3 PMC Auxiliary Function Library

In the section, we define some auxiliary functions in the module *PMCLib* to facilitate construction and operations of PMC terms. In the subsequent chapters, the functions are frequently exploited to build PMC terms.

The following functions are defined to build constructors having different arguments.

```
type C0     r =              CResult r
type C1 a   r = CArg a (     CResult r)
type C2 a b r = CArg a (CArg b (CResult r))
type C3 a b c r = CArg a (CArg b (CArg c (CResult r)))
mkC0 = CResult
mkC1 = CArg ∘ CResult
mkC2 = CArg ∘ CArg ∘ CResult
mkC3 = CArg ∘ CArg ∘ CArg ∘ CResult

type CA0 s r     = ConstrApp s (C0     r)
type CA1 s a r     = ConstrApp s (C1 a   r)
type CA2 s a b r   = ConstrApp s (C2 a b r)
type CA3 s a b c r = ConstrApp s (C3 a b c r)
mkCA0 = Constr ∘ mkC0
mkCA1 = Constr ∘ mkC1
mkCA2 = Constr ∘ mkC2
mkCA3 = Constr ∘ mkC3
```

The following two functions are defined to build pattern variables and expression variables.

```
mkPVar :: Typeable a ⇒ String → Pat a
mkPVar s = VarPat $ mkVar' s

mkEVar :: Typeable a ⇒ String → Expr a
mkEVar s = EVar $ mkVar' s
```

The following two functions are defined to build pattern constants and expression constants.

```
mkPat :: Typeable a ⇒ String → Pat a
mkPat s = cPat0 $ CResult s

mkExpr :: Typeable a ⇒ String → Expr a
mkExpr s = cExpr0 $ CResult s
```

The following functions are defined to build expressions from values built from the data constructors *CArg* and *CResult*.

```
cExpr0 :: (Typeable a) ⇒ CResult a → Expr a
cExpr0 c = ConstrExpr (Constr c)

cExpr1 :: (Typeable a, Typeable c) ⇒ CArg a (CResult c) → Expr a → Expr c
cExpr1 c a = ConstrExpr (Constr c `ConstrApply` a)

cExpr2 :: (Typeable a1, Typeable a2, Typeable c) ⇒
  CArg a1 (CArg a2 (CResult c)) → Expr a1 → Expr a2 → Expr c
cExpr2 c a1 a2 = ConstrExpr (Constr c `ConstrApply` a1 `ConstrApply` a2)

cExpr3 :: (Typeable a1, Typeable a2, Typeable a3, Typeable c) ⇒
  CArg a1 (CArg a2 (CArg a3 (CResult c))) →
  Expr a1 → Expr a2 → Expr a3 → Expr c
cExpr3 c a1 a2 a3 = ConstrExpr $
  Constr c `ConstrApply` a1 `ConstrApply` a2 `ConstrApply` a3
```

The following functions are defined to build patterns from values built from the data constructors *CArg* and *CResult*.

```
cPat0 :: (Typeable a) ⇒ CResult a → Pat a
cPat0 c = ConstrPat (Constr c)

cPat1 :: (Typeable a, Typeable c) ⇒ CArg a (CResult c) → Pat a → Pat c
cPat1 c a = ConstrPat (Constr c `ConstrApply` a)

cPat2 :: (Typeable a1, Typeable a2, Typeable c) ⇒
```

```
    CArg a1 (CArg a2 (CResult c)) → Pat a1 → Pat a2 → Pat c
cPat2 c a1 a2 = ConstrPat (Constr c `ConstrApply` a1 `ConstrApply` a2)
cPat3 :: (Typeable a1, Typeable a2, Typeable a3, Typeable c) ⇒
    CArg a1 (CArg a2 (CArg a3 (CResult c))) → Pat a1 → Pat a2 → Pat a3 → Pat c
cPat3 c a1 a2 a3 = ConstrPat $
    Constr c `ConstrApply` a1 `ConstrApply` a2 `ConstrApply` a3
```

2.4 Examples

In the section, we use the auxiliary functions in section 2.3 to build examples of type-indexed PMC terms, which are later used in the section 3.3 reduction examples, the section 3.5 normalising examples and the section 4.9 bimonadic semantics evaluation example.

It is obvious that any λ-*calculus* terms can be translated into PMC terms: variables and function application are translated directly, and λ-abstraction is translated into a matching abstraction over a pattern matching that has a single-variable pattern and a result matching that immediately returns the body:

$$\lambda v.e := \{\!\mid v \rhd \urcorner e \ulcorner \mid\!\}$$

In the following subsections, we first give examples in the untyped λ-calculus or *case* expressions and then use abstract syntax of PMC to describe examples. Finally, we demonstrate how to build corresponding examples in the type-indexed implementation.

All the code in the section is included in the module *PMCExmaple*.

2.4.1 Example 1

This example demonstrates the building of a PMC expression from the following λ-calculus term in Haskell:

```
    example1 = (λ((x : xs) : ((y : ys) : zss)) → (xs : (ys : zss))) [[1, 2, 3], [2, 3, 4], [3, 4, 5], [6]]
```

which can be translated into the following PMC expression:

$$\{\!\mid [[1,2,3],[2,3,4],[3,4,5],[6]] \rhd (\mathsf{x} : \mathsf{xs} : (\mathsf{y} : \mathsf{ys} : \mathsf{zss})) \Mapsto \urcorner xs : (ys : zss) \ulcorner \mid\!\} \ .$$

The PMC expression will be used in 3.3 to demonstrate PMC reduction.

We first build the expression $[[1,2,3],[2,3,4],[3,4,5],[6]]$.

The building of the subexpressions $[1,2,3]$, $[2,3,4]$, $[3,4,5]$ and $[6]$ need a constructor ":" of type *C2 Int* [*Int*] [*Int*].

```
    cons :: Typeable a ⇒ C2 a [a] [a]
```

```
cons = mkC2 ":"
```

We define two functions to facilitate building a list of two elements, respectively for patterns and expressions.

```
consP :: Typeable a ⇒ Pat a → Pat [a] → Pat [a]
consP = cPat2 cons

consE :: Typeable a ⇒ Expr a → Expr [a] → Expr [a]
consE = cExpr2 cons
```

We can use the function *foldr* to further define a function to facilitate building a list of arbitrary many elements.

```
mkEList :: Typeable a ⇒ [Expr a] → Expr [a]
mkEList = foldr consE nilE
```

Here we need define a empty list expression.

```
nilE :: Typeable a ⇒ Expr [a]
nilE = mkExpr "[]"
```

We can use the function *mkExpr* to build 1, 2, 3, 4, 5, 6 and [].

```
e1, e2, e3, e4, e5, e6 :: Expr Int
e1 = mkExpr "1"
e2 = mkExpr "2"
e3 = mkExpr "3"
e4 = mkExpr "4"
e5 = mkExpr "5"
e6 = mkExpr "6"
```

Now we can build subexpressions $[1, 2, 3]$, $[2, 3, 4]$, $[3, 4, 5]$ and $[6]$.

```
e123, e234, e345, e6nil :: Expr [Int]
e123 = mkEList [e1, e2, e3]
e234 = mkEList [e2, e3, e4]
e345 = mkEList [e3, e4, e5]
e6nil = mkEList [e6]
```

Thus, we can build the expression $[[1, 2, 3], [2, 3, 4], [3, 4, 5], [6]]$ now.

```
e :: Expr [[Int]]
e = mkEList [e123, e234, e345, e6nil]
```

We then build the pattern $(x : xs : (y : ys : zss))$.

```
px, py :: Pat Int
px = mkPVar "x"
py = mkPVar "y"

pxs, pxxs, pys, pyys :: Pat [Int]
```

$$
\begin{aligned}
&\mathit{pxs} = \mathit{mkPVar}\ \texttt{"xs"}\\
&\mathit{pxxs} = \mathit{consP}\ \mathit{px}\ \mathit{pxs}\\
&\mathit{pys} = \mathit{mkPVar}\ \texttt{"ys"}\\
&\mathit{pyys} = \mathit{consP}\ \mathit{py}\ \mathit{pys}\\
&\mathit{pzss}, \mathit{pyszss}, \mathit{pyyszss} :: \mathit{Pat}\ [[\mathit{Int}]]\\
&\mathit{pzss}\ \ = \mathit{mkPVar}\ \texttt{"zss"}\\
&\mathit{pyszss} = \mathit{consP}\ \mathit{pys}\ \mathit{pzss}\\
&\mathit{pyyszss} = \mathit{consP}\ \mathit{pyys}\ \mathit{pzss}\\
&p :: \mathit{Pat}\ [[\mathit{Int}]]\\
&p = \mathit{consP}\ \mathit{pxxs}\ \mathit{pyyszss}
\end{aligned}
$$

We also need build the matching $\urcorner \mathsf{xs} : (\mathsf{ys} : \mathsf{zss}) \ulcorner$.

$$
\begin{aligned}
&\mathit{exs}, \mathit{eys} :: \mathit{Expr}\ [\mathit{Int}]\\
&\mathit{exs} = \mathit{mkEVar}\ \texttt{"xs"}\\
&\mathit{eys} = \mathit{mkEVar}\ \texttt{"ys"}\\
&\mathit{ezss}, \mathit{eyszss}, \mathit{exsyszss} :: \mathit{Expr}\ [[\mathit{Int}]]\\
&\mathit{ezss}\ \ = \mathit{mkEVar}\ \texttt{"zss"}\\
&\mathit{eyszss} = \mathit{consE}\ \mathit{eys}\ \mathit{ezss}\\
&\mathit{exsyszss} = \mathit{consE}\ \mathit{exs}\ \mathit{eyszss}\\
&m :: \mathit{Match}\ [[\mathit{Int}]]\\
&m = \mathit{Return}\ \mathit{exsyszss}
\end{aligned}
$$

Finally, we can build the matching

$$[[1,2,3],[2,3,4],[3,4,5],[6]] \triangleright (\mathsf{x} : \mathsf{xs} : (\mathsf{y} : \mathsf{ys} : \mathsf{zss})) \mapsto \urcorner xs : (ys : zss) \ulcorner$$

and then the expression

$$\{\!|\ [[1,2,3],[2,3,4],[3,4,5],[6]] \triangleright (\mathsf{x} : \mathsf{xs} : (\mathsf{y} : \mathsf{ys} : \mathsf{zss})) \mapsto \urcorner xs : (ys : zss) \ulcorner\ |\!\} .$$

$$
\begin{aligned}
&\mathit{epm} :: \mathit{Match}\ [[\mathit{Int}]]\\
&\mathit{epm} = \mathit{Supply}\ e\ \$\ \mathit{PMatch}\ p\ m\\
&\mathit{epmE} :: \mathit{Expr}\ [[\mathit{Int}]]\\
&\mathit{epmE} = \mathit{MExpr}\ \mathit{epm}
\end{aligned}
$$

Using the text representation functions of PMC in the appendix B.1, we can show it in GHCi, GHC's interactive environment.

```
*PMCExample> epmE
{[[1,2,3],[2,3,4],[3,4,5],[6]] >> (x:xs:(y:ys:zss)) => |xs:(ys:zss)|}
```

2.4.2 Example 2

We first compare the following two case expressions in Haskell:

```
example2a = (λarg1 arg2 → case arg1 of
  (x : xs) → case arg2 of
    [] → 1
    _ → error "error: matching failure!"
  ys → case arg2 of
    (v : vs) → 2
    _ → error "error: matching failure!"
  )
```

and

```
example2b = (λarg1 arg2 → case (arg1, arg2) of
  (x : xs, []) → 1
  (ys, v : vs) → 2
  _ → error "error: matching failure!"
  )
```

When supplied with the arguments [2,3] [3,4], *example2a* return 1 but *example2b* return 2, which is because that case expressions do not have backtracking mechanism. When the second argument [3,4] mismatches against [], *example2a* cannot backtrack to match the first argument against the next pattern. *example2b* uses the method of paralleling all arguments to avoid the necessity of backtracking and is a "more successful" pattern matching.

Naturally, the pattern matching of PMC corresponds to the second "more successful" pattern matching. Therefore, we choose to translate the following case expression, which is based on the above second example *example2b*, into the PMC terms:

```
example2 = (λarg1 arg2 → case (arg1, arg2) of
  (x : xs, []) → 1
  (ys, v : vs) → 2
  _ → error "error: matching failure!"
  ) ⊥ (3 : [])
```

The corresponding PMC term is as following:

$$\{\!|\, ((\mathrm{x} : \mathrm{xs}) \mapsto [\,] \mapsto \rceil 1 \lceil) \,\|\, (\mathrm{ys} \mapsto (\mathrm{v} : \mathrm{vs}) \mapsto \rceil 2 \lceil) \,|\!\} \perp (3 : [\,])$$

It is easy to see that compared with case expressions, the PMC pattern matching saves variable names *arg1 and arg2* and always leads to the "more successful" pattern matching.

Actually, the PMC expression was first introduced in [11] to demonstrate the different reduction sequences of the two calculi $\mathsf{PMC}_{\not\leftrightarrow}$ and $\mathsf{PMC}_{\oslash}$. We will use the type-indexed reduction system to implement the two reduction sequences in 3.3.

We first build the patterns $(x : xs)$, $[\,]$, ys and $(v : vs)$.

```
xP :: forall a ∘ Typeable a ⇒ Pat a
xP  = mkPVar "x"
xsP, xxsP, ysP :: forall a ∘ Typeable a ⇒ Pat [a]
```

```
xsP = mkPVar "xs"
xxsP = consP xP xsP
ysP = mkPVar "ys"
vP :: Pat Int
vP  = mkPVar "v"
nilP, vsP, vvsP :: Pat [Int]
nilP = mkPat "[]"
vsP = mkPVar "vs"
vvsP = consP vP vsP
```

We then build the matching $\{\!|\, 1 \,|\!\}$ and $\{\!|\, 2 \,|\!\}$.

```
r1, r2 :: Match Int
r1 = Return $ mkExpr "1"
r2 = Return $ mkExpr "2"
```

Thus, we can build the matching $((\mathrm{x} : \mathrm{xs}) \mapsto [\,] \mapsto \rceil 1 \lceil) \,\|\, (\mathrm{ys} \mapsto (\mathrm{v} : \mathrm{vs}) \mapsto \rceil 2 \lceil)$ now.

```
l, r :: Match ([Int] → [Int] → Int)
l = PMatch xxsP $ PMatch nilP r1
r = PMatch ysP $ PMatch vvsP r2
pmpm :: Match ([Int] → [Int] → Int)
pmpm = MAlt l r
```

We also need the expressions $\perp$ and $3 : [\,]$.

```
emptyIntList :: Expr [Int]
emptyIntList = Empty
threeNilE :: Expr [Int]
threeNilE = mkEList [e3]
```

Finally, we build the PMC expression

$$\{\!|\, ((\mathrm{x} : \mathrm{xs}) \mapsto [\,] \mapsto \rceil 1 \lceil) \,\|\, (\mathrm{ys} \mapsto (\mathrm{v} : \mathrm{vs}) \mapsto \rceil 2 \lceil) \,|\!\}\ \perp\ (3 : [\,]) \ .$$

```
pmc' :: Expr Int
pmc' = (MExpr pmpm) `Apply` emptyIntList `Apply` threeNilE
```

We build the following PMC expression, which will be used in 3.3.

$$\{\!|\, (\perp \rhd (\mathrm{x} : \mathrm{xs}) \mapsto [\,] \mapsto \rceil 1 \lceil) \,\|\, (\perp \rhd \mathrm{ys} \mapsto (\mathrm{v} : \mathrm{vs}) \mapsto \rceil 2 \lceil) \,|\!\}\ (3 : [\,]) \ .$$

We first build the PMC matching $\{\!|\, (\perp \rhd (\mathrm{x} : \mathrm{xs}) \mapsto [\,] \mapsto \rceil 1 \lceil) \,\|\, (\perp \rhd \mathrm{ys} \mapsto (\mathrm{v} : \mathrm{vs}) \mapsto \rceil 2 \lceil) \,|\!\}$.

```
pmpm' :: Match ([Int] → Int)
pmpm' = MAlt (Supply emptyIntList l) (Supply emptyIntList r)
```

We then build the PMC expression.

$pmc :: Expr\ Int$
$pmc = MExpr\ \$\ Supply\ threeNilE\ pmpm'$

Finally, using the text representation functions of PMC in the appendix B.1, we can show then in GHCi, GHC's interactive environment.

```
*RedExample> pmc'
{(x:xs) => [] => |1| || ys => (v:vs) => |2|} empty [3]

*RedExample> pmc
{[3] >> (empty >> (x:xs) => [] => |1| || empty >> ys => (v:vs) => |2|)}
```

2.4.3 Example 3

The five expression examples in the subsection demonstrate how to build PMC expressions. The examples will also be evaluated in 4.9 to demonstrate the bimonadic semantics of PMC.

Before giving the examples in the subsection, we define a constructor "(,)".

$pair :: forall\ a\ b \circ (Typeable\ a, Typeable\ b) \Rightarrow C2\ a\ b\ (a, b)$
$pair = mkC2$ "(,)"

We define two functions to facilitate building a pair, respectively for patterns and expressions.

$pairP :: (Typeable\ a, Typeable\ b) \Rightarrow Pat\ a \rightarrow Pat\ b \rightarrow Pat\ (a, b)$
$pairP = cPat2\ pair$
$pairE :: (Typeable\ a, Typeable\ b) \Rightarrow Expr\ a \rightarrow Expr\ b \rightarrow Expr\ (a, b)$
$pairE = cExpr2\ pair$

The first expression example defines λ-calculus term in Haskell:

$ex1' = (\lambda(y : []) \rightarrow y)\ [5]$

which can be translated into the PMC expression $\{\!|\ [5] \rhd y : [] \Mapsto \rceil y \lceil\ |\!\}$.

$ex1 :: Expr\ Int$
$ex1 = MExpr\ \$\ Supply\ list1\ \$\ PMatch\ consyNil\ \$\ Return\ (mkEVar$ "y")
$headE :: Expr\ ([Int] \rightarrow Int)$
$headE = MExpr\ (PMatch\ consyNil\ \$\ Return\ (mkEVar$ "y"))
$list1 :: Expr\ [Int]$
$list1 = consE\ (mkExpr$ "5" $:: Expr\ Int)\ (mkExpr$ "[]" $:: Expr\ [Int])$
$consyNil :: Pat\ [Int]$
$consyNil = consP\ (mkPVar$ "y") $nilP$

we can show it in GHCi.

```
*EvalExample> ex1
{[5] >> (y:[]) => |y|}
```

The second expression example defines λ-calculus term in Haskell:

$ex2' = (\lambda(y : zs) \rightarrow zs)\ [5]$

which can be translated into the PMC expression $⦃\, [5] \rhd \mathsf{y : zs} \Mapsto \rceil \mathsf{zs} \lceil\, ⦄$.

$ex2 :: Expr\ [Int]$
$ex2 = MExpr\ \$\ Supply\ list1\ \$\ PMatch\ consyzs\ \$\ Return\ (mkEVar\ "zs")$
$consyzs :: Pat\ [Int]$
$consyzs = consP\ (mkPVar\ "y")\ (mkPVar\ "zs")$

we can show it in GHCi.

```
*EvalExample> ex2
{[5] >> (y:zs) => |zs|}
```

The third expression example defines λ-calculus term in Haskell:

$ex3' = (\lambda(x : (y : [])) \rightarrow y)\ ((+\!\!+)\ [5]\ [42])$

which can be translated into the PMC expression $⦃\, (++)\ [5]\ [42] \rhd (\mathsf{x} : (\mathsf{y} : [])) \Mapsto \rceil \mathsf{y} \lceil\, ⦄$.

$ex3 :: Expr\ Int$
$ex3 = MExpr\ \$\ Supply\ concList1List2\ \$\ PMatch\ consxyNil\ \$\ Return\ (mkEVar\ "y")$
$concList1List2 :: Expr\ [Int]$
$concList1List2 = Apply\ concList1\ \$$
  $consE\ (mkExpr\ "42" :: Expr\ Int)\ (mkExpr\ "[]" :: Expr\ [Int])$
$concList1 :: Expr\ ([Int] \rightarrow [Int])$
$concList1 = Apply\ (mkEVar\ "++" :: Expr\ ([Int] \rightarrow [Int] \rightarrow [Int]))\ list1$
$consxyNil = consP\ (mkPVar\ "x")\ consyNil$

we can show it in GHCi.

```
*EvalExample> ex3
++ [5] [42] >> (x:(y:[])) => |y|
```

The last expression example defines λ-calculus term in Haskell:

$ex4' = (\lambda(x : (y : zs)) \rightarrow y)\ ((+\!\!+)\ [5]\ [42])$

which can be translated into the PMC expression $⦃\, (++)\ [5]\ [42] \rhd (\mathsf{x} : (\mathsf{y} : \mathsf{zs})) \Mapsto \rceil \mathsf{y} \lceil\, ⦄$.

$ex4 :: Expr\ Int$
$ex4 = MExpr\ \$\ Supply\ concList1List2\ \$\ PMatch\ consxyzs\ \$\ Return\ (mkEVar\ "y")$
$consxyzs :: Pat\ [Int]$
$consxyzs = consP\ (mkPVar\ "x")\ consyzs$

we can show it in GHCi.

```
*EvalExample> ex4
++ [5] [42] >> (x:(y:zs)) => |y|
```

2.4.4 Example 4

The example in this subsection is a λ-calculus fixed-point function in Haskell:

$\mathit{returnOne}' = \lambda x \rightarrow 1$

which can be translated into a PMC expression $\{\!\!| x \rhd \rceil 1 \lceil |\!\!\}$.

The example function expression has a fixed-point 1 and will be used as an example of evaluating a fixed-point function in 3.5.

```
returnOne :: Expr (Int → Int)
returnOne = MExpr $ PMatch (mkPVar "x" :: Pat Int) $ Return (mkExpr "1" :: Expr Int)
```

It is shown in GHCi as follows.

```
*EvalExample> returnOne
{x => |1|}
```

2.4.5 Example 5

The following example define a case expression:

```
scopeGHC = case (5, 42) of
  (x, y) →
    case 22 of
      y → x + y
```

which can be translated into a PMC expression $\{\!\!| (\mathsf{x},\mathsf{y}) \mapsto \mathsf{y} \mapsto \rceil (+)\ \mathsf{x}\ \mathsf{y} \lceil |\!\!\}\ (5,42)\ 22$.

We build the expression in type-indexed PMC as follows.

```
scope :: Expr Int
scope = (MExpr (pairxy ‘PMatch‘ (y ‘PMatch‘ (Return plusxy))))
      ‘Apply‘ pair542 ‘Apply‘ e22
  where pairxy :: Pat (Int, Int)
    pairxy = (pairP (mkPVar "x" :: Pat Int) y)
    y :: Pat Int
    y = mkPVar "y"
    plusxy :: Expr Int
    plusxy = (mkEVar "+" :: Expr (Int → Int → Int))
      ‘Apply‘ (mkEVar "x" :: Expr Int)
      ‘Apply‘ (mkEVar "y" :: Expr Int)
    pair542 :: Expr (Int, Int)
    pair542 = (pairE (mkExpr "5" :: Expr Int) (mkExpr "42" :: Expr Int))
    e22 :: Expr Int
    e22 = mkExpr "22"
```

We can show it in GHCi.

```
*NormaliseExample> scope
{(x,y) => y => |+ x y|} (5,42) 22
```

2.5 Summary

In this chapter, we use the technique of type-indexed expressions to implement type-indexed syntax of PMC. By taking advantage of the technique, the type-indexed syntax of PMC mirrors the original theoretic definition in [11], which also makes it easy to show that the type-indexed PMC holds all the properties of the theoretic definition. The examples in the last section of this chapter show that the type-indexed implementation has the same expressive power as the theoretic definition.

Our experiences show that using type-indexed expressions in our implementation has led to not only more robust but also more efficient programs. On the one hand, the obvious advantage of using the technique is that the Haskell type system gives the validity of syntactic structures of the type-indexed PMC for free. On the other hand, the type-indexed implementation models the syntax of PMC with more accuracy and directness.

Chapter 3

Operational Semantics of PMC

This chapter includes our type-indexed implementation of operational semantics of PMC, which is introduced by Kahl in [11, 13].

The operational semantics of PMC has been briefly introduced in the subsection 1.2.2. The chapter provides a type-indexed implementation of the operational semantics of PMC. We first implement substitutions using TMap, which has been introduced in the subsection 1.4.3, in the section 3.1. Thus, in the section 3.2 we can use substitutions to implement type-indexed reduction rules in the section 3.2. We give reduction examples in the section 3.3, where reduction sequences are also provided to demonstrate the difference of the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\Leftarrow}$. We then implement normalisation in the section 3.4, which includes a leftmost-outermost strategy in the subsection 3.4.1 and a deterministic normalising strategy in the subsection 3.4.2. Finally, we give normalisation examples.

3.1 Substitutions

The module *Subst* includes a type-indexed implementation of substitution.

The module also imports α-conversion in the appendix D.1 to implement variable scoping.

The module imports *TIMap*, which is introduced in 1.4.3, as substitutions to help implement the reduction rule $(\rhd v)$ in the section 3.2, which corresponds to α-conversion in typed λ-calculus.

```
import TIMap as Su
```

The module also imports *AlphaConversion* in the appendix D to implement variable scoping.

```
import AlphaConversion
```

A value of type *Subst* is a type-indexed mapping from a value of type *Var a* to a value of type *Expr a*.

```
type Subst = Su.TIMap Var Expr
```

We define the type constructor *SubstFct* for convenience.

```
type SubstFct s = Subst → (forall a ∘ Typeable a ⇒ Q (s a))
```

Here the type constructor *Q* is defined in the appendix C.2:

```
type Q a = a -> Maybe a
```

A substitution function of type *SubstFct s* takes a substitution and a value of *s a* If the substitution process succeeds, it will return a value of type *Maybe* (*s a*), like Just *v*, where *v* is of type *s a*. Otherwise, it will return *Nothing*.

We define substitution of a single variable with an expression or pattern as special case of general substitution:

```
substitute :: (Ord (Var a), Typeable a) ⇒
  Var a → Expr a → (forall b ∘ Typeable b ⇒ Q (Match b))
substitute v e = substM (Su.singleton v e)
```

We define the substitution function *substE* for PMC expressions.

```
substE :: SubstFct Expr
substE su (EVar v)      = Su.lookup v su
substE su (ConstrExpr ca) = fmap ConstrExpr (substECA su ca)
  where
    substECA :: SubstFct (ConstrApp Expr)
    substECA su (Constr c) = Nothing
    substECA su (ConstrApply ca e) =
      qjoin ConstrApply (substECA su) (substE su) ca e
substE su (Apply e1 e2) = qjoin Apply (substE su) (substE su) e1 e2
substE su (MExpr m) = fmap MExpr $ substM su m
substE su Empty       = Just Empty
substE su EFix        = Just EFix
```

We define the substitution function *substM* for PMC matchings.

```
substM :: SubstFct Match
substM su (Return e) = fmap Return $ substE su e
substM su Fail         = Just Fail
substM su (PMatch p m) = let (p', m', su') = alphaP p m su
  in fmap (PMatch p') $ substM su' m'
substM su (Supply e m) = qjoin Supply (substE su) (substM su) e m
substM su (MAlt m1 m2) = qcomb MAlt (substM su) m1 m2
```

Here, the function *alphaP* is an α-conversion function. When the bound variables of the argument patterns occur in the range of the subsititutions, the function *alphaP* exploits a strategy to rename variable names to avoid name clashes.

The detailed implementation and examples of α-conversion in the type-indexed setting are included in the appendix D.1.

3.2 Reduction Rules

This module *Rule* provides an implementation of all PMC Reduction Rules. The explanation of the reduction rules has been directly taken from [11]. The rewriting system PMC consists of:

- nine first-order term rewriting rules,
- two rule-schemata ($\oslash \triangleright c$) and ($d \triangleright c$) — parameterised by the constructors and the arities — that involve the binding constructor $\mapsto$, but not any bound variables,
- the second-order rule ($\triangleright v$) involving substitution, and
- the second-order rule schema ($c \triangleright c$) for pattern matching that re-binds variables.

We define two type synonyms for convenience.

type *TrafoE* = *Trafo Expr*
type *TrafoM* = *Trafo Match*

A reduction rule of type *TrafoE* is a relation between two PMC expressions and correspondingly, a reduction rule of type *TrafoM* is a relation between two PMC matchings.
The type constructor *Trafo* in the above definitions is defined in the appendix C.3:

```
type Trafo s = forall a. (Typeable a) => Q (s a)
```

The definition of *Q* has been introduced in the section 3.1.

3.2.1 PMC Expressions Reduction Rules

All standard reduction rules of rewriting expressions here are first order.
A matching abstraction where all alternatives fail represents an ill-defined case — this is the motivation for the introduction of the empty expression into our language:

$$\{\!|\, ↯ \,|\!\} \xrightarrow[\mathsf{E}]{} \oslash \qquad (\{\!|\, ↯ \,|\!\})$$

redMExprFail :: *TrafoE*
redMExprFail (*MExpr Fail*) = *Just Empty*
redMExprFail _ = *Nothing*

Matching abstractions built from expression matchings are equivalent to the contained expression:

$$\{\!|\, \lceil e \rceil \,|\!\} \xrightarrow[\mathsf{E}]{} e \qquad (\{\!|\, \lceil\ \rceil \,|\!\})$$

```
redMExprReturn :: TrafoE
redMExprReturn (MExpr (Return e)) = Just e
redMExprReturn _                  = Nothing
```

Application of a matching abstraction reduces to argument supply inside the abstraction:

$$\{\!|\, m \,|\!\}\ a \ \xrightarrow[\mathsf{E}]{}\ \{\!|\, a \rhd m \,|\!\} \tag{$\{\!|\,|\!\}@$}$$

```
redApplyMExpr (Apply (MExpr m) a) = Just $ MExpr (Supply a m)
redApplyMExpr _ = Nothing
```

No matter which of our two interpretations of the empty expression we choose, it absorbs arguments when used as function in an application:

$$\oslash\ e \ \xrightarrow[\mathsf{E}]{}\ \oslash \tag{$\oslash@$}$$

```
redApplyEmpty :: TrafoE
redApplyEmpty (Apply Empty e) = Just Empty
redApplyEmpty _ = Nothing
```

3.2.2 First-order PMC Matchings Reduction Rules

The following are first-order standard reduction rules of rewriting matchings.

Failure is the (left) unit for $\mathbf{|}$; this enables discarding of failed alternatives and transfer of control to the next alternative:

$$\lightning\,\mathbf{|}\, m \ \xrightarrow[\mathsf{M}]{}\ m \tag{$\lightning\mathbf{|}$}$$

```
redMAltFail :: TrafoM
redMAltFail (MAlt Fail m) = Just m
redMAltFail _ = Nothing
```

Expression matchings are left-zeros for $\mathbf{|}$:

$$\rceil e \lceil\,\mathbf{|}\, m \ \xrightarrow[\mathsf{M}]{}\ \rceil e \lceil \tag{$\rceil\lceil\mathbf{|}$}$$

```
redMAltReturn :: TrafoM
redMAltReturn (MAlt (Return e) m) = Just $ Return e
redMAltReturn _ = Nothing
```

Argument supply to an expression matching reduces to function application inside the expression matching:

$$a \rhd \rceil e \lceil \ \xrightarrow[\mathsf{M}]{}\ \rceil e\ a \lceil \tag{$\rhd\rceil\lceil$}$$

```
redSupplyReturn :: TrafoM
redSupplyReturn (Supply a (Return e)) = Just $ Return (Apply e a)
redSupplyReturn _ = Nothing
```

The matching failure absorbs argument supply:

$$e \rhd \lightning \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad (\rhd\lightning)$$

```
redSupplyFail :: TrafoM
redSupplyFail (Supply e Fail) = Just Fail
redSupplyFail _ = Nothing
```

Argument supply distributes into alternatives:

$$e \rhd (m_1 \,|\, m_2) \quad \xrightarrow[\mathsf{M}]{} \quad (e \rhd m_1) \,|\, (e \rhd m_2) \qquad (\rhd|)$$

```
redSupplyMAlt :: TrafoM
redSupplyMAlt (Supply e (MAlt m1 m2)) = Just $ MAlt (Supply e m1) (Supply e m2)
redSupplyMAlt _ = Nothing
```

3.2.3 Second-order PMC Matchings Rules or Rule Schemas

Everything matches a variable pattern; this matching gives rise to substitution:

$$a \rhd v \mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad m[v\backslash a] \qquad (\rhd v)$$

```
redSupplyPMatchVarPat :: TrafoM
redSupplyPMatchVarPat (Supply a (PMatch (VarPat v) m)) = Just $
  qtry (substitute v a) m
redSupplyPMatchVarPat _ = Nothing
```

Matching constructors match, and the proviso in the following rule can always be ensured via α-conversion (for this rule to make sense, linearity of patterns is important):

$$c(e_1, \ldots, e_n) \rhd c(p_1, \ldots, p_n) \mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad e_1 \rhd p_1 \mapsto \cdots e_n \rhd p_n \mapsto m$$
$$\text{if } \mathsf{FV}(c(e_1, \ldots, e_n)) \cap \mathsf{FV}(c(p_1, \ldots, p_n)) = \{\} \qquad (c \rhd c)$$

Matching of different constructors fails:

$$d(e_1, \ldots, e_k) \rhd c(p_1, \ldots, p_n) \mapsto m \quad \xrightarrow[\mathsf{M}]{} \quad \lightning \qquad \text{if } c \neq d \text{ or } k \neq n \qquad (d \rhd c)$$

```
redConstrSupplyPMatch :: TrafoM
redConstrSupplyPMatch (Supply (ConstrExpr e) (PMatch (ConstrPat p) m)) = do
  f ← matchConstrApp' p e
  return $ f m
redConstrSupplyPMatch _ = Nothing
```

The following functions take a constructor pattern and match its first level against a constructor expression — success means equal types and therefore equal number of arguments, and equal constructor.

In case of success, the wrapping function for the rearrangement needed for the matching rule $(c \rhd c)$ is returned.

```
matchConstrApp' :: (Typeable a, Typeable b, Eq a, Typeable c) ⇒
  ConstrApp Pat a → ConstrApp Expr b → Maybe (Match c → Match c)
matchConstrApp' p e = cast e ≫= matchConstrApp p
matchConstrApp :: (Typeable a, Eq a, Typeable c) ⇒
  ConstrApp Pat a → ConstrApp Expr a → Maybe (Match c → Match c)
matchConstrApp (Constr c) (Constr c') = if c ≡ c' then Just id else Nothing
matchConstrApp (ConstrApply cap p) (ConstrApply cae e) = do
  e' ← cast e
  wrap ← matchConstrApp' cap cae
  return (wrap ∘ (e'`Supply`) ∘ (p`PMatch`))
matchConstrApp (ConstrApply c p) (Constr c') =
  error "error: Cannot happen in this kind of type-indexed expressions"
```

For the case where an empty expression is matched against a constructor pattern, we consider two different right-hand sides:

- With the first rule, corresponding to interpreting the empty expression as equivalent to non-termination, constructor pattern matchings are strict in the supplied argument:

  $$\oslash \rhd c(p_1, \ldots, p_n) \mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \upharpoonleft \oslash \upharpoonright \qquad (\oslash \rhd c \rightarrow \oslash)$$

  The calculus including this rule will be denoted $\mathsf{PMC}_{\oslash}$.

  ```
  redSupplyEmptyEMPTY :: TrafoM
  redSupplyEmptyEMPTY (Supply Empty (PMatch p m)) = Just $ Return Empty
  redSupplyEmptyEMPTY _ = Nothing
  ```

- With the second rule, corresponding to interpreting the empty expression as propagating the exception of matching failure, that failure is “resurrected”:

  $$\oslash \rhd c(p_1, \ldots, p_n) \mapsto m \quad \underset{\mathsf{M}}{\longrightarrow} \quad \lightning \qquad (\oslash \rhd c \rightarrow \lightning)$$

  The calculus including this rule will be denoted $\mathsf{PMC}_{\lightning}$; in this calculus, it is not possible to give $\oslash$ the same semantics as expressions without normal form.

```
redSupplyEmptyFAIL :: TrafoM
redSupplyEmptyFAIL (Supply Empty (PMatch p m)) = Just $ Fail
redSupplyEmptyFAIL _ = Nothing
```

For statements that hold in both $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$, we let $(\oslash \rhd c)$ stand for $(\oslash \rhd c \rightarrow \oslash)$ in $\mathsf{PMC}_{\oslash}$ and for $(\oslash \rhd c \rightarrow \lightning)$ in $\mathsf{PMC}_{\lightning}$.

3.2.4 Fixed-point Reduction Rules

The fixed-point combinator reduces via the fixed-point equation:

$$\mathsf{fix}\ e \ \xrightarrow[\mathsf{E}]{}\ e\ (\mathsf{fix}\ e) \qquad (\mathsf{fix}\ @)$$

We implement the fixed-point reduction rule as follows:

```
redApplyEFix :: TrafoE
redApplyEFix e@(Apply EFix f) = Just $ Apply f e
redApplyEFix _ = Nothing
```

3.2.5 Unioning All Reduction Rules

All the PMC reduction rules constitute the rewriting system PMC, which is intended as a basis for the operational semantics of functional programs.

All expressions reduction rules are united to constitute a resulting expression reduction rule *redExpr* for both $(\oslash \rhd c \rightarrow \oslash)$ and $(\oslash \rhd c \rightarrow \lightning)$.

```
redExpr :: TrafoE
redExpr = redMExprFail `alt` redMExprReturn `alt`
  redApplyMExpr `alt` redApplyEmpty `alt`
  redApplyEFix
```

All matchings reduction rules except $(\oslash \rhd c \rightarrow \oslash)$ and $(\oslash \rhd c \rightarrow \lightning)$ are united to constitute a matching reduction rule *redMatch*.

```
redMatch :: TrafoM
redMatch = redMAltFail `alt` redMAltReturn `alt`
  redSupplyReturn `alt` redSupplyFail `alt`
  redSupplyMAlt `alt` redSupplyPMatchVarPat `alt`
  redConstrSupplyPMatch
```

The above matching reduction rule *redMatch* and the matching reduction rule *redSupplyEmptyEMPTY* representing $(\oslash \rhd c \rightarrow \oslash)$ can be united to constitute a resulting matching reduction rule for $\mathsf{PMC}_{\oslash}$.

```
redMatchEMPTY :: TrafoM
redMatchEMPTY = redMatch `alt` redSupplyEmptyEMPTY
```

The above matching reduction rule *redMatch* and the matching reduction rule *redSupplyEmptyFAIL* representing $(\oslash \rhd c \longrightarrow \lightning)$ can be united to constitute a resulting matching reduction rule for PMC$_{\lightning}$.

```
redMatchFAIL :: TrafoM
redMatchFAIL = redMatch `alt` redSupplyEmptyFAIL
```

3.2.6 Type-Lost Problem of Implementing Rules Using Rewriting

In the definitions of reduction rules in [11], each rule r is considered to consist of two patterns (either two expression patterns, or two matching patterns), the *left-hand side* of r and the *right-hand side* of r.

In essence, each reduction rule is a rewriting rule. The reduction rules of PMC constitute a rewriting system. Therefore, naturally, we tried to implement the reduction rules using rewriting technique.

Let us directly translate the rewriting process in [2] into our PMC setting: we first match an expression (or a matching) argument with the left-hand side of expression (or matching) reduction rules to get a substitution and then apply this substitution as the environment to substitute the variables in the right-hand side of expression (or matching) reduction rules to get a new expression (or matching). The resulting expression (or matching) is the result of applying the expression (or matching) reduction rule to the initial expression (or matching).

In order to implement a substitution, which is a mapping from expression variables to expressions or from matching variables to matchings, we have to add a definition of matching variables into the definition of matchings.

```
MVar :: Typeable a => Var a -> Match a
```

We need define some type synonyms for convenience.

```
type Q a = a -> Maybe a

type Trafo s = forall a ∘ (Typeable a) => Q (s a)
type TrafoE = Trafo Expr
type TrafoM = Trafo Match

type Subst s = Su.TIMap Var s
type SubstE = Subst Expr
type SubstM = Subst Match
```

The expression substitution function *substE* takes two substitutions as an environment and transforms a expression argument into a new expression.

```
substE :: (SubstE, SubstM) -> TrafoE
substE (suE, suM) (EVar v) = Su.lookup v suE
substE su (MExpr m) =
  case (substM su m) of
    Just m' → Just $ MExpr m'
    _ → Nothing
```

The matching substitution function *substM* takes two substitutions as an environment and transforms a matching argument into a new matching.

```
substM :: (SubstE, SubstM) → TrafoM
substM (suE, suM) (MVar v) = Su.lookup v suM
substM su (Supply e m) =
  case (substE su e) of
    Just e' → case (substM su m) of
      Just m' → Just $ Supply e' m'
      _ → Nothing
    _ → Nothing
```

We proposed the following type definition for reduction rules:

```
type Rule s a = (s a, s a)
```

We took the following rule for example:

$$\{\!| m |\!\} \; a \quad \xrightarrow[\mathsf{E}]{} \quad \{\!| a \triangleright m |\!\} \qquad (\{\!| \; |\!\}@)$$

We can define the rule ($\{\!| \; |\!\}@$) as follows.

```
ruleApplyMExpr :: forall b a ∘ (Typeable a, Typeable (a → b), Typeable b)
  ⇒ Rule Expr b
ruleApplyMExpr = (Apply (MExpr m) e, MExpr $ Supply e m)
  where m :: Match (a → b)
    m = MVar (V "m")
    e :: Expr a
    e = EVar (V "e")
```

Then we need a matching function for expressions to match the left-hand side of reduction rules against the initial expression to produce new substitutions.

```
matchE :: (Typeable a, Ord (Var a)) ⇒
  (SubstE, SubstM) → Expr a → Expr a → Maybe (SubstE, SubstM)
matchE su (Apply e1 e2) (Apply e1' e2') = do
  e1" ← gcast e1'
  su' ← matchE su e1 e1"
  e2" ← gcast e2'
  matchE su' e2 e2"
matchE su (MExpr m1) (MExpr m2) = matchM su m1 m2
matchE (substE, substM) (EVar v) e = Just (Su.singleton v e, substM)
```

Similarly, we need a matching function for matchings to match the left-hand side of reduction rules against the initial matching to produce new substitutions.

```
matchM :: (Typeable a, Ord (Var a)) ⇒
(SubstE, SubstM) → Match a → Match a → Maybe (SubstE, SubstM)
matchM (substE, substM) (MVar v) (m :: Match a1) = Just (substE, Su.singleton v m)
```

Application of a expression reduction rule means matching the left-hand side of reduction rules against the expression argument to get a substitution and then applying the substitution to the right-hand side of reduction rules to get the resulting expression.

```
applyERule :: Typeable a ⇒ Rule Expr a → Expr a → Maybe (Expr a)
applyERule (lhs, rhs) e = do
  (suE, suM) ← matchE (Su.empty, Su.empty) lhs e
  return (qtry (substE (suE, suM)) rhs)
```

In the function *applyERule*, the following backtracking function *qtry* is used.

```
qtry :: Q a → a → a
qtry f x = maybe x id (f x)
```

Now we can test the rewriting system now. We have a PMC expression $\{\!| x \mapsto \rceil x \lceil |\!\}\ 5$, which can obviously be transformed by the expression reduction rule ($\{\!|\ |\!\}$@). We should be able to expect a resulting expression $\{\!| 5 \rhd x \mapsto \rceil x \lceil |\!\}$.

However, when we apply the rule application function *applyERule* to the expression reduction rule ($\{\!|\ |\!\}$@) and the expression $\{\!| x \mapsto \rceil x \lceil |\!\}\ 5$ in GHC v6.5, we met the following type-lost problem:

```
*TypeProblem> applyERule ruleApplyMExpr testRule

<interactive>:1:11:
    Ambiguous type variable 'a' in the constraint:
      'Typeable a' arising from use of 'ruleApplyMExpr' at
      <interactive>:1:11-19
    Probable fix: add a type signature that fixes these type variable(s)
```

The type-lost problem happened because current GHC cannot keep the information about relations of types of two values correctly during function evaluation so that type information is lost during the function is evaluated.

Let me explain more here. In the definition of the function *ruleApplyMExpr* in GHC, although *e* and *m* in the left-hand side *Apply* (*MExpr m*) *e* and the right-hand side *MExpr* $ *Supply e m* of the reduction rule *ruleApplyMExpr* should have the same type, respectively, when evaluation the function *applyERule* on the arguments *ruleApplyMExpr* and *testRule*, the Haskell type system can only express that the left-hand side *Apply* (*MExpr m*) *e* and the right-hand side *MExpr* $ *Supply e m* of the reduction rule *ruleApplyMExpr* have the same type but cannot express that within *Apply* (*MExpr m*) *e* and *MExpr* $ *Supply e m*, the two *e*'s is of type *Expr a* and the two *m*'s is of type *Match* $(a \to b)$. On the contrary, the Haskell type system think that *e* and *m* in the left-hand side *Apply* (*MExpr m*) *e* are of type *Expr a1* and *Match* $(a1 \to b)$ respectively and *e* and *m* in the right-hand side *MExpr* \$ *Supply e m* are of type *Expr a2* and *Match* $(a2 \to b)$ respectively. Thus, the substitutions failed to work because of type inequality.

Although we can restrict the types of the reduction rule explicitly to concrete types to go through with the type-lost problem, the new rewriting system will not be able to work on the rules of polymorphic types, which is not what we expected. Therefore, we have to implement reduction rules in a transformation style in the previous subsections.

Once this type-lost problem is solved in Haskell, we will be able to implement reduction rule using rewriting technique.

3.3 Reduction Examples

This module *RedExample* includes reduction examples, which demonstrate the type-indexed confluent reduction system of PMC.

3.3.1 One-Step Reduction Example

This subsection introduces a simple one-step reduction example. We first define a PMC matching $1 \rhd \mathrm{v} \Mapsto \rceil \mathrm{v} \lceil$ as *eg1*.

```
eg1 :: Match Int
eg1 = Supply (mkExpr "1" :: Expr Int) $
  PMatch (mkPVar "v" :: Pat Int) $
  Return $ (mkEVar "v" :: Expr Int)
```

It is shown in GHCi as follows.

```
*RedExample> eg1
1 >> v => |v|
```

Using a LaTeX generation mechanism provided by W. Kahl, the application of the reduction system to *eg1* gives rise to the following reduction sequence:

$$1 \rhd v \Mapsto \rceil v \lceil$$

$$\xrightarrow[(\rhd v)]{} \rceil 1 \lceil$$

3.3.2 Many-Step Reduction Example

We take the PMC matching

$$[[1,2,3],[2,3,4],[3,4,5],[5]] \rhd \mathsf{x : xs : (y : ys : zss)} \mapsto \rceil xs : (ys : zss)\lceil$$

for example, which is defined as a PMC matching *epm* in 2.4.1.
We show it in GHCi.

```
*NormExample> epm
[[1,2,3],[2,3,4],[3,4,5],[5]] >> x:xs:(y:ys:zss) => |xs:(ys:zss)|
```

Using a LaTeX generation mechanism provided by W. Kahl, the application of the reduction system to *epm* gives rise to the following reduction sequence:

$$[[1,2,3],[2,3,4],[3,4,5],[5]] \rhd x : xs : (y : ys : zss) \mapsto \rceil xs : (ys : zss)\lceil$$

$$\xrightarrow[(c\rhd c)]{\circ} ([1,2,3] \rhd (x : xs) \mapsto [[2,3,4],[3,4,5],[5]] \rhd (y : ys : zss) \mapsto \rceil xs : (ys : zss)\lceil)$$

$$\xrightarrow[(c\rhd c)]{\circ} (1 \rhd x \mapsto [2,3] \rhd xs \mapsto [[2,3,4],[3,4,5],[5]] \rhd (y : ys : zss) \mapsto \rceil xs : (ys : zss)\lceil)$$

$$\xrightarrow[(\rhd v)]{\circ} ([2,3] \rhd xs \mapsto [[2,3,4],[3,4,5],[5]] \rhd (y : ys : zss) \mapsto \rceil xs : (ys : zss)\lceil)$$

$$\xrightarrow[(\rhd v)]{\circ} ([[2,3,4],[3,4,5],[5]] \rhd (y : ys : zss) \mapsto \rceil [2,3] : (ys : zss)\lceil)$$

$$\xrightarrow[(c\rhd c)]{\circ} ([2,3,4] \rhd (y : ys) \mapsto [[3,4,5],[5]] \rhd zss \mapsto \rceil [2,3] : (ys : zss)\lceil)$$

$$\xrightarrow[(c\rhd c)]{\circ} (2 \rhd y \mapsto [3,4] \rhd ys \mapsto [[3,4,5],[5]] \rhd zss \mapsto \rceil [2,3] : (ys : zss)\lceil)$$

$$\xrightarrow[(\rhd v)]{\circ} ([3,4] \rhd ys \mapsto [[3,4,5],[5]] \rhd zss \mapsto \rceil [2,3] : (ys : zss)\lceil)$$

$$\xrightarrow[(\rhd v)]{\circ} ([[3,4,5],[5]] \rhd zss \mapsto \rceil [2,3] : ([3,4] : zss)\lceil)$$

$$\xrightarrow[(\rhd v)]{\circ} \rceil [[2,3],[3,4],[3,4,5],[5]]\lceil$$

The many-step reduction is shown in GHCi as follows.

```
*RedExample> (repeat' redMatch) epm
Just |[[2,3],[3,4],[3,4,5],[5]]|
```

3.3.3 Transformation Rule for Interpretting Operators

We take the operator + for example to demonstrate how to build a transformation rule to interpret operators in our implementation.

The function *intPlus* returns a PMC variable denoting operator +.

```
intPlus :: Var (Int → Int → Int)
intPlus = mkVar' "+"
```

The function *isExprInt* is to determine whether a PMC expression is of type *Expr Int*.

```
isExprInt :: Typeable a ⇒ Expr a → Bool
isExprInt e = typeOf e ≡ typeOf (⊥ :: Expr Int)
```

The function *getInt* is to get *Int* value from a PMC expression of type *Expr Int*.

```
getInt :: Expr Int → Maybe Int
getInt (ConstrExpr (Constr (CResult s))) = case reads s of
  (k, "") : _ → Just k
  _ → Nothing
getInt _ = Nothing
```

Thus, using the above functions, we implement the following rule to interpret operator +.

```
redPlus :: TrafoE
redPlus (Apply (Apply (EVar f) e1) e2) =
case gcast f of
  Nothing → Nothing
  Just f' → if f' ≡ intPlus ∧ isExprInt e1 ∧ isExprInt e2
    then do
      e1' ← gcast e1
      a1 ← getInt e1'
      e2' ← gcast e2
      a2 ← getInt e2'
      return $ mkExpr $ show $ a1 + a2
    else Nothing
redPlus _ = Nothing

redExprWithPlus :: TrafoE
redExprWithPlus = redExpr ‘alt‘ redPlus
```

The following example applies the above reduction rules. At first, we define a PMC expression denoting $1 + 3$

```
onePlusThree :: Expr Int
onePlusThree = (mkEVar "+" :: Expr (Int → Int → Int))
  ‘Apply‘ (mkExpr "1" :: Expr Int)
  ‘Apply‘ (mkExpr "3" :: Expr Int)
```

Then, we apply reduction rule *redExprWithPlus* to this expression.

> *resultOnePlusThree* :: *Maybe* (*Expr Int*)
> *resultOnePlusThree* = *redExprWithPlus onePlusThree*

Thus, we get the reduced result 4.

```
*RedExample> resultOnePlusThree
Just 4
```

3.3.4 Difference Reduction Sequences of the Two Calculi

Here we take the following pattern matching example directly from 5.2 of the PMC paper and implement them in our typed PMC settings to demonstrate different reduction sequences of the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$.

$$\{\!|\, ((x : xs) \mapsto\!\!\!\Rightarrow [\,] \mapsto\!\!\!\Rightarrow \rceil 1 \upharpoonleft) \,\|\, (ys \mapsto\!\!\!\Rightarrow (v : vs) \mapsto\!\!\!\Rightarrow \rceil 2 \upharpoonleft) \,|\!\}\ \bot\ (3 : [\,])$$

If we replace $\bot$ with the empty expression $\oslash$. then we obtain different behaviour according to which interpretation we choose for $\oslash$.

In the section 2.4.2, we have defined the corresponding PMC term *pmc'*, which is shown in GHCi as follows.

```
*RedExample> pmc'
{(x:xs) => [] => |1| || ys => (v:vs) => |2|} empty [3]
```

Although the module *PMCTrafo* of "transformation transformer", which are used in the normalising strategy, is already included in the appendix C.4, we present some transformation transformers here to help implement the reduction sequences in this subsection, for completeness. Every transformation transformer take a "primitive" reduction rule, which is a transformation, and return another new transformation.

The transformation transformer *inApplyL* applies a reduction rule as its first argument to the expression *f* in the expression (*Apply f a*) as its second argument.

> *inApplyL* :: *TrafoE* → *TrafoE*
> *inApplyL t* (*Apply f a*) = *fmap* (*flip Apply a*) \$ *t f*
> *inApplyL t* _ = *Nothing*

The transformation transformer *inMExpr* applies a reduction rule as its first argument to the matching *m* in the expression (*MExpr m*) as its second argument.

> *inMExpr* :: *TrafoM* → *TrafoE*
> *inMExpr t* (*MExpr m*) = *fmap MExpr* \$ *t m*

```
inMExpr t _ = Nothing
```

The transformation transformer *inSupplyR* applies a reduction rule as its first argument to the matching *m* in the matching (*Supply a m*) as its second argument.

```
inSupplyR :: TrafoM → TrafoM
inSupplyR t (Supply a m) = fmap (Supply a) $ t m
inSupplyR t _ = Nothing
```

The transformation transformer *inMAltL* applies a reduction rule as its first argument to the matching *m1* in the matching (*MAlt m1 m2*) as its second argument.

```
inMAltL :: TrafoM → TrafoM
inMAltL t (MAlt m1 m2) = fmap (flip MAlt m2) $ t m1
inMAltL t _ = Nothing
```

Now We can first execute the same reduction sequence for the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\leftsquigarrow}$ to get to *pmc*, which is also defined in the section 2.4.2 and shown in GHCi as follows.

```
*RedExample> pmc
{[3] >> (empty >> (x:xs) => [] => |1| || empty >> ys => (v:vs) => |2|)}
```

We can implement the reduction sequence as follows.

```
stepi, stepii, stepiii :: TrafoE
stepi   = inApplyL redApplyMExpr
stepii  = redApplyMExpr
stepiii = inMExpr (inSupplyR redSupplyMAlt)
```

Using a LaTeX generation mechanism provided by W. Kahl, the application of the reduction system to *pmc'* gives rise to the following reduction sequence:

$$\{\!|\,(((x : xs) \Mapsto [\,] \Mapsto \lceil 1 \rceil) \,\|\, (ys \Mapsto (v : vs) \Mapsto \lceil 2 \rceil))\,|\!\}\ \bot\ (3 : [\,])$$

$$\xrightarrow[(\{\!|@|\!\})]{\circ} \{\!|\,((\oslash \rhd (x : xs) \Mapsto [\,] \Mapsto \lceil 1 \rceil) \,\|\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \lceil 2 \rceil))\,|\!\}\ (3 : [\,])$$

$$\xrightarrow[(\{\!|@|\!\})]{\circ} \{\!|\,(3 : [\,]) \rhd ((\oslash \rhd (x : xs) \Mapsto [\,] \Mapsto \lceil 1 \rceil) \,\|\, (\oslash \rhd ys \Mapsto (v : vs) \Mapsto \lceil 2 \rceil))\,|\!\}$$

Now we get to the expression *pmc*.

We implement the reduction sequence in $\mathsf{PMC}_{\oslash}$ as follows:

```
step1, step2, step3, step4, step5 :: TrafoE
step1 = inMExpr (inSupplyR (inMAltL redSupplyEmptyEMPTY))
step2 = inMExpr (inSupplyR redMAltReturn)
step3 = inMExpr (redSupplyReturn)
step4 = redMExprReturn
step5 = redApplyEmpty
```

Using a LaTeX generation mechanism provided by W. Kahl, the application of the reduction system to *pmc* gives rise to the following reduction sequence:

$$\begin{aligned}
&\{\!|\,(3:[\,]) \rhd ((\oslash \rhd (x:xs) \Mapsto [\,] \Mapsto \lceil 1 \rceil) \mid (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \lceil 2 \rceil))\,|\!\}\\
&\xrightarrow[(\oslash \rhd c \to \oslash)]{} \{\!|\,(3:[\,]) \rhd (\lceil \oslash \rceil \mid (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \lceil 2 \rceil))\,|\!\}\\
&\xrightarrow[(\lceil\rceil \mid)]{} \{\!|\,(3:[\,]) \rhd \lceil \oslash \rceil\,|\!\}\\
&\xrightarrow[(\rhd \lceil\rceil)]{} \{\!|\,\lceil \oslash\ (3:[\,])\rceil\,|\!\}\\
&\xrightarrow[(\{\!|\lceil\rceil|\!\})]{} \oslash\ (3:[\,])\\
&\xrightarrow[(\oslash @)]{} \oslash
\end{aligned}$$

In $\mathsf{PMC}_{\oslash}$, empty expression propagates.

In $\mathsf{PMC}_{\lightning}$, however, this exception can be caught: matching the empty expression against list construction produces a failure, and the other alternative succeeds.

We implement the reduction sequence in $\mathsf{PMC}_{\lightning}$ as follows:

```
step1', step2', step3', step4', step5', step6', step7' :: TrafoE
step1' = inMExpr (inSupplyR (inMAltL redSupplyEmptyFAIL))
step2' = inMExpr (inSupplyR redMAltFail)
step3' = inMExpr (inSupplyR redSupplyPMatchVarPat)
step4' = inMExpr redConstrSupplyPMatch
step5' = inMExpr redSupplyPMatchVarPat
step6' = inMExpr redSupplyPMatchVarPat
step7' = redMExprReturn
```

Using a LaTeX generation mechanism provided by W. Kahl, the application of the reduction system to *pmc* gives rise to the following reduction sequence:

$$\begin{aligned}
&\{\!|\,(3:[\,]) \rhd ((\oslash \rhd (x:xs) \Mapsto [\,] \Mapsto \lceil 1 \rceil) \mid (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \lceil 2 \rceil))\,|\!\}\\
&\xrightarrow[(\oslash \rhd c \to \lightning)]{} \{\!|\,(3:[\,]) \rhd (\lightning \mid (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \lceil 2 \rceil))\,|\!\}\\
&\xrightarrow[(\lightning \mid)]{} \{\!|\,(3:[\,]) \rhd \oslash \rhd ys \Mapsto (v:vs) \Mapsto \lceil 2 \rceil\,|\!\}\\
&\xrightarrow[(\rhd v)]{} \{\!|\,(3:[\,]) \rhd (v:vs) \Mapsto \lceil 2 \rceil\,|\!\}\\
&\xrightarrow[(c \rhd c)]{} \{\!|\,3 \rhd v \Mapsto [\,] \rhd vs \Mapsto \lceil 2 \rceil\,|\!\}\\
&\xrightarrow[(\rhd v)]{} \{\!|\,[\,] \rhd vs \Mapsto \lceil 2 \rceil\,|\!\}\\
&\xrightarrow[(\rhd v)]{} \{\!|\,\lceil 2 \rceil\,|\!\}\\
&\xrightarrow[(\{\!|\lceil\rceil|\!\})]{} 2
\end{aligned}$$

From the above two reduction sequences in the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$, we can draw a conclusion that $\mathsf{PMC}_{\oslash}$ turns out to be a formalisation of the operational pattern matching semantics of current functional programming languages and $\mathsf{PMC}_{\lightning}$ has a "more successful" evaluation and can be turned into a basis for programming languages implementation.

3.4 Normalisation

The goal of this section is to provide a type-indexed implementation of the normalising strategy of PMC, which is introduced in [11]. The explanation of the normalising strategy has been directly taken from [11]. We first provide a leftmost-outermost strategy based on the transformation rules. We then implement a deterministic normalising strategy for reduction to SHNF.

The module *PMCTrafo* of "transformation transformer" in the appendix C.4 includes the transformation transformers over all the syntactic structures of PMC expressions and matchings. Every transformation transformer take a "primitive" reduction rule, which is a transformation, and return another new transformation. These transformation transformers are implementation basis for the leftmost-outermost strategy in 3.4.1 and the normalising strategy in 3.4.2. Some of the transformation transformers has already been in 3.3.4.

3.4.1 Leftmost-Outermost Strategy

Now we can implement a leftmost-outermost strategy easily, as a byproduct.

The following performs a single *tE* or *tM* transformation at the leftmost-outermost point where this is possible.

```
leftmostOutermostE :: TrafoE → TrafoM → TrafoE
leftmostOutermostE tE tM = tE
  `alt` inConstrExpr (leftmostOutermostE tE tM)
  `alt` inApplyL  (leftmostOutermostE tE tM)
  `alt` inApplyR  (leftmostOutermostE tE tM)
  `alt` inMExpr   (leftmostOutermostM tE tM)
  `alt` inEFix    (leftmostOutermostE tE tM)
leftmostOutermostM :: TrafoE → TrafoM → TrafoM
leftmostOutermostM tE tM = tM
  `alt` inSupplyL (leftmostOutermostE tE tM)
  `alt` inSupplyR (leftmostOutermostM tE tM)
  `alt` inPMatch  (leftmostOutermostM tE tM)
  `alt` inReturn  (leftmostOutermostE tE tM)
  `alt` inMAltL   (leftmostOutermostM tE tM)
  `alt` inMAltR   (leftmostOutermostM tE tM)
```

The leftmost-outermost strategy is deterministic but obviously not normalising. For example, in a PMC matching $a \rhd v \Mapsto m$, if a is non-terminating, then even when m is a constant, the leftmost-outermost strategy applied to $a \rhd v \Mapsto m$ is non-terminating.

3.4.2 Normalising Strategy

This module *Normalise* implements a SHNF strategy and uses the leftmost-outermost strategy to implement a normalisation strategy on top of the SHNF strategy.

PMC is equipped with a normalising strategy of the reduction rules in 3.2, which reduces expressions and matchings to *strong head normal form* (SHNF).

The definition of SHNFs is translated from [21] into the PMC setting for completeness.

The use of *metavariables* is made explicit. For example, e and m in the rule ($\rceil\lceil\|$) are metavariables:

$$\rceil e \lceil \| m \;\underset{\mathsf{M}}{\longrightarrow}\; \rceil e \lceil$$

Each reduction rule r in 3.2 is considered to consist of two patterns (either two expression patterns, or two matching patterns), the *left-hand side* of r and the *right-hand side* of r.

A rule *partially matches* a matching or expression t if its left-hand side partially matches t.

A non-variable matching pattern or expression pattern p *partially matches* a matching, respectively an expression, t, if firstly the top-level syntactic constructions of p and t are the same, and secondly, letting $p_1, \ldots, p_k$ be the immediate constituents of p and $t_1, \ldots, t_k$ the immediate constituents of t, if for each $i : \mathbb{N}$ with $1 \leqslant i \leqslant k$ for which p_i is not a variable, p_i partially matches t_i, or there exists a rule that partially matches t_i.

A term is in *strong head normal form (SHNF)* if no rule partially matches this term.

It is easy to see that a rule that matches an expression, respectively a matching, t, also partially matches t.

Now we give a reduction strategy that reduces expressions and matchings to *strong head normal form* (SHNF) as follows.

With the set of rules defined in 3.2, this definition of SHNFs directly induces the following facts:

- Variable expressions, constructor applications, the empty expression $\oslash$, failure $\lightning$, expression matchings $\rceil e \lceil$, and pattern matchings $p \Mapsto m$ are already in SHNF.

- All rules that have an application $f\ a$ at their top level have a variable for a, and none of these rules has a variable for f, so $f\ a$ is in SHNF if f is in SHNF and $f\ a$ is not a redex.

- A matching abstraction $⦃ m ⦄$ is in SHNF if m is in SHNF unless $⦃ m ⦄$ is a redex for one of the rules $(⦃ ↯ ⦄)$ or $(⦃ 1{\upharpoonright} ⦄)$.
- An alternative $m_1 \mathbin{\|} m_2$ is in SHNF if m_1 is in SHNF unless $m_1 \mathbin{\|} m_2$ is a redex for one of the rules $(↯\|)$ or $(1{\upharpoonright}\|)$, since all alternative rules have a variable for m_2.
- No rules for argument supply $a \rhd m$ have a variable for m, and all rules for argument supply $a \rhd m$ that have non-variable a have a constructor pattern matching for m. Therefore, if $a \rhd m$ is not a redex, it is in SHNF if m is in SHNF and, whenever m is of the shape $c(p_1, \ldots, p_n) \mapsto m'$, a is in SHNF, too.

Due to the homogenous nature of its rule set, PMC therefore has a deterministic strategy for reduction of applications, matching abstractions, alternatives, and argument supply to SHNF:

- For an application $f\ a$, if f is not in SHNF, proceed into f, otherwise reduce $f\ a$ if it is a redex.
- For a matching abstraction $⦃ m ⦄$, if m is not in SHNF, proceed into m, otherwise reduce $⦃ m ⦄$ if it is a redex.
- For an alternative $m_1 \mathbin{\|} m_2$, if m_1 is not in SHNF, proceed into m_1, otherwise reduce $m_1 \mathbin{\|} m_2$ if it is a redex.
- If an argument supply $a \rhd m$ is a redex, reduce it (this is essential for the case where m is of shape $m_1 \mathbin{\|} m_2$, which is not necessarily in SHNF, and $(\rhd\|)$ has to be applied). Otherwise, if m is not in SHNF, proceed into m.

  If m is of the shape $c(p_1, \ldots, p_n) \mapsto m'$, and a is not in SHNF, proceed into a.

Applications, matching abstractions, and alternatives, are redexes only if the selected constituent is in SHNF.

This deterministic strategy for reduction to SHNF induces a deterministic normalising strategy for PMC.

Directly translating the strategy from the PMC paper [11] yields the following transformations that fail on strong head normal forms, and perform a single reduction step towards the SHNF otherwise. For both expressions and matchings, a redex is obviously not a SHNF, so this is tried first. For non-redexes, only a few cases need to be covered:

```
shnfStepE :: TrafoE → TrafoM → TrafoE
shnfStepE redE redM = redE
  `alt` inApplyL (shnfStepE redE redM)
  `alt` inMExpr (shnfStepM redE redM)

shnfStepM :: TrafoE → TrafoM → TrafoM
```

shnfStepM redE redM = *redM*
  ‘*alt*‘ *inMAltL* (*shnfStepM redE redM*)
  ‘*alt*‘ (*inSupplyR guardPMatch* ‘*seq*′‘ *inSupplyL* (*shnfStepE redE redM*))
  ‘*alt*‘ *inSupplyR* (*notGuardPMatch* ‘*seq*′‘ *shnfStepM redE redM*)

Using the leftmost-outermost strategy, we can easily implement a normalisation strategy on top of the SHNF strategy:

nfStepE :: *TrafoE* → *TrafoM* → *TrafoE*
nfStepE redE redM = *shnfStepE redE redM* ‘*alt*‘
  leftmostOutermostE (*shnfStepE redE redM*) (*shnfStepM redE redM*)
nfStepM :: *TrafoE* → *TrafoM* → *TrafoM*
nfStepM redE redM = *shnfStepM redE redM* ‘*alt*‘
  leftmostOutermostM (*shnfStepE redE redM*) (*shnfStepM redE redM*)

3.5 Normalisation Examples

The module *NormaliseExample* includes normalisation examples.

3.5.1 Reduction to SHNF

We first let defaultly the deterministic strategy for reduction to SHNF to take the confluent reduction systems Rule.*redExpr* and Rule.*redMatch* in 3.2.5 as arguments.

shnfStepE0 :: *TrafoE*
shnfStepE0 = *shnfStepE* Rule.*redExpr* Rule.*redMatch*

shnfStepM0 :: *TrafoM*
shnfStepM0 = *shnfStepM* Rule.*redExpr* Rule.*redMatch*

We still take *epm* – $[[1,2,3],[2,3,4],[3,4,5],[5]] \rhd \mathrm{x} : \mathrm{xs} : (\mathrm{y} : \mathrm{ys} : \mathrm{zss}) \mapsto \rceil xs : (ys : zss) \lceil$ for example. We repeat applying the deterministic strategy *shnfStepM0* to *epm* and the resulting matching is $\{\!| [[2,3],[3,4],[3,4,5],[5]] |\!\}$. It is shown in GHCi as follows.

```
*NormaliseExample> (repeat' shnfStepM0) epm
Just |[[2,3],[3,4],[3,4,5],[5]]|
```

We also repeat applying the deterministic strategy *shnfStepE0* to *epmE* –

$$[[1,2,3],[2,3,4],[3,4,5],[5]] \rhd \mathrm{x} : \mathrm{xs} : (\mathrm{y} : \mathrm{ys} : \mathrm{zss}) \mapsto \rceil xs : (ys : zss) \lceil \ .$$

and the resulting matching is

$$[[2,3],[3,4],[3,4,5],[5]] \ .$$

It is shown in GHCi as follows.

```
*NormaliseExample> (repeat' shnfStepE0) epm'
Just [[2,3],[3,4],[3,4,5],[5]]
```

We implement a normalisation strategy on top of the SHNF strategy using the leftmost-outermost strategy.

```
nfStepE0 :: TrafoE
nfStepE0 = nfStepE (leftmostOutermostE Rule.redExpr Rule.redMatch)
  (leftmostOutermostM Rule.redExpr Rule.redMatch)
nfStepM0 :: TrafoM
nfStepM0 = nfStepM (leftmostOutermostE Rule.redExpr Rule.redMatch)
  (leftmostOutermostM Rule.redExpr Rule.redMatch)
```

We also repeat applying the strategy *nfStepE0* to *epmE* –

$$[[1,2,3],[2,3,4],[3,4,5],[5]] \triangleright \mathsf{x} : \mathsf{xs} : (\mathsf{y} : \mathsf{ys} : \mathsf{zss}) \mapsto \rceil xs : (ys : zss) \lceil \ .$$

and the resulting matching is

$$[[2,3],[3,4],[3,4,5],[5]] \ .$$

It is shown in GHCi as follows.

```
*NormaliseExample> (triply (triply (triply nfStepE0))) epmE
Just [[2,3],[3,4],[3,4,5],[5]]
```

The leftmost-outermost strategy is deterministic but obviously not normalising. For example,

3.5.2 Normalisation Examples of PMC Fixed-point Expressions

A fixed point is a value for which a function returns the same value. For example, the fixed point of

$$\texttt{return1} = \lambda \text{x}.1$$

is the value 1. `return1` only has that one fixed point, but functions can have more than one fixed point, e.g. the identity function has all values as fixed points.

In Haskell, "*fix*" is the fixed-point operator. *fix* is defined in Haskell as below:

```
fix :: (a → a) → a
fix f = f $ fix f
```

The above-mentioned funtion *return1* can be defined in Haskell.

```
return1 :: Int → Int
return1 = λx → 1
```

When we apply *fix* to *return1*, GHCi produce 1 as the fixed point of *return1*.

```
*NormExample> fix return1
1
```

Now we define *fix return1* in the type-indexed PMC.

> *fixReturnOne* :: *Expr Int*
> *fixReturnOne* = *Apply EFix returnOne*

Note that the PMC expression *returnOne* is defined in 2.4.4. GHCi can show it as follows.

```
*NormaliseExample> returnOne
{x => |1|}
```

When we apply the deterministic strategy for reduction to SHNF to the PMC expression *fixReturnOne*, the normalisation produces the result 1.

```
*NormaliseExample> (repeat' shnfStepE0) fixReturnOne
Just 1
```

3.6 Summary

The type-indexed implementation of the reduction rules and the normalising strategy of $\text{PMC}_{\oslash}$ constitute the operational semantics of type-indexed $\text{PMC}_{\oslash}$. From its confluent reduction rules and normalising strategy as well as the reduction and normalising sequence of its examples, we can conclude that $\text{PMC}_{\oslash}$ is a concise and elegant formalisation of the operational pattern matching semantics of modern functional programming languages.

By changing the single rule concerned with results of matching failure to “failure as exception”, we have $\text{PMC}_{\lightning}$, which is still confluent and normalising, but results in “more successful” evaluation.

Chapter 4

Bimonadic Semantics of PMC

This chapter includes the formalisation and implementation of the bimonadic semantics of PMC based on Kahl's proposal. We formalise the bimonadic semantics of PMC in the abstract categorical setting. The bimonadic semantics employs two monads to abstract two kinds of computations, which corrrespond to the two syntactic categories of PMC, i.e., *expressions* and *matchings*. In the type-indexed implementation, there are three semantic functions *evalP*, *evalE* and *evalM* that capture the meanings of the three kinds of PMC's syntactic terms, i.e., *patterns*, *expressions* and *matchings*. We also implement type semantics, variable semantics and constructor semantics to interpret the meanings of types, variables and operators, and constructors.

In this chapter, we first introduce categorical notation in the section 4.2 and then use them to formalise the bimonadic semantics of PMC in the section 4.3. In the implementation part, we first implement the type semantics in the bimonadic semantics in the section 4.5. We also implement the variable semantics and constructor semantics in the sections 4.6 and 4.7 respectively. Variables and constructors are two syntactic units of building *patterns* and *expressions* of PMC. We then implement the bimonadic semantics of PMC including the three semantic functions for the three syntactic categories *patterns*, *expressions*, and *matchings* respectively in the section 4.8. Finally, we implement examples to demonstrate the different semantics of the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$.

4.1 Introduction

In the denotational approach, the *effect* of executing a program is studied. The effect means an association between initial states and final states. The idea is to define a *semantic function* for each *syntactic category*. The function maps each *syntactic construct* to a *mathematical object* and describes the effect of executing that construct.

It has long been recognized, however, traditional denotational semantics lacks modularity and reusability [18], which makes difficult applying traditional denotational semantics to the design of realistic programming languages [22]. Moggi [17] took the notion of monad from category theory to structure various notions of computational effect. Liang and Hudak [15] introduced *modular monadic semantics* to take advantage of a monadic approach to structure denotational semantics, which achieves a high level of modularity and extensibility.

In modular monadic semantics, monads and monad transformers are used to separate values from computations. Modular monadic semantics maps *terms* in source languages into *computations* in meta languages, compared with that traditional denotational semantics maps *terms* in source languages into *values* in meta languages.

Kahl proposed the bimonadic semantics of PMC in an abstract categorical setting, which allows to use existing categorical concepts to formalise the bimonadic semantics and guide the implementation elaborating the idea. The formalisation and implementation of the bimonadic semantics of PMC is the main task of this thesis.

In PMC syntactical domain, PMC terms are divided into two major syntactic categories: *expressions* and *matchings*. Correspondingly, in the monadic semantics, Kahl proposed two monads to represent two kinds of computations, one for expressions and the other for matchings respectively. The resulting *bimonadic semantics* allows us to have an axiomatized formulation of well-known programming languages features such as environments.

Since the bimonadic semantics of PMC is defined in an abstract categorical setting, it is necessary to summarise relevant categorical notation in the section 4.2, which will be used in the section 4.3 to formalise the bimonadic semantics of PMC.

4.2 Categorical Notation

Considering the correspondence between *cartesian closed categories* and *typed λ-calculi*, we will define the bimonadic semantics in a *cartesian closed categories* setting. Relevant categorical notation is introduced in this section.

We adopt categorical notations from Barr and Wells' book [3] into our setting.

Over binary products $a \times b$, we define two projections $\mathsf{fst}_{a,b} : a \times b \longrightarrow a$ and $\mathsf{snd}_{a,b} : a \times b \longrightarrow b$. We abuse the notation of pairing $\langle\ \rangle$ to define morphism pairing $\langle f, g\rangle : c \longrightarrow a \times b$ for morphisms $f : c \longrightarrow a$ and $g : c \longrightarrow b$.

For every two objects a and b in a cartesian closed category, there are an exponential object (for "functions from a to b") written $[a \longrightarrow b]$, an "function application" morphism $\mathsf{eval}_{[a\longrightarrow b]}$: $[a \longrightarrow b] \times a \longrightarrow b$, and a currying operation λ that maps every morphism $f : c \times a \longrightarrow b$ to the unique morphism $\lambda f : c \longrightarrow [a \longrightarrow b]$ such that $(\lambda f \times \mathsf{id} a)\ \mathsf{eval}_{[a\longrightarrow b]} = f$.

We define $\Pi i : \mathcal{I} \bullet a(i)$ for the indexed (but not necessarily ordered) product over the *finite* index set $\mathcal{I}$, with component $a(i)$ for index i; the projection to the sub-product indexed by elements of a subset $\mathcal{J} \subseteq \mathcal{I}$ is

$$\mathsf{proj}^{a}_{I \succ J} : (\Pi i : \mathcal{I} \bullet a(i)) \longrightarrow (\Pi i : \mathcal{J} \bullet a(i))$$

(we assume singleton products to be identified with their components: $(\Pi i : \mathcal{J} \bullet a(i)) = a(\mathcal{J})$).

We will write both the object mapping and the morphism mapping of a functor as an application of the functor name (Haskell uses the *Functor* class member function *fmap* for the morphism mapping), so that for a functor H and a morphism $f : a \longrightarrow b$ we have

$$H\ f : H\ a \longrightarrow H\ b$$

A *monad* is a triple $(M, \mathsf{return}^M, \mathsf{join}^M)$ consisting of a endofunctor M together with two natural transformations, which, for readability, we just present as polymorphic morphisms:

$$\begin{aligned} &\mathsf{return}^M_a : a \longrightarrow M\ a \\ &\mathsf{join}^M_a : M\ (M\ a) \longrightarrow M\ a \end{aligned}$$

satisfying the following additional laws:

$$\begin{aligned} &\mathsf{join}^M_a;\ \mathsf{return}^M_a = \mathsf{id}(M\ (M\ a)) \\ &\mathsf{return}^M_a;\ \mathsf{join}^M_a = \mathsf{id}(M\ a) \\ &\mathsf{join}^M_a;\ \mathsf{join}^M_a = \mathsf{join}^M_{M\ a};\ \mathsf{join}^M_a \end{aligned}$$

Every monad M gives rise to a so-called Kleisli category; it has return^M morphisms as identities, and for two of its arrows $f : a \longrightarrow M\ b$ and $g : b \longrightarrow M\ c$, their composition is defined as follows:

$$\begin{aligned} &f \odot_M g : a \longrightarrow M\ c \\ &f \odot_M g = f;\ M\ g;\ \mathsf{join}^M_c \end{aligned}$$

A *monad with zero* has a natural transformation (assume $\mathsf{term}_a : a \longrightarrow \mathbb{1}$ is the unique morphism into the terminal object):

$$zero^M_a : \mathbb{1} \longrightarrow M\ a$$

with

$$\begin{aligned} &M\ \mathsf{zero}^M_a;\ \mathsf{join}^M_a = \mathsf{term}_{M\ \mathbb{1}};\ \mathsf{zero}^M_a \\ &\mathsf{zero}^M_{M\ a};\ \mathsf{join}^M_a = \mathsf{zero}^M_a \end{aligned}$$

In addition, an *additive monad* has a natural transformation

$$\mathsf{plus}^{\mathsf{M}}_{\mathsf{a}} : M\ a \times M\ a \longrightarrow M\ a$$

with (assuming a strict choice of direct products, i.e., with $\mathbb{1} \times A = A$ etc.):

$$\begin{aligned} &(\mathsf{zero}^M_a \times f);\ \mathsf{plus}^M_a = f \\ &(f \times \mathsf{zero}^M_a);\ \mathsf{plus}^M_a = f \\ &(\mathsf{id}M\ a \times \mathsf{plus}^M_a);\ \mathsf{plus}^M_a = (\mathsf{plus}^M_a \times \mathsf{id}M\ a);\ \mathsf{plus}^M_a \end{aligned}$$

As Moggi explains in [16], we need *strong monads* for being able to deal with expression with more than one free variable; a strong monad M has a natural transformation:

$$\mathsf{strengthL}^M_{a,b} : a \times M\ b \longrightarrow M\ (a \times b)$$

called *tensorial strength* satisfying

$$\begin{aligned} &r_{M\ a} = \mathsf{strengthL}^M_{1,a};\ M\ r_a \\ &\mathsf{strengthL}^M_{a\times b,c};\ M\ \mathsf{assoc}_{a,b,c} = \mathsf{assoc}_{a,b,M\ c};\ (\mathsf{id}\ a \times \mathsf{strengthL}^M_{b,c});\ \mathsf{strengthL}^M_{a,b\times c} \\ &\mathsf{return}^M_{a\times b} = (\mathsf{id}a \times \mathsf{return}^M_b);\ \mathsf{strengthL}^M_{a,b} \\ &\mathsf{strengthL}^M_{a,M\ b};\ M\ \mathsf{strengthL}^M_{a,b};\ \mathsf{join}^M_{a\times b} = (\mathsf{id}a \times \mathsf{join}^M_b);\ \mathsf{strengthL}^M_{a,b} \end{aligned}$$

We define the "swapped version" as

$$\begin{aligned}&\mathsf{strengthR}^{M}_{a,b} :: M\ a \times b \rightarrow M\ (a \times b)\\&\mathsf{strengthR}^{M}_{a,b} = \mathsf{swap}_{M\ a,b};\ \mathsf{strengthL}^{M}_{b,a};\ M\ (\mathsf{swap}_{b,a})\end{aligned}$$

This allows us to define:

$$\otimes_M : (M\ a \times M\ b) \rightarrow M\ (a \times b)$$

via ($\mathsf{swap}_{a,b}$ is the isomorphism from $a \times b$ to $b \times a$)

$$\otimes_M = \mathsf{strengthR}^{M}_{a,M\ b};\ M\ (\mathsf{strengthL}^{M}_{a,b});\ \mathsf{join}^{M}_{a\times b}$$

Notice that we chose to "execute the first component first" – this is in general different from proceeding the other way round.

We shall use the folding of this over ordered tuples:

$$\begin{aligned}&\otimes : (M\ a_1 \times \cdots \times M\ a_n) \rightarrow M\ (a_1 \times \cdots \times a_n)\\&\otimes = (\cdots((((M_{a_1} \times M_{a_2});\ \otimes_M) \times M_{a_3});\ \otimes_M) \cdots \times M_{a_n});\ \otimes_M\end{aligned}$$

4.3 Formalisation of the Bimonadic Semantics of PMC

Before we get to the formalisation of the bimonadic semantics of PMC, we introduce the idea of type semantics.

When we attempted to implement the bimonadic semantics of PMC, we found that given that any pattern matching (or a function) has a type $\alpha \rightarrow \beta$ we can easily evaluate this pattern matching (or this function) to some value of type $\mathsf{M}\ (\alpha \rightarrow \beta)$ using matching semantic function. From this, we can extract a function of type $\alpha \rightarrow \beta$ in a monadic computation. However, for the purpose of dealing properly with pattern matching failure, the result of function application should be type $\mathsf{M}\ \beta$ instead of just type β. Therefore, in order to continuing evaluation, we have to convert this value of type $\mathsf{M}\ (\alpha \rightarrow \beta)$ to another value of type $\alpha \rightarrow \mathsf{M}\ \beta$ so that we can directly supply an argument of type α to this pattern matching (or apply this function to an argument of type α) to get a result of type $\mathsf{M}\ \beta$. Thus, we introduce an explicit type semantics to solve this problem. Our basic type semantics rules are as follows:

$$\begin{aligned}[\![\alpha \rightarrow \beta]\!]_{\mathsf{M}} &= [\![\alpha]\!]_{\mathsf{M}} \rightarrow \mathsf{M}\ [\![\beta]\!]_{\mathsf{M}} &&\\ [\![T]\!]_{\mathsf{M}} &= T && \text{if } T \text{ is a primitive type}\\ [\![C\ \alpha_1 \ldots \alpha_n]\!]_{\mathsf{M}} &= C\ [\![\alpha_1]\!]_{\mathsf{M}} \ldots [\![\alpha_n]\!]_{\mathsf{M}} && \text{if } C \text{ is a polynomial type constructor}\end{aligned}$$

The second case is of course an instance of the third.

The idea of the explicit type semantics is the foundation of the formalisation of the bimonadic semantics of PMC in this section. However, In the bimonadic semantics of PMC, the type semantics depends on both E and M. Therefore, we have $[\![\alpha]\!]_{\mathsf{E,M}}$ instead of $[\![\alpha]\!]_{\mathsf{M}}$. For brevity, we will use the abbreviation $[\![\alpha]\!]$ for $[\![\alpha]\!]_{\mathsf{E,M}}$, where the monads are clear from the context.

Now we start from the term category $\mathcal{T}$ for typed patterns. Then we consider a functorial semantics in a cartesian closed category $\mathcal{C}$ via the functor denoted by superscripting with $\mathcal{C}$. Assume two monads in $\mathcal{C}$, $(\mathsf{E}, \mathsf{join}^{\mathsf{E}}, \mathsf{return}^{\mathsf{E}})$ for expressions, with an additional natural transformation

$$\mathsf{empty}^{\mathsf{E}}_a : \mathbb{1} \rightarrow \mathsf{E}\ a$$

and the additive monad $(\mathsf{M}, \mathsf{join}^{\mathsf{M}}, \mathsf{return}^{\mathsf{M}}, \mathsf{zero}^{\mathsf{M}}, \mathsf{plus}^{\mathsf{M}})$ for matchings.

The factoring of zero^M_a and return^M_a through the direct sum of $\mathbb{1}$ and a has to be a mono – this makes sure that their ranges are disjoint.

In particular, we will need distribution of addition over function application:

$$\begin{aligned}
&(\mathsf{plus}^M_{[a\to M\ b]} \times \mathsf{id}a);\ \mathsf{strengthR}^M_{[a\to M\ b],a} \odot_M \mathsf{eval}_{[a\to M\ b]} = \\
&\langle(\mathsf{fst}_{a,b} \times \mathsf{id}a);\ \mathsf{strengthR}^M_{[a\to M\ b],a} \odot_M \mathsf{eval}_{[a\to M]} \\
&,(\mathsf{snd}_{a,b} \times \mathsf{id}a);\ \mathsf{strengthR}^M_{[a\to M\ b],a} \odot_M \mathsf{eval}_{[a\to Mb]} \\
&\rangle;\ \mathsf{plus}^M_b
\end{aligned}$$

The two transformations $\mathsf{transfer}$ and eject are introduced.

- $\mathsf{transfer}_a : M\ a \rightarrow E\ a$, with

  $$\mathsf{zero}^M_a;\ \mathsf{transfer}_a = \mathsf{empty}^E_a \qquad (\mathsf{return}^M;\ \mathsf{transfer})$$

  and

  $$\mathsf{return}^M_a;\ \mathsf{transfer}_a = \mathsf{return}^E_a \qquad (\mathsf{return}^M;\ \mathsf{transfer})$$

- $\mathsf{eject}_a : E\ a \rightarrow M\ a$, with

  $$\mathsf{return}^E_a;\ \mathsf{eject}_a = \mathsf{return}^M_a$$

The condition

$$\mathsf{empty}^E_a;\ \mathsf{eject}_a = \mathsf{zero}^M_a \qquad (\mathsf{eject}\oslash)$$

is necessary only for the semantics of $\mathsf{PMC}_{\not\leftarrow}$.

We consider interpretation of types and data constructors in a cartesian closed category $\mathcal{C}$. For each type α, let $\alpha^{\mathcal{C}}$ denote the object of $\mathcal{C}$ that serves as interpretation of α.

While in strict languages, in the rewriting semantics only values can be substituted for variables, and analogously only values need to be bound to variables by the valuations in the denotational semantics, we are here targetting non-strict languages, where the operational semantics can substitute arbitrary expressions for variables, and therefore, analogously, the type of the denotational variable semantics has to coincide with that of the expression semantics. The object associated with a variable is therefore the images under the expression monad E of the object that interprets the variable's type.

For the sake of conciseness and readability, we abbrebriate this object corresponding to the type of a variable v by

$$v^{\mathsf{E}} := \mathsf{E}\ [\![\mathsf{type}(v)]\!]^{\mathcal{C}}$$

and also introduce similar notation for each sets $\mathcal{V}$ of variables:

$$\mathcal{V}^{\mathsf{E}} := \Pi v : \mathsf{FV}(e) \bullet \mathsf{E}\ [\![\mathsf{type}(v)]\!]^{\mathcal{C}}\ .$$

In the non-strict setting, data constructors always produce values, but accept arbitrary expressions as arguments. Therefore, for each constructor $c : \alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$, the *constructor morphism* that serves as interpretation of c goes from a product of expression semantics to an expression semantics:

$$c^{\mathcal{C}} : \mathsf{E}\ [\![\alpha_1]\!]^{\mathcal{C}} \times \cdots \times \mathsf{E}\ [\![\alpha_n]\!]^{\mathcal{C}} \rightarrow [\![\beta]\!]^{\mathsf{C}}$$

In addition, for each constructor $c : \alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$, we also assume existence of an arrow

$$\tilde{c}^{\mathcal{C}} : [\![\beta]\!]^{\mathcal{C}} \rightarrow \mathsf{M}\ (\mathsf{E}\ [\![\alpha_1]\!]^{\mathcal{C}} \times \cdots \times \mathsf{E}\ [\![\alpha_n]\!]^{\mathcal{C}})$$

such that $c^{\mathcal{C}};\ \tilde{c}^{\mathcal{C}} = \mathsf{return}^{\mathsf{M}}_{\mathsf{E}\ [\![\alpha_1]\!]^{\mathcal{C}} \times \cdots \times \mathsf{E}\ [\![\alpha_n]\!]^{\mathcal{C}}}$.

Since we want the reduction rules to be translated into semantic equations, both sides of a rule always have to be interpreted in a compatible way; since the reduction rules do not preserve all free variables, have to externally impose a start object for the semantic morphisms.

Therefore, given a variable set $\mathcal{V}$, we will define the semantics of an expression e of type α with $\mathsf{FV}(()e) \subseteq \mathcal{V}$ as a morphism from the product corresponding to the variable set $\mathcal{V}$ to the object corresponding to α:

$$[\![e]\!]^{\mathsf{E}}_{\mathcal{V}} : \mathcal{V}^{\mathsf{E}} \rightarrow \mathsf{E}\ [\![\alpha]\!]^{\mathcal{C}}$$

For each matching m of type α, we will define its semantics as a morphism in the Kleisli category for M from the variables to the result type:

$$[\![m]\!]^{\mathsf{M}}_{\mathcal{V}} : \mathcal{V}^{\mathsf{E}} \rightarrow \mathsf{M}\ [\![\alpha]\!]^{\mathcal{C}}$$

One might consider to use $\mathsf{M}\ (\mathsf{E}\ [\![\alpha]\!]^{\mathcal{C}})$ as the target type here, but we will see that we gain additional flexibility by the chosen setup.

Finally, to each pattern p of type α, we associate a morphism in the Kleisli category of M from the object used for expression semantics of type α to the object corresponding to the set of free variables of the pattern:

$$[\![p]\!]^{\mathsf{P}} : \mathsf{E}\ [\![\alpha]\!]^{\mathcal{C}} \rightarrow \mathsf{M}\ (\mathsf{FV}(p)^{\mathsf{E}})$$

We formalise the bimonadic semantics of PMC in the figure 4.1.

Pattern semantics: If $\underset{\mathsf{P}}{\vdash}\ p : \alpha$, then $\llbracket p \rrbracket^{\mathsf{P}} : \mathsf{E}\ \alpha^{\mathcal{C}} \to \mathsf{M}\ (\mathsf{FV}(p)^{\mathsf{E}})$

- $\llbracket v \rrbracket^{\mathsf{P}} = \mathsf{return}^{\mathsf{M}}_{v^{\mathsf{E}}}$
- $\llbracket c(p_1, \dots, p_n) \rrbracket^{\mathsf{P}} = \tilde{c}^{\mathcal{C}} \odot_{\mathsf{M}} ((\llbracket p_1 \rrbracket^{\mathsf{P}} \times \cdots \times \llbracket p_n \rrbracket^{\mathsf{P}});\ \otimes)$

  The target type is isomorphic to $\mathsf{M}\ (\Pi v : \mathsf{FV}(p) \bullet \mathsf{E}\ \llbracket \mathsf{type}(v) \rrbracket^{\mathcal{C}})$; for the sake of conciseness we consider these two types as identified.

Expression semantics: If $\underset{\mathsf{E}}{\vdash}\ e : \alpha$, then $\llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}} : \mathcal{V}^{\mathsf{E}} \to \mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}}$

- $\llbracket v \rrbracket^{\mathsf{E}}_{\mathcal{V}} = \mathsf{proj}^{\mathsf{E}}_{\mathcal{V} \succ \{v\}}$
- $\llbracket c(e_1, \dots, e_n) \rrbracket^{\mathsf{E}}_{\mathcal{V}} = \langle \llbracket e_1 \rrbracket^{\mathsf{E}}_{\mathcal{V}}, \dots, \llbracket e_n \rrbracket^{\mathsf{E}}_{\mathcal{V}} \rangle;\ c^{\mathcal{C}};\ \mathsf{return}^{\mathsf{E}}_{\alpha^{\mathcal{C}}}$
- If $\underset{\mathsf{E}}{\vdash}\ f : \alpha \to \beta$ and $\underset{\mathsf{E}}{\vdash}\ a : \alpha$, then $\llbracket f\ \ a \rrbracket^{\mathsf{E}}_{\mathcal{V}} =$

  $\langle \llbracket f \rrbracket^{\mathsf{E}}_{\mathcal{V}};\ \mathsf{eject}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V}} \rangle;\ \mathsf{strengthR}^{\mathsf{E}}_{[\mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}} \to \mathsf{M}\ \llbracket \beta \rrbracket^{\mathcal{C}}], \mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}}} \odot_{\mathsf{E}} \mathsf{eval}_{[\mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}} \to \mathsf{M}\ \llbracket \beta \rrbracket^{\mathcal{C}}]};\ \mathsf{transfer}_{\llbracket \beta \rrbracket^{\mathcal{C}}}$
- $\llbracket \{\!|\ m\ |\!\} \rrbracket^{\mathsf{E}}_{\mathcal{V}} = \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V}};\ \mathsf{transfer}$
- $\llbracket \oslash \rrbracket^{\mathsf{E}}_{\mathcal{V}} = \mathsf{empty}^{\mathsf{E}}$

Matching semantics: If $\underset{\mathsf{M}}{\vdash}\ m : \alpha$, then $\llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V}} : \mathcal{V}^{\mathsf{E}} \to \mathsf{M}\ \alpha^{\mathcal{C}}$

- $\llbracket \rceil e \lceil \rrbracket^{\mathsf{M}}_{\mathcal{V}} = \llbracket e \rrbracket^{\mathsf{E}}_{\mathcal{V}};\ \mathsf{eject}$
- $\llbracket \lightning \rrbracket^{\mathsf{M}}_{\mathcal{V}} = \mathsf{zero}^{\mathsf{M}}$
- If $\underset{\mathsf{P}}{\vdash}\ p : \alpha$ and $\underset{\mathsf{M}}{\vdash}\ m : \beta$, then $\llbracket p \Mapsto m \rrbracket^{\mathsf{M}}_{\mathcal{V}} =$

  $\lambda((\mathsf{proj}^{\mathsf{E}}_{\mathcal{V} \succ \mathcal{V} \setminus \mathsf{FV}(p)} \times \llbracket p \rrbracket^{\mathsf{P}});\ (\mathsf{strengthL}^{\mathsf{M}}_{\mathcal{V} \setminus \mathsf{FV}(p)^{\mathsf{E}}, \mathsf{FV}(p)^{\mathsf{E}}}) \odot_{\mathsf{M}} \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V} \cup \mathsf{FV}(p)});\ \mathsf{return}^{\mathsf{M}}_{[\mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}} \to \mathsf{M}\ \llbracket \beta \rrbracket^{\mathcal{C}}]}$
- if $\underset{\mathsf{E}}{\vdash}\ a : \alpha$ and $\underset{\mathsf{M}}{\vdash}\ m : \alpha \to \beta$, then

  $\llbracket a \rhd m \rrbracket^{\mathsf{M}}_{\mathcal{V}} = \langle \llbracket m \rrbracket^{\mathsf{M}}_{\mathcal{V}}, \llbracket a \rrbracket^{\mathsf{E}}_{\mathcal{V}} \rangle;\ (\mathsf{strengthL}^{\mathsf{M}}_{[\mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}} \to \mathsf{M}\ \llbracket \beta \rrbracket^{\mathcal{C}}], \mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}}}) \odot_{\mathsf{M}} \mathsf{eval}_{[\mathsf{E}\ \llbracket \alpha \rrbracket^{\mathcal{C}} \to \mathsf{M}\ \llbracket \beta \rrbracket^{\mathcal{C}}]}$
- $\llbracket m_1 \,|\, m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V}} = \langle \llbracket m_1 \rrbracket^{\mathsf{M}}_{\mathcal{V}}, \llbracket m_2 \rrbracket^{\mathsf{M}}_{\mathcal{V}} \rangle;\ \mathsf{plus}^{\mathsf{M}}$

Figure 4.1: Bimonadic Semantics of PMC

4.4 Monads

In this section, we will implement the monads in the bimonadic semantics of PMC. We have two sets of monads to respectively work for $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$. In every set of monads, the monads E_i and *M* are the *computation concepts* corresponding to expressions respectively matchings, and wrap *values*. *E1* and *M* work for $\mathsf{PMC}_{\oslash}$ and *E2* and *M* work for $\mathsf{PMC}_{\lightning}$. We first define the matching monad *M* and its relevant categorial functions. We then define the expression monad *E1* and its relevant categorial functions for $\mathsf{PMC}_{\oslash}$. Finally, We define the expression monad *E2* and relevant categorial functions for $\mathsf{PMC}_{\lightning}$. The categorical functions has been introduce in the abstract categorical setting in the section 4.3.

The matching monad is shared by thw two calculi:

```
newtype M a = M{unM :: Maybe a} deriving (Typeable1)
```

Matching failure *Fail* is translated into *M Nothing*.

The following expression monad *E1* is for $\mathsf{PMC}_{\oslash}$.

```
newtype E1 a = E1{unE1 :: Identity a} deriving (Typeable1)
```

Empty expression *Empty* is translated into *E* (*error* "Empty").

The following expression monad *E2* is for $\mathsf{PMC}_{\lightning}$.

```
newtype E2 a = E2{unE2 :: Maybe a} deriving (Typeable1)
```

Empty expression *Empty* is translated into *E Nothing*.

The following *Typeable1* instance of *Identity* allows *E1* to derive its *Typeable1* instance.

```
tcIdentity = mkTyCon "Control.Monad.Identity"
instance Typeable1 Identity where
  typeOf1 (_ :: Identity a) = mkTyConApp tcIdentity []
```

4.4.1 Matching Monad

In this subsection, we will implement the matching monad *M* for both the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$.

The monad *M* is in *Monad*, *MonadPlus* and *Functor* classes.

```
instance Monad M where
  return m = M $ return m
  fail s = M $ fail s
  (M m) ≫= k = M (m ≫= unM ∘ k)
instance MonadPlus M where
  mzero = M Nothing
  (M m1) `mplus` (M m2) = M (m1 `mplus` m2)
instance Functor M where
  fmap f (M m) = M (fmap f m)
```

We define the *Transfer* and *Eject* classes.

```
class Transfer m e where
  transfer :: m a → e a

class Eject e m where
  eject :: e a → m a
```

4.4.2 Expression Monad for $\mathsf{PMC}_{\oslash}$

The expression monad *E1* for $\mathsf{PMC}_{\oslash}$ has *Monad* and *Functor* instances.

```
instance Monad E1 where
  return e = E1 $ return e
  fail s = E1 $ fail s
  (E1 m) ≫= k = E1 (m ≫= unE1 ∘ k)

instance Functor E1 where
  fmap f e = e ≫= λa → return $ f a
```

We first create an instance *Transfer Maybe Identity* and then based on this, create an instance *Transfer M E1*.

```
instance Transfer Maybe Identity where
  transfer = maybe (fail "Transfer") return
instance Transfer M E1 where
  transfer (M m) = E1 (transfer m)
```

We first create an instance *Eject Identity Maybe* and then based on this, create an instance *Eject E1 M*.

```
instance Eject Identity Maybe where
  eject i = return $ runIdentity i
instance Eject E1 M where
  eject (E1 e) = M (eject e)
```

4.4.3 Expression Monad for Resurrection of Matching Failure

We now turn to $\mathsf{PMC}_{↯}$.

The expression monad *E2* for $\mathsf{PMC}_{↯}$ has *Monad* and *Functor* instances.

```
instance Monad E2 where
  return e = E2 $ return e
  fail s = E2 $ fail s
  (E2 m) ≫= k = E2 (m ≫= unE2 ∘ k)

instance Functor E2 where
```

```
    fmap f (E2 e) = E2 (fmap f e)
```

We first create an instance *Transfer Maybe Maybe* and then based on this, create an instance *Transfer M E2*.

```
instance Transfer Maybe Maybe where
  transfer = id
instance Transfer M E2 where
  transfer (M m) = E2 (transfer m)
```

We first create an instance *Eject Maybe Maybe* and then based on this, create an instance *Eject E2 M*.

```
instance Eject Maybe Maybe where
  eject = id
instance Eject E2 M where
  eject (E2 e) = M (eject e)
```

4.5 Implementation of Type Semantics

As said in the section 4.3, the idea of the explicit type semantics is the foundation of the formalisation of the bimonadic semantics of PMC in this section. Now, before implementing the bimonadic semantics of PMC, We implement the type semantics in this section as the foundation of the implementation of the bimonadic semantics of PMC.

We implement this type semantics through a type constructor *SemType*, which is used as the type-level mapping from type indices to their semantics.

Preliminary experiments using a type class instead showed that in that case, the compiler will not derive the premises of the instance for function types since it does no make use of closedness information of type classes.

GADTs are by definition closed, and therefore provide more guidance to the compiler, so we use these for the time being, even though that limits the type constructors we can use.

```
data SemType :: (* → *) → (* → *) → (* → *) where
  SemTypeFct :: (Typeable a, Typeable b) ⇒
    (SemTypeE e m a → SemTypeM e m b) → SemType e m (a → b)
  SemTypeTriv :: () → SemType e m ()
  SemTypeBool :: Bool → SemType e m Bool
  SemTypeInt :: Int → SemType e m Int
  SemTypeChar :: Char → SemType e m Char
  SemTypeInteger :: Integer → SemType e m Integer
  SemTypeFloat :: Float → SemType e m Float
  SemTypeDouble :: Double → SemType e m Double
  SemTypePair :: (SemTypeE e m a, SemTypeE e m b) → SemType e m (a, b)
```

```
SemTypeEither :: Either (SemTypeE e m a) (SemTypeE e m b) →
  SemType e m (Either a b)
SemTypeMaybe :: Maybe (SemTypeE e m a) → SemType e m (Maybe a)
SemTypeList :: ListRepr e (SemType e m a) → SemType e m [a]
```

We will need at least one inverse constructor — the following pattern matching is complete due to the GADT constraints.

```
unSemTypeList :: SemType e m [a] → ListRepr e (SemType e m a)
unSemTypeList (SemTypeList xs) = xs
```

We need a *Typeable1* instance for *SemType e m*.

```
tcSemType = mkTyCon "VarSem.SemType"
instance (Typeable1 e, Typeable1 m) ⇒ Typeable1 (SemType e m) where
  typeOf1 (_ :: SemType e m a) = mkTyConApp tcSemType
    [typeOf1 (⊥ :: e a)
    , typeOf1 (⊥ :: m a)
    ]
```

Recursive datatypes are based on a bifunctor, which, for lists, is the following:

```
type ListBiFunctor a b = Maybe (a, b)
```

Just for illustration, here is how lists are defined from this bifunctor via explicit recursion:

```
data List a = List (ListBiFunctor a (List a))
```

One could also use a second-order type constructor for recursive datatypes:

```
data RecType f = RecType (f (RecType f))
```

If *ListBiFunctor* was a newtype, we could partially apply it for the definition using *RecType*:

```
data List' a = List' (RecType (ListBiFunctor a))
```

Since the *RecType* overhead makes *List'* harder to use than *List*, we use a construction that is modelled on that for *List*, adding a "wrapper" monad around all type constructors:

```
data ListRepr w a = ListRepr (ListBiFunctor (w a) (w (ListRepr w a)))
```

We need a *Typeable1* instance, which has to be done manually because of the higher-order kind of *ListRepr*:

```
tcListRepr = mkTyCon "VarSem.ListRepr"
instance (Typeable1 w) ⇒ Typeable1 (ListRepr w) where
  typeOf1 (_ :: ListRepr w a) = mkTyConApp tcListRepr
    [typeOf1 (⊥ :: w a)
    ]
```

We implement the *show* instance for *SemType e m a* as follows.

```
instance (Functor e, ShowF e) ⇒ Show (SemType e m a) where
  showsPrec = showSemType

showSemType :: (Functor e, ShowF e) ⇒ ShowSPrec (SemType e m a)
showSemType _ (SemTypeTriv _) = ("()"++)
showSemType _ (SemTypeBool x) = shows x
showSemType _ (SemTypeInt x) = shows x
showSemType _ (SemTypeChar x) = shows x
showSemType _ (SemTypeInteger x) = shows x
showSemType _ (SemTypeFloat x) = shows x
showSemType _ (SemTypeDouble x) = shows x
showSemType _ (SemTypePair (x, y)) =
  ('(':) ∘ showsPrecF showSemType 0 x ∘
  (',':) ∘ showsPrecF showSemType 0 y ∘ (')':)
showSemType _ (SemTypeList (ListRepr Nothing)) = shows "[]"
showSemType _ (SemTypeList (ListRepr (Just (ea, eas)))) =
  ('(':) ∘ showsPrecF showSemType 0 ea ∘
  (" : "++) ∘ showsPrecF showSemType 0 (fmap SemTypeList eas) ∘ (')':)
```

We frequently need *SemTypes* inside the semantic monads:

```
type SemTypeE e m a = e (SemType e m a)
type SemTypeM e m a = m (SemType e m a)

newtype SemE e m a = SemE{unSemE :: SemTypeE e m a}
newtype SemM e m a = SemM{unSemM :: SemTypeM e m a}
```

4.6 Variable Semantics

In the section, by defining type-indexed mappings to construct dictionaries, we implement variable assignment and operator semantics.

4.6.1 Variable Assignments

We use a separate type *VarAssign* to handle variable semantics. It maps a variable of type *Var a* to a type semantics value of type *SemType e m a*, where *e* and *m* are two monad arguments and can be instantiated as *E1* and *M* for $\mathsf{PMC}_{\oslash}$, or instantiated as *E2* and *M* for $\mathsf{PMC}_{\Leftarrow}$. Thus, the corresponding variable assignments respectively work for $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\Leftarrow}$.

```
type VarAssign e m = VA.TIMap Var (SemE e m)
```

We also define insertion and lookup functions for convenience.

```
vaInsert :: (Typeable a, Monad e, MonadPlus m, Transfer m e, Eject e m) ⇒
  Var a → SemE e m a → VarAssign e m → VarAssign e m
vaInsert v a = VA.insert v a
vaLookup :: (Typeable a, Monad e, MonadPlus m, Transfer m e, Eject e m) ⇒
  Var a → VarAssign e m → Maybe (SemE e m a)
vaLookup v va = VA.lookup v va
```

4.6.2 Operator Semantics

Due to that we consider operators and primitive functions as variables, we also use *VarAssign* to implement operator semantics, which is used to interpret operators from operator names to its meaning in the semantic domain.

We introduce a dictionary including the two operators + and ++ as follows:

```
va0 :: (Functor e, Monad e, MonadPlus m, Transfer m e, Eject e m) ⇒
  VarAssign e m
va0
= vaInsert (mkVar' "+" :: Var (Int → Int → Int))
  (wrapIntBinary (+))
∘ vaInsert (mkVar' "++" :: Var ([Int] → [Int] → [Int]))
  (wrapIntListBinary conc)
$ VA.empty
```

The function *wrapIntBinary* is built to facilitate building binary operator or function over *Int* in the semantic domain. Therefore, it can be used to build the operator + in the semantic domain.

```
wrapIntBinary :: (Eject e m, Monad e, Monad m) ⇒
  (Int → Int → Int) → SemE e m (Int → Int → Int)
wrapIntBinary f = SemE $ return $
  SemTypeFct $ λx → do   -- in Monad m
    return $
      SemTypeFct $ λy → do   -- in Monad m
        SemTypeInt a ← eject x
        SemTypeInt b ← eject y
        return $ SemTypeInt $ f a b
```

The function *wrapIntListBinary* is built to facilitate building binary operator or function over [*Int*] in the semantic domain. Therefore, it can be used to build the operator ++ in the semantic domain.

```
wrapIntListBinary :: (Eject e m, Monad e, Functor e, Monad m) ⇒
  (SemTypeE e m [Int] → SemTypeE e m [Int] → SemTypeE e m [Int]) →
```

```
    SemE e m ([Int] → [Int] → [Int])
  wrapIntListBinary conc = SemE $ return $
    SemTypeFct $ λx → do    -- in Monad m
      return $
        SemTypeFct $ λy → do    -- in Monad m
          eject $ conc x y
```

We need a function *conc* of type
SemTypeE e m [Int] → SemTypeE e m [Int] → SemTypeE e m [Int] as an argument of the function *wrapIntListBinary*.

```
  conc :: (Functor e, Monad e) ⇒
    SemTypeE e m [Int] → SemTypeE e m [Int] → SemTypeE e m [Int]
  conc ass bss = do    -- in Monad e
    SemTypeList (ListRepr maybeValue) ← ass
    case maybeValue of
      Nothing → bss
      Just (a, as) → let
        cs = conc (fmap SemTypeList as) bss
        in return $ SemTypeList (ListRepr (Just (a, fmap unSemTypeList cs)))
```

4.7 Constructor Semantics

In this section, we implement a constructor semantics for constants and constructors.

4.7.1 Constructor Assignments

We define a type *Constructor* to be the type constructor of the source of a type-indexed mapping, which acts as constructor assignments.

```
  data Constructor :: * → *where
    Constructor :: (Show c, Ord c, Typeable c, CType c a) ⇒ c → Constructor a
```

The *CType* class has been introduced in the section 2.1.2.

Since the type *Constructor a* is intended to be the type of the source of a type-indexed mapping, it must have the *Ord* instance.

```
  instance Eq (Constructor a) where
    Constructor x ≡ Constructor y = case cast x of
      Nothing → False
      Just x' → x' ≡ y

  instance Ord (Constructor a) where
```

```
compare (Constructor x) (Constructor y) = case cast x of
  Nothing → error "this should not be possible"
  Just x' → compare x' y
Constructor x ⩽ Constructor y = case cast x of
  Nothing → error "this should not be possible"
  Just x' → x' ⩽ y
```

Now we define the type of the type-indexed mapping, a separate newtype *ConstrAssign e m*, to acts as constructor assignments.

```
type ConstrAssign e m = CA.TIMap Constructor (SemType e m)
```

The following two functions is used to facilitate the operations of insertion and lookup.

```
caInsert :: (Show c, Ord c, Typeable c, CType c a, Typeable a,
  Monad e, Monad m, Transfer m e, Eject e m) ⇒
  c → SemType e m a → ConstrAssign e m → ConstrAssign e m
caInsert c s = CA.insert (Constructor c) s
caLookup :: (Show c, Ord c, Typeable c, CType c a, Typeable a,
  Monad e, Monad m, Transfer m e, Eject e m) ⇒
  c → ConstrAssign e m → Maybe (SemType e m a)
caLookup c ca = CA.lookup (Constructor c) ca
```

We introduce a dictionary as follows:

```
ca0 :: (Functor e, Monad e, Monad m, Transfer m e, Eject e m) ⇒
  ConstrAssign e m
ca0
= caInsert (CResult "[]" :: CResult [Int]) (SemTypeList (ListRepr Nothing))
∘ caInsert (CArg (CArg (CResult ":"))) wrapIntList
∘ caInsert (CArg (CArg (CResult "(,)"))) wrapIntPair
∘ caInsert (CResult "1") (SemTypeInt (1 :: Int))
∘ caInsert (CResult "2") (SemTypeInt (2 :: Int))
∘ caInsert (CResult "5") (SemTypeInt (5 :: Int))
∘ caInsert (CResult "22") (SemTypeInt (22 :: Int))
∘ caInsert (CResult "42") (SemTypeInt (42 :: Int))
$ CA.empty
```

where *wrapIntList* is a list constructor in the semantic domain

```
wrapIntList :: (Functor e, Monad e, Monad m, Eject e m) ⇒
  SemType e m (Int → [Int] → [Int])
wrapIntList =
  SemTypeFct $ λ(a :: SemTypeE e m Int) →
    return $
      SemTypeFct $ λ(as :: SemTypeE e m [Int]) → do
```

```
        return $ SemTypeList $ ListRepr $ Just (a, fmap unSemTypeList as)
```

and *wrapIntPair* is a pair constructor in the semantic domain.

```
wrapIntPair :: (Monad e, Monad m, Eject e m) ⇒
  SemType e m (Int → Int → (Int, Int))
wrapIntPair =
  SemTypeFct $ λ(a :: SemTypeE e m Int) → do
    return $
      SemTypeFct $ λ(b :: SemTypeE e m Int) → do
      return $ SemTypePair (a, b)
```

4.7.2 Semantics of Pattern Constructors

Since Control.Monad.Identity.*Identity* has no *Typeable* and *Ord* instances, and also since we do not need the monad aspects, we define our own identity type constructor:

```
newtype I a = I{unI :: a} deriving (Eq, Ord, Typeable)
instance Functor I where
  fmap f (I a) = I (f a)
instance Monad I where
  return a = I a
  ia ≫= f = f (unI ia)
```

We also define the *ConstrUnCurry* class as follows.

```
class (Monad e, Monad m, Typeable1 e, Typeable1 m
  , Typeable c, Typeable r, Typeable as) ⇒
  ConstrUnCurry e m c r as | c e m → r, c e m → as
where
  constrResultType :: c → r
  constrArgTypes :: c → as
```

The following instances impose a restriction on the argument types of *ConstrUnCurry*: *c* is the type that a constructor has in typed PMC, *r* is the result type of constructor application of the constructor in the semantic domain, *as* is the type of a decomposed structure of a constructor application in the semantic domain.

```
instance (Monad e, Monad m, Typeable1 e, Typeable1 m, Typeable a) ⇒
  ConstrUnCurry e m (CResult a) (SemTypeE e m a) ()
where
  constrResultType _ = ⊥
  constrArgTypes _ = ⊥

instance (ConstrUnCurry e m c r as, Typeable a) ⇒
  ConstrUnCurry e m (CArg a c) r (e (SemType e m a), as) where
```

```
    constrResultType _ = ⊥
    constrArgTypes _ = ⊥
```

A value of the type *ConstrMatchFct e m c* wraps a function, which is used to decompose the result of a constructor application into the structure of the constructor application. The resulting structure is used to implement matching of constructor applications of patterns.

```
  data ConstrMatchFct :: (* → *) → (* → *) → * → *where
    ConstrMatchFct :: (Transfer m e, Eject e m,
      ConstrUnCurry e m c r as) ⇒ (r → m as) → ConstrMatchFct e m c
```

a function and the function, from the result of constructor application, decompose the structure of the constructor application to facilitate the implementation of matching expressions against patterns.

Now we can define a type-indexed mapping to implement the semantics of matching of constructor applicatons of patterns.

```
  type PatConstrMap e m = PCM.TIMap I (ConstrMatchFct e m)
```

The type-indexed mapping maps a constructor to its decomposition function. Thus, given an expression constructor application, we first get its constructor and then find its decomposition function from this type-indexed mapping. Finally, we can use this decomposition function to decompose the value of the comstructor application into the structure of its corresponding constructor. By using the decomposed structure, we can match it against the corresponding pattern.

A insertion function is defined for convenience.

```
  pcmInsert :: (Show c, Ord c, Typeable c,
    Transfer m e, Eject e m, ConstrUnCurry e m c r as) ⇒
    c → (r → m as) → PatConstrMap e m → PatConstrMap e m
  pcmInsert c f = PCM.insert (I c) (ConstrMatchFct f)
```

We introduce a dictionary as follows:

```
  pcm0 :: (Functor e, Monad e, Eject e m, Monad m, Typeable1 e, Typeable1 m,
    Eject e m, Transfer m e) ⇒ PatConstrMap e m
  pcm0
   = pcmInsert (CResult "[]" :: CResult [Int]) unwrapNil
   ∘ pcmInsert pair  unwrapPair
   ∘ pcmInsert cons unwrapCons
   $ PCM.empty
```

where *unwrapNil* decomposes the structure of a null list

```
  unwrapNil :: (Eject e m, Monad m) ⇒ SemTypeE e m [Int] → m ()
  unwrapNil x = do
```

```
    (SemTypeList (ListRepr Nothing)) ← eject x
    return ()
```

and *unwrapPair* decomposes the structure of a pair

```
unwrapPair :: (Eject e m, Monad m) ⇒
    SemTypeE e m (Int, Int) → m (SemTypeE e m Int, (SemTypeE e m Int, ()))
unwrapPair x = do
    (SemTypePair (ex, ey)) ← eject x
    return (ex, (ey, ()))
```

and *unwrapCons* decomposes the structure of a list

```
unwrapCons :: (Eject e m, Monad m, Functor e) ⇒
    SemTypeE e m [Int] → m (SemTypeE e m Int, (SemTypeE e m [Int], ()))
unwrapCons x = do
    (SemTypeList (ListRepr (Just (x, xs)))) ← eject x
    return (x, (fmap SemTypeList xs, ()))
```

Finally, the above two type-indexed mapping *constrAssign e m* and *PatConstrMap e m* constitute the semantics of constructors.

```
type ConstrSem e m = (ConstrAssign e m, PatConstrMap e m)
```

4.8 Implementation of Bimonadic Semantics

In this section, by using the monads in the section 4.4, the variable semantics in the section 4.6, and the constructor semantics in the section 4.7, we implement the formalised bimonadic semantics in the section 4.3. Our implementation exactly corresponds to the bimonadic semantics of PMC in the figure 4.1.

We definition the two evaluation function *evalE1* and *evalE2* for $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\Leftrightarrow}$ respectively. Note that we instantiate the monad variables *e* and *m* with *E1* and *M* respectively in *evalE1* and instantiate the monad variables *e* and *m* with *E2* and *M* respectively in *evalE2*. Considering the different instance functions will be used when the corresponding monads are different in the evaluation functions, the two different function types are sufficient to produce two functions of different evaluation processes, which actually are what we expect.

```
evalE1 :: (Typeable a) ⇒ ConstrSem E1 M → VarAssign E1 M → Expr a →
    E1 (SemType E1 M a)
evalE1 = evalE

evalE2 :: (Typeable a) ⇒ ConstrSem E2 M → VarAssign E2 M → Expr a →
    E2 (SemType E2 M a)
evalE2 = evalE
```

4.8.1 Semantic Function for Patterns

The semantic function for patterns maps every syntactical construct of patterns to the monad object in the semantic domain.

We define a newtype *UpdVA* for convenience.

```
type UpdVA e m = (VarAssign e m) → m (VarAssign e m)
```

We use the monad variables in the function type so that we can instantiate them with different monads later to gain reusability.

```
evalP :: (Typeable a, Typeable1 e, Typeable1 m,
  Monad e, MonadPlus m, Transfer m e, Eject e m) ⇒
  PatConstrMap e m → Pat a → SemTypeE e m a → UpdVA e m
```

When evaluating a value with structure *Varpat v*, the function adds it and its corresponding argument value into the variable assignments for later use.

```
evalP pcm (VarPat v) st va = return $ vaInsert v (SemE st) va
```

For a value with structure *ConstrPat ca*, *evalP* call *evalPCA* and provide *evalPCA* with a function argument to record the decomposed structure level by level to match supplied expression argument again this pattern.

```
evalP pcm (ConstrPat ca) st va = evalPCA pcm ca (λ() → return) st va

evalPCA :: (Typeable r, Typeable c, Ord c, ConstrUnCurry e m c r as,
  Monad e, MonadPlus m, Transfer m e, Eject e m) ⇒
  PatConstrMap e m → ConstrApp Pat c → (as → UpdVA e m) → (r → UpdVA e m)
```

When the patterns have structure *Constr c*, the function looks it up in the semantics of pattern constructors to get the decomposition function of constructor application of this constructor. Then, the function applies this decomposition function to the expression argument to get the decomposed structure of the expression argument. Finally, the function uses the functions *cont* and *cont'* to match the expression argument against the pattern level by level, by keeping all the *cont* functions hold.

```
evalPCA pcm (Constr c) cont st va = case PCM.lookup (I c) pcm of
  Nothing → fail "evalPCA: unknown constructor"
  Just (ConstrMatchFct cmf) → case cast st of
    Nothing → fail "evalPCA: cast error"
    Just r → do
      as ← cmf r
      case cast as of
        Nothing → fail "evalPCA: back-cast error"
        Just as' → cont as' va
evalPCA pcm (ConstrApply ca p) cont st va = evalPCA pcm ca cont' st va
  where
```

```
cont' (a, as) va = do
  va' ← cont as va
  evalP pcm p a va'
```

4.8.2 Semantic Function for Expressions

The semantic function for expressions maps every syntactical construct of expressions to the monad object in the semantic domain.

```
evalE :: (Typeable a, Typeable1 e, Typeable1 m, Monad e,
  Functor e, MonadPlus m, Transfer m e, Eject e m) ⇒
  ConstrSem e m → VarAssign e m → Expr a → SemTypeE e m a
```

The semantic function looks up the expression variable directly in the variable assignments to get the corresponding monad object in the semantic domain.

```
evalE cs va (EVar v) = case vaLookup v va of
  Just (SemE a) → a
  Nothing → error $ "evalE: " ++ show v ++ " is a free variable"
```

We define an auxiliary function *evalECA* to evaluate expression constructor applications.

```
evalE cs va (ConstrExpr ca) = evalECA cs va ca
```

When evaluating a expression with structure *Apply f a*, the function first evaluates *f* to get a function *f'* of type *SemTypeE e m a → SemTypeM e m b* and then evaluate *a* to a value *a'* of type *SemTypeE e m a*. The function applies *f'* to *a'* to get a value of type *SemTypeM e m b*. Finally, the function applies *transfer* to the value to get the expected result.

```
evalE cs va (Apply f a) = do
  SemTypeFct f' ← evalE cs va f
  let a' = evalE cs va a
  transfer (f' a')
```

For a expression with structure *MExpr m*, the function first calls *evalM* to evaluate *m* and then apply *transfer* to the evaluation value to get the expected result.

```
evalE cs va (MExpr m) = transfer $ evalM cs va m
```

The expression *Empty* is directly interpreted as *fail* "Empty".

```
evalE cs va Empty = fail "Empty"
```

The function *evalECA* is used to evaluate expression constructor applications.

```
evalECA :: (Show c, Ord c, Typeable c, Typeable a, CType c a, Typeable1 e, Typeable1 m,
  Monad e, Monad m, Transfer m e, Eject e m, Functor e, MonadPlus m) ⇒
  ConstrSem e m → VarAssign e m → ConstrApp Expr c → SemTypeE e m a
```

For a value with structure *Constructor c*, *evalECA* looks up it directly in the constructor assignments.

```
evalECA (ca, pcm) va (Constr c) = case caLookup c ca of
  Just a → return a
  Nothing → error $ "evalECA: " ++ show c ++ " is not in ConstrAssign"
```

For a value with structure *ConstrApply c e*, the evaluation is similar with values with structure *Apply f a*. However, here we need extra *gcast* operations.

```
evalECA (ca, pcm) va (ConstrApply c e) = do   -- in Monad e
  SemTypeFct f ← evalECA (ca, pcm) va c
  let a = evalE (ca, pcm) va e
  case gcast a of
    Nothing → error "evalECA: cast failed"
    Just a' → case gcast (SemM (f a')) of
      Nothing → error "evalECA: backcast failed"
      Just (SemM r) → transfer r
```

4.8.3 Semantic Function for Matchings

The semantic function for matchings maps every syntactical construct of matchings to the monad object in the semantic domain.

```
evalM :: (Typeable a, Typeable1 e, Typeable1 m, Monad e, Functor e, MonadPlus m,
  Transfer m e, Eject e m) ⇒
  ConstrSem e m → VarAssign e m → Match a → SemTypeM e m a
```

For a matching with structure *Return e*, the function first calls *evalE* to evaluate *e* and then apply *eject* to the result.

```
evalM cs va (Return e) = eject $ evalE cs va e
```

The matching *Fail* is directly interpreted as *fail* "Fail".

```
evalM cs va Fail = fail "Fail"
```

When evaluating a matching with structure *PMatch p m*, the function introduces a λ-abstraction to provide it with a expression argument. Then the function calls *evalP* to evaluate *p* and take the resulting variable assignments as an argument to evaluate *m*. Finally, the function wraps the result as a function into *SemTypeFct* and return it.

```
evalM cs@(ca, pcm) va (PMatch p m) = return $ SemTypeFct $
  λa → do va' ← evalP pcm p a va
     evalM cs va' m
```

When evaluating a matching with structure *Supply e m*, the function first evaluates *m* to get a function *f* of type *SemTypeE e m a* → *SemTypeM e m b* and then evaluate *e* to a value *a* of type *SemTypeE e m a*. Finally, the function applies *f* to *a* to get the result.

```
evalM cs va (Supply e m) = do
  SemTypeFct f ← evalM cs va m
  let a = evalE cs va e
```

$f\ a$

When evaluating a matching with structure *MAlt m1 m2*, because *m* is an additive monad, the function evaluates *m1* and *m2* as alternatives but *m1* is prior.

$$evalM\ cs\ va\ (MAlt\ m1\ m2) = (evalM\ cs\ va\ m1)\ `mplus`\ (evalM\ cs\ va\ m2)$$

4.9 Evaluation Examples

4.9.1 Four Simple Evaluation Examples

The four expression example that is evaluated in this subsection is defined in 2.4.3.

The first expression is the PMC expression $\{\!|\ [5] \rhd y : [] \mapsto \lceil y \rceil\ |\!\}$. we can show it in GHCi.

```
*EvalExample> ex1
{[5] >> (y:[]) => |y|}
```

When we apply *evalE1* and *evalE2* to it respectively, we get the expected result as follows.

$$\{\!|\ [5] \rhd y : [] \mapsto \lceil y \rceil\ |\!\}$$
$$\xrightarrow[\text{evalE1}]{} \text{E1 } 5$$

We can show it in GHCi.

```
*EvalExample> evalE1 (ca0,pcm0) va0 ex1
E1 5
```

$$\{\!|\ [5] \rhd y : [] \mapsto \lceil y \rceil\ |\!\}$$
$$\xrightarrow[\text{evalE2}]{} \text{E2 } 5$$

We can show it in GHCi

```
*EvalExample> evalE2 (ca0,pcm0) va0 ex1
E2 5
```

The second expression is the PMC expression $\{\!|\ [5] \rhd y : zs \mapsto \lceil zs \rceil\ |\!\}$. we can show it in GHCi.

```
*EvalExample> ex2
{[5] >> (y:zs) => |zs|}
```

When we apply *evalE1* and *evalE2* to it respectively, we get the expected result as follows.

$$\{\!|\, [5] \rhd \mathsf{y} : \mathsf{zs} \mapsto\!\!\!\Rightarrow \rceil \mathsf{zs} \upharpoonleft \,|\!\}$$
$$\xrightarrow[\text{evalE1}]{} \mathsf{E1}\ \text{"[]"}$$

We can show it in GHCi.

```
*EvalExample> evalE1 (ca0,pcm0) va0 ex2
E1 "[]"
```

$$\{\!|\, [5] \rhd \mathsf{y} : \mathsf{zs} \mapsto\!\!\!\Rightarrow \rceil \mathsf{zs} \upharpoonleft \,|\!\}$$
$$\xrightarrow[\text{evalE2}]{} \mathsf{E2}\ \text{"[]"}$$

We can show it in GHCi.

```
*EvalExample> evalE2 (ca0,pcm0) va0 ex2
E2 "[]"
```

The third expression is the PMC expression $\{\!|\, (++)\ [5]\ [42] \rhd (x : (y : [])) \mapsto\!\!\!\Rightarrow \rceil y \upharpoonleft \,|\!\}$. we can show it in GHCi.

```
*EvalExample> ex3
++ [5] [42] >> (x:(y:[])) => |y|
```

When we apply *evalE1* and *evalE2* to it respectively, we get the expected result as follows.

$$\{\!|\, (++)\ [5]\ [42] \rhd (x : (y : [])) \mapsto\!\!\!\Rightarrow \rceil y \upharpoonleft \,|\!\}$$
$$\xrightarrow[\text{evalE1}]{} \mathsf{E1}\ 42$$

We can show it in GHCi.

```
*EvalExample> evalE1 (ca0,pcm0) va0 ex3
E1 42
```

$$\{\!|\, (++)\ [5]\ [42] \rhd (x : (y : [])) \mapsto\!\!\!\Rightarrow \rceil y \upharpoonleft \,|\!\}$$
$$\xrightarrow[\text{evalE2}]{} \mathsf{E2}\ 42$$

We can show it in GHCi.

```
*EvalExample> evalE2 (ca0,pcm0) va0 ex3
E2 42
```

The last expression is the PMC expression $\{\!\!\{ (++)\ [5]\ [42] \rhd (x : (y : zs)) \mapsto \rceil y \lceil \}\!\!\}$. we can show it in GHCi.

```
*EvalExample> ex4
++ [5] [42] >> (x:(y:zs)) => |y|
```

When we apply *evalE1* and *evalE2* to it respectively, we get the expected result as follows.

$$\{\!\!\{ (++)\ [5]\ [42] \rhd (x : (y : zs)) \mapsto \rceil y \lceil \}\!\!\}$$
$$\xrightarrow[\text{evalE1}]{} \mathsf{E1}\ 42$$

We can show it in GHCi.

```
*EvalExample> evalE1 (ca0,pcm0) va0 ex3
E1 42
```

$$\{\!\!\{ (++)\ [5]\ [42] \rhd (x : (y : zs)) \mapsto \rceil y \lceil \}\!\!\}$$
$$\xrightarrow[\text{evalE2}]{} \mathsf{E2}\ 42$$

We can show it in GHCi.

```
*EvalExample> evalE2 (ca0,pcm0) va0 ex4
E2 42
```

4.9.2 Evaluation Example of Variable Scope

We will evaluate the following PMC expression

$$\{\!\!\{ (x, y) \mapsto y \mapsto \rceil (+)\ x\ y \lceil \}\!\!\}\ (5, 42)\ 22\ ,$$

which we implement as *scope* in the type-indexed PMC in 2.4.5.
We can show it in GHCi.

```
*EvalExample> scope
{(x,y) => y => |+ x y|} (5,42) 22
```

When we apply *evalE1* and *evalE2* to *scope*, we get the expected result as follows.

$$\{\!\!\{ (x, y) \mapsto y \mapsto \rceil (+)\ x\ y \lceil \}\!\!\}\ (5, 42)\ 22$$
$$\xrightarrow[\text{evalE1}]{} \mathsf{E1}\ 27$$

We can show it in GHCi.

```
*EvalExample> evalE1 (ca0,pcm0) va0 scope
E1 27
```

$$\{\!|\, (\mathsf{x},\mathsf{y}) \Mapsto \mathsf{y} \Mapsto \rceil(+)\ \mathsf{x}\ \mathsf{y}\upharpoonleft \,|\!\}\ (5,42)\ 22$$

$$\xrightarrow[\text{evalE2}]{} \mathsf{E2}\ 27$$

We can show it in GHCi

```
*EvalExample> evalE2 (ca0,pcm0) va0 scope
E2 27
```

4.9.3 Different Evaluation Results of the Two Calculi

Here we take the following pattern matching example directly from 5.2 of the PMC paper and evaluate it using *evalE1* and *evalE2* respectively to demonstrate different evaluation results of the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$.

$$\{\!|\, (3:[]) \rhd ((\oslash \rhd (x:xs) \Mapsto [] \Mapsto \rceil 1 \upharpoonleft) \,|\, (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \rceil 2 \upharpoonleft)) \,|\!\}$$

In the section 2.4.2, we have defined the corresponding PMC term *pmc*, which is shown in GHCi as follows.

```
*EvalExample> pmc
{[3] >> (empty >> (x:xs) => [] => |1| || empty >> ys => (v:vs) => |2|)}
```

As demonstrated in the section 3.3.4, when we apply *evalE1* and *evalE2* to *pmc*, we get the expected result as follows.

$$\{\!|\, (3:[]) \rhd ((\oslash \rhd (x:xs) \Mapsto [] \Mapsto \rceil 1 \upharpoonleft) \,|\, (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \rceil 2 \upharpoonleft)) \,|\!\}$$

$$\xrightarrow[\text{evalE1}]{} \mathsf{E1}\ \oslash$$

We can show it in GHCi.

```
*EvalExample> evalE1 (ca0,pcm0) va0 pmc
E1 *** Exception: Empty
```

$$\{\!|\, (3:[]) \rhd ((\oslash \rhd (x:xs) \Mapsto [] \Mapsto \rceil 1 \upharpoonleft) \,|\, (\oslash \rhd ys \Mapsto (v:vs) \Mapsto \rceil 2 \upharpoonleft)) \,|\!\}$$

$$\xrightarrow[\text{evalE2}]{} \mathsf{E2}\ 2$$

We can show it in GHCi

```
*EvalExample> evalE2 (ca0,pcm0) va0 pmc
E2 2
```

From the above two evaluation results in the two calculi $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\nleftrightarrow}$, we can draw a conclusion that evalE1 exactly abstracts the meaning of pattern matching of current functional programming languages and evalE2 has a "more successful" evaluation and can be turned into a basis for programming languages implementation.

4.10 Summary

Precisely and unambiguously, the bimonadic semantics of PMC defines the semantics of every syntatical structure of PMC. Thus, it can provide a basis for automatically generating compilers or interpreters. Besides, the bimonadic semantics of PMC implements the two calculus $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\nleftrightarrow}$ under the same framework, which produces flexibility and reusability. Thus, the bimonadic semantics is also useful to investigate other pattern matching model by providing the different monads for PMC's expressions and matchings.

Chapter 5

Conclusions and Future Work

5.1 Summary of the Thesis

In this thesis research, we formalised the bimonadic semantics of the pattern matching calculi (PMC) using categorical concepts and implemented the synatx, operational semantics, and bimonadic semantics of PMC using type-indexed expressions.

The pattern matching calculi are new calculi modelling non-strict pattern matching in modern functional programming languages, and cleanly internalise pattern matching via a modest abstraction that divides PMC terms into two major syntactic categories, namely *expressions* and *matchings*. By providing two different rules to interpret the *empty expression* that results from matching failures, Kahl presented two kinds of calculi, $\mathsf{PMC}_{\oslash}$ and $\mathsf{PMC}_{\lightning}$, both of which have a confluent reduction system and a same normalising strategy. Our type-indexed implementation of syntax and operational semantics of the two calculi shows that $\mathsf{PMC}_{\oslash}$ is a simple and elegant formalisation of the operational pattern matching semantics of current functional programming languages. $\mathsf{PMC}_{\lightning}$ has a "more successful" evaluation result and can be a useful basis for implementations of modern functional programming language.

As a new technique based on Haskell's language extensions of type-safe cast, arbitrary-rank polymorphism, and GADTs, type-indexed expressions demonstrate a uniform way of defining all expressions as type-indexed to capture more program abstraction. In the implementation, the technique of using type-indexed expressions to model PMC data structures can offer both convenience in programming and clarity in code. The type-indexed syntax of PMC mirrors the original theoretic definition of PMC in [11, 13] and the implementation of the operational semantics of the two calculi corresponds perfectly to the original design in [11, 13]. Evaluation examples of the operational semantics show that PMC can be a useful basis of modelling non-strict pattern matching.

Based on Kahl's proposal, we formalised and implemented the bimonadic semantics of PMC in an abstract categorical setting. The bimonadic semantics employs two monads to reflect two kinds of computational effects, which correspond to our two major syntactic categories, i.e. PMC *expressons* and *matchings*. Thus, our bimonadic semantics models the meaning of PMC with more accuracy. The resulting ***bimonadic semantics*** allows us to have an axiomatized formulation of well-known programming languages features such as environments.

Finally, from a practical programming viewpoint, our implementation is a good demonstration of how to program in the pure type-indexed setting by taking full advantage of Haskell's language extensions of type-safe cast, arbitrary-rank polymorphism and GADTs.

5.2 Related Work

In Peyton Jones' book [19], the chapter 4 by Peyton Jones and Wadler introduces a built-in value FAIL representing a pattern matching failure. However, compared with Kahl's PMC that we implemented in this thesis, they did not discover the relation $\{\!|\, \mathsf{FAIL} \,|\!\} = \mathsf{ERROR}$ between FAIL and ERROR, where ERROR corresponds to an empty expression that results from matching failures.

Wadler's chapter 5 in the same book has been one of the standard references for compilation of pattern matching, studying expressions containing alternative and FAIL.

Harrison and Keiburtz provided an abstract semantics and a logical characterization of pattern-matching in Haskell and the reduction order that it entails in [7], based on traditional syntactical structure of pattern matching.

Harrison, Sheard and Hook introduced a calculational semantics for Haskell that exposes the interaction of its strict features with its default laziness in [8]. Their implementation considered "case branches $p \rightarrow e$" as separate syntactical units, which is a PMC matching $p \mapsto e$ in our PMC implementation.

Mosses recognized that traditional denotational semantics lacks modularity and reusability in [18], Watt argued that the drawback makes difficult applying traditional denotational semantics to the design of realistic programming languages in [22]. In [17], Moggi took the notion of monad from category theory to structure various notions of computational effect. Based on the concept of monad in Haskell, Liang and Hudak in [15] introduced *modular monadic semantics* to take advantage of a monadic approach to structure denotational semantics, which achieves a high level of modularity and extensibility. Their work is based on only one monad and does not deal with applications of two monads in denotational semantics.

There is no work on type-indexed forms in the GADT setting yet, excepting Kahl's type-indexed expressions in [14], although there has been some work on type-indexed functions and type-indexed data types. Type-indexed functions were introduced more than a decade ago. The recent work on type-indexed functions includes Oliveira and Gibbons' paper [4], where they presented a design pattern *TypeCase* that allows the definition of closed type-indexed functions. Hinze, Jeuring and Löh defined a type-indexed data type in [10], which is constructed in a generic way from an argument data type.

5.3 Accomplishments

With respect to the purposes of the thesis, the following goals have been accomplished:

- The bimonadic semantics of PMC has been formalised using categorical concepts based on Kahl's proposal.
- The syntax, operational semantics, and bimonadic semantics have been implemented using type-indexed expressions based on Kahl's PMC paper [11, 13] and Kahl's work in type-indexed expressions in [14].
- Some sophisticated PMC evaluation examples have been provided to demonstrate the power of our semantics models.
- The technique of type-indexed expressions, based on Haskell's language extensions of type-safe cast, arbitrary-rank polymorphism and GADTs, has been taken full advantage of during the whole implementation process. Our implementation experiences demonstrate how to use this technique and show the advantages of the technique.

In addition, a type-lost problem in the Haskell type system has been discovered and described.

5.4 Future Work

The primary direction of future work will be a further investigation of how $\mathsf{PMC}_{\lightning}$ can be turned into a basis for programming language implementations. One of our important aims is to make the pattern matching calculi be a useful basis for an interactive program transformation and reasoning system for Haskell.

The next step in the short term can be the development of an automatic translation tool from Haskell code segments to an evaluable PMC terms. Thus, by interactively reasoning about the resulting evaluable PMC terms, we can analyse the properties of original Haskell code segments. Such an result would be inspiring.

The nature of functional languages makes it easier to reason about its extensional behavior, for example, the value returned by a program. However, its intensional behavior, such as the execution order of statements and the time complexity of a program, , is difficult to investigate. In future work, based on our fine-grained PMC syntactic structure and compositional reduction system, the interactive program transformation and reasoning system can be used to measure complexity of Haskell code segments.

Appendix A

Syntax of PMC

The appendix includes modules that define syntax of PMC.

A.1 Variable

Variables is one of two syntactic units of building *patterns* and *expressions* and can only occur as patterns or as expressions. Note that there are no matching variables.

In the type-indexed implementation of PMC, all syntactica elements are defined as type-indexed forms. Variables is defined as follows.

The module defines variables and some auxiliary functions.

```
module Variable
  (Var (), mkVar, mkVar'
  , varName
  , relevantSuffix, renameAvoidingSuffixes
  , eqVar
  , HasVar (..), var', isVar
  , FreeIn, freeInV
  , HasVarType
  )
  where
import Data.Typeable
import PrelExts
import Data.Char
import Control.Monad (guard)
import qualified Data.Set as Set
```

In the definition of variables, *String* is variable name's type and every type-indexed variable has of type *Var a*, which is a variable type with type *a* as index type.

```
newtype Var a = V String
  deriving (Eq, Ord, Typeable)
```

In the definition of variables, *String* is variable name's type and every type-indexed variable has of type *Var a*, which is a variable type with type *a* as index type.

Since the module *Variable* exports *Var* as an abstract type, the constructor *V* is hidden and not exported. The following partial function *mkVar'* is provided to as the only interface to build a variable from a variable name of type *String*.

```
mkVar' :: forall a ∘ String → Var a
mkVar' = either error id ∘ mkVar
```

The function *mkVar* is used to facilitate defining the function *mkVar'*; it return a variable if the argument is a valid variable name or return an error message otherwise.

```
mkVar :: forall a ∘ String → Either String (Var a)
mkVar s = if isVarName s ∨ isOperator s then Right (V s)
  else Left $ "mkVar: illegal variable name or operator name ``" ++ s ++ "''"
```

Note that primitive operators are considered as variables in the implementation. For every primitive operator, a corresponding reduction rule has to be added in order to interpret it in the operational semantics and a correspondence between its variable in the implementation and real function in the source language has to be added into a semantic dictionary of type *TIMap* in the bimonadic semantics.

Variable names are directly showed.

```
instance Show (Var a) where
  show (V s) = s
  showsPrec _ (V s) = (s++)
```

eqVar is a type-indexed equality function of comparing two variables.

```
eqVar :: EQ1 Var
eqVar = eqCast (≡)

instance Eq1 Var where
  eq1 = eqVar
```

Var has an instance of *Functor* class.

```
instance Functor Var where
  fmap f (V s) = V s
```

The function *varName* returns variable names from variables.

```
varName :: Var a → String
varName (V s) = s
```

The function *isVarName* tells whether a string is a valid variable name or not.

```
isVarName :: String → Bool
isVarName = all (λc → isAlphaNum c ∨ c ∈ "'")
```

The function *isOperator* tells whether a string is a valid variable name or not. In the implementation, operators are considered as variables to implement.

```
isOperator :: String → Bool
isOperator s = s ∈ ["+", "-", "*", "/", "==", "/=", "<=", "++", "fix'"]
```

When renaming variables, we avoid existing variables in a context by first collecting their *relevant* suffixes, where relevance depends on the renaming tactic, which here is adding primes.

```
relevantSuffix :: Var a → Var b → Maybe String
relevantSuffix (V v1) (V v2) = do
  suffix ← dropPrefix v1 v2
  guard (all ('\'' ≡) suffix)
  return suffix

renameAvoidingSuffixes :: Var a → Set.Set String → Var a
renameAvoidingSuffixes (V v) ss = V $ v ++ head (filter ok $ iterate ('\'':) "'")
  where ok suff = ¬ (Set.member suff ss)
```

The following code defines class *HasVar* and some auxiliary functions.

```
class HasVar s where
  var :: (Typeable a) ⇒ Var a → s a
  hasVar :: (Typeable a) ⇒ s a → Bool
  getVar :: (Typeable a) ⇒ s a → Maybe (Var a)
  freeIn :: FreeIn s

isVar :: (HasVar s, Typeable a) ⇒ s a → Bool
isVar = maybe False (const True) ∘ getVar

instance HasVar Var where
  var = id
  hasVar = hasVarV
  getVar = Just
  freeIn = freeInV

type HasVarType s = forall a ∘ Typeable a ⇒ s a → Bool

hasVarV :: HasVarType Var
hasVarV = const True

var' :: (HasVar s, Typeable a) ⇒ String → s a
var' s = var (mkVar' s)

type FreeIn s = forall a b ∘ (Typeable a, Typeable b) ⇒ Var a → s b → Bool

freeInV :: FreeIn Var
freeInV v v' = case gcast v' of
  Nothing → False
  Just v'' → v ≡ v''
```

A.2 Constructors

We try to provide an abstract datatype for constructors that are type-indexed in a disciplined way, enabling syntactic distinction between full and partial constructor application.

```
module Constructor
(CResult (..), CArg (..)
, CType
) where
import Data.Typeable
import qualified TIMap as ECM
import qualified TIMap as PCM
import Control.Monad.Identity
```

We use the Haskell type system to enforce full application of constructors to all arguments by defining a special encoding of constructor types.

Constants expecting no arguments have a *CResult* type:

```
data CResult a = CResult String
  deriving Typeable
```

Constructors expecting arguments have a *CArg* type:

For adding an additional first expected argument of type *a*, the constructor type is wrapped in *CArg c*

```
data CArg a c = CArg c
  deriving Typeable
```

The following class relates constructor type encodings with the encoded types:

```
class CType c t | c → t where
instance CType (CResult a) a
instance CType c b ⇒ CType (CArg a c) (a → b)
```

The *Show*, *Ord*, and *Typeable* constraints are necessary since GHC cannot use closed type classes (*CType* is closed since not exported).

We need some standard instances:

```
instance Eq (CResult a) where
  CResult x ≡ CResult y = x ≡ y
instance Eq c ⇒ Eq (CArg a c) where
  CArg x ≡ CArg y = x ≡ y
instance Ord (CResult a) where
  compare (CResult x) (CResult y) = compare x y
  CResult x ⩽ CResult y = x ⩽ y
```

```
instance Ord c ⇒ Ord (CArg a c) where
  compare (CArg x) (CArg y) = compare x y
  CArg x ⩽ CArg y = x ⩽ y
instance Show (CResult a) where
  showsPrec _ (CResult s) = (s++)
instance Show c ⇒ Show (CArg a c) where
  showsPrec p (CArg c) = showsPrec p c
```

Since we could not express the functional dependency $t \to c$ in class *CType*, we need to *cast* before being able to compare two *Constant* arguments — this is the reason for the *Typeable* constraint in *Constant*.

A.3 Patterns

```
module Pattern where
import Variable
import Constructor
import Data.Typeable
  -- import TypeCombinators
import PrelExts
```

A.3.1 The Definition of Patterns

The following defintion mirros exactly the abstract syntax of *patterns*.

```
data Pat :: * → *where
  VarPat :: Typeable a ⇒ Var a → Pat a
  ConstrPat :: ConstrApp Pat (CResult a) → Pat a
```

Variables should be type-indexed. Therefore, we use *Var a* instead of *Var*.

We parameterise the type of fully applied constructor applications with the syntactic category *s* so that we can use this both for patterns and expressions.

```
data ConstrApp :: (* → *) → * → *where
  Constr :: c → ConstrApp s c
  ConstrApply :: Typeable a ⇒ ConstrApp s (CArg a c) → s a → ConstrApp s c

infixl 9 'ConstrApply'

tcConstrApp = mkTyCon "ConstrApp"
instance (Typeable1 s) ⇒ Typeable1 (ConstrApp s) where
  typeOf1 (x :: ConstrApp s c) = mkTyConApp tcConstrApp
    [typeOf1 (⊥ :: s c)]
```

A.3.2 HasVar and HasConstructorApp classes and instances

```
class HasConstructorApp s where
  constrApp :: (Typeable a) ⇒ ConstrApp s (CResult a) → s a
  getConstrApp :: (Typeable a) ⇒ s a → Maybe (ConstrApp s (CResult a))

instance HasConstructorApp Pat where
  constrApp = ConstrPat
  getConstrApp (ConstrPat ca) = Just ca
  getConstrApp _ = Nothing

instance HasVar Pat where
  var = VarPat
  hasVar = hasVarP
  getVar (VarPat v) = Just v
  getVar _ = Nothing
  freeIn = freeInP

hasVarP :: HasVarType Pat
hasVarP (VarPat v) = True
hasVarP (ConstrPat ca) = hasVarCA ca

hasVarCA :: HasVar s ⇒ HasVarType (ConstrApp s)
hasVarCA (Constr c) = False
hasVarCA (ConstrApply ca s) = hasVarCA ca ∨ hasVar s

freeInP :: FreeIn Pat
freeInP v (VarPat v') = freeInV v v'
freeInP v (ConstrPat ca) = freeInCA freeInP v ca

freeInCA :: FreeIn s → FreeIn (ConstrApp s)
freeInCA freeIn v (Constr c) = False
freeInCA freeIn v (ConstrApply ca s) = freeInCA freeIn v ca ∨ freeIn v s
```

A.4 Type-Indexed Syntax of Pattern Matching Calculi

```
module PMC where
import Variable
import Constructor
import Data.Typeable
import PrelExts
import Pattern
```

A.4.1 Type-Indexed Implementation of Syntax of Pattern Matching Calculi

The mechanism for using type-indexed expressions to model PMC data structures can offer both convenience in programming and clarity in code. By using type-indexed expressions, we can model PMC data structures with surprising accuracy. The following definitions of *expressions* and *matchings* exactly mirror the original definitions in [11].

```
data Expr :: * → *where
  EVar   :: Typeable a ⇒ Var a → Expr a
  ConstrExpr :: Typeable a ⇒ ConstrApp Expr (CResult a) → Expr a
  Apply  :: (Typeable a, Typeable (a → b), Typeable b) ⇒
            Expr (a → b) → Expr a → Expr b
  MExpr :: Typeable a ⇒ Match a → Expr a
  Empty :: Typeable a ⇒ Expr a
  EFix   :: Typeable a ⇒ Expr ((a → a) → a)
```

In order to be able to *match* patterns' *constructor functions* with expressions' *constructor functions*, we have to define *Expr*' data type regarding *constructor functions* in the same way as we define *Pat*'s data type.

```
tcExpr = mkTyCon "PMC.Expr"
instance Typeable1 Expr where
  typeOf1 (x :: Expr a) = mkTyConApp tcExpr []
instance Ord a ⇒ Ord (Expr a) where
instance Eq a ⇒ Eq (Expr a) where
```

For convenience, we declare the infix form of the application constructors as high-priority infix operators:

```
infixl 9 ‘Apply‘
infixr 3 ‘PMatch‘
infixr 3 ‘Supply‘
infixr 2 ‘MAlt‘

data Match :: * → *where
  Return :: Typeable a ⇒ Expr a → Match a
  Fail   :: Typeable a ⇒ Match a
  PMatch :: (Typeable a, Typeable b) ⇒ Pat a → Match b → Match (a → b)
  Supply :: (Typeable a, Typeable b) ⇒ Expr a → Match (a → b) → Match b
  MAlt :: Typeable a ⇒ Match a → Match a → Match a

tcMatch = mkTyCon "PMC.Match"
instance Typeable1 Match where
  typeOf1 (x :: Match a) = mkTyConApp tcMatch []
```

Note that *CResult* and *CArg* only serve to ensure that constructor applications apply constructors to the correct number of arguments. They will **never** show up in expression types.

A.4.2 HasVar instance

```
instance HasVar Expr where
  var = EVar
  hasVar = hasVarE
  getVar (EVar v) = Just v
  getVar _ = Nothing
  freeIn = freeInE

hasVarE :: HasVarType Expr
hasVarE (EVar v) = True
hasVarE (ConstrExpr ca) = hasVarCA ca
hasVarE Empty = False
hasVarE EFix = False
hasVarE (MExpr m) = hasVarM m
hasVarE (Apply f a) = hasVarE f ∨ hasVarE a

hasVarM :: HasVarType Match
hasVarM (Return e) = hasVarE e
hasVarM Fail = False
hasVarM (Supply a m) = hasVarE a ∨ hasVarM m
hasVarM (MAlt m1 m2) = hasVarM m1 ∨ hasVarM m2
hasVarM (PMatch p m) = hasVarP p ∨ hasVarM m
```

Appendix B

Text Representations of PMC Terms

The appendix includes modules that define Text Representations of PMC Terms.

B.1 Text Representation of PMC Terms

```
module PMCText where
import Pattern
import PMC
import Variable
import PrelExts
import Data.Typeable
```

The *Show* instances for expressions and patterns are built with functions that for typing reasons have to be defined separately:

```
instance Typeable a ⇒ Show (Pat a) where
  showsPrec = showsPrecPat
instance Typeable a ⇒ Show (Expr a) where
  showsPrec = showsPrecExpr
```

The *showsPrec* functions for expressions and patterns call *showsPrecConstrApp* with *themselves at explicitly polymorphic type as arguments*, so this is a somewhat unusual instance of polymorphic recursion.

```
showsPrecPat :: forall a ∘ Typeable a ⇒ ShowSPrec (Pat a)
showsPrecPat p (VarPat v) = showsPrec p v
showsPrecPat p (ConstrPat c) = showsPrecConstrApp showsPrecPat p c

showsPrecExpr :: forall a ∘ Typeable a ⇒ ShowSPrec (Expr a)
showsPrecExpr p (EVar v)      = showsPrec p v
showsPrecExpr p (ConstrExpr c) = showsPrecConstrApp showsPrecExpr p c
showsPrecExpr p (Apply f a) = parenShows (p > 10) $
  showsPrecExpr 10 f ∘ (' ':) ∘ showsPrecExpr 11 a
showsPrecExpr p (MExpr a)   = encloseShows '{' '}' $ shows a
showsPrecExpr p Empty       = ("empty"++)
showsPrecExpr p EFix        = ("fix"++)
```

Using these (or directly their *showsPrec* names, we can also define *Show* instances for the relevant constructor application types:

```
instance (Typeable a, Show a) ⇒ Show (ConstrApp Expr a) where
  showsPrec = showsPrecConstrApp showsPrecExpr
instance (Typeable a, Show a) ⇒ Show (ConstrApp Pat a) where
  showsPrec = showsPrecConstrApp showsPrecPat
```

The *Show* instance for matchings is not affected by all this.

```
instance Typeable a ⇒ Show (Match a) where
  showsPrec p (Return e)     = encloseShows '|' '|' $ shows e
  showsPrec p Fail = ("fail"++)
  showsPrec p (PMatch pat m) = parenShows (p > 3) $
    showsPrec 4 pat ∘ (" => "++) ∘ showsPrec 3 m
  showsPrec p (Supply e m)   = parenShows (p > 3) $
    showsPrec 4 e ∘ (" >> "++) ∘ showsPrec 3 m
  showsPrec p (MAlt m1 m2) = parenShows (p > 2) $
    showsPrec 2 m1 ∘ (" || "++) ∘ showsPrec 2 m2
```

For constructor applications *ConstrApp*, we pass in a *polymorphic showsPrec* function for the arguments; the function itself uses polymorphic recursion, i.e., the recursive call is at a different type from the occurrence in the left-hand side — this is only possible with an explicit type signature.

Meanwhile, we deal in particular with list and pair show. Empty list is shown as "[]" and singleton list [*a*] as "[a]". Many-element list [*a'1*, *a'2*, ..., *a'n*] is shown exactly in default Haskell style as well. We also deal with pair show in similar way. As for other constructors, we show them as normal functions, that is, first constructor functiona name, then the first parameter and so on.

A normal pattern constructor function application is like
ConstrApply (...(*ConstrApply* (*Constr* (*CArg* (...(*CArg* (*CResult c*))...)) *varPat'1*)...)$*varPat'n*$
As stated before, the *polymorphic showsPrec* can be used to show *varPat'i*.

However, (*CArg* (...(*CArg* (*CResult c*))...)) can only be shown using *show* instance in *Constructor* module, considering that we cannot use a recursive function to show it.

Considering that both list and pair constructors are binary function, we can write *showsPrecConstrApp* as follows to show list and pair as we expect.

The following *showsPrecConstrApp* shows all constructors as prefix notation, excepting ":" and "(,)"".

```
showsPrecConstrApp :: (Show c, Typeable c, HasVar s) ⇒
    (forall a ∘ Typeable a ⇒ ShowSPrec (s a)) → Int → ConstrApp s c → ShowS
showsPrecConstrApp showsPrecS p (Constr c) = showsPrec p c
showsPrecConstrApp showsPrecS p (ConstrApply c s) = case hasVarCA c ∨ hasVar s of
False →
  case c of
```

```
        ConstrApply (Constr c2) s2 →
          case showsPrec p c2 "" of
            ":" →
              case showsPrecS p s "" of
                "[]" → bracketShows (p > −1) $ showsPrecS 0 s2
                _ → bracketShows (p > −1) $
                  showsPrecS 0 s2 ∘ (',':) ∘ showsPrecS (−1) s
            "(,)" → parenShows True $ showsPrecS 1 s2 ∘ (',':) ∘ showsPrecS 1 s
            infixOp → parenShows (p > 1) $
              showsPrecS 2 s2 ∘ (' ':) ∘ (infixOp++) ∘ (' ':) ∘ showsPrecS 2 s
        _ → parenShows (p > 1) $
          showsPrecConstrApp showsPrecS 1 c ∘ (' ':) ∘ showsPrecS 0 s
    True →
      case c of
        ConstrApply (Constr c2) s2 →
          case showsPrec p c2 "" of
            ":" → parenShows (p > 1) $ showsPrecS 1 s2 ∘ (':':) ∘ showsPrecS 2 s
            "(,)" → parenShows True $ showsPrecS 2 s2 ∘ (',':) ∘ showsPrecS 2 s
            infixOp@(':' : _) → parenShows (p > 1) $
              showsPrecS 2 s2 ∘ (' ':) ∘ (infixOp++) ∘ (' ':) ∘ showsPrecS 2 s
            prefixConstr → parenShows (p > 1) $
              showsPrecConstrApp showsPrecS 1 c ∘ (' ':) ∘ showsPrecS 2 s
        _ → parenShows (p > 1) $
          showsPrecConstrApp showsPrecS 1 c ∘ (' ':) ∘ showsPrecS 2 s
```

B.2 Examples of Text Representations of PMC Terms

```
module PMCTextExample where
import Pattern
import PMC
import PMCLib
import Variable
import Constructor
import Data.Typeable
import PMCText
```

Some *Show* examples:

```
cons :: CArg Int (CArg [Int] (CResult [Int]))
cons = mkC2 ":"
```

```
cons2 :: CArg [Int] (CArg [[Int]] (CResult [[Int]]))
cons2 = mkC2 ":"
list23 = cExpr2 cons (mkExpr "2" :: Expr Int) list3
list3 = cExpr2 cons (mkExpr "3" :: Expr Int) (mkExpr "[]" :: Expr [Int])
x = mkEVar "x" :: Expr [Int]
ys = mkEVar "ys" :: Expr [[Int]]
list23xys = cExpr2 cons2 list23 $
  cExpr2 cons2 x ys
listx23ys = cExpr2 cons2 x $
  cExpr2 cons2 list23 ys
list23x23ys = cExpr2 cons2 list23 $
  cExpr2 cons2 x $
  cExpr2 cons2 list23 ys
```

```
*PMCTextExample> list23xys
[2,3]:(x:ys)

*PMCTextExample> listx23ys
x:([2,3]:ys)

*PMCTextExample> list23x23ys
[2,3]:(x:([2,3]:ys))
```

Appendix C

Tool Modules from Kahl's work

The appendix includes modules from Kahl's work [14].

C.1 Type-Indexed Maps

This module provides an implementation of *type-indexed maps*, that is, values *m* :: *TIMap k r* representing type-indexed families $m = (m_a)_{a::*}$ of maps m_a::*Map* (*k a*) (*r a*) where both the source and the target type may depend on the index.

This is made possible by the type-safe casts from Data.*Typeable* and the arbitrary-rank polymorphism supported by GHC with `-fglasgow-exts`.

This module is intended for *qualified import*, and exports an interface that is an appropriately adapted sub-interface of the interface of Data.*Map*, the new finite-map module shipping with GHC-6.4.

```
module TIMap
  (TIMap ()
  , lookup
  , null, size, member
  , fold, foldWithKey
  , empty, insert, singleton, delete
  )
  where
import Prelude hiding (lookup, filter, foldr, foldl, null, map)
import qualified Data.Map as Map
import Data.Typeable
import Data.Maybe (isJust)
```

We define a type-indexed map as a list of *Map*s, where each *Map* is the component map for a specific type.

For these *type-specific maps*, we need a newtype so that *gcast* can be applied to them directly:

```
newtype TSMap k r a = TSMap (Map.Map (k a) (r a))
```

Since *k*, *r* :: $* \rightarrow *$ are higher-kind type variables, GHC currently does not *derive* any *Typeable* instances for this, but it is straight-forward to produce the basic instance ourselves:

```
instance (Typeable1 k, Typeable1 r, Typeable a) ⇒ Typeable (TSMap k r a) where
  typeOf (_ :: TSMap k r a) = mkTyConApp (mkTyCon "TIMap.TSMap")
```

```
    [typeOf1 (⊥ :: k a)
    , typeOf1 (⊥ :: r a)
    , typeOf (⊥ :: a)
    ]
```

A type-indexd map is then implemented essentially as a list of existentially quantified type-specific maps — we use GADT notation to define this in a single definition as a specialised list type (the *Typeable* instance has to be done manually again).

```
data TIMap :: (* → *) → (* → *) → *where
  Empty :: TIMap k r
  Cons :: (Typeable a, Ord (k a)) ⇒ TSMap k r a → TIMap k r → TIMap k r
instance (Typeable1 k, Typeable1 r) ⇒ Typeable (TIMap k r) where
  typeOf (_ :: TIMap k r) = mkTyConApp (mkTyCon "TIMap.TIMap")
    [typeOf1 (⊥ :: k ())
    , typeOf1 (⊥ :: r ())
    ]
```

The constructors are not exported. The exported interface will guarantee the *invariant* that no two elements of such a list have the same type, and that no list element is an empty type-specific map.

A more efficient implementation could be implemented via a *Map TypeRep* (*ETSMap k r*) — this would need an *Ord* instance for *TypeRep* (currently not provided in Data.*Typeable*), and a wrapper type *ETSMap* for the existentially quantified version of *TSMap*.

For lookup, we use *gcast* on each list element to test whether it has the right type for the argument; if it has, then, according to the *TIMap k* invariant, it is the only list element of that type, and Map.*lookup* produces the result.

```
lookup :: (Typeable a, Ord (k a)) ⇒ k a → TIMap k r → Maybe (r a)
lookup v Empty = Nothing
lookup v (Cons tsm tim) = case gcast tsm of
  Nothing → lookup v tim
  Just (TSMap m) → case Map.lookup v m of
    Nothing → lookup v tim
    j → j
```

Essentially the same pattern is used for implementing *insert* and *delete*:

```
insert :: (Typeable a, Ord (k a)) ⇒ k a → r a → TIMap k r → TIMap k r
insert v x Empty = Cons (TSMap $ Map.singleton v x) Empty
insert v x (Cons tsm tim) = case gcast tsm of
  Just (TSMap m) → Cons (TSMap $ Map.insert v x m) tim
  Nothing → Cons tsm (insert v x tim)

delete :: (Typeable a, Ord (k a)) ⇒ k a → TIMap k r → TIMap k r
delete v Empty = Empty
delete v (Cons tsm tim) = case gcast tsm of
  Just (TSMap m) →
    let m' = Map.delete v m
    in if Map.null m'
      then tim
      else Cons (TSMap m') tim
  Nothing → Cons tsm (delete v tim)
```

```
union :: (Typeable a, Ord (k a)) => TIMap k r -> TIMap k r -> TIMap k r
union = Map.union
```

For the folding functions, the plymorphic argument function can rely on being invoked only at instances *a* where *k a* has an *Ord* instance and *a* has a *Typeable* instance. If we were to omit this last constraint, many natural applications, as for example TISet.*isSubsetOf*, would become impossible.

```
fold :: (forall a ∘ (Typeable a, Ord (k a)) ⇒
  r a → b → b) → b → TIMap k r → b
fold f = foldWithKey (const f)
foldWithKey :: (forall a ∘ (Typeable a, Ord (k a)) ⇒
  k a → r a → b → b) → b → TIMap k r → b
foldWithKey f e = h
  where
    h Empty = e
    h (Cons (TSMap tsm) tim) = Map.foldWithKey f (h tim) tsm
```

The remaining items from the *Map* interface that we choose to implement right now can be implemented directly or via the functions already shown without further complications.

```
empty :: TIMap k r
empty = Empty

singleton :: (Typeable a, Ord (k a)) ⇒ k a → r a → TIMap k r
singleton v x = insert v x empty

null :: TIMap k r → Bool
null Empty = True
null _ = False
size :: TIMap k r → Int
size Empty = 0
size (Cons (TSMap tsm) tim) = Map.size tsm + size tim
member :: (Typeable a, Ord (k a)) ⇒ k a → TIMap k r → Bool
member v tsm = isJust (lookup v tsm)
```

C.2 Q-Combinators

The *q*-combinators, adapted from John Harrison's HOL-Light, serve for saving unneccessary updates and thereby maximising sharing: If an argument function of type *a* → *Maybe a* returns *Nothing*, this is taken to mean "no change".

```
module QCombinators where
import Control.Monad (mplus)

type Q a = a → Maybe a

qtry :: Q a → a → a
qtry f x = maybe x id (f x)

qalt :: Q a → Q a → Q a
qalt t1 t2 e = t1 e ‘mplus‘ t2 e

qseq :: Q a → Q a → Q a
qseq f g x = case f x of
  Nothing → g x
  Just x' → case g x' of
    Nothing → Just x'
    j → j

qjoin :: (a → b → c) → Q a → Q b → a → b → Maybe c
qjoin f gx gy x y =
  case gx x of
    Just x' → Just $ f x' $ qtry gy y
    Nothing → fmap (f x) (gy y)
qjoin' :: ((a, b) → c) → Q a → Q b → (a, b) → Maybe c
qjoin' f gx gy (x, y) = qjoin (curry f) gx gy x y
qcomb :: (a → a → b) → Q a → a → a → Maybe b
qcomb con fn = qjoin con fn fn

qjoin3 :: (a → b → c → d) → Q a → Q b → Q c → a → b → c → Maybe d
qjoin3 f gx gy gz x y z =
  case gx x of
    Just x' → Just $ uncurry (f x') $ qtry (qjoin' id gy gz) (y, z)
    Nothing → qjoin (f x) gy gz y z

qpupd1 :: Q a → Q (a, b)
qpupd1 f (x, y) = fmap (λx → (x, y)) $ f x
qpupd2 :: Q b → Q (a, b)
qpupd2 f (x, y) = fmap (λy → (x, y)) $ f y
```

```
qmaybe :: Q a → Q (Maybe a)
qmaybe f Nothing = Nothing
qmaybe f (Just x) = fmap Just $ f x

qmap :: Q a → Q [a]
qmap f [] = Nothing
qmap f (x : xs) = case f x of
  Just x' → Just (x' : qtry (qmap f) xs)
  Nothing → fmap (x:) (qmap f xs)
```

With general monads:

```
type QM m a = a → m (Maybe a)

mqtry :: Monad m ⇒ QM m a → a → m a
mqtry f x = do mx ← f x
  return $ maybe x id mx

mqjoin :: (Functor m, Monad m) ⇒
  (a → b → c) → QM m a → QM m b → a → b → m (Maybe c)
mqjoin f gx gy x y =
do mx ← gx x
  case mx of
    Just x' → do y' ← mqtry gy y
      return $ Just $ f x' y'
    Nothing → fmap (fmap (f x)) (gy y)
mqcomb :: (Functor m, Monad m) ⇒
  (a → a → b) → QM m a → a → a → m (Maybe b)
mqcomb con fn = mqjoin con fn fn
```

C.3 Transformations and Transformation Combinators

```
module Trafo where
import PMC
import QCombinators
import Data.Typeable
import PrelExts

type Trafo s = forall a ∘ (Typeable a) ⇒ Q (s a)
```

```
mkTrafo :: Typeable a => Q (s a) -> Trafo s
mkTrafo f a = gcast a >>= f >>= gcast
```

```
seq', alt :: Trafo s → Trafo s → Trafo s
```

```
seq' = qseq
alt = qalt
twice :: Trafo s → Trafo s
twice t = t =>>= t
triply :: Trafo s → Trafo s
triply t = t =>>= t =>>= t
repeat' :: Trafo s → Trafo s
repeat' t = t'
where
  t' x = case t x of
    Nothing → Nothing
    j@(Just x') → case t' x' of
      Nothing → j
      j' → j'
```

C.4 Transformation Transformers

The module *PMCTrafo* includes the transformation rules over all the syntactic structures of PMC expressions and matchings. The transformation rules are implementation basis for the leftmost-outermost strategy in 3.4.1 and the normalising strategy in 3.4.2.

We first define the following type synonym for convenience. The type constructor *Trafo* in the definitions is defined in appendix C.3; it has the kind $* \rightarrow *$.

```
type TrafoE = Trafo Expr
type TrafoM = Trafo Match
type TrafoCA s = Trafo (ConstrApp s)
```

"Transformation transformers" apply transformations inside determined constructor arguments, i.e., every transformation transformer take a "primitive" reduction rule, which is a transformation, and return another new transformation.

- The syntactic definition of *Expr* gives rise to the following transformation transformers.
  - The following transformer transforms a PMC expression with the syntactic structure *ConstrExpr c*.

```
inConstrExpr :: TrafoE → TrafoE
inConstrExpr t (ConstrExpr ca) = fmap ConstrExpr $ inCA t ca
inConstrExpr t _ = Nothing
inCA :: forall c ∘ TrafoE → ConstrApp Expr c → Maybe (ConstrApp Expr c)
inCA t (Constr c) = Just (Constr c)
inCA t (ConstrApply ca e) = do
  e' ← t e
  ca' ← inCA t ca
  return $ ConstrApply ca' e'
```

- The two following transformers transform a PMC expression with the syntactic structure *Apply f a* in two different ways.

```
inApplyL :: TrafoE → TrafoE
inApplyL t (Apply f a) = fmap (flip Apply a) $ t f
inApplyL t _ = Nothing

inApplyR :: TrafoE → TrafoE
inApplyR t (Apply f a) = fmap (Apply f) $ t a
inApplyR t _ = Nothing
```

- The following transformer transforms a PMC expression with the syntactic structure *MExpr m*.

```
inMExpr :: TrafoM → TrafoE
inMExpr t (MExpr m) = fmap MExpr $ t m
inMExpr t _ = Nothing
```

- The following transformer transforms a PMC expression with the syntactic structure *Apply EFix f*.

```
inEFix :: TrafoE → TrafoE
inEFix t e@(Apply EFix f) = t $ Apply f e
inEFix t _ = Nothing
```

• The syntactic definition of *Match* gives rise to the following transformation transformers.

- The following transformer transforms a PMC matching with the syntactic structure *PMatch p m*.

```
inPMatch :: TrafoM → TrafoM
inPMatch t (PMatch p m) = fmap (PMatch p) $ t m
inPMatch t _ = Nothing
```

- The two following transformers transform a PMC matching with the syntactic structure *Supply a m* in two different ways.

```
inSupplyL :: TrafoE → TrafoM
inSupplyL t (Supply a m) = fmap (flip Supply m) $ t a
inSupplyL t _ = Nothing

inSupplyR :: TrafoM → TrafoM
inSupplyR t (Supply a m) = fmap (Supply a) $ t m
inSupplyR t _ = Nothing
```

- The two following transformers transform a PMC matching with the syntactic structure *MAlt m1 m2* in two different ways.

```
inMAltL :: TrafoM → TrafoM
inMAltL t (MAlt m1 m2) = fmap (flip MAlt m2) $ t m1
inMAltL t _ = Nothing

inMAltR :: TrafoM → TrafoM
inMAltR t (MAlt m1 m2) = fmap (MAlt m1) $ t m2
inMAltR t _ = Nothing
```

- The following transformer transforms a PMC matching with the syntactic structure *Return e*.

```
inReturn :: TrafoE → TrafoM
inReturn t (Return e) = fmap Return $ t e
inReturn t _ = Nothing
```

The three following transformations are to determine whether a PMC matching has some structure or not. These transformations succeed (without changing anything) for their selected constructors, and fail otherwise. Ihe result will decide which transformations have to be applied next.

```
guardSupply :: TrafoM
guardSupply m@(Supply _ _) = Just m
guardSupply _ = Nothing

guardPMatch :: TrafoM
guardPMatch m@(PMatch _ _) = Just m
guardPMatch _ = Nothing
notGuardPMatch :: TrafoM
notGuardPMatch (PMatch _ _) = Nothing
notGuardPMatch m = Just m
```

C.5 Prelude Extensions

```
module PrelExts where
import Data.Typeable
```

C.5.1 Material Related to **Show**

```
type PrecShowS = Int → ShowS
type ShowSPrec a = Int → a → ShowS
```

```
class Show1 f where
  shows1 :: Show a ⇒ f a → ShowS

class ShowF f where
  showsPrecF :: ShowSPrec a → ShowSPrec (f a)

encloseShows :: Char → Char → ShowS → ShowS
encloseShows open close shows = (open:) ∘ shows ∘ (close:)
parenShows :: Bool → ShowS → ShowS
parenShows False shows = shows
parenShows True shows = encloseShows '(' ')' shows
bracketShows :: Bool → ShowS → ShowS
bracketShows False shows = shows
bracketShows True shows = encloseShows '[' ']' shows
```

C.5.2 Lists

```
dropPrefix :: Eq a ⇒ [a] → [a] → Maybe [a]
dropPrefix [] ys = Just ys
dropPrefix (x : xs) (y : ys) = if x ≡ y then dropPrefix xs ys else Nothing
dropPrefix _ _ = Nothing
```

C.5.3 Monads

```
(=>>=) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f =>>= g = λx → f x ≫= g
```

C.5.4 Other Datatypes

```
class Functor f ⇒ Container f where
  elems :: f a → [a]

type EQ1 f = forall a b ∘ (Typeable a, Typeable b) ⇒ f a → f b → Bool
class Eq1 (f :: * → *) where
  eq1 :: EQ1 f

eqCast :: (forall a ∘ s a → s a → Bool) → EQ1 s
eqCast eq x x' = case gcast x of
  Nothing → False
  Just x˙ → eq x˙ x'
```

Appendix D

α-conversion

D.1 α-conversion

This module is used to implement variable scoping in the section 3.1.

```
module AlphaConversion where
import Pattern
import PMC
import Variable
import Constructor
import TIMap as Su   -- used here as substitutions
import QCombinators
import Data.Set as Set
import Data.Typeable
```

D.1.1 α-conversion

```
type Substitution = Su.TIMap Var Expr
```

α-conversion to avoid range variables of a substitution inside a binder, at the same time eliminating the bound variables from the domain of the substitution:

```
type Alpha s = forall a b ∘ (Typeable a, Typeable b) ⇒
  s a → Match b → Substitution → (s a, Match b, Substitution)
alphaV :: Alpha Var
alphaV v m su = let
  su' = Su.delete v su
  ranSuffixes = Su.fold (λe → Set.union (varSuffixesE v e)) Set.empty su'
  mSuffixes = varSuffixesM v m
in if Set.member "" ranSuffixes
  then let v' = renameAvoidingSuffixes v $ Set.union ranSuffixes mSuffixes
    in (v', qtry (renameVarM v v') m, su')
  else (v, m, su')
alphaP :: Alpha Pat
alphaP (VarPat v) m su = let (v', m', su') = alphaV v m su
  in (VarPat v', m', su')
```

```
alphaP (ConstrPat ca) m su = let (ca', m', su') = alphaCA ca m su
  in (ConstrPat ca', m', su')
alphaCA :: Alpha (ConstrApp Pat)
alphaCA ca@(Constr c) m su = (ca, m, su)
alphaCA (ConstrApply ca p) m su = let
  (ca', m', su') = alphaCA ca m su
  (p', m'', su'') = alphaP p m' su'
in (ConstrApply ca' p', m'', su'')
```

D.1.2 Variable Suffixes

For variable renaming, we collect all suffixes of variables (free and bound) occurring in an expression that have the bound variable name as prefix:

```
type GetVarSuffixes s = forall a b ∘ Var a → s b → Set.Set String

varSuffixesV :: GetVarSuffixes Var
varSuffixesV v v' = case relevantSuffix v v' of
  Nothing → Set.empty
  Just s → Set.singleton s

varSuffixesE :: GetVarSuffixes Expr
varSuffixesE v (EVar v') = varSuffixesV v v'
varSuffixesE v (ConstrExpr c) = varSuffixesConstrApp varSuffixesE v c
varSuffixesE v Empty = Set.empty
varSuffixesE v EFix = Set.empty
varSuffixesE v (Apply f a) = Set.union (varSuffixesE v f) (varSuffixesE v a)
varSuffixesE v (MExpr m) = varSuffixesM v m

varSuffixesM :: GetVarSuffixes Match
varSuffixesM v (Return e) = varSuffixesE v e
varSuffixesM v Fail = Set.empty
varSuffixesM v (Supply a m) = Set.union (varSuffixesE v a) (varSuffixesM v m)
varSuffixesM v (MAlt m1 m2) = Set.union (varSuffixesM v m1) (varSuffixesM v m2)
varSuffixesM v (PMatch p m) = Set.union (varSuffixesP v p) (varSuffixesM v m)

varSuffixesP :: GetVarSuffixes Pat
varSuffixesP v (VarPat v') = varSuffixesV v v'
varSuffixesP v (ConstrPat c) = varSuffixesConstrApp varSuffixesP v c

varSuffixesConstrApp :: GetVarSuffixes s → GetVarSuffixes (ConstrApp s)
varSuffixesConstrApp varSuffixes v (Constr c) = Set.empty
varSuffixesConstrApp varSuffixes v (ConstrApply ca s) =
  Set.union (varSuffixesConstrApp varSuffixes v ca) (varSuffixes v s)
```

D.1.3 Renaming Variables

```
type Rename s = forall a b ∘ (Typeable a, Typeable b) ⇒ Var a → Var a → Q (s b)
```

Renaming assumes that the new variable is not captured by any binders. This had to be defined separately since calling substitution in *Alpha* would have produced mutually recursive functions with different contexts.

```
renameVarV :: Rename Var
renameVarV u v w = case gcast u of
  Nothing → noChange
  Just u' → if u' ≢ w then noChange
    else case gcast v of
      Nothing → noChange
      Just v' → changed v' w
  where noChange = Just w
    changed v' w = Just v'

renameVarM :: Rename Match
renameVarM v v' Fail          = Just Fail
renameVarM v v' (Return e) = fmap Return $ renameVarE v v' e
renameVarM v v' (MAlt m1 m2) = qcomb MAlt (renameVarM v v') m1 m2
renameVarM v v' (Supply e m) = qjoin Supply (renameVarE v v')
  (renameVarM v v') e m
renameVarM v v' (PMatch p m) = if v `freeInP` p then Just (PMatch p m)
  else fmap (PMatch p) $ renameVarM v v' m

renameVarE :: Rename Expr
renameVarE v v' (EVar w)       = fmap EVar $ renameVarV v v' w
renameVarE v v' (Apply e1 e2) = qjoin Apply (renameVarE v v')
                                    (renameVarE v v') e1 e2
renameVarE v v' (MExpr m)     = fmap MExpr $ renameVarM v v' m
renameVarE v v' Empty          = Just Empty
renameVarE v v' EFix           = Just EFix
renameVarE v v' (ConstrExpr ca) = fmap ConstrExpr $
                                    renameVarCA renameVarE v v' ca

renameVarCA :: Rename s → Rename (ConstrApp s)
renameVarCA rename v v' (Constr c) = Just (Constr c)
renameVarCA rename v v' (ConstrApply ca e)
  = qjoin ConstrApply (renameVarCA rename v v') (rename v v') ca e
```

D.2 α-conversion Examples

```
module AlphaConversionExample where
import Pattern
import PMC
import PMCLib
import Variable
import Constructor
import TIMap as Su  -- used here as substitutions
import QCombinators
import Data.Set as Set
import Data.Typeable
import AlphaConversion
```

alphaV Examples

```
v'm'su1 = alphaV (mkVar' "x" :: Var Int)
  (PMatch (mkPVar "x" :: Pat Int) (Return (mkEVar "x") :: Match Int))
  (Su.insert
    (mkVar' "z" :: Var Int)
    (cExpr1 (mkC1 "+5" :: CArg Int (CResult Int))
      (mkEVar "x" :: Expr Int)
    )
    Su.empty
  )
v'm'su2 = alphaV (mkVar' "x" :: Var Int)
  (PMatch (mkPVar "y" :: Pat Int) (Return (mkEVar "x") :: Match Int))
  (Su.insert
    (mkVar' "x" :: Var Int)
    (cExpr1 (mkC1 "+5" :: CArg Int (CResult Int))
      (mkEVar "x" :: Expr Int)
    )
    Su.empty
  )
v'm'su3 = alphaV (mkVar' "x" :: Var Int)
  (PMatch (mkPVar "y" :: Pat Int) (Return (mkEVar "x") :: Match Int))
  (Su.insert
    (mkVar' "z" :: Var Int)
    (cExpr1 (mkC1 "+5" :: CArg Int (CResult Int))
      (mkEVar "x" :: Expr Int)
    )
```

```
        Su.empty
      )

*NormExample> case v_m_su1 of (v,m,su) -> v
x'
*NormExample> case v_m_su1 of (v,m,su) -> m
x => |x|

*NormExample> case v_m_su2 of (v,m,su) -> v
x
*NormExample> case v_m_su2 of (v,m,su) -> m
y => |x|

*NormExample> case v_m_su3 of (v,m,su) -> v
x'
*NormExample> case v_m_su3 of (v,m,su) -> m
y => |x'|
```

varSuffixesV Examples

```
*NormExample> varSuffixesV (mkVar' "abc"::Var Int)
              (mkVar' "abc"::Var Int)
{""}

*NormExample> varSuffixesV (mkVar' "abc"::Var Int)
              (mkVar' "abc'''"::Var Int)
{"'''"}

*NormExample> varSuffixesV (mkVar' "abc"::Var Int)
              (mkVar' "abc123"::Var Int)
{}
```

varSuffixesE Examples

```
    set1 = varSuffixesE (mkVar' "abc" :: Var Int) (mkEVar "abc'''" :: Expr Int)
    set2 = varSuffixesE (mkVar' "abc" :: Var Int) (mkEVar "abc123" :: Expr Int)
    set3 = varSuffixesE (mkVar' "abc" :: Var Int)
      (cExpr1 (CArg (CResult "f") :: CArg Int (CResult Int))
              (mkEVar "abc'''" :: Expr Int)
      )
    set4 = varSuffixesE (mkVar' "abc" :: Var Int)
      (cExpr2 (CArg (CArg (CResult "f")) :: CArg Int (CArg Int (CResult Int)))
            (mkEVar "abc'''" :: Expr Int)
            (mkEVar "abc'" :: Expr Int)
    )

*NormExample> set1
{"'''"}
*NormExample> set2
{}
*NormExample> set3
{"'''"}
*NormExample> set4
{"'","'''"}
```

renameVarM Examples

```
renVM1 = Supply (mkExpr "22" :: Expr Int) $
  PMatch (mkPVar "y" :: Pat Int) $
  Return $ (mkEVar "+" :: Expr (Int → Int → Int))
    `Apply` (mkEVar "x" :: Expr Int)
    `Apply` (mkEVar "y" :: Expr Int)
renVM2 = Supply (mkExpr "22" :: Expr Int) $
  PMatch (mkPVar "z" :: Pat Int) $
  Return $ (mkEVar "+" :: Expr (Int → Int → Int))
    `Apply` (mkEVar "x" :: Expr Int)
    `Apply` (mkEVar "y" :: Expr Int)
testRenVM1 = renameVarM (mkVar' "x" :: Var Int) (mkVar' "a" :: Var Int) renVM1
testRenVM2 = renameVarM (mkVar' "y" :: Var Int) (mkVar' "a" :: Var Int) renVM1
testRenVM3 = renameVarM (mkVar' "x" :: Var Int) (mkVar' "a" :: Var Int) renVM2
testRenVM4 = renameVarM (mkVar' "y" :: Var Int) (mkVar' "a" :: Var Int) renVM2
testRenVM5 = renameVarM (mkVar' "z" :: Var Int) (mkVar' "a" :: Var Int) renVM2
```

```
*NormExample> renVM1
22 >> y => |+ x y|
*NormExample> renVM2
22 >> z => |+ x y|

*NormExample> testRenVM1
Just (22 >> y => |+ a y|)
*NormExample> testRenVM2
Just (22 >> y => |+ x y|)
*NormExample> testRenVM3
Just (22 >> z => |+ a y|)
```

```
*NormExample> testRenVM4
Just (22 >> z => |+ x a|)
*NormExample> testRenVM5
Just (22 >> z => |+ x y|)
```

D.2.1 Closure

```
test x y z = case (x, y) of
  (5, 42) → f z
  _ → error "should not happen"
  where f y = case y of a → x + a
pairOf2Int :: Expr (Int, Int)
pairOf2Int = cExpr2 (mkC2 "(,)") (mkExpr "5" :: Expr Int)
  (mkExpr "42" :: Expr Int)
pairxy :: Pat (Int, Int)
pairxy = cPat2 (mkC2 "(,)") (mkPVar "x" :: Pat Int) (mkPVar "y" :: Pat Int)
exprInt :: Expr Int
exprInt = mkExpr "22"
paty :: Pat Int
paty = mkPVar "y"
scopeM :: Match Int
scopeM = Return $ (mkEVar "+"
  `Apply` (mkEVar "x" :: Expr Int)
  `Apply` (mkEVar "y" :: Expr Int)
  )
scopeTest :: Match Int
scopeTest = Supply pairOf2Int $ PMatch pairxy $
  Supply exprInt $ PMatch paty scopeM
scopeTest2 :: Match Int
scopeTest2 = Supply exprInt $ PMatch paty scopeM
scopeM2 :: Match Int
scopeM2 = Return $ (mkEVar "+"
  `Apply` (mkEVar "z" :: Expr Int)
  `Apply` (mkEVar "z2" :: Expr Int)
  )
scopeTest4 :: Match Int
scopeTest4 = Supply exprInt $ PMatch paty scopeM2
```

When we use normalization without α-conversion, we get the following *wrong* results.

```
*Eval> test 5 42 22
```

```
27

*Eval> scopeTest
(5,42) >> (x,y) => 22 >> y => |+ x y|

*Norm> normM scopeTest
Just |+ 5 42|
```

The examples show that our operatinal semantics have to deal with variable scoping by using such mechanisms as renaming. When we use normalization with α-conversion, we get the following *correct* results.

```
*Norm> normM scopeTest4
Just |+ z z2|

*Norm> scopeTest4
22 >> y => |+ z z2|

*Norm> scopeTest2
22 >> y => |+ x y|

*Norm> normM scopeTest2
Just |+ x 22|

*Norm> scopeTest
(5,42) >> (x,y) => 22 >> y => |+ x y|

*Norm> normM scopeTest
Just |+ 5 22|
```

Bibliography

[1] http://www.nationmaster.com/encyclopedia/.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] Michael Barr and Charles Wells. *Category Theory for Computing Science.* 1999. 3rd edition.

[4] Bruno C. d. S. Oliveira and Jeremy Gibbons. Typecase: A design pattern for type-indexed functions. In *Haskell Workshop 2005*, September 2005.

[5] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop 2000*, volume 41 of *ENTCS*, pages 12.1–12.27, 2001.

[6] Glasgow Haskell Compiler home page. http://www.haskell.org.

[7] William L. Harrison and Richard B. Kieburtz. Pattern-driven reduction in Haskell. In *2nd International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, 2002.

[8] William L. Harrison, Tim Sheard, and James Hook. Fine control of demand in Haskell. In *Sixthe International Conference on the Mathematics of Program Construction*, July 2002.

[9] The Haskell 98 language report, December 2002. available from http://www.haskell.org/onlinereport/.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, May 2004.

[11] Wolfram Kahl. Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation. SQRL Report 16, Software Quality Research Laboratory, McMaster Univ., October 2003. available from http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[12] Wolfram Kahl. Confluence proof of the pattern matching calculi in Isabelle, October 2003. available from
http://www.cas.mcmaster.ca/ kahl/PMC/Confluence/PMC/.

[13] Wolfram Kahl. Basic pattern matching calculi: A fresh view on matching failure. In *Proceedings of The Seventh International Symposium on Functional and Logic Programming (FLOPS 2004)*, volume 2998 of *LNCS*, pages 276–290. Springer-Verlag GmbH, 2004.

[14] Wolfram Kahl. Type-indexed expressions in Haskell. Manuscript, April 2005.

[15] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058 of *LNCS*, pages 219–234. Springer-Verlag, 1996.

[16] Engenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[17] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.

[18] Peter D. Mosses. Theory and practice of action semantics. In *21st Int. Symp. on Mathematical Foundations of Computer Science*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, September 1996.

[19] Simon Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, 1987. available from
http://research.microsoft.com/Users/simonpj/papers/slpj-book-1987/.

[20] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. available from
http://research.microsoft.com/Users/simonpj/papers/gadt/, July 2004.

[21] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[22] David A. Watt. Why don't programming language designers use formal methods? In *Seminario Integrado de Software e Hardware - SEMISH'96*, pages 1–6, 1996.