Люка Карделли, Мартин Абади: Классы и типы в языках, основанных на классах

Перевод: Пискунов А.Г.

1 ноября 2010 г.

АННОТАЦИЯ

Документ содержит некоторые выдержки из монографии 'Теория Объектов' Карделли, Абади, посвященные обсуждению понятий тип, класс, подкласс, выделение типа (subtyping), наследование (inheritance) и их отличий. Может быть использован как введение в объектно - ориентированное программирование.

Содержание

1 ВВЕДЕНИЕ 3

2 ЯЗЫКИ, ОСНОВАННЫЕ НА КЛАССАХ 3

2.1 Классы и объекты 3
2.2 Выбор метода 5
2.3 Подклассы и наследование 7
2.4 Категоризация и динамическая диспетчеризация 9
2.5 Пропущенная и восстановленная информанция о типе . . . 12
2.6 Ковариантность, контравариантность и инвариантность . 13
2.7 Конкретизация метода 15
2.8 Конкретизация типа у слова $self$ 17

3 БОЛЕЕ ТОНКИЕ СВОЙСТВА, ОСНОВАННЫЕ НА КЛАССАХ 19

3.1 Типы объектов 19
3.2 Различия образования подклассов от выделения подтипов 21
3.3 Параметры типа 23
3.4 Образование классов без выделения типа 25
3.5 Протоколы обьекта 27

4 ЯЗЫКИ, ОСНОВАННЫЕ НА ОБЪЕКТАХ 29

1 ВВЕДЕНИЕ

Текст документа является не совсем полным переводом четвертой и пятой глав монографии Martin Abadi и Luca Cardelli 'A theory of objects' [2]. В основном, пропущены общие рассуждения во введениях к разделам и исторические ссылки.

2 ЯЗЫКИ, ОСНОВАННЫЕ НА КЛАССАХ

К языкам, основанным на классах, относятся такие языки как *Simula*, *Smolltalk* и $C++$. К языкам, основанным на объектах, относятся такие языки как *Cecil*, *Omega* и *Emerald*. В этой главе обсуждаются основные свойства языков, основанных на классах, которые, несмотря на бесчисленные особенности, с минимальными возражениями могут трактоваться как характеризующие классические объектно - ориентированные языки.

2.1 Классы и объекты

Языки, основанные на классах, объединяются понятием класса, как средством описания объектов. Начнем с примера объявления класса:

```
class cell is
    var contents: Integer := 0;
    method get(): Integer is
        return self.contents;
    end;
    method set(n: Integer) is
        self.contents := n;
    end;
end;
```

Класс предназначен для описания структуры всех объектов, генерируемых классом. Этот класс *cell* описывает объекты, имеющие целое поле, называемое *contents*, инициализируемое нулем и двумя методами: *get* и *set*, которые выполняют действия над полем *contents*. Метод *get* не имеет параметров, будучи выполненным, он возвращает содержимое поля *contents*. Метод *set* имеет целочисленный формальный параметр. Он запоминает полученный фактический параметр в поле *contents*. Внутри этих методов используется специальный идентификатор *self*, который обозначает текущий объект. Поля и методы объекта называются его атрибутами.

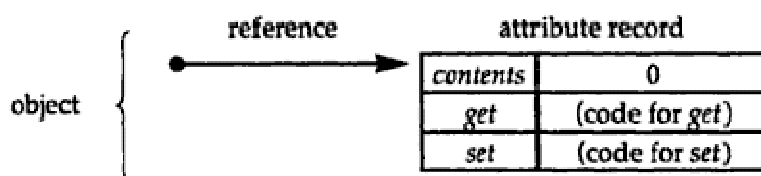

Рис. 1: Наивная модель памяти

Планируемое поведение объектов может быть понято в терминах наивной модели памяти на рис. 1 . В этой модели объекты представляются как ссылки на записи атрибутов. Любое обращение к атрибуту объекта неявно получает такую ссылку на запись, чтобы иметь доступ к атрибуту. Переменные программы содержат ссылки, но не записи атрибутов (далее просто запись). Передача параметров и присвоение копирует ссылки, но не ассоциированные записи. Таким образом передача фактических параметров и присвоение предоставляет доступ к записям атрибутов.

Объект может быть создан из класса c при помощи конструктора *new c*. Более точно, *new c* отводит память под запись и возвращает ссылку на нее. Запись содержит начальные значения и методы, указанные классом c. Различное выполнение конструктора *new c* создает различные объекты. То есть, конструктор создает ссылки на различные записи. А объект, который генерируется по описанию класса c при помощи конструктора *new*, для упрощения называется 'объектом класса c' или 'экземпляром класса c'.

Символом $InstanceTypeOf(c)$ будем обозначать тип объектов класса c. В следующем примере, *new cell* привязывается к переменной $myCell$ типом $InstanceTypeOf(cell)$:

```
var myCell: InstanceTypeOf(cell) := new cell;
```

Введнеие символа типа $InstanceTypeOf(cell)$ означает наличие важного различия между классами и типами. Можно рассматривать $cell$, вместо $InstanceTypeOf(cell)$, как тип объектов сгенерированных классом $cell$. Тем не менее, такая идентификация класса как тип будет позднее сбивать с толку.

У переменной с названием $myCell$, можно извлекать значения поля *contents* обращением $myCell.contents$ и можно обновлять его при помощи $myCell.contents := n$. Поля *contents* у различных объектов класса $cell$ изменяются независимо. То есть, каждый объект класса $cell$ имеет свое, отдельное поле $contents$.

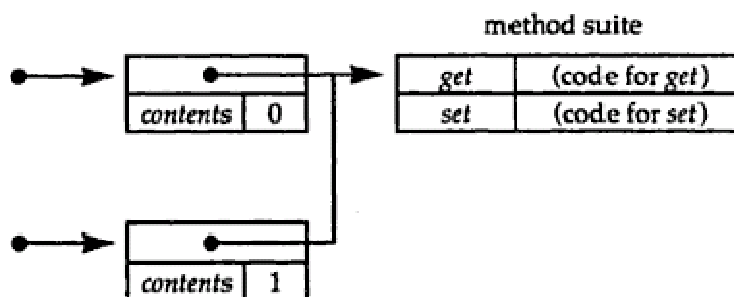

Рис. 2: Хранилище методов

У такой переменной можно вызывать её методы. Например, $myCell.set(3)$ это способ вызова кода для set с формальным параметром n равным числом 3 и с символом $self$, связанным с myCell. В качестве еще одного примера использования метода, определем процедуру $double$, которая удваивает значение, хранящаяся в формальном параметре $aCell$:

```
procedure double(aCell: InstanceTypeOf(cell)) is
    aCell.set(2 * aCell.get());
end;
```

2.2 Выбор метода

Получив запрос в форме $o.m(...)$, некоторый, независимый от языка процесс, называемый выбор метода (method lookup), должен подобрать подходящий метод m объекта o который будет выполниться. Выбор метода не слишком тривиален, так как стандартная модель памяти, используемая в языках, основанных на классах, оказывается несколько более сложная, чем наивная модель на рис. 1 . Мы коротко обсудим стандартную модель памяти, так как она часто представляется в качестве семантики языков, основанных на классах.

Хотя класс является средством для описания структуры объектов, сгенерированных с его помощью, методы не встраиваются напрямую в объекты, как это может показаться из синтаксиса классов и рис. 1 . Вместо этого, для экономии памяти, они (factored) складируются в специальные хранилища методов (method suites) и совместно используются различными объектами одного класса (см. рис. 2).

Выбор метода должен иметь доступ к этому хранилищу; будем говорить, что объекты делегируют вызовы метода хранилищам методов, ассоциированным с соответствующими классами.

При использовании в языке, коротко обсужденного наследования, хранилища методов могут быть организованны в виде дерева, и выбор метода может потребовать просмотра последовательностей таких хранилищ. С учетом множественного наследования, хранилища методов могут образовывать направленные графы (даже включающие циклы). Выбор метода должен иметь стратегию для просмотра таких графов. В зависимости от языка, выбор метода может выполняться на шаге компиляции, основываясь на информации о типах, или во время исполнения, основываясь на хранилищах методов, поддерживаемым для каждого объекта.

Хотя описание и реализация выбора метода может быть вполне сложной задачей, обычно производится впечатление, что методы встроены непосредственно в объекты, как показано в наивной модели памяти на рис. 1 . Иллюзия встроенности в объект и методов, и полей усиливается похожестью обычной записи $o.x$ для поля и $o.m(...)$ для вызываемых методов. Кроме того, во всех реализациях выбора метода, вхождение слова $self$ внутри метода ссылается на объект, с которым изначально запрашивался метод. Это слово $self$ постоянно ссылается на объект так, что производит впечатление содержания метода в объекте.

Интересно обратить внимание, что некоторые свойства, отличающие встраивание методов от делегирования оных, не проявляются в языках, основанных на классах. Для языков, основанных на классах, типичным является невозможность извлечения метода из объекта как функции, и, таким образом, невозможность обновления метода внутри объекта (в результате замены его другим методом). Как мы увидим позже, извлечение методов должно быть запрещено, чтобы сохранить возможность (удовлетворить необходимость) типизации. Обновление метода не всегда небезопасно (Method update is not necessarily unsound).

> Безопасность означает, что если программа правильно типизированна, она не приведет к ошибкам на шаге исполнения.

При реализации встроенных методов - обновление метода оказывает влияние на отдельный объект. При реализации делегированных методов - обновление метода оказывает влияние на все объекты класса.

Многие языки, основанные на классах, с равным успехом могут быть реализованы или на технике встраивания, или на технике делегирования. Для простоты, мы будем использовать модель встраивания в будущих объяснениях наследования, но иногда будем делать замечания по поводу

модели делегирования и хранилищ методов. Различия между встраиванием и делегированием появятся при обсуждении языков, основанных на объектах в главе 4 .

2.3 Подклассы и наследование

Хотя понятие класса является интересным само по себе, оно является промежуточным шагом на пути к понятиям подкласса и наследования. Подобно любому классу, подкласс (subclass) описывает структуру множества объектов. Но это происходит пошагово, при помощи описания дополнений и изменений его непосредственного надкласса (superclass). Поля надкласса по умолчанию добавляются в подкласс и к ним могут быть добавлены новые. Методи надкласса могут быть либо добавлены в подкласс по умолчанию, либо явно переопределены методами подкласса. Под отношением подкласса (subclass relation) будем понимать отношение частичного порядка, которое появляется в результате обьявлений различных классов.

Предыдущий пример расширяется при помощи объявления подкласса *reCell* (восстанавливаемая ячейка) класса *cell*. В объектах нового класса по сравнению с объектами надкласса появляется метод который восстанавливает поле *contents* в предыдущее состояние. Метод *set* переопределен так, что он сохраняет текущее значение поля *contents* перед его изменением; *super.set*(*n*) вызывает предыдущую версию метода *set* из класса *cell*.

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is
        self.contents := self.backup;
    end;
end;
```

Наследование означает, что аттрибуты класса принадлежат так же и подклассу. Эта совместная принадлежность включает инициализацию полей и код методов. Так, инициализация поля *contents* и код метода *get* наследуются классом *reCell* из класса *cell*. Метод *set* может быть наследован по умолчанию подобно методу *get*, но вместо этого он переопределяется в подклассе *reCell*.

Вообще говоря, подразумевается, что выражение c' наследует от c означает, что класс c' является подклассом c. Заметьте, однако, что подкласс класса без полей может переопределить все методы надкласса и, таким образом, ничего не наследовать. И, наоборот, существуют механизмы для совместного использования кода метода, который не полагается на отношение подкласса (например, strahtforward procedural abstraction). Таким образом, надо тщательно отличать наследование от образования подкласса (subclassing).

Без подклассов вхождение слова $self$ в объявлении класса всегда ссылается на объект этого класса. С подклассами получается иначе. В методе, который подкласс c' наследует из класса c, слово $self$ ссылается на объект класса c', но не на объект исходного класса c. Точнее, используя слово $self$ можно получить доступ к методам, переопределенным в c', но не к исходным методам класса c.

Специальный идентефикатор *super* в методе подкласса может использоваться для вызова версии метода надкласса. Более точно, $super.m(a)$ обращается к методу m текущего непосредственного надкласса с аргументом a, при этом слово $self$ продолжает иметь значение текущего $self$. Последнее условие - критично, оно гарантирует, что любое вхождение $self$ внутри метода m трактуется как ссылка на объект текущего подкласса, а не как ссылка на объект любого надкласса.

Реализация выбор метода должна быть пересмотрена в свете текущего обсуждения подклассов. В стиле модели делегирования методов, реализация наследования означает, что хранилища методов создаются для каждого (под)класса. В случае языков с динамической типизацией, эти хранилища просматриваются от подклассов к надклассам, то тех пор пока не будет обнаружен подходящий метод (рис. 3). В случае языков со статической типизацией (при котором иерархия хранилищ методов известна в момент компиляции) хранилища собраны на каждом уровне, так что циклический просмотр не является необходимым; в том смысле, что после первого шага делегирования будет использоваться модель встраивания методов (рис. 4).

Множественное наследование возникает, когда класс наследует атрибуты более чем одного надкласса. (В противном случае это одиночное наследование.) Не существует ничего особенного относительно множественного наследования, кроме того, что у класса обычно указывается несколько надклассов. Это влечет необходимость разрешения конфликтов и неоходимость избегать дублирования среди идентичных атрибутов надклассов. Наиболее простым

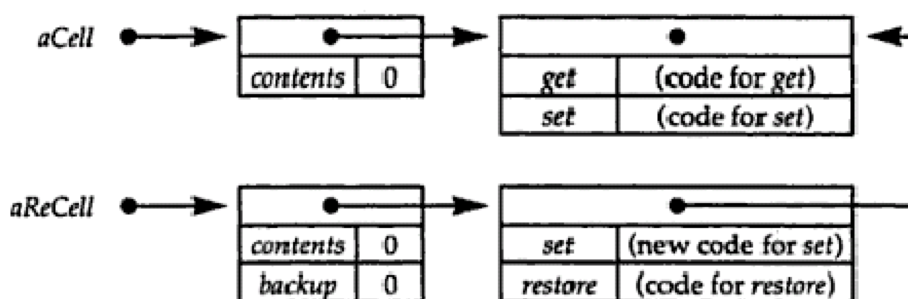

Рис. 3: Иерархия хранилищ методов

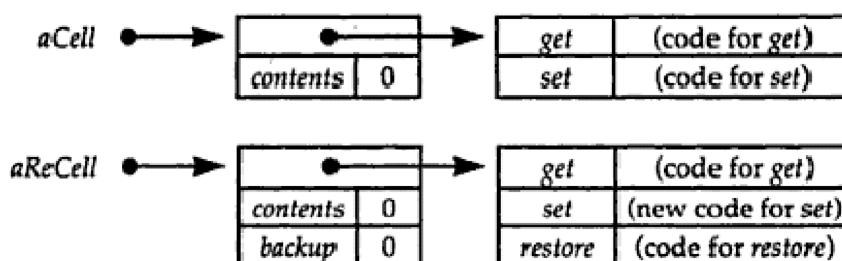

Рис. 4: Свернутые (collapsed) хранилища методов

решением является требование идентифицирования всех дублированных идентичных атрибутов и запрета любых конфликтов. Так же, используется различные другие решения. При множественных надклассах для процедуры выбора метода уникального нет однозначного пути вперед и назад, по которому она могла бы следовать и, поэтому, значение идентификатора *super* должно быть более точным.

2.4 Категоризация и динамическая диспетчеризация

Из уже сказанного может показаться, что подклассы - это просто удобный способ чтобы не повторять определения, уже сделанные в надклассе. Рассмотрим понятие категоризации, что бы получить дополнительные полезные возможности. Рассмотрим следующие определения:

```
var myCell: InstanceTypeOf(cell) := new cell;
var myReCell: InstanceTypeOf(reCell) := new reCell;
procedure f(x: InstanceTypeOf(cell)) is ... end;
```

Рассмотрим также следующий фрагмент кода, в области видимости этих определений:

```
myCell := myReCell;
f(myReCell);
```

В этом коде, значение экземпляра класса $reCell$ присвоено в переменную, объявленную как экземпляр класса $cell$. Похожим образом, экземпляр класса $reCell$ передается в процедуру, ожидающую экземпляр класса ell. Оба фрагмента будут не допустимыми в языках вроде *Pascal*-я, потому что типы $InstanceTypeOf(cell)$ и $InstanceTypeOf(reCell)$ не совпадают. В объектно - ориентированных языках такой код становятся допустимым вследствие следующего правила, которое воплощает то, что частно называют полиморфизмом (подтипов):

> Если c' подкласс класса c и o' это экземпляр $'$, то o' будет экземпляром c.

или, с точки зрения проверки типов

(1)

Если c' является подклассом c, и $o' : InstanceTypeOf(c')$, тогда $o' : InstanceTypeOf()$

Проанализируем утверждение (1), с помощью рефлексивного и транзитивного отношения подтипа (<:) между типами $InstanceTypeOf$. Интуитивно предполагается, что это отношение подтипа вводится как теоретико - множественное включение между множествами. Пока не будем определять отношения подтипа точно, но будем считать, что оно удовлетворяет следующим двум свойствам:

(2)

если $a : A$ и $A <: B$, тогда $a : B$

(3)

$InstanceTypeOf(c') <: InstanceTypeOf(c)$ тогда и только тогда, когда c' есть подклассом c

Вместе (2) и (3) влекут (1).

Свойство (2), называемое категоризация (subsumption, правило включения) является характеристическим свойством отношения подтипа. При использовании категоризации, величина типа A может рассматриваться как величина надтипа B. Можно говорить, что величина типа A относится к типу B.

Свойство (3), которое мы будем называть подкласс-это-подтип, является характеристическим свойством определения подкласса в классических языках, основанных на классах. Так как наследование очень связано с определением подкласса, мы можем понимать (3) как свойство наследование-это-выделение_подтипа. Как будет показано ниже, некоторые другие, позже созданные языки, основанные на классах, используют подход отличный от приведенного: наследование-это-не_выделение_подтипа.

После введения понятия категоризации, можно пересмотреть смысл вызова метода. Например, в коде:

```
procedure g(x: InstanceTypeOf(cell)) is
    x.set(3);
end;
g(myReCell);
```

необходимо определить смысл выражения $x.set(3)$ во время вызова g. Объявленный тип формального параметра x это $IntanceTypeOf(cell)$, в тоже время, фактическим параметром является переменная $myReCell$, которая является экземпляром класса $reCell$. Так как метод set переопределен в классе $reCell$, существует две возможности:

Static dispatch:	*x.set*(3) **executes the code of** ***set*** **from class** ***cell***
Dynamic dispatch:	*x.set*(3) **executes the code of** ***set*** **from class** ***reCell***

Статическая диспетчеризация (static dispatch, иначе - раннее связывание) основана на информации о x, доступной во время компиляции кода. Динамическая диспетчеризация (dynamic dispatch, позднее связывание) основана на информации о значении x, доступной во время выполнения.

Можно говорить, что $InstanceTypeOf(reCell)$ это настоящий тип x во время выполнения $g(myReCell)$, и этот настоящий тип определяет выбор метода.

Динамическая диспетчеризация настолько широко используется во всех объектно - ориентированных языках, что она может трактоваться как одно из определяющих свойств этих языков. Динамическая диспетчеризация является важной характеристикой абстракции объектов: каждый объект имеет автономную информацию о том, как он должен себя вести, таким образом, что нет нужды в исследовании объекта для того, что бы выбрать операцию, применимую к объекту.

Интересным следствием динамической диспетчеризации оказывается то, что категоризация не должна иметь побочных эффектов во время выполнения. Например, если применение категоризации из $InstanceTypeOf(reCell)$ в $InstanceTypeOf(cell)$ вызвало бы преобразование $reCell$ в ell таким

образом, что оказались бы отброшены дополнительные атрибуты (*backup* и *restore*), то динамическая диспетчеризация, вызвавшая метод *set*, сбойнула бы. Тот факт, что категоризация не имеет побочных эффектов, полезен и для эффективности и для семантики.

2.5 Пропущенная и восстановленная информанция о типе

Хотя категоризация не имеет побочных эффектов во время выполнения, она все таки, уменьшает доступную статическую информацию о настоящем типе объекта. Вообразим себе корневой класс, который вообще не имеет атрибутов, так что все остальные классы является его подклассами. Тогда любой объект может рассматриваться, как элемент корневого класса и может трактоваться как бесполезный объект без атрибутов.

Более слабым примером, будет категоризация объекта из типа $InstanceTypeOf(reCell)$ в $InstanceTypeOf(cell)$. В этом случае теряется доступ к полю *backup* (так же, как и к методу *restore*). Это не означает, однако, бесполезности этих полей, так как они продолжают использоваться через переопределенный метод *set*. Таким образом, атрибуты, пропущенные изза категоризации могут быть использованы, благодаря динамической диспетчеризации.

С точки зрения чистой объектно - ориентированной методологии, динамическая диспетчеризация является единственной возможностью использовать атрибуты, пропущенные изза категоризации. Эта позиция основывается на требованиях абстрагирования: Любую информацию об объекте должна быть получена через вызовы его методов. При чистом подходе категоризация предлагает простой и эффективный механизм для сокрытия приватных атрибутов. Создав объект $reCell$ и, после категоризации, используя его как объект *cell* надо быть уверенным, что отсутствует возможность изменять поле *backup*.

Многие языки, однако, имеет возможности узнать настоящий тип объекта и, следовательно, восстановить доступ к опущенным атрибутам ([13], [15]). Например, некоторая процедура с формальным параметром x типа $InstanceTypeOf(cell)$ может содержать следующий код. Оператор преобразования типа (*typecast*) связыает параметр x с объектом c или rc в зависимости от настоящего (во время выполнения) типа x:

```
typecase x
    when rc: InstanceTypeOf(reCell) do ... rc.restore() ... ;
    when c: InstanceTypeOf(cell) do ... c.set(3) ... ;
end;
```

Через соответствующую ветку можно получить доступ к ранее недоступному атрибутом.

Механизм *typecast* очень полезен, но считается не совсем правильным (impure - нечистый) вследствие нескольких методологических причин (равно как изза теоретических). Во первых, он нарушает объектную абстракцию, раскрывая информацию, которая может трактоваться как приватная. Во вторых, это делает программу менее надежной, так как создает возможность сбоя во премя выполнения, если не будет выбрана ни одна из известных веток. В третьих, и, возможно, по наиболее важной причине, код становится менее гибким: после добавления еще одного подкласса класса *cell* программист должен пересмотреть и расширить оператор *typecast* в существующем коде. В правильном фреймворке, дописывание новых подклассов не должно вызывать необходимость дополнительного кодирования в ранее созданных классах. Это является хорошим свойством, еще и по той причине, что исходный код коммерческих библиотек может быть не доступен.

Хотя отказ от использования *typecast* может оказаться недостижим, все таки необходимым остается требование благоразумного использования этого оператора. Стремление сократить область использования оператора *typecast* прослеживается во многих структурах объектно - ориентированных языков. А именно, *typecast* необходим при эмулировании объектов записями процедур не объектно ориентированными языками; и, напротив, обычное приведение типа методов в объектно - ориентированных языках избегает его использование. Некоторые тонкие способы типизации методов, возможные без *typecast*, обсуждаются позднее.

2.6 Ковариантность, контравариантность и инвариантность

В заключение главы исследуем более гибкие техники типизирования для классов и методов, которые помогают избежать использования *typecast*. Для объяснения этих техник, сначала пересмотрим базовые свойства выделения типа для декартовых произведений и функциональных типов.

Рассмотрим типы декартовых произведений. Тип А $\times B$ является типом пар, у которых левая компонента принадлежит типу A, а правая - B. Операции $fst(c)$ и $snd(c)$ извлекают левую и правую компоненты, соответственно, из элемента c типа А $\times B$.

Будем говорить, что $\times$ является ковариантным оператором (на обоих аргументах), так как А $\times B$ изменяется в том же смысле, как A или B:

$A \times B <: A' \times B'$ при условии, что $A <: A'$ и $B <: B'$

Далее, идет объяснение этого свойства.

Обоснование ковариантности декартового произведения

> Каждая пара (a, b) у которой левая компонента типа A и правая типа B имеет тип А $\times B$. Если $A <: A'$ и $B <: B'$, то после категоризации, мы получаем $a : A'$ и $b : B'$, то есть, пара (a, b) так же имеет тип $A' \times B'$. Это значит, что любая пара типа А $\times B$ так же является парой типа $A' \times B'$, всегда, когда $A <: A'$ и $B <: B'$. Другими словами включение декартовых произведений $A \times B <: A' \times B'$ выполняется всегда, когда выполняется $A <: A'$ и $B <: B'$.

Теперь рассмотрим функциональные типы. Тип $A \to B$ это тип функций с типом аргумента A и типом результата B.

Будем говорить, что $\to$ является контравариантным оператором на левом аргументе так как, $A \to B$ изменяется в смысле противоположном изменению A; правый аргумент, напротив, остается ковариантным:

$A \to B <: A' \to B'$ при условии, что $A' <: A$ и $B <: B'$

Обоснование ко/контравариантности функционального типа

> Если $B <: B'$, то функция f типа $A \to B$ вырабатывает результаты типа B' вследствие категоризации. Если $A' <: A$, то A' входит в область определения f, так как она определена на A. Тогда каждая функция типа $A \to B$ имеет также тип $A' \to B'$ всегда когда $A' <: A$ и $B <: B'$. Другими словами, включение $A \to B <: A' \to B'$ выполняется всегда когда выполняются $A' <: A$ и $B <: B'$.

В случае функций от многих аргументов, например, типа $(A_1 \times A_2) \to B$, контравариантность проявляется на обоих сомножителях A_1 и A_2, потому, что декартово произведение, которое ковариантно на обоих сомножителях, появилось в контравариантном контексте.

Наконец, рассмотрим пару, у которой можно обновлять компоненты. Временно обозначим ее тип как $A \oplus B$. Пусть дано $p : A \oplus B$, $a : A$ и $b : B$, пусть есть операции $getLft(p) : A$ и $getRht(p) : B$ такие, что извлекают компонеты и операции $setLft(p, a)$ и $setRht(p, b)$ котороые обновляют компоненты.

Оператор $\oplus$ не удовлетворяет ни условию ковариантности, ни условию контравариантности.

$A \oplus B <: A' \oplus B'$ при условии, что $A' = A$ и $B = B'$

Говорится, что оператор $\oplus$ инвариантнен (на обоих аргументах).

Обоснование инвариантности $A \oplus B$

> Пусть $A <: A'$ и $B <: B'$, можно ли полагать ковариантность $A \oplus B <: A' \oplus B'$? Предположим это включение, тогда из $p : A \oplus B$ получим $p : A' \oplus B'$ и можно применить $setLft(p, a')$ для любого $a' : A$. Тогда $getLft(p)$ может возвращать элементы типа A', не являющиеся элементами типа A. Это означает, что включение $A \oplus B <: A' \oplus B'$ не выполняется.
>
> Наоборот, если $A'' <: A$ и $B'' <: B$, можно ли предположить контравариантность $A \oplus B <: A'' \oplus B''$? Из $p : A \oplus B$ получаем $p : A'' \oplus B''$. Значит мы можем некорректно вывести, что $getLft(p) : A''$. То есть, включение $A \oplus B <: A'' \oplus B''$ не выполняется тоже.

Большинство споров в обществе объектно - ориентированного программирования вращаются вокруг вопроса должны ли типы аргументов у методов изменятся контравариантно или ковариантно от классов к подклассам. Мы должны настаивать на том, что свойства $\times$, $\rightarrow$, $\oplus$ неизбежно следует из наших предположений. Считать, что функциональный тип $\rightarrow$ ковариантен на его левом аргументе так же не правильно, как то, что π равно 3.0 (any more than we can take π to be 3.0). В следующих разделах, при помощи простого анализа убедимся в невозможности произвольного изменения свойств методов. Нет возможности считать типы аргументов у метода изменяющимися ковариантно, при неизменном значении терминов ковариантность, выделение подтипа, категоризация (как, например, в [9]). Наши определения влекут невозможность ковариантности параметров типа у методов. Ковариантность формальных параметров методов может повлечь непредсказуемое поведение.

2.7 Конкретизация метода

При обсуждении подклассов был выбран наиболее простой подход к переопределению методов. Он требует, что новый метод имеет точно тот же тип, как и переопределяемый. Это условие может быть ослаблено с тем, чтобы позволять уточнение метода (method specialization, конкретизация). Уточнение позволяет новому методу иметь другие аргументы и тип результата, более подходящие для подкласса. Как и ранее, не будет позволено переопределять и уточнять типы полей: поля можно обновлять, подобно компонентам типа $A \oplus B$, и, следовательно, их типы должны быть инвариантными.

Переопределим метод m с помощью новых типов B' и A' следующим образом:

```
class c is
    method m(x: A): B is ... end;
    method m1(x1: A1): B1 is ... end;
end;

subclass c' of c is
    override m(x: A'): B' is ... end;
end;
```

Выяснение о допустимости типов B' и A' ограничивается только возможностью категоризации между типами $InstanceTypeOf(c')$ и $InstanceTypeOf(c)$. Если o' типа $InstanceTypeOf(c')$ категоризируется к типу $InstanceTypeOf(c)$, и вызывается метод $o'.m(a)$, то для аргумента оказывается приемлимым иметь статический тип A и для результата - тип B. Следовательно, оказывается достаточным требовать что $B' <: B$ (ковариантно) и $A' <: A$ (контравариантно). Это называется методом конкретизации при переопределении: тип результата B уточняется до B' и тип параметра A обобщается до A', что означает общую конкретизацию функционального типа $A \to B$ к типу $A' \to B'$.

Существует другая форма метода конкретизации, которая применяется неявно в случае наследования. Слово $self$ при вхождении в методе класса c может трактоваться как имеющее тип $InstanceTypeOf(c)$, в том смысле, что все объекты, связанные к $self$ имеют тип $InstanceTypeOf(c)$ или как имеющее подтип $InstanceTypeOf(c)$. Если методы c наследуются классом c', то же самое слово $self$ может, подобным образом, трактоваться как имеющее тип $InstanceTypeOf(c')$ (can be considered). То есть, тип слова $self$ по умолчанию конкретизируется при наследовании (ковариантно!).

Кроме того, рассматривая метод в качестве функции с первым параметром $self$, получаем еще одну возможность взглянуть на конкретизацию; будем называть такие функции - пре-методами. Например, в случае метода m_1 класса c, его пре-метод pm_1 будет иметь тип $(InstanceTypeOf(c) \times A_1) \to B_1$. Этот тип будет подтипом $(InstanceTypeOf(c') \times A_1) \to B_1$, так что pm_1 будет имет тоже этот тип по правилу включения. Значит pm_1 имеет тип правильно преметода для c', и может быть наследован классом c'. Вообще, говоря, каждый наследуемый пре-метод типа $(InstanceTypeOf(c) \times A_1) \to B_1$ по правилу включения имеет тип $(InstanceTypeOf(c') \times A'_1) \to B'_1$ для любых $A'_1 <: A_1$ и $B_1 <: B'_1$. То есть, параметры, включая $self$ могут конкретизироваться, а выход функции может обобщаться.

Надо заметить, что включение для типов параметров и результата, которое получено для наследования противоположно включению для типов при переопределения: для наследования, типы параметров можно конкретизировать, а тип результата - обобщать; для переопределения -

типы параметров можно обобщать, а тип результата - конкретизировать. В любом случае, необходимые правила для конкретизации метода и для наследования, и для переопределения будут непосредственными следствиями ограничений, возникающими из за выделения подтипа и категоризации.

sound rules - необходимые правила. В смысле необходимости и достаточности - soundness and completness.

2.8 Конкретизация типа у слова $self$

Конкретизация метода добавляет гибкостии в определении подкласса, позволяя изменения типа результата работы метода. Еще одна возможность появляется, если тип результата метода оказывается типом текущего класса. В этом разделе будут обсуждены конструкции, придуманные для конкретизации таких методов.

Определения класса часто бывают рекурсивные, в том смысле, что можно определить класс c, который будет содержать вхождения $InstanceTypeOf(c)$. Например, рассмотрим класс c, содержащий метод m с типом результата $InstanceTypeOf(c)$: :

```
class c is
    var x: Integer := 0;
    method m(): InstanceTypeOf(c) is ... self ... end;
end;
subclass c' of c is
    var y: Integer := 0;
end;
```

При наследовании, рекурсивные типы, по умолчанию, остаются точно такими же как и другие. Например, для o' из класса c', получаем, что $o'.m()$ имеет тип $InstanceTypeOf(c)$, а не, к примеру, $InstanceTypeOf(c')$. Вообще говоря, использование $InstanceTypeOf(c')$ в качестве типа результата для наследованного метода m в классе c' не необходимо, потому что m может создавать и возвращать экземпляры класса c, которые не являются экземплярами класса c'.

Предположим, однако, что m возвращает $self$, возможно, после изменения поля x. Тогда это влечет необходимость (then it would be sound to give) присвоения типу результата наследованного метода тип ($InstanceTypeOf(c')$, так как $self$ всегда связан с текущим объектом (bounded to the receiver of

invocations), который вызвал метод. В этом случае, более точного типа, необходимо избегать использования оператора *typecase*: преобразование типа результата к $(InstanceTypeOf(c)$ означает потерю информации.

Этот довод подводит к понятию типов $Self$. Ключевое слово $Self$ представляет тип $self$. Вместо присвоения типу результата m тип $InstanceTypeOf(c)$, можно писать:

```
class c is
    var x: Integer := 0;
    method m(): Self is ... self ... end;
end;
```

Код метода m полагается на предположение, что $Self$ является подтипом типа $(InstanceTypeOf(c)$, что означает, что $self$ имеет тип $Self$. Поле, добавленное к $self$, не изменяет его тип $Self$ (Field updates to $self$ preserve its $Self$ type). То есть, необходимо для метод m тип результата как имеющий тип $Self$, а не просто $InstanceTypeOf(c)$.

Если c' объявлен как подкласс c, результат типа m все равно останется $Self$. Однако, теперь тип $Self$ будет трактоваться как подтип $(InstanceTypeOf(c')$. Значит, тип $Self$, как тип результата метода, автоматически конкретизируется (is automatically specialized on subclassing).

Не существует никаких препятствий для расширения классический языков, основанных на класах,

добавлением к ним возможности использовать типа $Self$ на ковариантных местах, например как тип результата методов. Такое расширение усилит выразительную мощь языка и предотвратит потерю информации о типе без особых накладных расходов, кроме необходимости корректного отслеживания типа слова $self$. Можно даже позволить использовать $Self$ в качестве типа полей. Это необходимо так как такие поля обновляются только толи $self$, толи обновленной версией $self$.

Следующим шагом, можно было бы позволить использование $Self$ на контравариантных местах (как тип параметров). Это позволено в Eiffel-е ([14]). К сожалению, как демонстрирует простой контрпример ([10]), контравариантное использование типа $Self$ не совместимо с правилом категоризации. Как мы убедились, типы аргументов у наследуемых методов должны обобщаться, но не конкретизироваться. Надлежащее обращение с типом $Self$ у параметров относится к новым разработкам в языках, основанных на классах, и хорошо совмещается с их классическими качествами. Эти вопросы еще будут обсуждаться в разделах 3.4 и 3.5 .

3 БОЛЕЕ ТОНКИЕ СВОЙСТВА, ОСНОВАННЫЕ НА КЛАССАХ

Одной из наиболее главных характеристик языков, основанных на классах, является строгая корреляция между наследованием, образованием подклассов и выделением подтипов. При идентичной трактовке этих отношений достигается существенная экономия в синтаксисе и концепциях. Эта идентичность так же влечет дополнительную гибкость при категоризации: объект подкласса, некоторые методы которого могут быть наследованы, всегда может использоваться вместо объекта надкласса, как относящийся к надтипу.

Существуют ситуации однако, при которых наследование, образование подклассов и выделение подтипов конфликтуют. Оба способа для повторного использования кода: и наследование, и параметризация ограничивают совпадение этих понятий. Значит, необходимо потратить усилия на их отличия. Некоторые отличия между образования подклассов от выделения подтипов стали широкоизвестной банальностью; другие отличия менее очевидны. В этой главе рассмотрим некоторые возможности для несовпадения понятий.

3.1 Типы объектов

В первых языках (например, Simula), описание типа объектов перемешивалось с реализацией методов. Эта ситуация конфликтует с широко признанными теперь преимуществами хранения спецификаций отдельно реализации, частично для того, чтобы иметь возможность раздельной работы программистов большой команды.

Относительно недавные изменения классической модели решают эти проблемы. Разделение спецификации объекта и его реализации может быть достигнуто введением типов для объектов, которые независимы от конкретных классов ([4], [16]). Этот подход поддерживается в Modula-3 и в других языках, имеющими классы и интерфейсы.

Смысл такого разделения несколько ограниченно проявлялся во время начального обсуждения типа $InstanceTypeOf(cell)$. Этот тип обозначал то, что все его элементы получаются при помощи выполнения оператора *new* для класса *cell*, и, следовательно, все они имеют идентичные методы. Позднее оказалось, что используя свойства образования подклассов переопределение методов, категоризацию и динамическую диспетчеризацию возможно получать объекты типа $InstanceTypeOf(cell)$, отличающиеся дополнительными

атрибутами и различным кодом методов. Стало, следовательно, более понятно, что тип $InstanceTypeOf(cell)$ должен явно зависить от сути класса $cell$, которая описывает только некоторую специфику кода методов. На деле, код класса $cell$ не обязательно может быть обнаружен в объектах типа $InstanceTypeOf(cell)$.

Вообще говоря, общими для элементов типа $InstanceTypeOf(cell)$ являются только сигнатуры атрибутов, указанных в классе $cell$. Это множество всех сигнатур класса иногда называется протоколом объекта.

Было бы естественно такой протокол рассматривать как тип объектов класса $cell$. Рассмотрим следующие классы $cell$ и $reCell$:

```
class cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;

subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
        self.backup := self.contents;
        super.set(n);
    end;
    method restore() is self.contents := self.backup end;
end;
```

Затем вводем два объектных типа: $Cell$ и $ReCell$, которые соответствуют этим классам. Они записаны как независимые типы, однако нужно ввести правильный синтаксис чтобы избежать повторения общих компонент текста.

```
ObjectType Cell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

ObjectType ReCell is
    var contents: Integer;
    var backup: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;
```

В типах содержатся атрибуты и их типы, но отсутствует реализация. Возможно использование этих типов в интерфейсах функций, так же как и различная реализация.

Обозначение $ObjectTypeOf(cell)$ используется в качестве мета-обозначение для типа $Cell$. Этот тип может быть механически извлечен из класса $cell$. Следовательно, допустимы записи либо $o : ObjectTypeOf(cell)$ либо $o : Cell$. Главное, что ожидается от $ObjectTypeOf$ это

$newc : ObjectTypeOf(c)$ для любого класса c

Могут быть различные классы $cell$ и $cell_1$, производящие объекты одного и того же типа $Cell$, эквивалентному обоим типам: как $ObjectTypeOf(cell)$, так и $ObjectTypeOf(cell_1)$. Следовательно, от объектов, имеющих тип $Cell$ требуется только соответствие определенному протоколу, независящему от реализации.

3.2 Различия образования подклассов от выделения подтипов

Отношение подтипа первоначально основывалось на отношении подкласса. Если типы объектов оказались независимы от классов, стало необходимо сделать независимое определение. Возможно несколько выборов: либо выделение подтипов определяется по структуре типа, либо при помощи имен типов в объявлениях, и ожидать, как в первом случае, что часть структуры типов совпадет.

Стуруктурное выделение типа (выделение типа, основанное на структуре типа) является желательным свойством, так как помогает подпобру типов в распределенных и постоянных (persistent) системах([3], [15]). К недостаткам относится возможность случайного, незапланированного совпадения типов. Однако, всегда можно избежать такого совпадения, вводя различия на вершине структурной иерархии типов([15]). Напротив, выделение типа, основанное на именах типов, тяжело определить точно и, при этом, не поддерживается структурное выделение типов.

Для текущих целей достаточно использовать вполне простую форму структурного выделения типа. Считается, что для двух объектных типов O и O':

$O' <: O$ если O' имеет те же компоненты как O и, возможно, больше

где компонентой типа считается имя поля или метода вместе с его типом. Так, например, $ReCell <: Cell$.

Из этого определения следует, что типы объектов станут инвариантами в их компонентных типах (component types), хотя конкретизации методов с необходимостью влечет расширение, как обсуждалось в разделе 2.7 . Для случаев, когда типы объектов определяются рекурсивно, необходимо более аккуратно определить выделение подтипов. Отложим дальнейшее обсуждение этого вопроса до разделов с исчислением первого порядка.

Такое определение выделения типа и типов объектов, естественно, приводит к возможности выделения нескольких типов (multiple subtyping), вследствие неупорядоченности компонент. Например, рассмотрим тип объектов:

```
ObjectType ReInteger is
    var contents: Integer;
    var backup: Integer;
    method restore();
end;
```

Тогда $ReCell <: Cell$ и $ReCell <: ReInteger$.

Вследствие этого нового определения выделения подтипов получаем:

(4)

если c' подкласс класса c тогда $ObjectTypeOf(c') <: ObjectTypeOf(c)$.

Это правило выполняется потому, что подкласс может только добавить атрибуты к классу и вследствие сохранения у методов существующих типов при переопределении.

Свойство (4) является переформулировкой свойства (3) (наследование-это-выделение_подтипа) для $ObjectTypeOf$. Хотя свойство (3) является свойством двойной импликации, утверждение, обратное к (4) не выполняется: существуют несвязанные типы c и c', такие, что $ObjectTypeOf(c) = O$ и $ObjectTypeOf(c') = O$, если $O' <: O$.

Следовательно, в результате ослабления свойства двойной импликации (3) и одиночной импликации (4), получается некоторое отделение образования подклассов от выделения подтипов. Как и раньше, образование подкласса влечет выделение подтипа, так что все использования правила категоризации остаются допустимыми. Но, так как категоризация основано на выделении подтипа а не на образовании подкласса, категоризация получает большую гибкость.

В заключение, понятие 'наследование-это-выделение_подтипа' может быть ослаблено до понятия 'наследование-влечет-выделение_подтипа' без потери выразительности и с выигрышем при отделении реализации кода от объявления интерфейсов.

3.3 Параметры типа

Параметризация типа - это обычная техника для использования одного и того же кода для различных типов. Она становится общей в современных объектно - ориентированных языках, независимо от объектно - ориентированных свойств. Параметры типа описаны в этом разделе, так как они вызывают интерес сами по себе и, потому что, используются позже в этой главе при обсуждении бинарных методов.

В сочетании с выделением типа, параметризацию можно использовать для избавления от трудностей, связанных с контравариантностью, например, при конкретизации (уточнении) методов. Рассмотрим типы объектов $Person$ и $Vegetarian$, предположим, $Vegetables <: Food$ (но не наоборот) [5]:

```
ObjectType Person is
    ...
    method eat(food: Food);
end;
ObjectType Vegetarian is
    ...
    method eat(food: Vegetables);
end;
```

Предположив, что вегетарианец - это человек, можно было бы ожидать $Vegetarian <: Person$. Однако, такое включение не может выполняться, вследствие контравариантности входных параметров у метода eat. Если ошибочно предположить такое включение, то вегетарианец, вследствие правила категоризации, может трактоваться как едящий мясо.

Корректное выполнения этого правила между вегетарианцами и людьми можно получить, преобразовав типы объектов в типы операторов (функций из типов в типы), параметризованных по типу параметра $food$:

```
ObjectOperator PersonEating[F <: Food] is
    ...
    method eat(food: F);
end;
ObjectOperator VegetarianEating[F <: Vegetables] is
    ...
    method eat(food: F);
end;
```

Механизм, который используется при этом, называется связанная параметризация типа (bounded type parametrization) Переманная F

называется параметром типа. Связывание, такое как $F <: Vegetables$, ограничивает возможные подстановки F только подтипами $Vegetables$. То есть, $VegetarianEating[Vegetables]$ это корректно сформированный тип, а $VegetarianEating[Food]$ - нет. Тип $VegetarianEating[Vegetables]$ это экземпляр $VegetarianEating$, эквивалентный типу $Vegetarian$.

Не существует (злементов) экземпляров операторов, таких как $PersonEating$; существуют только экземпляры в виде $PersonEating[F]$ для заданного F. Поэтому нельзя получить никакого отношения включения между операторами $VegetarianEating$ и $PersonEating$. Однако можно утверждать следующее:

для всех $F <: Vegetables, VegetarianEating[F] <: PersonEating[F]$

Потому, что для любого $F <: Vegetables$ два экземпляра включаются из за обычных правил для выделения типа. В частности, выполняется $Vegetarian = VegetarianEating[Vegetables] <: PersonEating[Vegetables]$. Такое включение полезно для категоризации: оно, в частности, корректно утверждает, что вегетарианцы это люди, которые едят только вегератианскую еду.

Типы, относящиеся к связанным параметрам типа, называются связанными абстрактными типами (bounded abstract types). Так же они называются частичными абстрактными типами. Связанные абстрактные типы позволяют другое решение задачи преставления $Vegetarian$ как подтипа $Person$. Переопределим наши объектные типы, добавляя параметр F как подтипа $Food$ в качестве одного из атрибутов:

```
ObjectType Person is
    type F <: Food;
    ...
    var lunch: F;
    method eat(food: F);
end;

ObjectType Vegetarian is
    type F <: Vegetables;
    ...
    var lunch: F;
    method eat(food: F);
end;
```

Компонета типа $F <: Food$ в типе $Person$ означает, что данный человек ест определенную еду, но не знаем какую именно. Атрибут $lunch$ содержит некоторую еду, которую этот человек может есть.

Выбрав конкретный подтип типа $Food$, $F = Dessert$, указав десерт для поля $lunch$ и реализуя метод с параметром типа $Dessert$, можно построить объект типа $Person$. В результате получили объект $Person$, забыв какое конкретное значения F использовано при реализации.

Теперь включение $Vegetarian <: Person$ выполняется. Вегетарианцев, которые рассматриваются как принадлежащих типу $Person$, можно безопасно кормить ихней едой, так как они созданы при $F <: Vegetables$. Ограниченность такого подхода проявляется в том, что человека можно кормить едой, которую он носит с собой в качестве компоненты типа F, но не едой, полученной независимым способом.

3.4 Образование классов без выделения типа

В разделе 3.2 было показано, что разделение понятий выделение подтипа и образование подкласса увеличивает возможности правила категоризации. Еще один подход выявил, что дальнейшее разделение этих понятий приводит к усилению возможностей наследования. Этот подход окончательно отказывается от идеи, что образование подкласса подразумевает выделение подтипа (свойство (4)), и известен под именем наследование-это-не-выделение_типа [11]). Этот подход, в основном, мотивируется желанием обрабатывать контравариантные (формальные параметры функций) вхождения типа $Self$ так, чтобы позволять наследование методов; примеры таких методов естественным образом появляются из прикладных задач. Платой за гибкость в наследовании является сужение области применения правила категоризации. Если тип $Self$ используется в контравариантных местах, подклассы не обязательно влекут подтипы.

Рассмотрим два типа Max и $MinMax$ для целых, к которым добавили методы min и max. Оба эти типа определяются рекурсивно:

```
ObjectType Max is
    var n: Integer;
    method max(other: Max): Max;
end;

ObjectType MinMax is
    var n: Integer;
    method max(other: MinMax): MinMax;
    method min(other: MinMax): MinMax;
end;
```

Рассмотрим также два класса:

```
class maxClass is
    var n: Integer := 0;
    method max(other: Self): Self is
        if self.n > other.n then return self else return other end;
    end;
end;
subclass minMaxClass of maxClass is
    method min(other: Self): Self is
        if self.n < other.n then return self else return other end;
    end;
end;
```

Методы min и max называются бинарными (binary) потому что они работают с двумя объектами: $self$ и еще одним - *other*, причем второй объект вводится контравариантным вхождением типа $Self$ ([7]). Заметьте, что метод max, имеющий формальный параметр типа $Self$, наследуется в классе $minMaxClass$ из $maxClass$.

Интуитивно кажется, что тип Max соответствует классу $maxClass$, а $MinMax$ - классу $minMaxClass$. Что бы сделать это соответствие более точным, необходимо определить значение $ObjectTypeOf$ для классов, содержащих вхождение типа $Self$. Положим $ObjectTypeOf(maxClass) = Max$ и $ObjectTypeOf(minMaxClass) = MinMax$. Для выполнения этих равенств, надо использованию типа $Self$ в классе поставить в соответствие использование рекурсии в типе. Так же неявно конкретизируем тип $Self$ для наследуемых методов; например, отобразим использование $Self$ в наследуемом методе max в $MinMax$. Тогда оказывается, что любое вхождение $maxClass$ имеет тип Max, а любое вхождение $minMaxClass$ - тип $MinMax$.

При этом оказывается что, хотя $minMaxClass$ является подклассом $maxClass$, $MinMax$ не может быть подтипом Max. Рассмотрим класс:

```
subclass minMaxClass' of minMaxClass is
    override max(other: Self): Self is
        if other.min(self) = other then return self else return other end;
    end;
end;
```

Любая переменная mm' класса $minMaxClass'$ имеет тип $MinMax$. Если бы $MinMax$ был бы подтипом Max, то mm' имела бы тип Max и можно было применять метод $mm'.max(m)$ к любому значению m типа Max. Так как m вообще не имеет аттрибута min, перегруженный метод max у объекта mm' не может быть выполнен. Таким образом

$MinMax <: Max$ не выполняется

То есть, подклассы с контравариантным вхождением класса $Self$ не всегда принадлежат отношению 'быть подтипом'.

3.5 Протоколы обьекта

Можно установить другое отношение между типом класса и типом его подкласса, даже если образование подкласса не влечет выделение типа. В таком случае к типам, находящиеся в таком отношении, нельзя применять свойство категоризации. Изучим это новое отношение между типами объектов.

В примере раздела 3.4 нельзя получить пользу от квантификации над подтипами Max вследствие невозможности установить отношение типизации (In the exapmle of Section 3.4 , we cannot usefully quantify over the subtypes of Max because of the failure of subtyping). Такое параметрическое определение как:

```
ObjectOperator P[M <: Max] is ... end;
```

оказывается не слишком полезным; можно подставить P в $P[Max]$, но тип $P[MinMax]$ оказывается некорректным (is not well-formed).

Одноко интуитивно очевидно, что любой объект, удовлетворяющий протоколу $MinMax$, должен удовлетворять протоколу Max. Это наталкивет на мысль о введении некоторого отношения подпротокола, для того, что бы позволять полезную параметризацию. Для того, что бы обнаружить это отношение подпротокола введем два оператора типа $MaxProtocol$ и $MinMaxProtocol$:

```
ObjectOperator MaxProtocol[X] is
    var n: Integer;
    method max(other: X): X;
end;

ObjectOperator MinMaxProtocol[X] is
    var n: Integer;
    method max(other: X): X;
    method min(other: X): X;
end;
```

Обобщение это примера означает, что всегда можно любой тип T заменить оператором $T\!-\!Protocol$, заменяя все рекурсивные вхождения T. Оператор $T - Protocol$ является функцией на типах; взяв ее неподвижную точку, получаем тип T;

Можно обнаружить два формальных отношения между типами Max и $MinMax$. Во первых, $MinMax$ это неподвижная точка протокола $MaxProtocol$, то есть:

$MinMax <: MaxProtocol[MinMax]$

Во вторых, пусть $\prec:$ означает отношение подтипа высшего порядка между двумя операторами:

$P \prec: P'$ тогда и только тогда, когда $P[T] <: P'[T]$ для всех типов T

Тогда протоколы Max и $MinMax$ удовлетворяют ему:

$MinMaxProtocol \prec: MaxProtocol$

Любое из этих двух отношений может использоваться для введения понятия подпротокол:

S подпротокол T, если $S <: T - Protocol[S]$

или

S подпротокол T, если $S - Protocol \prec: T - Protocol$

Второе выражение точнее отражает тот факт, что это отношение подпротокола и, что оно отношение между операторами, а не между типами.

Всегда, когда появляются свойства общие для нескольких типов, можно задумываться об их параметризации. Для этого используется одна из следующих форм параметризации:

ObjectOperator $P_1[X <: MaxProtocol[X]]$ **is … end;**
ObjectOperator $P_2[P <: MaxProtocol]$ **is … end;**

Далее, можно подставлять P_1 вместо $P_1[MinMax]$, а $P_2 P_2[MinMaxProtocol]$.

На практике эти две формы параметризации оказыватся одинаково выразительными. Первая называется F-связанной параметризацией (F-bounded parametrization, [6], [8], [12]). Вторая форма - это связанная параметризация высшего порядка (higher - order bounded parametrization), определенная через (pointwise subtyping) неподвижную точку операторов типа. Формально она будет обсуждена в третьей части (Исчисление второго порядка). Для сравнения между этими формами параметризации, смотри ([1]).

Вместо работы с операторами типа, языки программирования, поддерживающие подпротоколы, могут определить отношение совпадения (matching relation) на типах, обозначаемое $< \#$. Свойства отношения совпадения

проектируются так, что бы соответствовать определению протокола. В зависимости от выбора отношения подпротокола, получаем:

$S <\# T$ если $S <: T - Protocol[S]$ (F- связанная интерпретация)

или

$S <\# T$ если $S - Protocol \prec: T - Protocol$ (интерпретация высшего порядка)

При любом определении получаем $MinMax <\# Max$.

Отношение совпадения не удовлетворяет свойству категоризации (то есть, $S <\# T$ и $s : S$ не влечет выполнение $s : T$); однако, совпадение полезно для параметризации всех типов, которые совпадают (находятся в отношении совпадение) с данным:

ObjectOperator $P_3[X <\# Max]$ **is** ... **end;**

Подстановка $P_3[MinMax]$ законна.

В заключение отметим, что даже при наличие контравариантных вхождений типа $Self$ и без выполнения отношения типизации, может существовать наследование бинарных методов вроде max. К сожалению, категоризация не выполняется в таком контексте, а квантификация по подтипам не помогает. Эти недостатки частично компенсируются существованием отношения подпротокола и возможностями параметризации.

4 ЯЗЫКИ, ОСНОВАННЫЕ НА ОБЪЕКТАХ

Главные свойства языков, основанных на классах, появились полностью оформленными в языке Simula. Языки, основанных на объектах, развивались поэтапно, при этом предполагалось, что такие языки должны быть более простыми и более гибкими чем традиционные языки, основанные на классах. В таких языках без типизирования существует только понятие объектов и динамической диспетчеризации. При наличии типизирования, языки поддерживают типы объектов, выделение подтипа и категоризацию.

Главными свойствами языков, основанных на объектах, является отсутствие классов и существование конструкций для создания отдельных объектов. В отсутствие классов совершенно естественно непосредственно ввести типы объектов и отделить интерфейс объектов от их реализации. Например:

```
ObjectType Cell is
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

object cell: Cell is
    var contents: Integer := 0;
    method get(): Integer is return self.contents end;
    method set(n: Integer) is self.contents := n end;
end;
```

Этот определение *object* создает объект типа *Cell* и называет его *cell*. Даже без классов, можно создавать набор однородных объектов, используя специальную процедуру для их построения. Она может иметь параметры для инициализации полей. Например:

```
procedure newCell(m: Integer): Cell is
    object cell: Cell is
        var contents: Integer := m;
        method get(): Integer is return self.contents end;
        method set(n: Integer) is self.contents := n end;
    end;
    return cell;
end;

var cellInstance: Cell := newCell(0);
```

Такая процедура, генерирующая объекты жутко похожа (uncannily similar) на классы *Simula*, которые тоже имеют параметры и выполнимое тело. Роль оператора *new* выполняется такой процедурой.

Главным свойством модели, основанной на объектах, есть ненужность понятий, основанных на классах, которые можно заменять более простыми понятиями. И, более того, эти простые понятия можно использовать более гибкими способами чем при строгом отношении к классам.

Предметный указатель

атрибут, 3
атрибутов., 4
динамическая диспетчеризация, 11
динамической диспетчеризации., 29
динамическую диспетчеризацию, 19
хранилища методов, 8
хранилище методов, 5
категоризации,, 23
категоризации., 14, 18, 25
категоризации:, 19
категоризация, 10, 12
конкретизации (уточнении), 23
конкретизация (уточнение) метода, 15
конкретизируем , 26
множественное наследование, 8
надкласс, 7
наследование, 7
наследования,, 6
необходимость типизации, 6
непротиворечивость типизации, 6
образование подкласса, 8
образования подклассов, 19
отношение подкласс, 7
отношение подпротокола, 27
подкласс, 7
правилу включения, 16
правилу включения., 16
преобразование типа, 12
протокол объектов, 20
статическая диспетчеризация, 11
структурное выделение типа, 21
связанная параметризация высшего порядка, 28
выбор метода, 5, 8
выбора метода, 9
выделение типа, 11
выделение типа,, 21
выделением типа,, 23

бинарный метод, 26
конкретизироваться., 18
оператор, 23
отношение совпадения, 28
связанная параметризация типа, 23
связанный абстрактный тип, 24
bounded abstract types, 24
bounded type parametrization, 23
F-bounded parametrization, 28
higher - order bounded parametrization, 28
matching relation, 28
method lookup, 5
method specialization , 15
operator, 23
structural subtyping, 21
subtyping, 11

binary method, 26

dynamic dispatch, 11

F-связанная параметризация, 28

inheritance, 7

method suites , 5
multiple inheritance, 8

object protocol , 20

protocol relation, 27

static dispatch, 11
subclass, 7
subclass relation, 7
subclassing , 8
subsumption, 10

superclass, 7

typecast, 12
typing soundess, 6

ССЫЛКИ

[1] Cardelli Abadi. On subtyping and matching, 1994. In Proceeding of the Europian Conference on Object - Oriented Programming. Lecture Notes in Computer Science 952, 145-167. Springer-Verlag.

[2] Cardelli Abadi. A theory of objects, 1996. Springer, http://books.google.com.ua/books?id=4xT3LgCPP5UC.

[3] Ghelli Albano and Orsini. Galileo: A strongly typed, interactive conceptual language, 1985. ACM Transactions on Database Systems 10(2), 230-360.

[4] America. A behavioural approach to subtyping in object - oriented programming languages, 1989. Philips Research Journal 44(2-3), 365-383.

[5] Browne. Eiffel: Frequency asked questions, 1994. <comp.lang.eiffel> newsgroup.

[6] Bruce. A paradigmatic object - oriented: programming language: Design, static typing and semantics, 1994. Journal of Functional Programming 4(2); 127-206.

[7] Castsgna The Hopkings Objects Group Leavens Bruce, Cardelli and Pierce.

[8] Hill Olthoff Canning, Cook and Mitchel. F-bounded polymorphism for object - oriented programming, 1989. In Proceeding on Functional Programming and Computer Architecture, 273-280.

[9] Catanga. Cavariance and contravariance: Conflict without a cause, 1995. ACM Transactions on Programming Languages and Systems 17(3), 431-447.

[10] Cook. A proposal for making eiffel type-safe, 1989. In Proceeding of the European Conference of Object-Oriented Programming, 57-72.

[11] Hill Cook and Canning. Inheritance is mot subtyping, 1990. In Proceccing of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, 125-135.

[12] Luckham Katiyar and Mitchell. Polymorphism and subtyping in interfaces, 1994. ACM SIGPLAN Notice 29(8), 22-34.

[13] Magnusson Madsen and M0ller-Pedersen. Strong typing of object-oriented languages revisited. Proceeding of the ACM Conference on Object-Oriented Programming Systems, etc, 141-160.

[14] Meyer. Object - oriented software construction, 1988. Prentice Hall.

[15] Nelson. Systems programming with modula-3, 1991. Prentice Hall.

[16] Snyder. Inheritance and the development of encapsulated software systems, 1987. In Research directions in object - oriented programming, 165-188. MIT Press.