関数型言語Haskellでオブジェクト指向プログラミングをサポートするモジュールの設計と実装

桂　直人[1,a)]　木山　真人[1,b)]　芦原　評[1,c)]

概要：関数型プログラミング言語は，簡潔なコーディングを可能にする，デバッグを容易にするなどの利点を持ち，近年注目されている．また，オブジェクト指向プログラミングは，プログラムの保守性・再利用性を高められる利点が知られている．本研究の目的は，両者の利点が両立した，関数型言語でのオブジェクト指向プログラミングの実現である．本研究では，関数型言語 Haskell でオブジェクト指向プログラミングをサポートするモジュール OOClass を設計し，実装した．OOClass は，オブジェクト指向におけるクラスを Haskell の型クラスで表現する．また，クラス間の継承を可能にし，Mixin とトレイトの機能を備える．

Design and Implementation of a Module Supporting Object-Oriented Programming in Functional Language Haskell

Abstract: Functional programming languages are widely noticed with its merits such as the possibility of concise coding and the easiness to debug in recent years. Object-oriented programming is known for its merits which can improve serviceability and reusability of programs. The purpose of our study is an introduction of object-oriented programming in the functional language to obtain the both merits. We designed and implemented OOClass: the module supporting object-oriented programming in the functional language Haskell. OOClass can represent classes in object-oriented programming as type classes in Haskell. OOClass have features for class inheritance, mixin and traits.

1. はじめに

関数型プログラミング言語は，副作用を持たない関数の集まりによってプログラミングを行う言語である．簡潔なコーディングを可能にする，デバッグを容易にするなどの利点から，近年注目を集めている．また，オブジェクト指向プログラミングは，データの集合と，そのデータに対する手続きの組み合わせを一つのまとまりとして管理するプログラミング手法である．オブジェクト指向プログラミングによってプログラムの保守性・再利用性を高められる．

本研究では，関数型プログラミング言語の一つである Haskell に，オブジェクト指向プログラミングをサポートする仕組みを追加する．具体的には，オブジェクト指向におけるクラスや，クラスを拡張する機能である Mixin とトレイトを実現させるモジュールを設計し実装する．本研究で設計・実装するモジュール (以下，OOClass) で目的とする機能は，以下の通りである．

- クラスの定義と，クラス間の継承
- Mixin の定義と，Mixin のクラスへの適用
- トレイトの定義と，トレイトのクラスへの適用

本論文の構成は次の通りである．第 2 章では，予備知識について述べる．第 3 章では，OOClass の設計について述べる．第 4 章では，OOClass の実装について述べる．第 5 章では，OOClass に関する議論を行う．第 6 章でまとめを行う．

2. 予備知識

本章では，事前に必要な予備知識について述べる．

2.1 Template Haskell

Template Haskell は，関数型プログラミング言語 Haskell の拡張であり，プログラムの記述を自動化する．通常，

[1] 熊本大学工学部情報電気電子工学科
Department of Computer Science and Electrical Engineering, Faculty of Engineering, Kumamoto University
[a)] c9733@st.cs.kumamoto-u.ac.jp
[b)] masato@cs.kumamoto-u.ac.jp
[c)] ashihara@cs.kumamoto-u.ac.jp

Haskell はコンパイル時にコードを構文解析し，構文木に変換する．ここで，Template Haskell を使用すると，通常の構文木の間に，プログラマが作成した構文木を接合できる．接合されるのは”$( )”で囲んだ内側の構文木である．接合する構文木は，Haskell のデータとして扱う．例として，式を表す構文木は Exp 型，宣言を表す構文木は Dec 型のデータとして扱う．本研究で主に注目する構文木の型は Dec 型である．

2.2 Mixin

Mixin は，クラスを拡張するために用いられる仕組みである．Mixin は，任意のクラスに対して，特定のメソッドの集合を追加する．クラス同士の継承と異なり，継承関係を考える必要なく，任意のクラスに対して任意の Mixin を自由に適用できる．

2.3 トレイト

トレイトは，メソッドの集合をクラスに追加するという点で Mixin と同じである．トレイトと Mixin の違いは，個々のメソッドを sum，exclude，alias のようなトレイト演算子で操作できる点である．sum は二つのトレイトのメソッドを合成し，exclude は特定のメソッドを取り除き，alias は特定のメソッドを新たな名前でコピーする．この仕組みを利用してメソッドの衝突を明示的に回避し，共通のメソッドを持つ二つのトレイトを結合できる．[1]

3. 設計

本章では，OOClass におけるクラス，Mixin，トレイトについて述べる．3.1 節でクラスについて，3.2 節で Mixin について，3.3 節でトレイトについて，3.4 節で OOClass の使用方法について述べる．

3.1 クラス

本節では，OOClass におけるクラスについて述べる．クラスは，以下の仕組みを持つ仕様である．

- メソッド
  クラスは，通常のオブジェクト指向言語と同様に，任意の関数をメソッドとして持つ．ただし，メソッド名は，スーパークラスのメソッドを継承する場合と Mixin・トレイトのメソッドを適用する場合を除き，他のクラス・Mixin・トレイトのメソッド名と重複できない．
- メンバ変数
  クラスは，0 個以上のメンバ変数を持つ．それぞれのメンバ変数が，String 型の名前と Member 型の値を持つ．メンバ変数の集合は [(String,Member)] 型である．Member 型は自作の型である．メンバ変数の値を表現するために使用する．データコンストラクタとして Int，String，NotFound を持つ．データコンストラクタ Int は Int 型の引数 1 つを持ち，Int 型のメンバ変数の値を意味する．データコンストラクタ String は String 型の引数 1 つを持ち，String 型のメンバ変数の値を意味する．データコンストラクタ NotFound は引数を持たず，メンバ変数が存在しない場合を意味する．
- クラスの継承
  クラスは，他のクラス 1 つをスーパークラスに取り，継承を行える．継承した場合，スーパークラスの持つメンバ変数を得る．また，メソッドのオーバーライドができる．ただし，メソッドの型は変更できない．
- Mixin の適用
  クラスは，0 個以上の Mixin を適用できる．動作が定義されていないメソッドを含む Mixin を適用し，そのメソッドをオーバーライドによって再定義しない場合，エラーメッセージを表示する．
- トレイトの適用
  クラスは，0 個以上のトレイトを適用できる．Mixin の適用と同様に，動作が定義されていないメソッドを含むトレイトを適用し，そのメソッドをオーバーライドによって再定義しない場合，エラーメッセージを表示する．適用するトレイトは Trait 型のデータとして指定する．

本研究では，オブジェクト指向プログラミング言語におけるクラスを，Haskell の仕組みの 1 つである型クラスを用いて表現する．

クラスを表現するために型クラスを宣言する．そして，クラス階層を表現するために，全ての型クラスのスーパークラスとして，型クラス Object を宣言する．

型クラス Object の型クラスメソッドとして objField と member を定義する．objField はオブジェクト指向におけるインスタンスを表す変数を引数に取り，そのインスタンスが持つメンバ変数を返す関数である．member はオブジェクト指向におけるインスタンスを表す変数と任意の文字列を引数に取り，その文字列と変数名が一致するメンバ変数を 1 つ探し，そのメンバ変数の値を返す関数である．条件に合うメンバ変数が存在しない場合，Member 型の値 NotFound を返す．

1 つのクラス定義ごとに，以下を 1 つずつ宣言する．ここで，定義するクラス名を”X”とする．

- 型クラス X
  型クラス X は，型クラス Object のサブクラスである．クラス X が持つメソッドのうち，継承や Mixin・トレイトの適用によって引き継いだメソッドでないメソッドは，この型クラス X の型クラスメソッドとして表現する．
- 型 XInstance
  定義するクラスのインスタンスを意味する型である．この型は，型クラス X，型クラス Object，およびクラ

ス X の全ての祖先クラスに対応する型クラスのインスタンスである．また，Mixin を適用する場合は，適用する Mixin とその祖先 Mixin に対応する型クラスのインスタンスになる．トレイトを適用する場合は，適用するトレイトメソッドに対応する型クラスのインスタンスになる．

- [(String,Member)] 型の関数 fieldXInstance
  デフォルトのメンバ変数の集合を表す．
- XInstance 型の関数 makeXInstance
  fieldXInstance をメンバ変数として持つ，クラス X のインスタンスを表す．クラス X のインスタンスを作成する場合で，メンバ変数に変更を加えない場合に使用する．

3.2 Mixin

本節では，OOClass における Mixin について述べる．Mixin は，以下の仕組みを持つ仕様である．

- メソッド
  Mixin は，クラスと同様に，任意の関数をメソッドとして持つ．ただし，メソッド名は，スーパー Mixin のメソッドを継承する場合を除き，他のクラス・Mixin・トレイトのメソッド名と重複できない．他のクラス・Mixin・トレイトのメソッド名と重複できない．
- Mixin の継承
  Mixin は，他の Mixin1 つをスーパー Mixin に取り，継承を行える．また，メソッドのオーバーライドができる．ただし，メソッドの型は変更できない．

本研究では，オブジェクト指向プログラミング言語における Mixin を，型クラスを用いて表現する．1 つの Mixin 定義ごとに，定義する Mixin 名と同じ名前の型クラスを宣言する．Mixin が持つメソッドのうち，継承によって引き継いだメソッドでないメソッドは，この型クラスの型クラスメソッドとして表現する．

3.3 トレイト

本節では，OOClass におけるトレイトについて述べる．トレイトは，以下の仕組みを持つ仕様である．

- メソッド
  トレイトは，任意の関数をメソッドとして持つ．ただし，メソッド名は，スーパートレイトのメソッドを継承する場合を除き，他のクラス・Mixin・トレイトのメソッド名と重複できない．
- トレイトの継承
  トレイトは, 0 個以上のトレイトをスーパートレイトとして継承できる．継承するトレイトは Trait 型のデータとして指定する．

クラスがトレイトを適用する場合，およびトレイトがトレイトを継承する場合には，トレイトを表す型として Trait 型を使用する．

Trait 型は，データコンストラクタとして Trait，TExclude，TAlias を持つ．以下に，それぞれのデータコンストラクタについて述べる．

- データコンストラクタ Trait
  データコンストラクタ Trait は，String 型の引数 1 つを持ち，定義したままの内容のトレイトを表す．引数はトレイトの名前である．
- データコンストラクタ TExclude
  データコンストラクタ TExclude は，特定のメソッドを取り除くデータコンストラクタである．Trait 型の第 1 引数と String 型の第 2 引数を持ち，元となるトレイトからメソッド 1 つを取り除いたトレイトを表す．第 1 引数は元となるトレイトであり，第 2 引数は取り除くトレイトの名前である．
- データコンストラクタ TAlias
  データコンストラクタ TAlias は，特定のメソッドを新たな名前でコピーするデータコンストラクタである．Trait 型の第 1 引数と String 型の第 2 引数と String 型の第 3 引数を持つ．元となるトレイトに，そのトレイトが持つメソッドと同じ内容の，別の名前のメソッドを新たに追加したトレイトを表す．第 1 引数は元となるトレイトであり，第 2 引数は元となるトレイトが持つメソッドの名前であり，第 3 引数は新たに追加するメソッドの名前である．

本研究では，オブジェクト指向プログラミング言語におけるトレイトを，0 個以上の型クラスを用いて表現する．定義したトレイトが持つメソッドごとに，そのメソッドに応じた型クラスを宣言する．その型クラスは，メソッドの名前を”x”とすると”TraitMethod_x”であり，メソッド x を型クラスメソッドとして持つ．

3.4 使用方法

本節では，OOClass の使用方法について述べる．

OOClass では，クラス，Mixin もしくはトレイトの定義を行う場合，関数 newClass，newMixin もしくは newTrait から返された構文木を接合する．ここで，接合は上から順番に処理されるため，自分自身より下のコードで定義されるクラス，Mixin，トレイトは扱えない．

以下で，関数 newClass，newMixin，newTrait の各々の用途と引数について述べる．

- 関数 newClass
  新たなクラスを定義する場合は，関数 newClass を使用する．関数 newClass が取る引数を**表 1** に示す．TheClass のインスタンス ins を宣言している．
- 関数 newMixin
  新たな Mixin を定義する場合は，関数 newMixin を使用する．関数 newMixin が取る引数を**表 2** に示す．

表 1 関数 newClass が取る引数

順番	型	内容	備考
1	String	定義するクラスの名前	型クラス名に使用可能な文字列に限る
2	Maybe String	継承するクラスの名前	継承する場合は Just (クラス名) / 継承しない場合は Nothing
3	[String]	適用する Mixin の名前のリスト	-
4	[Trait]	適用するトレイトのリスト	-
5	[(String,Member)]	定義するクラスのデフォルトの メンバ変数のリスト	-
6	Q [Dec]	メソッドの宣言構文木	-

表 2 関数 newMixin が取る引数

順番	型	内容	備考
1	String	定義する Mixin の名前	型クラス名に使用可能な文字列に限る
2	Maybe String	継承する Mixin の名前	継承する場合は Just (クラス名) / 継承しない場合は Nothing
3	Q [Dec]	メソッドの宣言構文木	関数の内容の定義を保留する場合は (関数名) = abstractMethod と宣言する

- 関数 newTrait
  新たなトレイトを定義する場合は，関数 newTrait を使用する．関数 newTrait が取る引数を**表 3** に示す．

4. 定義処理の実装

本章では，OOClass における，クラス，Mixin，トレイトを定義する処理の実装について述べる．

本章で用いる Field 型は，[(String,Member)] 型の別名である．また，Category 型は，自作の型であり，ClassC，MixinC，TraitC のいずれかの値を持つ．

実際にクラス・Mixin・トレイトの定義処理を行う関数は，関数 newAnything である．関数 newAnything は，クラス・Mixin・トレイトを定義する場合に使用する関数 newClass，newMixin，newTrait から呼び出される．呼び出しの際，Category 型の値が識別子として渡される．関数 newAnything は 7 つの引数を取る．引数の型と内容を**表 4** に示す．引数 category の値が ClassC であればクラスを，MixinC であれば Mixin を，TraitC であればトレイトを定義する処理を行う．関数 newAnything の返り値は Q [Dec] 型の構文木である．

クラス・Mixin・トレイトの定義処理において，グローバル変数を使用する．グローバル変数の実装は，外部モジュール Data.IORef の各種関数と外部モジュール System.IO.Unsafe の関数 unsafePerformIO を使用して行う．グローバル変数に記録する情報は，定義したクラス・Mixin・トレイトの継承元，定義したクラスが適用した Mixin とトレイト，各クラス・Mixin・トレイトが持つメソッドの実装などである．これらの情報は，継承や Mixin・トレイトの適用の処理に必要である．定義するグローバル変数の，名前と記録する内容を**表 5** に示す．

定義処理の流れを**図 1** に示す．以下の節では，各処理の詳細を述べる．

表 5 グローバル変数

名前	記録する内容
inheritRec	クラス・Mixin の継承元，適用元
allMethodsRec	クラス・Mixin のメソッドの情報
traitMethodsRec	トレイトメソッドの情報

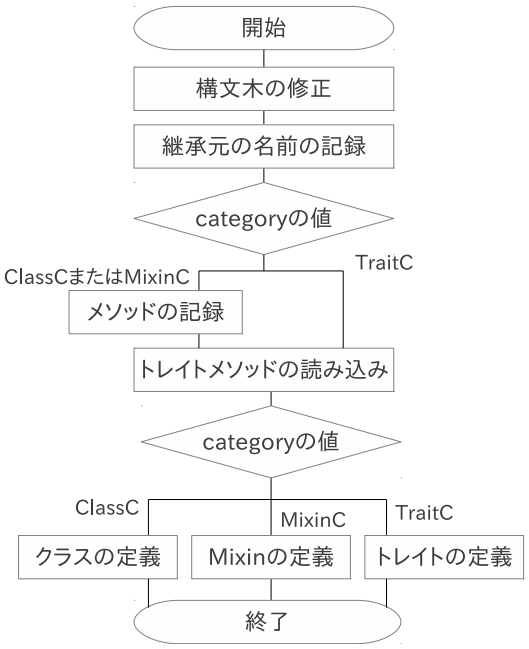

図 1 関数 newAnything の処理の流れ

4.1 構文木の修正

まず，引数 qMethods の宣言構文木から，不必要な宣言を取り除く処理を行う．型宣言，関数宣言，変数宣言のいずれでもない宣言を構文木から取り除く．

次に，構文木中の名前の修正を行う．引数 qMethods の宣言構文木中に含まれる関数の名前は，コンパイラによっ

表 3 関数 newTrait が取る引数

順番	型	内容	備考
1	String	定義するトレイトの名前	型クラス名に使用可能な文字列に限る
2	[Trait]	継承するトレイトのリスト	-
3	Q [Dec]	メソッドの宣言構文木	関数の内容の定義を保留する場合は (関数名) = abstractMethod と宣言する

表 4 関数 newAnything が取る引数

順番	型	名前	内容
1	Category	category	クラス・Mixin・トレイトのどれを定義するか
2	String	nameStr	定義するクラス・Mixin・トレイトの名前
3	Maybe String	superNameStr	クラスが継承するクラス Mixin が継承する Mixin
4	[String]	mixinStrs	クラスが継承する Mixin
5	[Trait]	useTraits	クラス・トレイトが継承するトレイト
6	Field	field	デフォルトのメンバ変数
7	Q [Dec]	qMethods	メソッドの宣言構文木

て自動的に名前を変更される．このため，異なるクラス宣言で同じ関数の名前を使用しても，コンパイラはそれぞれ違う名前の関数として認識する．このままではオーバーライドの処理を行うのに不都合であるため，引数 qMethods の宣言文中で使用された通りの名前に変換する．

これらの修正を行った宣言構文木を，変数 methods に束縛する．

4.2 継承元の名前の記録

定義するクラス・Mixin が継承する，継承元のクラス・Mixin の名前のリストを，グローバル変数 inheritRec に記録する．また，クラスが適用する Mixin の名前も同様に記録する．トレイトを定義する場合や，何も継承・適用しないクラス・Mixin を定義する場合，空リストを記録する．

4.3 メソッドの記録

継承したメソッドを含めた，クラス・Mixin が持つ全てのメソッドの情報を，グローバル変数 allMethodsRec に記録する．

クラス・Mixin が持つ各メソッドは，メソッドが初めて定義されたクラス・Mixin に対応する型クラスの型クラスメソッドである．型クラスメソッドの実装を宣言するためには，メソッドの実装の情報に加え，メソッドがどの型クラスの型クラスメソッドであるのかという情報も記録する必要がある．したがって，グローバル変数 allMethodsRec は，各クラス・Mixin について，自分自身，自分の祖先，自分が適用する Mixin それぞれに対応する型クラスの型クラスメソッドの実装を記録できる形とする．

変数 methods の宣言構文木のうち，オーバーライド以外の関数宣言を抜き出し，定義するクラス・Mixin 自身に対応する型クラスの型クラスメソッドとして，グローバル変数 allMethodsRec に記録する．続いて，グローバル変数 inheritRec と allMethodsRec の情報を参照し，継承する関数の情報をグローバル変数 allMethodsRec に記録する．オーバーライドせずに継承する関数は，allMethodsRec の内容を記録し，オーバーライドする関数は，変数 methods の内容を記録する．

4.4 トレイトメソッドの読み込み

定義するクラスまたはトレイトがトレイトを継承する場合，引数 useTraits とグローバル変数 traitMethodsRec の値から，実装するトレイトメソッドの情報を得る．実装するトレイトメソッドすべての，対応する型クラスの名前，型宣言の構文木，実装の宣言の構文木を記録する．また，メソッドの別名を作成する場合，新たなトレイトメソッドに対応した型クラスの宣言が必要となるので，型クラスの宣言に必要な構文木を作成し，保持する．

トレイトメソッドの情報と別名宣言のための構文木を求める処理は，引数 useTraits の要素を 1 つずつ，関数 makeTraits” に渡して行う．関数 makeTraits” で行う処理の詳細を以下に述べる．

- データコンストラクタ Trait が使用された場合
  グローバル変数 traitMethodsRec の値から，データコンストラクタ Trait の引数と同じ名前のトレイトの情報を探し，そのトレイトが持つ各トレイトメソッドの情報を記録する．トレイトメソッドの情報とは，対応する型クラスの名前，型宣言の構文木，実装の宣言の構文木である．
- データコンストラクタ TExclude が使用された場合
  データコンストラクタ TExclude の第 1 引数のトレイトの情報から，第 2 引数で指定したメソッドに関する情報を取り除いた結果を取得する．
- データコンストラクタ TAlias が使用された場合
  データコンストラクタ TAlias の第 1 引数のトレイト

の情報から，トレイトメソッド 1 つの情報を異なる名前で複製し追加した結果を取得する．複製するメソッドはデータコンストラクタ TAlias の第 2 引数であり，複製後の名前はデータコンストラクタ TAlias の第 3 引数である．
また, 新たなトレイトメソッドに対応する型クラスが必要になるため, 型クラスの宣言構文木を保持する．データコンストラクタ TAlias の第 3 引数の文字列を”a”とすると，宣言する型クラスの名前は TraitMethod_a であり，名前を a に変更し複製した後の型宣言を型クラスメソッドに持つ．

4.5 クラスの定義

クラスを定義する場合，以下の構文木を返す．ここで，定義するクラスの名前を”X”とする．

- 型クラス宣言
  型クラス X を宣言する．型クラス X は型クラス Object のサブクラスである．変数 methods 中の型宣言を，型クラスメソッドの型宣言として持つ．ここで，型クラス X のインスタンスである型を表す型変数として”myInstance”を使用する．したがって，型宣言中の”myInstance”型変数は，クラス X のインスタンスを意味する．
- 型宣言
  型 XInstance を宣言する．型 XInstance は Field 型の値 1 つを引数に取る．
- 変数宣言
  変数 fieldXInstance を宣言する．クラス X がスーパークラスを持たない場合，引数 field に束縛される．クラス X がスーパークラス Y を持つ場合，引数 field と関数 fieldYInstance の返り値を連結したリストに束縛される．
- 変数宣言
  変数 makeXInstance を宣言する．型 XInstance を fieldXInstance に適用した値に束縛される．
- インスタンス宣言
  型 XInstance を型クラス Object のインスタンスにする．型クラス Object の型クラスメソッドである関数 objField の実装を宣言する．
- インスタンス宣言
  型 XInstance を型クラス X のインスタンスにする．変数 methods を参照し，型クラス X の型クラスメソッドの実装を宣言する．
- 0 個以上のインスタンス宣言
  クラス X の祖先クラス，クラス X が適用する Mixin，およびクラス X が適用する Mixin の祖先 Mixin に対応する型クラスすべてに対して，型 XInstance をインスタンスにするためのインスタンス宣言を行う．各型クラスの型クラスメソッドの実装の宣言も行う．変数 methods にオーバーライドの記述がある場合はその宣言構文木を使用し，それ以外の場合はグローバル変数 allMethodsRec を参照し，親クラスや継承した Mixin の持つメソッドの構文木を使用する．
- 0 個以上のインスタンス宣言
  型 XInstance を，トレイトメソッドに対応する型クラスのインスタンスにするインスタンス宣言を行う．各型クラスの型クラスメソッドの実装の宣言も行う．変数 methods にオーバーライドの記述がある場合は，その宣言構文木を使用する．それ以外の場合は，トレイトメソッド読み込みの際に記録した実装の宣言の構文木を使用する．
- 0 個以上の型クラス宣言
  継承するトレイトメソッドに別名が新たに使われる場合，新たなトレイトメソッドに対応する型クラスを追加するための型クラス宣言を行う．

クラスを定義する際，型クラスメソッドの実装を宣言する構文木中の名前”NewMyInstance”を”XInstance”に変更する．ここで”X”は定義するクラスの名前である．データコンストラクタ NewMyInstance は，Field 型の値を 1 つ引数に取る．クラスのメソッドを定義する際，メソッドを呼び出したインスタンスと同じクラスのインスタンスを作成する関数として使用できる．

4.6 Mixin の定義

Mixin を定義する場合，以下の構文木を返す．

- 型クラス宣言
  型クラス X を宣言する．変数 methods 中の型宣言を，型クラスメソッドの型宣言として持つ．ここで，型クラス X のインスタンスである型を表す型変数として”myInstance”を使用する．したがって，型宣言中の”myInstance”型変数は，クラス X のインスタンスを意味する．

4.7 トレイトの定義

トレイトを定義する場合，以下の構文木を返す．

- 0 個以上の型クラス宣言
  トレイトメソッドの型宣言の数だけ，型クラス宣言を行う．型宣言されるメソッドの名前を”a”とすると，TraitMethod_a という名前の型クラスの宣言を行う．そして，型クラス TraitMethod_a は，メソッド a の型宣言を，型クラスメソッドの型宣言として持つ．
- 0 個以上の型クラス宣言
  継承するトレイトメソッドに別名が新たに使われる場合，新たなトレイトメソッドに対応する型クラスを追加するための型クラス宣言を行う．

トレイトを定義する際, 宣言した全てのトレイトメソッド

a.hs
```
import OOClass
import Language.Haskell.TH
$(newClass "ClosedCounter"
  Nothing [] [("val",Int 0)]
  [d|
   add :: myInstance -> myInstance
   add this = MyNewInstance [("val", Int $
    (memberToInt $ member this "val") + 1)]
  |])
counter = makeClosedCounterInstance
```

図 2 オブジェクトのメンバ変数の外部参照

について，対応する型クラスの名前，型宣言の構文木，実装の宣言の構文木の情報を，グローバル変数 traitMethodsRec に記録する．

5. 議論

本章では，OOClass に関する議論を行う．5.1 節では OOClass を使用する場合と使用しない場合のコード記述の比較について，5.2 節では実行速度の低下について，5.3 節ではデータの隠蔽について，5.4 節ではエラーメッセージについて述べる．

5.1 記述の比較

OOClass を使用してオブジェクト指向モデルを記述すると，Mixin やトレイトの実装を一ヶ所に集められる．一方，OOClass を使用せずに同様のオブジェクト指向モデルを記述する場合，Mixin やトレイトの実装を別々の場所に繰り返し書かなければならず，また，インスタンス宣言を多く記述する必要があるので，コード量が大きくなる．

5.2 実行速度

OOClass の処理による，プログラムの実行速度の低下は考えられない．2.1 節で述べている通り，Template Haskell での構文木の作成と接合処理はコンパイル時に行われるためである．

5.3 データ隠蔽

OOClass では，関数 member によって外部からもオブジェクトのメンバ変数を参照できる．外部から参照される例を図 2 に示す．図のコードでは，クラス ClosedCounter を定義している．クラス ClosedCounter は，数値 1 つを保持し，数値を増加させる機能を持つカウンタを表す．ここで，関数 member を使用すれば，外部からカウンタの数値を参照できる．この仕様では，データが隠蔽されず，カプセル化できない．

ClosedCounter.hs
```
module ClosedCounter
(add,ClosedCounter,
 makeClosedCounterInstance,(#)) where
import OOClass
import Language.Haskell.TH
$(newClass "ClosedCounter"
  Nothing [] [("val",Int 0)]
  [d|
   add :: myInstance -> myInstance
   add this = MyNewInstance [("val", Int $
    (memberToInt $ member this "val") + 1)]
  |])
```

outside.hs
```
import ClosedCounter
import Language.Haskell.TH

counter = makeClosedCounterInstance
```

図 3 モジュール化による隠蔽

```
$(newClass "TheClass"
  Nothing [] [] [] [d| |])
$(newClass "ExClass"
  (Just "ThheClass") [] [] [] [d|  |])
```

図 4 誤ったコード

カウンタの数値を隠蔽する場合，クラス ClosedCounter の宣言部分をモジュール化すればよい．その例を図 3 に示す．この例では，モジュール ClosedCounter に OOClass をインポートする．モジュール ClosedCounter からは各メソッドと，インスタンスを表すデータ型，インスタンスを作成するための関数をエクスポートし，関数 member はエクスポートしない．外部では，直接 OOClass をインポートせず，モジュール ClosedCounter をインポートする．この場合，モジュールの外部では関数 member を使用できないので，カウンタの数値は隠蔽される．

5.4 エラーメッセージ

OOClass を使用したプログラム中に誤りがあり，それによるエラーメッセージが表示される場合，現状の OOClass では，表示されるメッセージが内部処理に関する理解しづらい内容である場合がある．例として，図 4 に誤ったコードを示す．このコードには，クラス ExClass の継承元の記述に誤りがある．クラス名”TheClass”を記述すべき箇所に，誤ったクラス名”ThheClass”を記述している．この

```
ex.hs:1:1:
    Exception when trying to run compile-time
 code:
      Maybe.fromJust: Nothing
      Code: newClass
              "ExClass"
              (Data.Maybe.Just "ThheClass")
              (GHC.Types.[])
              (GHC.Types.[])
              (GHC.Types.[])
              [d| |]
Failed, modules loaded: OOClass.
```

図 5 エラーメッセージ

コードをコンパイルした場合に表示されるエラーメッセージを図 5 に示す．このメッセージは，クラス名に誤りがあるという情報を含んでいないため，OOClass のユーザにとって適切ではない．

上述の問題は，将来的には OOClass のバージョンアップによって回避できる．入力されたクラス名が存在しなかった場合，その旨を知らせるエラーメッセージを表示する機能を追加すればよい．

6. 関連研究

本研究と関連する研究を以下に挙げる．

- Scheme with Classes, Mixins, and Traits[1]
  オブジェクト指向におけるクラス，Mixin，トレイトの概念を，関数型言語 Scheme に導入する研究である．Scheme は関数型言語として純粋ではない点，型付けが動的である点などで Haskell と異なる．したがって，OOClass とは仕組みが異なる．
- Binding Haskell to Object-Oriented Component Systems via Reflection[2]
  Haskell のシステムを，外部のオブジェクト指向システム，もしくはコンポーネントシステムと通信させる研究である．Haskell 上でオブジェクト指向クラス階層のモデルを表現させるために，Haskell における型クラスを利用する．Haskell 上でのオブジェクト指向プログラミングの実現が目的ではない点で，本研究とは異なる．
- OOHaskell[3]
  Haskell にオブジェクト指向を導入する研究である．クラスの定義と継承，Mixin の適用が可能だが，トレイトの機能は無い．OOClass ではトレイトの機能を実現している．

7. まとめ

本論文では，関数型言語 Haskell でオブジェクト指向プログラミングをサポートするモジュール OOClass の設計と実装を行った．OOClass は，オブジェクト指向におけるクラスと，クラスをサポートする Mixin とトレイトの機能を備える．

OOClass では，オブジェクト指向におけるクラスを Haskell における型クラスとして表現する．クラス・Mixin・トレイトを定義するには，関数 newClass,newMixin,newTrait を用いる．関数に渡す引数として，クラス名等の情報を記述する．関数からは，定義するクラス・Mixin・トレイトのモデルとなる型クラスを宣言するための構文木の情報が返される．この構文木を，Haskell の拡張である Template Haskell を用いてコンパイル時に接合する．

クラス・Mixin・トレイトの定義処理は，OOClass の関数 newAnything で行われる．既に定義したクラス・Mixin・トレイトの情報を扱うために，グローバル変数を使用する．この処理によって，オブジェクト指向のコードを記述する際に，同一のメソッドの実装を繰り返し記述する必要や，多くのインスタンス宣言を記述する必要がなくなる．

OOClass では，コンパイル時に構文木の作成が全て行われるため，実行速度の低下の問題はない．また，Haskell でサポートされているモジュール化を行えば，作成したクラスのカプセル化も可能である．

参考文献

[1] Matthew Flatt, Robert Bruce Findler, Matthias Felleisen (2006)「Scheme with Classes, Mixins, and Traits」『APLAS'06 Proceedings of the 4th Asian conference on Programming Languages and Systems』pp.270-289
[2] Springer-Verlag Berlin, Heidelberg André T.H.Pang (2003)「Binding Haskell to Object-Oriented Component Systems via Reflection」The University of New South Wales Schools of Computer Schience and Engineering B. Sc(Computer Science & Psychology) Honours Thesis
[3] Oleg Kiselyov, Rälf Lammel (2005)「Haskell's overlooked object system」『Computing Research Repository』