Luca Cardelli: СЕМАНТИКА МНОЖЕСТВЕННОГО НАСЛЕДОВАНИЯ

перевод: ПИСКУНОВ А.Г.

16 апреля 2008 г.

АННОТАЦИЯ

Перевод статьи Luca Cardelli A Semantics of Multiple Inheritance в журнале Information and Computation 76, 138-164, 1988.

Большие части текста со сложными формулами вставлены в документ в графическом виде. Особых усилий по масштабированию размеров не делалось. Символ $\mathbb{E}$ в графике из оригинальной статьи соответствует символу $\mathbb{E}$, символ $\mathbb{D}$ - $\mathbb{D}$, символ $\mathbb{T}$ - $\mathbb{T}$. Вместо квадратных скобок с 'пустотой' используются $\dot{[}$ и $\dot{]}$.

Вопросы алгебраического проектирования классов рассматривались в работе [20] . Альтернативный подход к наследованию, при котором множество обьектов класса потомка не включается во множество объектов класса родителя изложен в [19] . Этот подход обьединяет математический подход к структурованию данных с таксонометрическими (обьектно - ориентированными) возможностями по созданию иерархии классов.

Содержание

1 ВВЕДЕНИЕ

Существуют, по крайней мере, два пути структурирования данных в языках программирования. Первый и общепринятый путь, который используется, например, в Pascal-е, можно сказать берет свое начало из математики. Данные организованы как прямые произведения (т.е. записи), непересекающиеся множества и пространства функций (т.е. процедуры и функции).

Второй - взят из биологии или таксономии. Данные организованы в иерархию классов и подклассов, причем данные любого уровня иерархии наследуют все атрибуты данных, расположенных выше в иерархии. Верхний уровень иерархии обычно называется классом всех объектов; каждое данное является объектом и каждое данное наследует базовые свойства верхнего класса, например, есть возможность сказать, являются ли два экземпляра некоторого класса одним и тем же объектом или нет. Процедуры и функции рассматриваются как локальные действия объектов, в противоположность глобальным операциям над объектами.

Эти разные пути структурирования данных порождают различные классы языков программирования и влекут к различным стилям программирования. Программирование с таксонометрически организованными данными часто называется объектно - ориентированным программированием, и может пропагандироваться как эффективный путь структурирования программ, базовых типов и, вообще, больших систем.

Понятия наследования и объектно - ориентированного программирования впервые появилось в Simula 67 ([1]). В Simula объекты сгруппированы в классы, а классы организованны в иерархию классов. Объекты похожи на записи, которые имеют функции в качестве компонентов. Экземплярам подкласса позволено появляться везде, где могли появляться экземпляры соответствующего надкласса. В Simula вопрос немного усложнялся изза того, что объекты использовались как сопрограммы, таким образом, что обмен между ними мог быть реализован как передача сообщений между процессами.

В Smalltalk ([2]) была принята и использована идея наследования с некоторыми изменениями. Несмотря на подчеркивание парадигмы передачи сообщений, объект языка Smalltalk обычно не является отдельным процессом. Передача сообщений реализована через вызовы функций, хо-

тя связь между именами сообщений и функциями (называемых методами) не очевидна. Подобно Simula, Smalltalk использует статические области видимости, добиваясь удобства интерактивного использования, и строгую проверку типов, позволяя вводить самодиагностику системы и понятие мета-классов.

Наследование может быть простым или множественным. В случае простого наследования, как в Simula или Smalltalk, иерархия подклассов имеет форму дерева, то есть каждый класс имеет не более одного родителя (superclass). Но иногда класс может рассматриваться как подкласс двух несовместимых родителей; что вынуждает к произвольному выбору одного из суперклассов в качестве родителя. Эта проблема естественным образом приводит к идее множественного наследования.

Множественное наследование встречается когда объект может принадлежить нескольким несовместимым суперклассам: это приводит к тому, что иерархия классов не ограничиватся формой дерева, а может образовывать сеть. Множественное наследование является идеей более элегантной, чем идея простого наследования, однако более трудной в реализации. До настоящего времени, множественное наследования обычно рассматривается в контексте безтипных языков с динамической областью видимости и реализованых как расширения языков Lisp или Smalltalk ([3] , [4] , [5] , [6]), или как подмножества языков представления знаний (knowledge representation langauges) [7] . Такими расширениями Smalltalk-а или Lisp-а с множественным наследованием являются, например, Galileo [8] или OBJ [9] .

Определение того, что делает язык объектно ориентированным остается спорным. Исследование разницы между Simula, Smalltalk и другими языками подсказывают мысль, что единственной критически важной идеей, ассоциирующейся с объектно-ориентированным программированием, является наследование. Сопрограммы, передача сообщений, статическая / динамическая область видимости, провека типов, простые / множественные родительские классы - все эти явно независимые характеристики могут присутствовать или отсутствовать в языке, который считается объектно - ориентированным. Поэтому теория объектно-ориентированного программирования должна прежде всего сосредоточится на значении термина наследование.

Целью работы является представление ясной семантики множествен-

ного наследования, демонстрация ее в контексте типизированных (strongly-typed) языков со статической областью видимости и доказательство существования алгоритма проверки типа (a sound typechecking algorithm exists). Множественное наследование интерпретируется в широком смысле: вместо того, чтобы ограничиваться объектами, оно расширяется естественным образом чтобы включать типы объединения и типы функций высшего порядка. Это конструирует семантический базис для объединения функционального и объектно - ориентированного программирования.

Преимуществом ясной семантики является то, что она дает возможность отличать фундаментальные вопросы от вопросов реализации или оптимизации.

Предлагаемая реализация множественного наследования является очень наивной, однако не препятствует более изощренным техникам реализации. Надо отдавать себе отчет, что улучшение техники реализации является абсолютно необходимым для получения реальных систем, основанных на наследовании (см. [10]).

Первая часть статьи неформальная, ее примеры представляет базовые понятия и ожидания от объектно - ориентированного программирования. Вторая часть формальная: в ней вводится язык, семантика, система построения типов и алгоритм проверки типов. Доказана правильность алгоритма в рамках системы вывода типов и правильность системы вывода в рамках семантики ([11]).

2 ОБЪЕКТЫ КАК ЗАПИСИ

Существует несколько возможностей представлять чем являются объекты. С точки зрения чистого Smalltalk-а объектами называтся физические сущности, вроде ящиков или машин. К несчастью, физические сущности достаточно бесполезны как семантические модели объектов, вследствие того, что они слишком сложны для формального описания.

Две простые интерпретации объектов, похоже, берут свое начало из различных реализаций объектно ориентированных языков. Первая интерпретация происходит из Simula, в которой объекты, по существу, являются записями с, возможно, функциональными компонентами. Передача сообщения выполняется выбором поля (компонентов функциональ-

ной записи) и наследование имеет дело с количеством и типом полей, принадлежащих записи.

Вторая интерпретация происходит из Lisp-а. Объект является функцией, которая получает сообщение (строчку или атом) и выбирает метод, подходящий для этого сообщения. Тут передача сообщения достигается применением функции, а наследование имеет дело с путем, которым передаются сообщения.

В некотором смысле, эти две интерпретации являются эквивалентными так как записи могут представляться как функции из меток (сообщений) в значения. Однако, утверждение что объекты являются функциями приводит к путанице и не пониманию, так как мы должны трактовать объекты как функции над сообщениями. Вместо этого мы можем уверенно полагать, что объекты являются записями, так как метки являются существенной частью записей.

Так же мы будем трактовать объекты как записи для целей проверки типов. В то время как (строка символов) сообщение может быть результатом произвольного вычисления, выбор из записи (record selection) обычно требует, что выбираемые метки (поля) записи должны быть известны в момент компиляции. В последнем случае возможно статически определить множество сообщений, поддерживаемых объектом и становится возможной диагностика ошибок типа во время компиляции при попытках посылать недопустимые сообщения. Это свойство поддерживается в Simula, но утеряно во всех производных языках (but has been lost in all the succeeding languages).

От переводчика

> Фраза 'сообщение x, поддерживаемое объектом класса y' трактуется как существование в классе y метода x.

Мы покажем, что парадигму объекты-как-записи можно использовать для всех основных характеристик объектов, при условии, что соответствующий язык является достаточно богатым. Будут рассмотрены следующие характеристики: множественное наследование, передача сообщений, приватные экземпляры переменных и концепция "self". Однако, дуализм между записями и функциями остается: в нашем языке объекты являются записями, однако семантика интерпретирует записи как функции.

3 ЗАПИСИ

Записью будет называться связывание конечного множества значений с метками, например:

$\{a = 3, b = true, c = "abc"\}$

Это запись с тремя полями: a, b, c хранящими значения целого 3, логического $true$ и строчки $"abc"$ соответственно. Метки a, b, c принадлежат к различным доменам меток; они не идентификаторы, не строки, и не могут быть вычислены как результат выражения. Будет считаться, что записи неупорядоченные и не могут содержать одну и ту же метку дважды.

Базовая операция на записях есть выбор, выраженный через обычную нотацию с точкой:

$\{a = 3, b = true, c = "abc"\}.a \ \equiv \ 3$

Выражение может иметь один или более типов; мы пишем
e:τ что бы показать, что выражение e имеет тип τ.

Записи имет тип записи, который является множеством помеченных различными метками типов. Например:

$\{a = 3, b = true\} : \{a : int, b : bool\}$

В общем случае, мы можем записать следующее неформальное правило для типов записи:

$$[Rule1] \quad if \ e_1 : \tau_1 \ and \ \ ... \ \ and \ e_n : \tau_n \ then$$
$$\{a_1 = e_1, \ ..., a_n = e_n\} : \{a_1 : \tau_1, \ ..., a_n : \tau_n\}$$

Это первое из серии неформальных правил, которые должны выразить наши начальные интуитивные представления о типизации. Не предполагается, что они образуют законченное множество правил или будут независимы одно от другого.

Существует отношение подтипа на типах записей, которое соответствует отношения подкласса языков Simula и Smalltalk. Например, мы можем определить следующие типы (определению типов предшествует ключевое слово type):

$$
\begin{array}{lcl}
type\ any & = & \{\} \\
type\ object & = & \{age : int\} \\
type\ vehicle & = & \{age : int, speed : int\} \\
type\ machine & = & \{age : int, fuel : string\} \\
type\ car & = & \{age : int, speed : int, fuel : string\}
\end{array}
$$

Интуитивно ясно, что запись типа *vehicle* является записью типа object, запись *machine* является записью object и запись *car* есть записью *vehicle* и записью *machine* (и, тем более, записью object). Можно говорить, что тип *car* есть подтипом типов *machine* и *vehicle*; тип *machine* является подтипом object; и так далее. Вообще говоря, тип записи τ является подтипом (записывается $\leq$) типа записи $\tau^{'}$ если τ имеет все поля $\tau^{'}$, и, возможно, еще некоторые, кроме того, все общие поля τ и $\tau^{'}$ тоже находятся в отношении подтипа $\leq$. Все базовые типы, такие как (*int* и *bool*) являются подтипом самих себя (где ι базовый тип):

[*Rule*2]
- $\iota \ \leq \iota$ (ι *a basic type*)
- $\tau_1 \leq \tau_1^{'},\ \ldots\ , \tau_n \leq \tau_n^{'} \Rightarrow$
  $\{a_1 : \tau_1, ..., a_{n+m} : \tau_{n+m}\} \leq \{a_1 : \tau_1^{'}, ..., a_n : \tau_n^{'}\}$

Рассмотрим конкретный автомобиль (*car*) (определению значений предшествует ключевое слово value):

$$value\ mycar = \{age = 4, speed = 140, fuel = "gasoline"\}$$

Конечно, *mycar* имеет тип *car* (*mycar*: *car*), однако мы можем также захотеть считать *mycar*: object. Чтобы иметь такую возможность мы должны потребовать, что если значение имеет тип τ , то оно имеет имеет также все типы $\tau^{'}$, для которых τ является подтипом $\tau^{'}$. Таким образом получаем третье неформальное правило:

$$[Rule3] \quad if\ a : \tau\ \ and\ \tau\ \leq \tau^{'}\ \ then\ a : \tau^{'}$$

Определив следующую функцию:

$$value\ age(x : object) : int = x.age$$

мы сможем вычислять значение $age(mycar)$, так как, согласно пра-

вилу [Rule3], *mycar* имеет тип требуемый функций *age*. На деле *mycar* принадлежит типам *car*, *vehicle*, *machine*, object, типу пустой записи и многим другим.

Когда значение функции определено на заданном аргументе? Это задается следующими правилами:

[*Rule*4] $if\ f : \sigma \rightarrow \tau\ and\ a : \sigma\ then\ f(a)$ определено и $f(a) : \tau$
[*Rule*5] $if\ f : \sigma \rightarrow \tau\ and\ a : \sigma^{'}, where\ \sigma^{'} \leq \sigma\ then\ f(a)$ определено и $f(a) : \tau$

Правило [Rule5] является по сути следствием правила [Rule3] и правила [Rule4]. Из [Rule3] и $a : \sigma^{'}$ мы получаем $a : \sigma$; и, значит, вычисление f(a) как функции f: $\sigma \rightarrow \tau$ определено.

Общепринятое отношение подкласса определяется только на объектах или классах. Наше отношения подтипа естественным образом расширяется на типы функций. Рассмотрим функцию

$$serial\_number : int \rightarrow car$$

Можно считать, что *serial_number* возвращает значения типа *vehicle*, так как каждая величина типа *car* является величиной типа *vehicle*. Вообще говоря, все функции, возвращающие тип *car*, также являются функциями, возвращающими тип *vehicle*. То есть, для любой области определения t мы можем утверждать, что тип t $\rightarrow$ *car* (множество функций из t в *car*) есть подмножеством функций типа t $\rightarrow$ *vehicle*, то есть, подтипом:

$t \rightarrow car \leq t \rightarrow vehicle$, так как $car \leq vehicle$

Далее, рассмотрим функцию:

$$speed : vehicle \rightarrow int$$

Так как все величины типа *car* являются величинами типа *vehicle*, мы можем использовать эту функцию для вычисления скорости величины типа *car*, потому что *speed* является также функций из *car* в *int*. Вообще говоря, каждая функция, заданная на множестве *vehicle* является функцией, заданной на множестве *car* и, поэтому, можно говорить что тип $vehicle \rightarrow int$ является подтипом типа $car \rightarrow int$:

$vehicle \ \rightarrow \ t \leq car \ \rightarrow \ t$ так как $car \leq vehicle$

Тут произошло нечто интересное: заметьте что отношение подтипа изменило порядок. Почему это произошло мы объясним позже, после формального определения смысла оператора $\rightarrow$ в следующих секциях. (Семантически, мы работаем в универсальном домене V всех вычислимых величин. Каждая функция f является функцией из V в V, что записывается как f : $V \rightarrow V$, где $\rightarrow$ есть привычным пространством непрерывных функций. Задавая f : $\sigma \ \rightarrow \ \tau$ мы определяем функцию f : $V \rightarrow V$, которая каждому элементу из множества $\sigma \ \subseteq V$, ставит в соответствие элемент множества $\tau \ \subseteq V$; ничего не предполагается про поведение f на дополнении к множеству σ).

Пусть дана функция f: $\sigma \ \rightarrow \tau$ из некоторого домена σ в некоторый кодомен τ , мы всегда можем рассматривать ее как функцию из меньшего домена $\sigma' \ \subseteq \sigma$ в больший кодомен $\tau' \ \supseteq \tau$. Например, функция f: $vehicle \rightarrow vehicle$ может использоваться в контексте age(f($mycar$)), где она использована как функция f: $car \ \rightarrow \ object$(применение f($mycar$) имеет смысл, так как каждая величина из множества car принадлежит множеству $vehicle$; v = f($mycar$) принадлежит $vehicle$; далее, возможно применять функции $age(v)$ так как каждый элемент множества $vehicle$ принадлежит множеству object).

Общее правило подтипа на типах функций может быть записано в виде:

$[Rule6] \quad if \ \sigma' \ \leq \sigma \ \ and \ \tau \ \leq \tau' \ then \ \sigma \ \rightarrow \ \tau \ \leq \sigma' \ \rightarrow \ \tau'$

Как уже сказано, отношения подтипа можно расширять на типы высшего порядка. Например, пусть дано определение функции $mycar$r_attribute которая любую целочисленную функцию, заданную на множества car, применяет к величине $mycar$:

$value \ mycar\_attribute(f : car \ \rightarrow \ int) : int = f(mycar)$

Мы можем применить ее к функции любого типа, которая является подтипом $car \ \rightarrow \ int$, например, $age : object \ \rightarrow \ int$. (Почему? Потому что car есть подтипом типа object, значит $object \ \rightarrow \ int$ есть подти-

пом $car \rightarrow int$ по правилу [Rule6], значит выражение $(mycar\_attribute : (car \rightarrow int) \rightarrow int)(age$: $object$-$> int)$ имеет смысл по правилу [Rule5]).

$$mycar\_attribute(age) \equiv 4$$
$$mycar\_attribute(speed) \equiv 140$$

До сих пор мы назначали определенным типам определенные значения. Однако отношение подтипа имеет сильный вкус включения типов, рассматриваемых как множества объектов, и мы хотим убедиться в справедливости наших суждений на строгой семантической основе.

Семантически бы можем рассматривать тип *vehicle* как множество всех записей с полем *age* и полем *speed*, имеющих соответствующие типы, однако тогда записи из множества *car* не будут принадлежать ко множеству *vehicle* так как они имеют три поля, в то время как записи из *vehicle* имеют только два. Чтобы получить включение, которое мы интуитивно ожидаем, мы должны сказать, что тип *vehicle* является множеством всех записей которые имеют по, крайней мере, два поля, как отмечено выше, и, также, могут иметь еще дополнительные поля. В этом смысле запись типа *car* является записью типа *vehicle*, а множество всех записей *car* включено во множество всех записей *vehicle*, в соответствие с нашими ожиданиями. Однако для определения этих "множеств"требуются дополнительные усилия. И это будет формально сделано в следующих секциях.

Мы закончим эту секцию прагматическими рассуждениями о нотации записей. Типы записей могут иметь большое число полей, поэтому необходима возможность быстрого определения подтипа некоторого типа записи, без повторного перечисления всех полей записи. Следующие три множества определений считаются эквивалентными:

$$
\begin{array}{lcl}
type\ object & = & \{age : int\} \\
type\ vehicle & = & \{age : int, speed : int\} \\
type\ machine & = & \{age : int, fuel : string\} \\
type\ car & = & \{age : int, speed : int, fuel : string\}
\end{array}
$$

$$
\begin{array}{lcl}
type\ object & = & \{age : int\} \\
type\ vehicle & = & object\ and\ \{speed : int\} \\
type\ machine & = & object\ and\ \{fuel : string\} \\
type\ car & = & vehicle\ and\ machine
\end{array}
$$

$$
\begin{array}{lcl}
type\ object & = & \{age : int\} \\
type\ car & = & object\ and\ \{speed : int, fuel : string\} \\
type\ vehicle & = & car\ ignoring\ fuel \\
type\ machine & = & car\ ignoring\ speed
\end{array}
$$

Оператор *and* образует объединение полей двух типов записей; если два типа записей имеют несколько общие метки (подобно *vehicle* и *machine*), то типы общих меток должны быть одинаковыми. В этот момент мы не будем точо формулировать что означает термин 'одинаковыми' , кроме того, что в приведенном примере 'одинаковыми' означает 'быть теми же'. И, в более общем случае, соответствовать операции И на выражениях типа, как будет объяснено в последующих секциях.

Оператор *ignoring* просто устраняет компоненты из типа записи. Оба оператора *and* и *ignoring* неопределены на типах, отличных от типов записей.

4 ВАРИАНТЫ

В денотационной семантике (denotational semantics) известны две базовые констукции для нефункциональных типов данных: прямое произведение и объединение (disjoint sums). Мы убедились, что наследование может быть выражено как отношение подтипа на типах записей, с возможностью расширения на высшие типы (типы функций). Типы записей - это просто помеченные метками (labeled) прямые произведения и, по аналогии, мы можем задаться вопросом возможны ли подобные способы

записи, выводимые из помеченных метками объединений (labeled disjoint sums).

Помеченное объединение называется вариантом. Тип вариант выглядит очень похоже на тип записи: это тоже неупорядоченное множество пар из типов и меток, заключенных в квадратные скобки:

$$type\ int\_or\_bool = [a : int, b : bool]$$

Элементом типа вариант является помеченное значение, причем меткой может быть одна из меток в типе вариант и тип значения должен совпадать с типом метки. Элементом int_or_bool может быть либо целое значение, помеченное меткой a, либо логическое, помеченное меткой b.

$$\begin{array}{lcl} value\ an\_int & = & [a = 3] : int\_or\_bool \\ value\ a\_bool & = & [b = true] : int\_or\_bool \end{array}$$

Базовыми операциями на вариантах будут is, которая проверяет имеет ли объект типа вариант заданную метку, и as, которая извлекает содержимое объекта типа вариант имеющего заданную метку:

$$\begin{array}{llll} an\_int & is & a & \equiv\ true \\ an\_int & is & b & \equiv\ false \\ an\_int & as & a & \equiv\ 3 \\ an\_int & as & b\ does\ not\ have\ a\ value & \end{array}$$

Вариант типа σ будет подтипом варианта τ (записывается $\sigma \leq \tau$) если τ имеет все те же метки что и σ , причем их типы совпадают. То есть, int_or_bool является подтипом $[a : int, b : bool, c : string]$.

В случае если типом ассоциированным с меткой есть тип $unit$ (тривиальный тип, с единственным элементом величина $unity$), мы будем опускать тип; вариант, у которого все поля имеют тип $unit$ будет так же называться перечислением. Типы перечисления приводятся в следующем примере:

$$\begin{array}{lcll} type\ precious\_metal & = & [gold, silver] & (\text{т.е.}[gold : unit, silver : unit]) \\ type\ metal & = & [gold, silver, steel] & \end{array}$$

Величину типа перечисление можно сокращать похожим образом, опуская часть '$= unity$', то есть, писать не $[gold = unity]$, а просто $[gold]$.

Будем считать, что функция, возвращающая тип $precious\_metal$, является функцией, возвращающей тип $metal$.

$t \rightarrow precious\_metal \leq t \rightarrow metal$ так как $precious\_metal \leq metal$

а функция, заданная на типе $metal$, также задана на типе $precious\_metal$

$metal \rightarrow t \leq precious\_metal \rightarrow t$ так как $precious\_metal \leq metal$

Очевидно, что правило [Rule6] остается неизменным для вариантов. Это дает основание для использования символа $\leq$ для взятия подтипов у обоих типов: у записей и у вариантов. Семантически отношение подтипа на вариантах отображается во включение подмножеств, точно так же как в случае записей: $metal$ является множество с тремя определенными элементами $[gold], [silver], [steel]$ и $precious\_metal$ является множеством из двух элементов $[gold], [silver]$.

Существуют два пути для вывода типов вариантов из других, ранее определенных. Мы можем определить $metal$ и $precious\_metal$:

$$\begin{aligned} &type\ precious\_metal &&= [gold, silver] \\ &type\ metal &&= precious\_metal\ or\ [steel] \end{aligned}$$

или

$$\begin{aligned} &type\ metal &&= [gold, silver, steel] \\ &type\ precious\_metal &&= metal\ dropping\ steel \end{aligned}$$

Оператор *or* выполняет объединение двух типов вариантов, оператор *dropping* удаляет значение из типа варианта.

От переводчика

Не совсем понятно, почему в одном случае *steel*, а в другом $[steel]$. Как видно ранее из обсуждение величины $[gold]$, где эта запись эквивалетна записи $[gold = unity]$, ее тоже надо брать в квадратные скобки. На всякий случай править не стал. *dropping* можно было бы рассматривать как разность множеств.

5 ИДИОМЫ НАСЛЕДОВАНИЯ

В наших, уже описанных конструкциях (framework), появляется возможность различить некоторые черты того, что обычно называется множественным наследованием между объектами. Например, экземпляр типа *car* имеет (наследует) все атрибуты экземпляра типа *vehicle* и все атрибуты экземпляра типа *machine*. Некоторые аспекты однако остаются необычными; например отношение наследования зависит только от структуры объектов и может не объявляться явно.

Далее, в этой секции наш подход будет сравниваться с другими подходами к наследованию, далее в ней будет показано как промоделировать некоторые известные техники наследования. Однако мы не пытаемся подробно объяснить существующие схемы наследования (например, Smalltalk-a), но, скорее, пытаемся представить новые перспективы вопроса.

Some differences between this and other inheritance result in net gains.

От переводчика

> Совершенно не представляю, что это предложение значит. Похоже на: 'Существуют некоторые отличия между этим (в смысле, нашим) и другими наследованиями, которые дают заметный выигрыш'.

Например, нам не известны языки в которых проверка типов сосуществует со множественным наследованием и функциями высшего порядка. Исключением является Galileo [8] и Amber [12] которые были разработаны, собственно, для этого исследования.

От переводчика

> в смысле для текущей статьи или для какой - то отдельного исследования сосуществования множественного наследования с проверкой типов?

Проверка типов предоставляет защиту против очевидных ошибок времени компиляции (подобных применению функции *speed* к объекту *machine*, который не есть объектом *vehicle*) и других, менее очевидных ошибок. Ведь можно построить сложную иерархию типов, в которой "everything *is* also something else"(что угодно может оказаться также чем то другим), что будет затруднять запоминание свойств объектов.

Замечание

> В предложении 'It can be difficult to remember which objects support which messages', сообщения трактуются как свойства объекта. фраза 'объект поддерживает сообщение *speed*' может трактоваться как 'объект имеет метод - член класса (или функцию) *speed*' (согласно только что приведенного примера применения функции *speed* к объекту типа *machine*).

Отношение подтипа выполняется на типах и не существует похожего отношения на объектах. То есть, мы не можем непосредственно промоделировать отношение "подобъекта которое используется, например, Omega-ой [7] , в котором мы можем определить класс *gasoline car*, как машины ("*car*") у которой атрибут $fuel$ имеет значение "*gasoline*".

Однако, в простых случаях можно добиться того же эффекта, давая некоторым атрибутам класса тип вариант. Например, вместо того, чтобы задавать поле $fuel$ строкой, можно определить:

$$\begin{aligned} type\ fueltype &= [coal, gasoline, electricity] \\ type\ machine &= \{age : int, fuel : fueltype\} \\ type\ car &= \{age : int, speed : int, fuel : fueltype\} \end{aligned}$$

Тогда можно задать

$$\begin{aligned} type\ gasoline\_car &= \{age : int, speed : int, fuel : [gasoline]\} \\ type\ gasoline\_car &= \{age : int, speed : int, fuel : [gasoline, coal]\} \end{aligned}$$

что дает $gasoline\_car \leq combustion\_car \leq car$. То есть, функция, определенная на множестве $combustion\_car$, может быть применена к объектам из класса $gasoline\_car$, а попытка применить ее к обьекту, у которой поле $fuel$ имеет значение $[electricity]$ будет приводить к ошибке времени компиляции.

Часто у функции, содержащейся в некотором поле записи, имеется необходимость в доступе к другим компонентам той же записи. В Smalltalk это обеспечивается передачей в функцию ссылки на всю запись (т.е. объект) как на $self$ и, затем, выбирается желаемая компонента. В Simula применяется похожая идея с ключевым словом *this*.

Возможность ссылаться на самого себя может быть получена как специальный случай оператора *rec*, который мы сейчас введем. Оператор *rec* используется для определения рекурсивных функций и данных. Например, рекурсивная функция для вычисления факториала может быть записана как:

$$rec\ fact : int \to int.\lambda n : int.\ if\ n = 0\ then\ 1\ else\ n * fact(n-1)$$

(Это не декларация, а выражение).

Тело оператора *rec* будет считаться конструктором (The body of rec ≡ restricted to be a constructor); этот неясный термин означает то, что при реализации вычисления могут быть временно прерваны чтобы избежать возможного зацикливания [13] . В ранее обсужденных языках, конструктор может быть либо константой, либо записью, либо вариантом, либо функцией, либо выражением *rec*, для которого выполняются эти ограничения.

Случаи рекурсивного определения данных (circular data definition) исключительно часто встречаются в объектно - ориентированном программировании. В следующем примере функциональная компонента записи ссылается на другие компоненты этой же записи. Функциональная компонента *d* вычисляет расстояние активной точки в записи *this* от заданной точки.

```
type point =
    {x : real, y : real}
type active_point =
    point and{d : point → real}
value make_active_point(px : real, py : real) : active_point =
    rec self : active_point.
        {x = px, y = py, d = λ p : point.
        sqrt((p.x − self.x) ∗ ∗2 + (p.y − self.y) ∗ ∗2)}
```

Объекты часто обладают переменными с атрибутом private, которые очень полезны для поддержания и обновления локального состояния объекта, и в то же время, предотвращают случайное внешнее взаимодействие. Далее, приводится пример объекта *counter*, которому присваивается начальное заданное число с возможностью увеличения значения

на один шаг за одно обращение. Ячейка n является обновляемой ячейкой, чье начальное содержимое равно n; ячейка может быть обновлена при помощи оператора присвоить := и ее содержимое может быть извлечено функцией *get* (побочные эффекты не будут обсуждаться в формальной семантике). Далее, $\lambda().e$ является сокращением для $\lambda x : unit.e$, где x не встречается в e и выражение $let\ x = a\ in\ b$ вводит новую локальную переменную x (инициализированную значением a) с областью видимости b, возвращая значение этой переменной.

$$
\begin{array}{l}
type\ counter = \{increment : unit\ \rightarrow\ unit, fetch : unit \rightarrow unit\} \\
value\ make\_counter(n : int) : counter = \\
\quad let\ count = cell\ n \\
\quad in\{increment = \lambda().count := (get\ count) + 1, \\
\qquad fetch = \lambda().get\ count\}
\end{array}
$$

Приведенная выше техника использования статической области видимости позволила получить (промоделировать) в полной общности переменные с атбируботм *private* (в источнике: Private variables are obtained in full generality by the above well known static scoping technique).

Наличие сторонних эффектов может быть полезно для последовательного выполнения операциий над объектами. Например, мы может захотеть определить счетчик другого типа, который можно будет использовать следующим образом (f() это аббревиатура для f(unity)):
$make\_counter(0).increment().increment().fetch()\ \equiv\ 2$
В этом случае, метод из записи должен возвращать свою запись, которой он принадлежит. Для этого потребуются как рекурсивно определенные объекты, так и рекурсивно заданные типы:

$$
\begin{array}{l}
type\ counter = \{increment : unit\ \rightarrow\ counter, fetch : unit \rightarrow unit\} \\
value\ make\_counter(n : int) : counter = \\
\quad let\ count = cell\ n \\
\quad inrecself : counter.\quad \{increment = \lambda().count := (get\ count) + 1; self \\
\qquad fetch = \lambda().get\ count\}
\end{array}
$$

где символ ';' разделяет последовательность операций. (Рекурсивные типы не будут более обсуждаться в формальной семантике; нам кажется случай рекурсивных типов можно бы охватить, но возникшие трудности

отвлекали бы нас от темы этой статьи).

В терминологии Smalltalk-a, подкласс автоматически наследует методы всех его надклассов. Подкласс может переопределять наследованные методы. В любом случае, все объекты как конкретного класса, так подкласса будут иметь те же самые методы. В следующем примере, в котором класс $Class\_A$ имеет методы f и g; функция $make\_A$ создает объекты класса $Class\_A$ формируя записи с компонентами f и g.

$$
\begin{array}{lcl}
type\ Class\_A & = & \{f : X \to X`,\ g : Y \to Y`\} \\
value\ fOfA(a : X) : X` & = & ... \\
value\ gOfA(a : Y) : Y` & = & ... \\
value\ make\_A() : Class\_A & = & \{f = fOfA,\ g = gOfA\}
\end{array}
$$

Далее, мы определим подкласс $A\_Subclass\_B$ класса $Class\_A$, с дополнительным методом методом h. Функция $make\_B$ создает объекты подкласса из

- явно наследованной компоненты f;
- новой компонеты g, изменяя при этом наследованный метод;
- совершенно новой компонеты h.

$$
\begin{array}{lcl}
type\ A\_Subclass\_B & = & Class\_A\ and\ \{h : Z \to Z`\} \\
value\ gOfB(a : Y) : Y` & = & ... \\
value\ hOfB(a : Z) : Z` & = & ... \\
value\ make\_A() : A\_Subclass\_B & = & \{f = fOfA,\ g = gOfB,\ h = hOfB\}
\end{array}
$$

В противоположность к Simula и Smalltalk, ничто не препятствует нам иметь абсолютно различные методы в различных объектнах одного класса, если эти методы имеют тип, предписанный классом.

И в Simula, и в Smalltalk объекты могут обращаться к методам родительских надклассов. В наших конструкциях (framework) эти возможности не могут быть проэмулированы общим способом, частично, вследствие использования множественного наследования.

6 АНОМАЛИИ ПРОВЕРКИ ТИПОВ

Проверка типов при наследовании, которое было нами представлено, имеет неожиданные аспекты. Они являются следствием отсутствия параметризованного полиморфизма и наличия побочных эффектов. Рассмотрим функцию тождественности (identity function) записей, имеющих целочисленную компоненту a:

$$\begin{array}{lcl} type\ A & = & \{a : int\} \\ value\ id(x : A) : A & = & x \end{array}$$

Возможно применять функцию id к подтипу B типа A, однако информация о типе окажется потерянной, так как результат будет иметь тип A, а не B. Например, следующее выражение не может быть проверено на правильность типа.:

$$(id(\{a = 3, b = true\})).b$$

Может это и не приводит к серьезным последствиям на практике, однако вынуждает программиста пользоваться менее полиморфный стиль, чем тому хотелось бы: в приведенном примере приходится писать различные функции тождественности для различных типов.

Следующей пример показывает, что полиморфизм наследования приводит к почти такому же эффекта как и параметрический полиморфизм:

$$\begin{array}{lcl} type\ anyList & = & rec\ list.[nil : unit,\ cons : \{rest : list\}] \\ type\ intList & = & rec\ list.[nil : unit,\ cons : \{first : int,\ rest : list\}] \\ type\ intPairList & = & rec\ list.[nil : unit,\ cons : \{first : int,\ second : int,\ rest : list\}] \end{array}$$

$$\begin{array}{lcl} value\ rest(I : anyList) : anyList & = & (I\ as\ cons).rest \\ value\ intFirst(I : intList) : int & = & (I\ as\ cons).first \\ value\ intSecond(I : intPairList) : int & = & (I\ as\ cons).second \end{array}$$

$$\begin{array}{lcl} value\ rec\ length(I : anyList) : int & = & \\ & & if\ \ I\ is\ nil\ then\ 0\ else\ (1 + length(rest\ I)) \end{array}$$

Здесь $intPairList$ является подтипом $intList$, который, в свою очередь,

является подтипом $anyList$. Оператор $rest$ может работать на любом из этих списков и можно использовать полиморфную функцию $length$. Но невозможно определить полиморфный оператор $first$. Функция $intFirst$, приведенная выше, работает на типах $intList$ и $IntPairList$, но $intSecond$ работает только на $intPairList$. Решение этих проблем предлагается в работе [14] , в которой обьединяются множественное наследование и параметрический полиморфизм.

Проверка типов для наследования должна быть ограничена, чтобы сохранять непротиворечивость при наличии побочных эффектов.

Замечание

Термин soundness трактуется как семантическая непротиворечивость как в работе [15] , корректность.

Параметрический полиморфизм тоже должен быть ограничен, так что бы иметь возможность обрабатывать побочные эффекты, но, похоже, природа проблема несколько отличается. Рассмотрим следующий пример (due to Antonio Albano), в котором мы предполагаем что поля записи позволяется обновлять оператором присваивания := (в предыдущих секциях использовался другой механизм):

$$
\begin{array}{l}
value\ f(r : \{a : \{\}\}) : unit = \\
\quad r.a := \{\} \\
value\ r = \\
\quad \{a = \{b = 3\}\} \\
f(r) \\
r.a.b
\end{array}
$$

Последнее выражение приведет к ошибке во время выполнения, так как значение поля a записи r было изменено в $\{\}$ после выхова функции f. Что бы избежать этого, достаточно синтаксически различать обновляемые и необновляемые поля записи и потребовать эквивалентность типа (вместо включения типа) во время проверки типов у обновляемых полей. Опять напомним, это обсуждение неформальное; побочные эффекты не будут обсуждаться далее в этой статье.

7 ВЫРАЖЕНИЯ

Далее, начнем формальное изложение множественного наследования. Сначала, мы определеним простой аппликативный язык с поддержкой наследования. Затем представим денотационную семантику в домене значений V. Некоторые подмножества V будут рассматриваться как типы, причем наследование непосредственно соответствует включению множеств. Так же будут представлены система выведения типов и алгоритм проверки типов. Корректность алгоритма будет доказана демонстрацией того, что алгоритм согласуется (не противоречит) с системой выведения типов и сама система выведения типов, в свою очередь, согласуется (не противоречит) с семантикой.

Наш язык является вариантом типизированного лямбда - исчисления с включением типов, рекурсией и типами данных, включающими записи и инварианты. Следующая нотация часто используется для записей (и, подобным образом, для типов записей и вариантов):

$$\{a_1 = e_1,\ \ldots\ , a_n = e_n\} \ \equiv\ \{a_i = e_i\}\ i \in 1..n$$

$$\{a_1 = e_1,\ \ldots\ , a_n = e_n, a_1^{'} = e_1^{'},\ \ldots\ , a_m^{'} = e_m^{'}\} \ \equiv$$
$$\{a_i = e_i, a_j^{'} = e_j^{'}\}\ \ i \in 1..n, j \in 1..m$$

Далее, приводим синтаксис выражений и выражений типов.

e	::=	*expressions*
	$x \mid$	*identifiers*
	$b \mid$	*constants*
	$if\ e\ then\ e\ else\ e \mid$	*conditionals*
	$\{a_j = e_j\} \mid e.a \mid (i \in 1..n, n \geq 0)$	*records*
	$[a = e] \mid e\ is\ a \mid e\ as\ a \mid$	*variants*
	$\lambda x : \tau.e \mid e\ e \mid$	*functions*
	$rec\ x : \tau.e$	*recursive data*
	$e : \tau \mid$	*type specs*
	(e)	

$$
\begin{array}{lll}
\tau & ::= & \textit{type expressions} \\
 & \iota \mid & \textit{type constants} \\
 & \{a_i : \tau_i\} \mid (i \in 1..n, n \geq 0) & \textit{record types} \\
 & [a_i : \tau_i] \mid (i \in 1..n, n \geq 0) & \textit{variant types} \\
 & \tau \rightarrow t \mid & \textit{function types} \\
 & (\tau) &
\end{array}
$$

где $i \neq j \Rightarrow a_i \neq a_j$
полагая $\iota_0 = unit, \iota_1 = bool, \iota_2 = int$, и так далее. Ограничения синтаксиса: тело выражения e из $rec\ x : \tau$ может быть только константой, записью, вариантом, лямбда выражением или другим выражением rec с этими ограничениями.

Метки a и идентификаторы x имеют одинаковый синтаксис, но отличаются контекстом. Среди типов (type constants) есть *unit* (домен с одним определенным элементом), *bool* и *int*. Среди констант мы имеем *unity* (типа *unit*), логические значения ($false$, $true$) и целые числа (0, 1, ...).

Вместо двух операций *is* и *as* для вариантов можно использовать одну конструкцию case. Первые две операции понятные и хорошо иллюстрируют семантику обработки исключительных состояний, хотя последняя более элегантена (одна конструкция вместо двух) и делает ненужной обработку исключительных состояний.

Введем следующие стандартные аббревиатуры:

$$
\begin{array}{lll}
let\ x : \tau = e\ in\ e^{'} & for & (\lambda x : \tau . e^{'}) e \\
f(x : \tau) : \tau^{'} = e & for & f : \tau \rightarrow \tau^{'} = \lambda\ x : \tau . (e : \tau^{'}) \\
rec\ f(x : \tau\ ) : \tau^{'} = e & for & f : \tau \rightarrow \tau^{'} = rec\ f : \tau \rightarrow \tau^{'} . \lambda x : \tau . e
\end{array}
$$

Выражения типов записей или вариантов считаются неупорядоченными, так что для любой перестановки π(n) для 1..n, считаются неразличимыми:

$$
\begin{array}{lcll}
\{a_i : \tau_i\} & \equiv & \{a_{\pi(n)(i)} : \tau_{\pi(n)(i)}\} & i \in 1..n \\
[a_i : \tau_i] & \equiv & [a_{\pi(n)(i)} : \tau_{\pi(n)(i)}] & i \in 1..n
\end{array}
$$

8 РАЗДЕЛ СЕМАНТИКИ

Семантика выражений дается в рекурсивно определенном множестве значений V. Ниже используются следующие операторы на множествах: (disjoint sum) сумма (+), прямое произведение ($\times$) и пространство непрерывных функций ($\rightarrow$).

$$V = B_0 + B_1 + \ ... \ + R + U + F + W$$

$$R = L \ \rightarrow \ V$$

$$U = L \times V$$

$$F = V \ \rightarrow \ V$$

$$W = \{ \ w \ \}$$

Множество L это множество символьных строк, которое мы называли метки, каждое из множеств B_i - один из базовых типов. Мы имеем:

$$B_0 \ \equiv \ O \ \equiv \ \{ \ \perp_O, \ unity \ \}$$

$$B_1 \ \equiv \ T \ \equiv \ \{ \ \perp_T, \ true, \ false \ \}$$

$$B_2 \ \equiv \ N \ \equiv \ \{ \ \perp_N, \ 0, \ 1, \ ... \ \}$$

b_{ij} является j-м элементом i-го базового домена B_i

W - домен, содержащий единственный элемент w, неправильное значение (the wrong value). Это значение w используется для моделирования ошибок времени исполнения (например, попытка использования целого значения в качестве функции) которые (согласно нашему желанию) компилятор должен будет диагностировать перед выполнением (trap before execution). Это не попытка моделировать исключительные ситуации времени выполнения (вроде попытки взятия первого элемента пустого списка); в нашем контексте такого рода ошибки могут быть сгенерированы только оператором *as*. Имя wrong используется чтобы обозначить w как элемент V (а не просто как элемента W). Исключительные состояния времени выполнения должны бы быть моделироваться дополнительным слагаемым V, но для простоты, мы, вместо этого, будем использовать неопределенный элемент V записываемый как $\perp_V$ (часто сокращаемый до $\perp$).
$R = L \ \rightarrow \ V$: множество записей, являющиеся ассоциациями значений с метками.

$U = L \times V$: множество вариантов, которые являются парами $< l, v >$ для некоторой метки l и значения v.
$F = V \rightarrow V$: множество непрерывных функций из V в V, используется для придания смысла лямбда выражениям (to give semantics to lambda expressions).

9 СЕМАНТИКА ВЫРАЖЕНИЙ

Функцию семантики обозначим через $\mathbb{E} \in Exp \rightarrow Env \rightarrow V$, где Exp синтаксическое выражение, составленное по правилам нашей грамматики, а $Env = Id \rightarrow V$ это окружение для идентификаторов. Семантика базовых значений определяется как $\mathbb{B} \in Exp \rightarrow$ V, и, поскольку она очевидна, ее определение опущено. Используя соглашения, мы определяем:

$$\mathbb{E}[\![x]\!]\eta = \eta[\![x]\!]$$
$$\mathbb{E}[\![b_{ij}]\!]\eta = \mathcal{B}[\![b_{ij}]\!]$$
$$\mathbb{E}[\![\text{if e then e' else e"}]\!]\eta =$$
$$\quad \text{if } \mathbb{E}[\![e]\!]\eta \varepsilon T \text{ then (if } (\mathbb{E}[\![e]\!]\eta \mid T) \text{ then } \mathbb{E}[\![e']\!]\eta \text{ else } \mathbb{E}[\![e"]\!]\eta) \text{ else wrong}$$
$$\mathbb{E}[\![\{a_1 = e_1, \ldots, a_n = e_n\}]\!]\eta =$$
$$\quad (\lambda b.\ \text{if } b=a_1 \text{ then } \mathbb{E}[\![e_1]\!]\eta \text{ else } \ldots \text{ if } b=a_n \text{ then } \mathbb{E}[\![e_n]\!]\eta \text{ else wrong}) \text{ in } V$$
$$\mathbb{E}[\![e.a]\!]\eta = \text{if } \mathbb{E}[\![e]\!]\eta \varepsilon R \text{ then } (\mathbb{E}[\![e]\!]\eta \mid R)(a) \text{ else wrong}$$
$$\mathbb{E}[\![[a = e]]\!]\eta = \langle a, \mathbb{E}[\![e]\!]\eta \rangle \text{ in } V$$
$$\mathbb{E}[\![e \text{ is a}]\!]\eta = \text{if } \mathbb{E}[\![e]\!]\eta \varepsilon U \text{ then } (\text{fst}(\mathbb{E}[\![e]\!]\eta \mid U) = a) \text{ in } V \text{ else wrong}$$
$$\mathbb{E}[\![e \text{ as a}]\!]\eta =$$
$$\quad \text{if } \mathbb{E}[\![e]\!]\eta \varepsilon U \text{ then (let } \langle b,v \rangle \text{ be } (\mathbb{E}[\![e]\!]\eta \mid U) \text{ in if } b = a \text{ then } v \text{ else } \perp) \text{ else wrong}$$
$$\mathbb{E}[\![\lambda x{:}\tau\ .\ e]\!]\eta = (\lambda v.\ \mathbb{E}[\![e]\!]\eta\{v/x\}) \text{ in } V$$
$$\mathbb{E}[\![e\ e']\!]\eta =$$
$$\quad \text{if } \mathbb{E}[\![e]\!]\eta \varepsilon F \text{ then (if } \mathbb{E}[\![e']\!]\eta \varepsilon W \text{ then wrong else } (\mathbb{E}[\![e]\!]\eta \mid F)(\mathbb{E}[\![e']\!]\eta)) \text{ else wrong}$$
$$\mathbb{E}[\![\text{rec } x{:}\tau\ .\ e]\!]\eta = \mathbf{Y}(\lambda v.\ \mathbb{E}[\![e]\!]\eta\{v/x\})$$
$$\mathbb{E}[\![e{:}\tau]\!]\eta = \mathbb{E}[\![e]\!]\eta$$

Комментарии к равенствам:

- d in V (где d$\in D$ и D слагаемое V) является иньекцией значения d в подходящее слагаемое из V. Следовательно, d in $V \in V$. Это выражение не должно путать в записи let ... be ... in ... для локальных переменных.

- v ε V (где v $\in V$ и D слагаемое V) это функция вычисляющая $\perp_T$ если v $= \perp_T$; $true$ если v = d in V для некоторого d $\in D$; в остальных случаях - false.

- v | D (где D слагаемое V) это функция вычисляющая: d если v = d in V для некоторого d $\in D$; $\perp_D$ в противном случае.

- if ... $then$ ... $else$ ... это синтаксис для функции условия : $T \to V \to V \to V$ отображающей $\perp_T$ в $\perp_V$.

- равенство в L влечет $\perp_L$ всегда, когда аргументом есть $\perp_L$.

- fst извлекает первый элемент из пары, snd извлекает второй.

- Y оператор фиксированной точки типа $(V \to V) \to V$.

- $\mathbb{E}$ определяет вызов фукции (a call by value semantics) при помощи семантики значения, однако позволяет строить рекурсивные структуры.

Интуитивно кажется верным, что правильно - типизированные программы никогда не возвращают значение wrong во время выполнения. Например, рассмотрим вхождение значения wrong в семантике записей. Проверяльшик типа (typechecker) должен убедится, что выбор поля записи применяется к записям, имеющим подходящие поля, следовательно, значение wrong никогда не будет возвращаться. Подобные размышления могут быть проделаны по поводу любого вхождения значения wrong в семантику: wrong является ошибкой типа времени исполнения программы, которая может быть обнаружена во время компиляции. Исключительные состояния времени выполнения, которые не могут быть обнаружены представляются символом $\perp$; единственное вхождение этого символа в приведенной выше семантике появляется в равенстве e as a.

Определив $\mathbb{E}$ таким образом, что оно удовлетворяет рассуждениям про интуитивное восприятие ошибок времени исполнения, в следующих

секциях мы будем считать, что фраза 'е семантически является правильно типизированым' надо понимать как ' $\mathbb{E}[\![e]\!]\eta \neq$ wrong', после чего мы запишем алгоритм, выполняющий статическую проверку правильности типа.

10 СЕМАНТИКА ВЫРАЖЕНИЙ ТИПОВ

Семантика типов дается в слабой идеальной модели (weak ideal model) [16] $\mathfrak{T}(V)$ (множество непустых замкнутых слева подмножеств V которые ограничены наименьшими верхними границами возрастающих последовательностей и не содержат значения wrong - the set of non-empty left-closed subset of V which are closed under least upper bounds of increasing sequences *and* do not contain wrong). $\mathfrak{T}(V)$ представляет собой решетку доменов, отношением порядка на которых является включение множеств. $\mathfrak{T}(V)$ замкнуто относительно пересечений и конечнных объединений и относительно обычных операций на множествах. Далее, $\mathbb{D} \in TypeExp \rightarrow \mathfrak{T}(V)$:

$$\mathbb{D}[\![\iota_i]\!] = B_i \text{ in } V$$
$$\mathbb{D}[\![\{a_i : \tau_i\}]\!] = \bigcap_i \{r \in R \mid r(a_i) \in \mathbb{D}[\![\tau_i]\!]\} \text{ in } V \qquad (\text{where } \mathbb{D}[\![\{\}]\!] = R \text{ in } V)$$
$$\mathbb{D}[\![[a_i : \tau_i]]\!] = (\{\langle \bot_L, v\rangle \mid v \in V\} \cup \bigcup_i \{\langle a_i, v\rangle \in U \mid v \in \mathbb{D}[\![\tau_i]\!]\}) \text{ in } V$$
$$\mathbb{D}[\![\sigma \rightarrow \tau]\!] = \{f \in F \mid v \in \mathbb{D}[\![\sigma]\!] \Rightarrow f(v) \in \mathbb{D}[\![\tau]\!]\} \text{ in } V$$

где D in $V = \{$ d in $V \mid$ d $\in D \} \cup \{ \bot_V \}$

10.1 Теорема (свойства $\mathbb{D}$)

$\forall \tau$. $\mathbb{D}[\![\tau]\!]$ является идеалом (так как $\bot \in \mathbb{D}[\![\tau]\!]$)

$\forall \tau$, v. v$\in \mathbb{D}[\![\tau]\!] \Rightarrow$ v $\neq$ wrong

Значение wrong было сознательно оставлено вне доменов типа, так что если значение имеет тип, то это значение не является ошибокой времени исполнения. Другими словами, значение wrong не имеет типа.

11 ВКЛЮЧЕНИЕ ТИПА

Отношение подтипа может быть задано синтаксически на структуре выражений типа. Это определение формализует наше начальное обсуждение о подтипах записей, вариантов и функций.

$$\iota_i \le \iota_i$$
$$\{a_i : \sigma_i , a_j : \sigma_j\} \le \{a_i : \sigma'_i\} \quad \text{iff} \quad \sigma_i \le \sigma'_i \qquad (i \in 1..n, n \ge 0;\ j \in 1..m, m \ge 0)$$
$$[a_i : \sigma_i] \le [a_i : \sigma'_i , a_j : \sigma'_j] \quad \text{iff} \quad \sigma_i \le \sigma'_i \qquad (i \in 1..n, n \ge 0;\ j \in 1..m, m \ge 0)$$
$$\sigma \to \tau \le \sigma' \to \tau' \quad \text{iff} \quad \sigma' \le \sigma \ \text{ and } \ \tau \le \tau'$$

11.1 Предложение.

Отношение $\leq$ является отношением частичного порядка.

К выражениям типа можно добавить две константы anything и nothing, так чтобы выполнялось nothing $\leq \tau \leq$ anything для любого τ. Тогда отношение $\leq$ задаст структуру решетки на выражениях типа, которая будет подрешеткой для $\mathfrak{T}\ (V)$. (Althought this *is* mathematical appealing, we have chosen not to do so in view of our intended application) Хотя это было бы привлекательно с точки зрения математики, но в связи с нашим планируемым применением, мы не будем этого делать. Например, выражение $if x$ then 3$else\ true$, должно привести к сообщению об ошибке типа вследствие конфликта между типами int и bool в разных ветках условного оператора. При наличии же полной решетки выражений типа, будет возможно вернуть anything в качестве типа для приведенного выражения и продолжать проверку типа. Это не правильно по двум причинам. Во-первых, по крайней мере, в наших конструкциях нет возможности для использования типа anything. Во-вторых, ошибки типа будет трудно локализовать так как их присутствие отмечается исключительно возможным появлением константы anything или nothing как результирующий тип.

Как уже было сказано, упорядоченность доменов в $\mathfrak{T}\ (V)$ задается включением множеств. Это позволяет получить непосредственную семантику для выделения подтипов (subtyping), как простое теоретико-множественное включение доменов.

11.2 Теорема (Семантическое выделение подтипов)

$$\tau \leq \tau' \;\Leftrightarrow\; \mathbb{D}[\tau] \subseteq \mathbb{D}[\tau'] \;.$$

Доказательство выполняется индукцией на структуре τ и τ' . При этом, мы будем нуждаться только в достаточности ($\Rightarrow$).

12 ПРАВИЛА ВЫВОДА ТИПА

В этой секции мы формально определим идею о 'синтаксически корректно сформированном выражении' (syntactically well-formed expression). Выражение называется корректно сформированно, если его тип может быть выведен в соответствии с правилами, образующими систему вывода (inference system). Если тип выражения не может быть получен, то говорится, что выражение содержит ошибки типа.

Вообще говоря, для одного и того же выражения можно вывести несколько типов. При условии, что система вывода непротиворечива, все эти типы должны быть в каком - то смысле совместимы. Алгоритм проверки типа может выбрать любой из допустимых типов как тип выражения (можно задать лучший тип, или наиболее общий тип, или главный тип). Должно быть показано, что система вывода совместима с семантикой языка (consistent with respect to the semantics of the language), что мы и увидим в конце этой секции.

Далее приводится система вывода для нашего языка. Она спроектирована так, чтобы

1. содержать в точности одно правило для каждой синтаксической конструкции;
2. удовлетворять свойству интуитивного выделения подтипов, выраженному в теореме о синтаксическом выделении подтипов, излагаемой ниже;
3. удовлетворять теореме о семантической непротиворечивости, относительно семантики нашего языка.

Использование предиката отношения подтипа $\leq$ очень важно во многих правилах типа. Однако надо заметить, что выделения подтипов не

влияет на фундаментальные правила выделения типов λ - исчисления, [ABS] и [COMB]. Это означает, что наш стиль выделения подтипов может быть естественным образом объединен с типами функций.

[VAR] $A.x: \tau \vdash x: \tau'$ where $\tau \leq \tau'$

[BAS] $A \vdash b_{ij} : \iota_i$

[COND] $$\frac{A \vdash e: bool \quad A \vdash e': \tau \quad A \vdash e'': \tau}{A \vdash (\text{if } e \text{ then } e' \text{ else } e''): \tau}$$

[RECORD] $$\frac{A \vdash e_1 : \tau_1 \quad \dots \quad A \vdash e_n : \tau_n}{A \vdash \{a_1 = e_1, \dots , a_n = e_n\} : \{a_i : \tau_i\}}$$ where $i \in I \subseteq 1..n$

[DOT] $$\frac{A \vdash e: \{ \dots a: \tau \dots \}}{A \vdash e.a : \tau}$$

[VARIANT] $$\frac{A \vdash e: \tau}{A \vdash [a = e] : [\dots a: \tau \dots]}$$

[IS] $$\frac{A \vdash e: [\dots]}{A \vdash (e \text{ is } a) : bool}$$

[AS] $$\frac{A \vdash e: [\dots a: \tau \dots]}{A \vdash (e \text{ as } a) : \tau}$$

[ABS]
$$\frac{A.x: \sigma \vdash e: \tau}{A \vdash (\lambda x: \sigma.\ e) : \sigma \to \tau}$$

[COMB]
$$\frac{A \vdash e: \sigma \to \tau \quad A \vdash e': \sigma}{A \vdash (e\ e') : \tau}$$

[REC]
$$\frac{A.x: \sigma \vdash e: \rho}{A \vdash (\text{rec } x: \sigma\ .\ e): \tau} \qquad \text{where } \rho \le \sigma \text{ and } \rho \le \tau$$

[SPEC]
$$\frac{A \vdash e: \sigma}{A \vdash (e: \sigma) : \tau} \qquad \text{where } \sigma \le \tau$$

Некоторые комментарии к правилам:

- A (называемое множеством допущений) это конечное отображение переменных в типы; A(x) тип ассоциированный с x в A; A.x : τ это множество предположений A, расширенное при помощи ассоциации x: τ , то есть, это отображение x в τ и любую другую переменную y в A(y).
- Если существует некоторое нетривиальное включение в базовые типы (например. int $\le$ real)б то правило [BAS] должно быть изменено в $A \vdash b_{ij} : \tau$ где $\iota_i \le \tau$.
- В правиле [RECORD], тип выведенной записи может иметь меньше полей, чем соответствующий объект запись.
- В правиле [VARIANT], тип выведенного варианта может иметь любое число полей, до тех пор, пока он включает поле, соответствующее объекту варианта.
- Правило [IS] предполагает что множество базовых типов не может содержать супертип типа *bool*, в противном случае требуются

дополнительные уточняющие правила. И правило [COND] предполагает, что не существует подтипа типа *bool*.

Базовые синтаксические свойства данной системы вывода выражены в теореме синтаксического выведении подтипов: если выражение имеет тип τ и τ является подтипом типа $\tau^{'}$, то выражение также имеет тип $\tau^{'}$. Лемма необходима, чтобы доказать случай [ABC]. И лемма, и теорема доказаны по индукции на структуре выводов.

12.1 Лемма (Синтаксическое выведение подтипов)

A.x: $\sigma \vdash$ e: τ и $\sigma^{'} \leq \sigma \Rightarrow$ A.x: $\sigma^{'} \vdash$ e: τ.

12.2 Теорема (Синтаксическое выведение подтипов)

A $\vdash$ e: τ и $\tau \leq \tau^{'} \Rightarrow$ A $\vdash$ e: $\tau^{'}$.

Следующая теорема постулирует корректность системы типов относительно семантики: если возможно вывести что e имеет тип ,то значение обозначенное e принадлежит домену τ . Множество предположений A согласовано с окружением η если для всех x из домена A, A(x) = τ влечет $\eta \dot{[}$ x $\dot{]} \in \mathbb{D} \dot{[} \tau \dot{]}$.

12.3 Теорема (Семантическая непротиворечивость)

Если A $\vdash$ e: τ и A согласовано с η тогда $\mathbb{E} \dot{[}$ e $\dot{]} \eta \in \mathbb{D} \dot{[} \tau \dot{]}$.

Доказательство выполняется по индукции на структуре выводов A $\vdash$ e : τ , используя семантическое выделение подтипов и $\mathbb{D}$ - свойства теорем.

На словах, если выражение e синтаксически правильно типизировано (то есть, для некоторого τ , A $\vdash$ e: τ)(well-typed), то оно также семантически правильно типизированно (то есть, для некоторого η такого, что A согласовано с η, $\mathbb{E} \dot{[}$ e $\dot{]} \eta \in \mathbb{D} \dot{[} \tau \dot{]}$), что влечет $\mathbb{E} \dot{[}$ e $\dot{]} \eta \neq$ wrong).

13 ОБЬЕДИНЕНИЕ И ПОДБОР ТИПОВ

В примерах в начале этой статьи, мы использовали операторы *and* и *or* для выражений над типами, теперь они нам потребуются в определении алгоритма проверки типа. Однако эти операторы не являются частью синтаксиса выражений над типами, точно также, как операторы *ignoring* и *dropping*.

Это потому, что эти операторы работают только на ограниченном числе выражений типов. В применении к произвольному выражению типа они или не определены, или могут быть ограничены процессом нормализации. Даже если выражение типа содержит перечисленные выше операторы, мы сможем обработать это выражение проверяя можно ли использовать данные операторы в этом контексте и в таком случае возможно их удаление (normalize them away) и приведение выражения типа к нормальному виду.

Оператор *and* интерпретируется как частичная операция *meet* - подбор- на типах (записывается как $\downarrow$), оператор *or* - как частичныя операция *join* - обьединение - (записывается $\uparrow$). Обьединение ($join$) и подбор ($meet$) случаются на частичном порядке, заданным отношением $\leq$, когда они существуют.

Определение операторов также немедленно определяет процесс нормализации, который исключает их (из выражений):

$\iota_i \uparrow \iota_i = \iota_i$

$\{a_i : \tau_i , b_j : \sigma_j\} \uparrow \{a_i : \tau'_i, c_k : \rho_k\} = \{a_i : \tau_i \uparrow \tau'_i\}$
if all $\tau_i \uparrow \tau'_i$ are defined $(\forall\, j,k.\ b_j \neq c_k)$

$[a_i : \tau_i , b_j : \sigma_j] \uparrow [a_i : \tau'_i, c_k : \rho_k] = [a_i : \tau_i \uparrow \tau'_i , b_j : \sigma_j , c_k : \rho_k]$
if all $\tau_i \uparrow \tau'_i$ are defined $(\forall\, j,k.\ b_j \neq c_k)$

$(\sigma \to \tau) \uparrow (\sigma' \to \tau') = (\sigma \downarrow \sigma') \to (\tau \uparrow \tau')$

$\tau \uparrow \tau'$ undefined otherwise

$\iota_i \downarrow \iota_i = \iota_i$

$\{a_i : \tau_i , b_j : \sigma_j\} \downarrow \{a_i : \tau'_i, c_k : \rho_k\} = \{a_i : \tau_i \downarrow \tau'_i , b_j : \sigma_j , c_k : \rho_k\}$
if all $\tau_i \downarrow \tau'_i$ are defined $(\forall\, j,k.\ b_j \neq c_k)$

$[a_i : \tau_i , b_j : \sigma_j] \downarrow [a_i : \tau'_i, c_k : \rho_k] = [a_i : \tau_i \downarrow \tau'_i]$
if all $\tau_i \downarrow \tau'_i$ are defined $(\forall\, j,k.\ b_j \neq c_k)$

$(\sigma \to \tau) \downarrow (\sigma' \to \tau') = (\sigma \uparrow \sigma') \to (\tau \downarrow \tau')$

$\tau \downarrow \tau'$ undefined otherwise

$\{a_i : \tau_i\}$ ignoring $a = \{a_j : \tau_j\}$ $(i \in 1..n,\ j \in 1..n - \{k \mid a_k = a\})$

τ ignoring a undefined otherwise

$[a_i : \tau_i]$ dropping $a = [a_j : \tau_j]$ $(i \in 1..n,\ j \in 1..n - \{k \mid a_k = a\})$

τ dropping a undefined otherwise

Отметим, что $\uparrow$ может быть неопределена даже если существует минимальная верхняя граница относительно $\leq$ (With Regards To) для его операндов. Тоже самое для $\downarrow$.

13.1 Утверждение (свойства $\uparrow$ и $\downarrow$)

Если $\sigma \uparrow \tau$ определено, тогда существует наименьшее ρ (w.r.t $\leq$) такое, что $\sigma \leq \rho$ и $\tau \leq \rho$.

Если $\sigma \downarrow \tau$ определено, тогда существует наибольшее ρ (w.r.t $\leq$) такое, что $\rho \leq \sigma$ и $\rho \leq \tau$.

Пусть S это множество идеалов, обозначенных некоторым выражением типа без операторов $\downarrow$ (and) и $\uparrow$ (or), где r/a = (λ b. *if* b = a *then* $\perp$ *else* r(b)).

13.2 Утверждение

$\mathbb{D}[\![\ \sigma \text{ and } \tau\ ]\!] = \text{the largest ideal in } \boldsymbol{S} \text{ contained in } \mathbb{D}[\![\ \sigma\ ]\!] \cap \mathbb{D}[\![\ \tau\ ]\!]$ when defined

$\mathbb{D}[\![\ \tau \text{ ignoring a}\ ]\!] = \{r \in \boldsymbol{R} \mid ((r / a) \text{ in } \boldsymbol{V}) \in \mathbb{D}[\![\ \tau\ ]\!]\} \text{ in } \boldsymbol{V}$ when defined

$\mathbb{D}[\![\ \sigma \text{ or } \tau\ ]\!] = \text{the smallest ideal in } \boldsymbol{S} \text{ containing } \mathbb{D}[\![\ \sigma\ ]\!] \cup \mathbb{D}[\![\ \tau\ ]\!]$ when defined

$\mathbb{D}[\![\ \tau \text{ dropping a}\ ]\!] = \mathbb{D}[\![\ \tau\ ]\!] - (\{\langle a, v\rangle \in \boldsymbol{U}\} \text{ in } \boldsymbol{V})$ when defined.

14 ПРОВЕРКА ТИПОВ

$\mathbb{T} \in Exp \rightarrow TypeEnv \rightarrow TypeEnv$ - это (частичная) функция проверки типов, где Exp - выражения и $TypeEnv$ - выражения типа, согласно нашей грамматики, а $TypeEnv = Id \rightarrow TypeExp$ окружение типа для идентификаторов.

Следующее описание должно было бы быть схемой для программ, которые возвращают выражения типа, обозначая тип специальным термом или отказом в случае ошибок типа. Слово fail является глобальным переходом на завершение программы (global jump-out): в случае, если получена ошибка, это слово означает остановку работы. Точно так же проверка типа не сможет работать когда операторы $\downarrow$ и $\uparrow$ не определены. Если мы утверждаем $\mathbb{T}\ [\![\ e\ ]\!]\ \mu = \tau$, мы требуем, что проверка типа выражения e закончилась успешно.

$$\mathcal{T}[\![x]\!]\mu = \mu[\![x]\!]$$
$$\mathcal{T}[\![b_{ij}]\!]\mu = \iota_j$$
$$\mathcal{T}[\![\text{if e then e' else e"}]\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu = \text{bool then } \mathcal{T}[\![e']\!]\mu \uparrow \mathcal{T}[\![e"]\!]\mu \text{ else fail}$$
$$\mathcal{T}[\![\{a_1 = e_1, \ldots, a_n = e_n\}]\!]\mu = \{a_1 : \mathcal{T}[\![e_1]\!]\mu, \ldots, a_n : \mathcal{T}[\![e_n]\!]\mu\}$$
$$\mathcal{T}[\![e.a]\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu = \{ \ldots a: \tau \ldots \} \text{ then } \tau \text{ else fail}$$
$$\mathcal{T}[\![[a = e]]\!]\mu = [a: \mathcal{T}[\![e]\!]\mu]$$
$$\mathcal{T}[\![e \text{ is } a]\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu = [\ldots a: \tau \ldots] \text{ then bool else fail}$$
$$\mathcal{T}[\![e \text{ as } a]\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu = [\ldots a: \tau \ldots] \text{ then } \tau \text{ else fail}$$
$$\mathcal{T}[\![\lambda x: \tau . e]\!]\mu = \tau \to \mathcal{T}[\![e]\!]\mu\{\tau / x\}$$
$$\mathcal{T}[\![e\ e']\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu = (\tau \to \tau') \text{ and } \mathcal{T}[\![e']\!]\mu \leq \tau \text{ then } \tau' \text{ else fail}$$
$$\mathcal{T}[\![\text{rec } x: \sigma . e]\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu\{\sigma / x\} = \tau \text{ and } \tau \leq \sigma \text{ then } \tau \text{ else fail}$$
$$\mathcal{T}[\![e: \sigma]\!]\mu = \text{if } \mathcal{T}[\![e]\!]\mu = \tau \text{ and } \tau \leq \sigma \text{ then } \sigma \text{ else fail}$$

Этот алгоритм проверки типов корректен относительно системы вывода типов: если алгоритм закончился удачно и вернул тип τ для выражение e, тогда можно доказать, что выражение e имеет тип τ . Окружение типа μ согласовано со множеством предположений A, если для каждого x в домене A, $\mu\,\dot{[}\,x\,\dot{]} = A\,(x)$.

14.1 Теорема (Синтаксическая непротиворечивость)

Если $\mathbb{T}\,\dot{[}\,e\,\dot{]}\,\mu = \tau$ тогда μ согласовано с некоторым A, таким что $A \vdash e: \tau$.

Доказательство теоремы проводится по по индукции на структуре e, используя свойства $\uparrow$, $\downarrow$ и $\leq$.

Комбинируя синтаксическую непротиворечивость, сематническую непротиворечивость и теоремы о $\mathbb{D}$ -свойствах мы немедленно получаем:

14.2 Следствие (О предотвращении ошибок типа):

Если $\mathbb{T}\,\dot{[}\,e\,\dot{]}\,\mu = \tau$ тогда $\mathbb{E}\,\dot{[}\,e\,\dot{]}\,\eta \neq \text{wrong}$ (когда $\eta\,\dot{[}\,x\,\dot{]} \in \mathbb{D}\,\dot{[}\,\mu\,\dot{[}\,x\,\dot{]}\,\dot{]}$ для всех x).

То есть, если выражение e может быть успешно проверено на правильность типа, то оно не будет приводить к ошибками типа во время выполнения.

Алгоритм проверки типов намеренно сделан более ограничивающий, чем система вывода типов; возможно вывести A.x : bool $\vdash$ *if* x *then* { a=*true* } *else* { a=3 } : { }, однако практически мы хотим чтобы такое выражение вызывало ошибку типа, по тем же причинам, по которым мы ввели правило для типа anything (for the same reasons that made us rule out the anything type). Это ограничение навязывается определением $\uparrow$, $\downarrow$. Таким же образом, можно вывести любой тип для выражения [a=3] *as* b, и в то же время, на нем проверка типов сбойнет (typechecker fails); это справедливо, потому что [a=3] *as* b всегда приведет к сбою во время исполнения.

По этим причинам, мы не формулировали теорему синтаксической непротиворечивости в форме: если A $\vdash$ e: τ и μ согласовано с A, то $\mathbb{T}\,\dot{[}$ e $\dot{]}\,\mu$ определено и $\mathbb{T}\,\dot{[}$ e $\dot{]}\,\mu \leq \tau$. Синтаксической законченности можно достичь используя (частичные) $\vee$ и (w.r.t $\leq$) вместо $\uparrow$, $\downarrow$ в алгоритме проверки типов (после чего надо изменить алгоритм вычисления $\mathbb{T}\,\dot{[}$ *if* x *then* { a= *true* } *else* {a=3 } $\dot{]}\,\mu$ = { }) и заменив примитивы *is* и *as* конструктцией *case*.

15 ВЫВОДЫ

Эта работа появилась как попытка проверить конструкцию множественного наследования, представленную в языке баз данных Galileo [8] , и доказать непротиворечивость алгоритма проверки типов для этого языка. Язык Amber [12] , разработанный позже в качестве эксперимента, кроме всего прочего, содержал проверку наследованных типов. Я полагаю, что эта статья адекватно решает главную задачу, хотя некоторые практические и теоретические вопросы могут потребовать дополнительной работы.

Параметрический полиморфизм не рассматривался в этой статье. Главной целью было изучение множественного наследования в совершенно понятных конструкциях, без взаимодействия с другими возможностями (ооп). Побочные эффекты и циклические типы также должны быть добавлены к рассмотрению при полном формальном доказательстве.

Некоторое смущение может вызвать тот факт, что языки подобные Smalltalk- у часто представляются как полиморфные языки. Это кор-

ректно, если под полиморфизмом мы понимаем то, что объект или функция может иметь много типов. Сейчас, однако, понятно, что существует два почти не различимых типа полиморфизма: полиморфиз наследования, основанных на включениях типа и параметрический полиморфизм, основанный на переменных типа и кванторах типа.

Эти два полиморфизма не совместимы. Как мы видели в статье, наследование может быть объяснено в терминах семантики, обычно используемых для параметрического полиморфизма. Более того, техническое объяснение полиморфизма будет тем же самым в обоих случаях: пересечение доменов. Кажется, что объединение этих двух типов полиморфизма не должно приводить к новым семантическим проблемам. Далее, взаимодействие наследования и параметрического полиморфизма при проверке типов смотри в [14] .

Существует несколько конкурирующих (хотя не полностью независимых) стилей параметрического полиморфизма замеченных в [11] , [17] , [18] , [16] .

Наследование является ортогональным к любому из них, то есть, кажется правильным изучать эти вопросы независимо, по крайней мере на начальном этапе. Однако окончательной целью должно быть полная интеграция параметрического полиморфизма и множественного наследования, и слияние функционального программирования с объектно - ориентированным на уровнях семантики и типов. Эта проблема начала привлекать к себе больше внимания.

16 СМЕЖНЫЕ РАБОТЫ И БЛАГОДАРНОСТИ

I would like to mention here [Reynolds 80, Oles 84] which expose similar semantic ideas in a different formal framework, [Ait-Kaci 83] again exposing very similar ideas in a Prolog-related framework, [Mitchell 84] this time presenting different, but related, ideas in the same formal framework, and [Futatsugi 85] whose OBJ system implements a first-order multiple inheritance typechecker, and whose subsorts have much to do with subtypes.

Finally, I would like to thank David MacQueen for many discussions, John Reynolds and the referees for detailed suggestions and corrections, and Antonio Albano and Renzo Orsini for motivating me to carry out this work.

Предметный указатель

ССЫЛКИ

[1] O.Dahl, K.Nygaard: *Simula, an Algol-based simulation language*, Comm. ACM. Vol 9, pp.671-678, 1966.

[2] A.Goldberg, D.Robson: *Smalltalk-80. The language and its implementation*, Addison-Wesley, 1983.

[3] D.G.Bobrow, M.J.Stefik: *The Loops manual*, Memo KB-VLSI-81-13, Xerox PARC.

[4] J-M.Hullot: *Ceyx: a Multiformalism programming environment*, IFIP 83, R.E.A.Mason (ed), North Holland, Paris 1983.

[5] L.Steels: *Orbit: an applicative view of object-oriented programming*, in: *Integrated Interactive Computing Systems*, pp. 193-205, P.Degano and E.Sandewall editors, North-Holland 1983.

[6] D.Weinreb, D.Moon: *Objects, Message Passing, and Flavors*, chapter 20 *of Lisp machine manual*, Fourth Edition, Symbolics Inc., 1981.

[7] G.Attardi, M.Simi: *Semantics of inheritance and attributions in the description system Omega*, M.I.T. A.I. Memo 642, August 1981.

[8] A.Albano, L.Cardelli, R.Orsini: *Galileo: a strongly typed, interactive conceptual language*, IEEE Transactions on Database Systems, June 1985.

[9] K.Futatsugi, J.A.Gorguen, J.P.Jouannaud, J.Meseguer *Principles of OBJ2*, Proc. POPL '85.

[10] P.Deutsch: *An efficient implementation of Smalltalk-80*, Proc. POPL '84.

[11] R.Milner: *A theory of type polymorphism in programming*, Journal of Computer and System Science 17, pp. 348-375, 1978.

[12] L.Cardelli: *Amber, Combinators and Functional Programming Languages*, Proc. of the 13th Summer School of the LITP, Le Val D'Ajol, Vosges (France), May 1985, Lecture Notes in Computer Science n.242, Springer-Verlag, 1986.

[13] L.Morris, J.Scwarz, *Computing cyclic list constuctures*, Conference Record of the 1980 Lisp Conference, pp144-153.

[14] L.Cardelli, P.Wegner: *On understanding types, data abstraction and polymorphism*, Computing Surveys, Vol 17, n.4, pp 471-522, December 1985.

[15] А.С.Косачев, В.Н.Пономаренко: *Анализ подходов к верификации функций безопасности и мобильности*, http://ipv6.ispras.ru/Verification_of_security.pdf.

[16] D.B.MacQueen, G.D.Plotkin, R.Sethi: *An ideal model for recursive polymorphic types*, Information and Control 71, pp.95-130, 1986.

[17] J.C.Reynolds: *Towards a theory of type structure*, in Colloquium sur la programmation pp. 408-423, Springer-Verlag, Lecture Notes in Computern Science, n.19, 1974.

[18] N.McCracken: *The typechecking of programs with implicit type structure*, in Semantics of Data Types, Lecture Notes in Computern Science, n.173, Springer-Verlag, 1984.

[19] А.Г. Пискунов. Формализация парадигмы обьектно-ориентированного программирования: наследование абстрактных автоматов, 2007. http://i.com.ua/~agp1/ru/oopFormalizm.html.

[20] А.Г. Пискунов. The RAISE Method Group: АЛГЕБРАИЧЕСКОЕ ПРОЕКТИРОВАНИЕ КЛАССА, 2007. http://www.realcoding.net/article/view/4538.