Реализация гибридных типов владения в Java посредством атрибутных грамматик

И. Д. Сергей
Dept. Computer Science,
Katholieke Universiteit Leuven
ilya.sergey@cs.kuleuven.be

Типы владения (Ownership Types) являются механизмом для статической спецификации структурных свойств графа памяти в объектно-ориентированных программах. Этот механизм позволяет организовать эффективное управление памятью и сборку мусора, предотвратить возможные «состояния гонки» в многопоточных программах и т. д. Однако несмотря на данные преимущества, типы владения до сих пор не нашли широкого применения.

В данной работе мы исследуем применение *гибридного* подхода к типам владения и описываем реализацию препроцессора для анализа кода, лишь частично аннотированного типами владения. Таким образом, программист должен выполнить аннотирование лишь для небольшой части программы, а препроцессор добавит в код динамические проверки корректности для тех случаев, когда статическая проверка типов не даёт полной гарантии сохранения инварианта владения. Препроцессор реализован посредством парадигмы атрибутных грамматик, что делает его легко адаптируемым для возможных расширений и добавления новых видов статического анализа.

Ключевые слова: типы владения, статический анализ, Java.

Введение

Типы владения (*Ownership Types*) представляют собой механизм для статической спецификации структурных свойств графа памяти в объектно-ориентированных программах [9, 10]. Различные варианты типов владения позволяют организовать эффективное управление памятью и сборку мусора [6], предотвратить возможные «состояния гонки» в многопоточных программах [2, 5], а также интерференцию эффектов в изменяемых частях кода [8, 26]. Несмотря на данные преимущества, типы владения до сих пор не нашли широкого применения в практическом программировании главным образом из-за большого количества «вспомогательных» типовых аннотаций, которые должны быть добавлены в код.

Существуют разные способы справляться с «многословностью» типовых аннотаций и произвести миграцию с *нетипизированного* на *типизированный* код. В случае последнего компилятор получает возможность обнаружить и исключить ряд ошибок статически, т. е. обеспечить *статическую типовую корректность* (static type soundness) на этапе анализа кода программы. В противном случае соответствующие проверки корректности состояния программы должны быть произведены на этапе исполнения, что отрицательно сказывается на производительности программы и недопустимо для ряда приложений, работающих, к примеру, во встроенных системах.

Наиболее хорошо изученным способом для добавления явных типовых аннотаций в код является процедура *вывода типов*, при которой производится составление уравнений, содержащих аннотации для разных точек в коде программы с последующей их унификацией. Данный подход описывается типовым схемами Хиндли-Милнера [24] и широко распространен в функциональных языках программирования, таких как Haskell и семейство языков ML. К сожалению, такой подход неприменим в полной мере для вывода типов владения в объектно-ориентированных программах. Данная проблема объясняется тем, что корректная (и, возможно, тривиальная) схема аннотирования программы типами владения *всегда* существует [12]. Кроме того, полный вывод типов владения обладает сомнительной пользой в силу того, что посредством аннотаций программист сам *специфицирует* ограничения, налагаемые на

отношения между динамически создаваемыми объектами. В отсутствие каких бы то ни было аннотаций процедура вывода типов не обладает какими-либо данными об ограничениях, а стало быть, не может вывести аннотаций, отличных от тривиальных.

В работе [32] был выбран альтернативный подход для привнесения типовых аннотаций в код, основанный на *гибридных типах* [17, 33, 34]. При данном подходе программист должен предоставить лишь незначительную часть типовых аннотаций. В результате этого на этапе статического анализа компилятору остаётся проверить указанные аннотации на *непротиворечивость* и добавить динамические проверки корректности состояния программы в тех местах, где данная информация не может быть получена статически посредством анализа типов. Если программа обладает некоторым набором аннотаций, необходимость в дополнительных проверках на этапе исполнения может отсутствовать, что соответствует статической типовой корректности программы. С этой точки зрения гибридные типы представляют собой гибкий компромисс между объёмом типовой информации, необходимой для статической проверки кода, и гарантиями по безопасности исполнения программы.

В случае применения гибридного подхода для типов владения существует некоторый минимальный набор аннотаций, который предоставляет спецификацию сведений об отношениях между объектами. Остальные аннотации считаются *вспомогательными* и могут быть опущены при гибридном подходе, что, однако, грозит дополнительными динамическими проверками.

В рамках данной работы мы опускаем теоретические аспекты гибридного подхода к типам владения, равно как и формулировку и доказательства теорем типовой корректности[1]. Целью данной статьи является описание технических деталей реализации препроцессора. Мы предлагаем решение, основанное на парадигме атрибутных грамматик [13], расширяя существующую версию компилятора языка Java гибридными типами владения. Предлагаемое решение не требует изменения оригинальной реализации компилятора, равно как и само может быть дополнено новыми алгоритма-

[1]Мы отсылаем заинтересованного читателя к полной версии технического отчёта о теоретических деталях описываемого подхода, представленной в [31].

ми статического и динамического анализа, использующими типы владения [30]. Насколько нам известно, это первый опыт реализации типов владения при помощи атрибутных грамматик. В данной работе мы также обсуждаем применимость разработанного прототипа для существующих библиотек Java, таких как Java Collection Framework.

Изложение материала в данной статье построено следующим образом. В *разделе 1* изложены основные сведения о типах владения и гибридном подходе к ним, а также приведён основной мотивирующий пример. *Раздел 2* даёт краткий обзор основных концепций атрибутных грамматик и реализующего их программного инструмента JastAdd, а также описывает основные аспекты исходной реализации компилятора языка Java в среде JastAdd. В *разделе 3* изложены особенности реализации расширения компилятора JastAddJ для поддержки гибридных типов владения. В *разделе 4* мы приводим отчёт о применении гибридных типов владения на практике. *Раздел 5* содержит обзор существующих аналогичных подходов для реализации *подключаемых систем типов* (pluggable type systems). В конце статьи приведены выводы по данной работе.

1. Предварительные сведения о типах владения и мотивирующий пример

В данном разделе приводятся краткие теоретические сведения о типах владения и дано определение семантического инварианта *владелец-как-доминатор* (owner-as-dominator, OAD), реализуемого в строго типизированных языках, поддерживающих типы владения. Кроме того, приводится мотивирующий пример для использования типов владения и иллюстрируется гарантируемый инвариант владения. Раздел заканчивается интуитивными рассуждениями о применении гибридного подхода к типам владения.

1.1. Типы владения

Идея типов владения опирается на отношение *вложенности* ($\prec$), устанавливаемое между динамически создаваемыми объектами в программе. Во время исполнения программы каждому объекту o_1 назначается *владелец* — некоторый другой объект o_2, такой, что

$o_1 \prec o_2$. Отношение вложенности является частичным порядком на множестве динамически создаваемых объектов: оно обладает рефлексивностью, транзитивностью и антисимметричностью. *Наибольший* элемент отношения вложенности в литературе традиционно обозначается как world.

Будем называть *графом объектов* программы такой граф, вершинами которого являются динамически создаваемые объекты, а дугами — ссылки между ними. Кроме того, в таком графе обычно выделяют *корневой элемент* — уникальный объект, созданный при запуске программы. Инвариант *владелец-как-доминатор* (в дальнейшем — *инвариант владения*) может быть сформулирован следующим образом: *для произвольного объекта o_1 и его владельца o_2 любой путь в графе объектов программы, начинающийся в корневом элементе и заканчивающийся в o_1, проходит через o_2*. Другими словами, в программе нет ссылок-полей, указывающих на o_1, таких, что они «обходят» o_2. Это означает, что один объект может ссылаться на другой только в том случае, если он находится «внутри» ($\prec$) владельца первого объекта. Таким образом, каждый объект определяет виртуальный *регион владения*, «внутри» которого объекты могут ссылаться друг на друга.

Описанный инвариант является усилением традиционного свойства *инкапсуляции*, типичного для объектно-ориентированных языков программирования, перенесенного с уровня *классов* на уровень *объектов*. Инкапсуляция в традиционном смысле гарантирует взаимодействие классов программы друг с другом через объявленные интерфейсы, описывающие их поведение. Инкапсуляция на уровне объектов посредством типов владения гарантирует взаимодействие с объектами только через специально выделенные объекты-владельцы. Кроме того, любой объект не может «покинуть» регион владения своего объекта-владельца.

1.2. Пример использования типов владения

В примере 1 приведён фрагмент кода на языке Java версии 1.4, дополненного аннотациями владения в форме типовых параметров в угловых скобках: `<...>`. Класс `List` обладает двумя параметрами владения: `owner` и `data`. Первый параметр владения `owner` ссылается на непосредственного владельца данного объекта класса

List. Второй параметр data ссылается по принятым соглашениям на некоторый объект, такой, что owner $\prec$ data. Конструкция this означает то же, что в языке Java. Эти же рассуждения применимы к двум другим классам данного примера Link и Iterator.

Пример 1. классы, реализующие логику связанных списков (List) и итераторов (Iterator) с необходимыми (подчёркнуты) и вспомогательными аннотациями

```
class List<owner, data> {
  Link<this, data> head;
  void add(Data<data> d) {
    head = new Link<this, data>(head, d);
  }
  Iterator<this,data> makeIterator() {
    return new Iterator<this, data>(head);
  }
}
class Link<owner,data> {
  Link<owner, data> next;
  Data<data> data;
  Link(Link<owner, data> next, Data<data> data) {
    this.next = next;
    this.data = data;
  }
}
class Iterator<owner, data> {
  Link<owner, data> current;
  Iterator(Link<owner, data> first) { current = first; }
  void next() { current = current.next; }
  Data<data> elem() { return current.data; }
  boolean done() { return (current == null); }
}
```

В четвёртой строчке рассматриваемого примера создаётся экземпляр класса Link с указанными владельцами this и data соответственно. Передавая данные параметры, программист обозначает тот факт, что данный конкретный экземпляр находится во владении текущего объекта класса List, создавшего его, а также что содержимое созданного экземпляра класса Link доступно только через объект, на который в данном контексте ссылается параметр владения data класса List. Те же рассуждения справедливы для экземпляра класса Iterator, создаваемого в примере тремя строками ниже. На рис. 1 изображена схема отношений между объек-

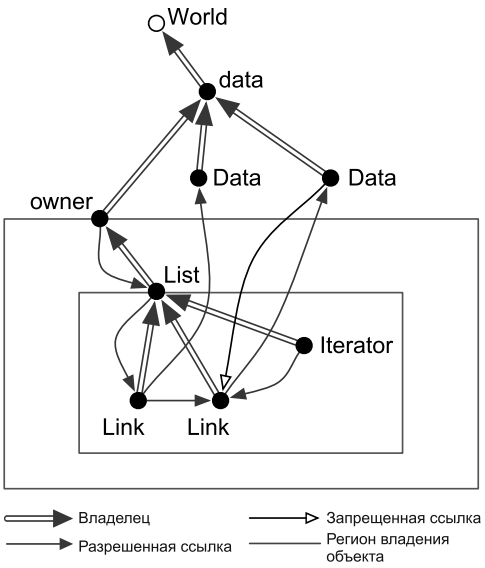

Рис. 1. Отношения владения, разрешённые и запрещённые ссылки для объектов, созданных с использованием классов из примера 1

тами, которые могут быть созданы динамически в программе, использующей классы из *примера 1*. Прямоугольные участки на схеме указывают на то, что любые экземпляры классов `Link` и `Iterator` заключены *внутри* региона владения экземпляра класса `List`, создавшего их. Точно так же любой экземпляр класса `List` находится строго внутри своего объекта-владельца, обозначенного параметром `owner`, который, в свою очередь, находится внутри объекта, обозначенного параметром `data`. Несложно видеть, что отношение владения формирует древовидную структуру на графе объектов программы. Построенное таким образом дерево аппроксимирует дерево непосредственных доминаторов в графе объектов [23]. На рис. 1 также изображено, какие ссылки являются разрешёнными и запрещёнными в данной системе классов. Так, к примеру, экземпляр класса `Data` не может ссылаться на экземпляр класса `Link`, потому как данная ссылка соответствовала бы пути из корневого элемента `world` в экземпляр класса `Link` в обход владельца — эк-

земпляра класса `List`, что противоречило бы инварианту владения. Различные формализации систем типов, гарантирующих статическую корректность программы в отношении данного инварианта, изложены в работах [8, 9, 32].

1.3. Гибридные типы владения

Гибридный подход к типам владения позволяет опустить существенную часть аннотаций при сохранении инварианта владения. Цена, которую приходится платить за краткость по сравнению с оригинальным подходом, — наличие динамических проверок инварианта. Разумеется, чем больше типовых аннотации предоставлено программистом, тем меньше динамических проверок будет добавлено препроцессором в код.

При этом в любом случае программист должен добавить некоторое *минимальное* число аннотаций в код для того, чтобы обозначить свои намерения касательно отношения владения между объектами. В примере 1 часть аннотаций отмечена подчёркиванием. Такие аннотации являются обязательными. В их число входит параметризация классов параметрами владения, как, к примеру, в случае с классом `List<owner, data>`. Кроме того, при создании каждого объекта необходимо указывать явно значения параметров владения, как это сделано, к примеру, в случае с созданием экземпляра `new Link<this, data>()`. Разумеется, отсутствие данных аннотаций может быть разрешено путём введения дополнительных соглашений. К примеру, можно принять по умолчанию, что `new Link()` является сокращением для `new Link<world, world>()`.

Аннотации, не отмеченные подчёркиванием в примере 1, являются вспомогательными. Они не специфицируют отношения владения между объектами, а служат лишь для передачи этой информации посредством типов в том случае, если некоторый объект присвоен локальной переменной или полю объекта, передан в качестве параметра или возвращён как результат вызова метода. При гибридном подходе эти аннотации могут быть опущены, и соответствующие проверки будут добавлены препроцессором. В этом случае неизвестные аннотации владения могут быть полностью опущены либо заменены символом «?», который указывает на точки

программы, которые могут быть возможной причиной нарушения инварианта владения. Проиллюстрируем эти рассуждения примером:

```
List list; // ≡ List<?,?>
list = new List<p, world>();
list = new List<this, world>();
List<p, world> newList = list; // необходима динамическая проверка типа
```

В последней строке приведенного примера в операторе присваивания переменная `list` обладает типом `List<?,?>`. При этом её ожидаемый тип `List<p, world>` определяется переменной `newList`. Необходимая операция приведения типа `List<?, ?>` ⇒ `List<p, world>` будет добавлена в код препроцессором. Другие примеры использования гибридных типов владения описаны в [31].

Статическая типовая корректность в отношении гибридных типов владения может быть сформулирована следующим образом: *частично аннотированная корректно типизированная программа не нарушает инвариант владения, но может быть аварийно прервана в случае попытки его нарушения.* При этом справедливо следующее утверждение: *полностью аннотированная и корректно типизированная программа не нарушает инвариант владения.* С нашей точки зрения, гибридный подход является разумным компромиссом для постепенного аннотирования кода. В случае, если программа не удовлетворяет установленным ограничениям, сообщения об ошибках и нарушении инварианта могут быть использованы для отладки и устранения проблемы либо изменения изначальных аннотаций.

2. Среда JastAdd

В данном разделе читатель познакомится со средой построения компиляторов JastAdd [20]. Данная среда предоставляет набор инструментов для декларативного описания структуры абстрактных синтаксических деревьев (*Abstract Syntactic Trees*, AST) разрабатываемого языка и спецификации поведения синтаксических единиц. Помимо создания стандартных функций для работы с абстрактными грамматиками, JastAdd позволяет реализовать дополнительную логику для анализа и трансформации синтаксических деревьев посредством механизма *атрибутных грамматик* (*Rewritable Circular*

Reference Attribute Grammars [13, 22]). JastAdd предоставляет также возможность добавлять новые атрибуты в вершины абстрактного синтаксического дерева без изменения исходного кода классов вершин. Для этого применяется механизм композиции или замещения существующих реализаций, что достигается путём использования *аспектной* модели, сходной со средой AspectJ [21]. Добавленное или дополненное поведение синтаксических элементов языка может быть описано как *декларативно* (в терминах атрибутов), так и *императивно* (в терминах стандартной спецификации языка Java).

2.1. Описание абстрактного синтаксиса языка

Абстрактная грамматика языка, описываемого в JastAdd, формирует иерархию классов, которые используются в качестве вершин AST. В примере 2 показан фрагмента кода, описывающего абстрактную грамматику. Для каждого нетерминала в грамматике генерируется отдельный класс. Если нетерминал обозначен как абстрактный (`abstract`), то для него генерируется абстрактный класс, а разделение нетерминалов в левой части продукции двоеточием (например, `ClassDecl : TypeDecl`) соответствует установлению отношения наследования между сгенерированными классами вершин AST. В правой части продукции располагается список элементов, соответствующих данному нетерминалу. Имя по умолчанию для каждого элемента является таким же, как и для соответствующей вершины, если только оно не указано явно, как, например, в случае с элементом-ссылкой на суперкласс `SuperClassAccess:Access`. Элементы в угловых скобках, например `<ID:String>`, соответствуют значениям, которые являются терминалами абстрактной грамматики, тогда как другие элементы, например `Modifiers`, соответствуют вершинам дерева. Необязательные элементы заключены в квадратные скобки. Элементы, помеченные `<<*>>`, например `BodyDecl*`, обозначают списки из нуля или более элементов.

Пример 2. Абстрактная грамматики описания класса в языке Java

```
abstract TypeDecl ::= Modifiers <ID:String> BodyDecl*;

ClassDecl : TypeDecl ::= Modifiers <ID:String>
    [SuperClassAccess:Access] Implements:Access* BodyDecl*;
```

По данному описанию грамматики JastAdd генерирует иерархию классов, конструкторы, а также методы доступа к элементам-детям, например `Modifiers getModifiers()` для класса `TypeDecl`. Элементы списков могут быть получены при помощи целочисленных индексов посредством сгенерированных методов доступа: `BodyDecl getBodyDecl(int index)`.

JastAdd не ограничивает выбор реализаций лексического и синтаксического анализаторов, которые могут быть использованы для разбиения входных файлов на лексемы и построения абстрактного синтаксического дерева. В реализации компилятора JastAddJ, взятой нами за основу, используются инструменты JFlex[2] и Beaver[3].

2.2. Описание атрибутов в среде JastAdd

Атрибутные грамматики являются мощным инструментом для описания контекстно-зависимой информации при реализации компиляторов [22]. При атрибутном подходе поведение вершин описывается атрибутами, а их значения определяются уравнениями. JastAdd расширяет модель атрибутных грамматик понятием *ссылочных атрибутов* (*reference attributes*), то есть атрибутов, которые ссылаются на другие вершины AST [13]. Поведение вершин может быть расширено новыми атрибутами посредством аспектной модели [1]. Аспекты могут интерферировать друг с другом, а также быть частично упорядочены при помощи механизма уточнения (refining), рассмотренного подробнее в *разделе 3*.

Синтезированные атрибуты. Наиболее часто используемый тип атрибутов — так называемый *синтезированный* (*synthesized attributes*). Синтезированные атрибуты аналогичны виртуальным методам без побочных эффектов, которые могут быть добавлены в классы вершин AST и эффективно вычислены благодаря механиз-

[2]URL: http://jflex.de
[3]URL: http://beaver.sourceforge.net

мам кэширования. Последнее возможно, если атрибут объявлен *ленивым* (`lazy`). Следующий фрагмент кода описывает атрибут, который определяет для вершины, является она ссылкой на объявление поля некоторого класса или нет:

```
syn boolean Expr.isFieldName(String name);
eq Expr.isFieldName(String name) { return false; }
eq FieldName.isFieldName(String name) = getName().equals(name);
```

Важно отметить, что в силу того, что `FieldName` является подтипом класса `Expr`, второе уравнение переопределяет метод `isFieldName` для класса `FieldName`. В противном случае для класса `FieldName` была бы использована изначальная реализация данного атрибута для класса `Expr`.

Унаследованные атрибуты (*inherited attributes*) используются для эффективной реализации логики поиска по абстрактному синтаксическому дереву методом прохода «снизу вверх» (lookup). Информация, вычисленная в какой-либо вершине, может быть передана вниз по дереву посредством унаследованных атрибутов (context propagation). Рассмотрим задачу о нахождении определения метода для выражения, находящегося внутри определения этого метода. Фрагмент кода, представленный ниже, иллюстрирует решение этой задачи средствами унаследованных атрибутов:

```
inh MethodDecl Expr.enclosingMethodDecl();
eq MethodDecl.getChild().enclosingMethodDecl() = this;
eq Program.getChild().enclosingMethodDecl() = null;
```

Класс `MethodDecl` определяет контекст для всех вершин типа `Expr`, находящихся «ниже» его в AST. Сама же логика унаследованного атрибута определяется в терминах *контекста*, в данном случае — `MethodDecl` (а не вершины типа `Expr`, для которой определён атрибут). Первое уравнение должно быть прочитано следующим образом: определим значение атрибута `enclosingMethodDecl()` для всего поддерева, чей корень является «ребёнком» некой вершины типа `MethodDecl`, как значение данной вершины `MethodDecl`. Второе уравнение доопределяет данный атрибут для всех выражений, не находящихся в поддереве определения какого-либо метода. Это делается путём определения атрибут в контексте вершины типа `Program`, которая является *уникальной* для данной программы и

корнем всего AST. Уравнения для унаследованных атрибутов справедливы для всего поддерева, что позволяет не копировать реализацию всех возможных типов вершин поддерева.

Собирательные атрибуты. Если унаследованные атрибуты позволяют решить задачу передачи информации по дереву сверху вниз, то собирательные атрибуты (collecting attributes) позволяют решить задачу сбора информации снизу вверх путём обхода поддерева, типично решаемую при помощи образца объектно-ориентированного программирования Visitor [18]. К примеру, для того чтобы собрать имена всех типов, объявленных в программе на языке Java, можно объявить следующий собирательный атрибут:

```
coll Set<String> Program.typeNames()
  [new HashSet<String>()]
  with add
  root Program;
```

Этот фрагмент кода объявляет атрибут `typeNames`, определённый для вершин типа `Program` и обладающий типом `Set<String>`. Атрибут инициализируется значением `new HashSet<String>()`, обновление атрибута производится с помощью вызова метода `add()` класса `Set`. Все возможные вершины, способные дополнить значение данного атрибута, должны находиться в поддереве элемента типа `Program`, т. е. во всём коде программы.

Второй этап определения собирательного атрибута — описание логики пополнения значений.

```
TypeDecl contributes name()
  to Program.typeNames()
  for getProgram();
```

Этот фрагмент кода указывает, что все объявления типов (классов, интерфейсов и т. п.) добавляют значение определённого для них метода `name()` в собирательное значение атрибута `typeNames()` вершины типа `Program`, возвращаемого методом `getProgram()` и определённого для класса `TypeDecl`. Важно заметить, что можно зарегистрировать более одного типа пополнений, и в этом случаи они будут производиться в произвольном порядке.

Прочие атрибуты. В числе прочих атрибутов, предоставляемых средой JastAdd, важно отметить так называемые *циклические*

атрибуты (circular attributes). Они позволяют вычислять значения путём обхода ряда вершин дерева и итерации некоторой монотонной функции на частично-упорядоченном множестве значений до тех пор, пока не будет достигнута наименьшая неподвижная точка функции перехода. Данные атрибуты незаменимы при решении задачи статического анализа потока данных методом итерации потоковых функций на графе потока управления [25]. Более подробные примеры использования циклических атрибутов можно найти в JastAdd [19].

2.3. JastAddJ: базовая реализация Java средствами атрибутных грамматик

За основу нашего препроцессора, расширяющего язык Java типами владения, взят проект JastAddJ — полная реализация компилятора языка программирования Java версии 1.4 посредством среды для работы с атрибутными грамматиками JastAdd. JastAddJ состоит из нескольких модулей, реализующих разные подзадачи компилятора: анализ типов и связывание имён, генерацию байт-кода и т. п. Также существуют расширения JastAddJ, добавляющие элементы языка Java версии 1.5 и `@NotNull`-аннотации [14, 15]. Наибольшую часть реализации JastAddJ составляет связывание имён и проверка типов. Наш препроцессор, описанный в *разделе 3*, дополняет и расширяет эту функциональность.

3. Расширение языка Java гибридными типами владения

В этом разделе мы описываем основные технические детали реализации расширения компилятора JastAdd гибридными типами владения, описанными в *разделе 1*. За основу взята модель языка Java версии 1.4. Типы владения могут быть скомбинированы с параметризованными типами Java (Java Generics) [29], однако мы выбрали упрощённую модель языка для сохранения ясности реализации прототипа. Для обозначения параметров владения в нашем прототипе был выбран синтаксис, стандартный для типовых параметров. В практической реализации параметры владения могут быть также выражены при помощи аннотаций, введённых в версии языка Java 1.5.

Объем кода реализованного прототипа составляет примерно 2600 строк без учёта комментариев, пустых строк и тестовых данных[4].

3.1. Подстановки

В спецификации языка Java 1.4 каждый тип, не являющийся примитивным, тождественен классу, который его описывает. В случае с типами владения тип описывается не только классом, но и конкретными значениями параметров владения. Например, классу

```
class C<p, q> extends D<p> { ... }
```

в различных контекстах могут соответствовать типы: T_1 = `C<this, owner>`, T_2 = `C<owner, world>`, где значения `this` и `owner` зависят от конкретного контекста. В нашем прототипе мы реализуем каждый тип как пару $C\langle\sigma\rangle$, где C — имя базового класса, а σ — соответствующая подстановка. Подстановка — это частично-определённая функция, которая сопоставляет параметрам конкретного класса их значения для некоторого типа владения. Например, упомянутые выше типы T_1 и T_2 для базового класса $C < p, q >$ определяются подстановками $\sigma_1 = \{p \mapsto \texttt{this}, q \mapsto \texttt{owner}\}$ и $\sigma_2 = \{p \mapsto \texttt{owner}, q \mapsto \texttt{world}\}$, так что $T_1 = C\langle\sigma_1\rangle$, $T_2 = C\langle\sigma_2\rangle$.

Описание типов с помощью базового класса и подстановки удобно, т. к. подстановка обладает свойствами частично определённой функции, что позволяет вычислять композицию подстановок $\sigma_2 \circ \sigma_1$ при переводе типа из одного семантического контекста в другой, а также производить переопределение подстановок на конкретно взятых элементах. Подстановки реализуются следующими классами:

```
public interface Fun<Dom, Im> { /* интерфейс функции */
  Im apply(Dom arg);

}
public class Substitution implements Fun<String, OwnerDeclaration> {
  public boolean isDefinedAt(String str) { ... }
  public OwnerDeclaration apply(String name) { ... }
```

[4]Исходный код прототипа с примерами и инструкциями по запуску доступен по адресу http://people.cs.kuleuven.be/ ilya.sergey/gradual/.

```
    public Substitution compose(final Substitution inner) { ... }
    public static Substitution empty() { ...}
    public Substitution update(String newName,
                               OwnerDeclaration value) { ... }
}
```

Помимо стандартных операций с частично определёнными функциями, реализация класса подстановки позволяет создать «пустую» (т. е., нигде не определённую) подстановку (`empty()`) и переопределить подстановку для конкретного значения параметра (`update()`).

3.2. Параметры владения и расширение синтаксиса

Наша следующая цель — расширить абстрактный синтаксис для поддержки типов владения. Для этого мы добавляем описания новых вершин абстрактного синтаксического дерева в исходную реализацию компилятора JastAddJ. К стандартным объявлениям классов и интерфейсов добавляются объявления классов, параметризованных владельцами (пример 3). Абстрактный синтаксис для объявлений интерфейсов, параметризованных владельцами, определяется аналогичным образом.

Пример 3. Абстрактный синтаксис описания класса, параметризованного владельцами

```
OwnerClassDecl : ClassDecl ::=
   Modifiers <ID:String> OwnerVariable*
   [SuperClassAccess:Access] Implements:Access* BodyDecl*
   <OwnershipClassTypes:Map<Substitution, OwnershipClassType»;
```

На приведённом листинге подчёркиванием выделены фрагменты, добавленные к исходному описанию абстрактного синтаксиса объявления классов `ClassDecl`:

• `OwnerVariable*` служит для обозначения списка параметров владения, указанных при описании класса;

• `OwnerhipClassTypes` является синтетическим элементом, который представляет собой ассоциированное с классом отображение для кэширования выведенных типов владения по соответствующим подстановкам. Подробнее об этом элементе речь пойдёт в *разделе 4*.

Параметры владения представляют собой идентификаторы, реализованные посредством класса `String`. Кроме того, мы вводим

синтетический элемент `BadOwnerVariable` для обозначения неудачного результата попытки связывания ссылки на владельца с его объявлением:

```
OwnerVariable ::= <ID:String>;
BadOwnerVariable;
```

Любая именная ссылка, такая как ссылка на поле, переменную, вызов метода или тип, в терминологии JastAddJ реализована классом, наследующим тип `Access`, например `SuperClassAccess` в примере 3. Любой владелец, упомянутый в произвольном типе программы, является либо ссылкой на `world`, `this` или некоторый параметр владения, либо неопределённым владельцем «?»:

```
abstract Owner : Access;
World : Owner;
RefOwner : Owner;
ThisOwner : Owner;
UnknownOwner : Owner;
```

Наконец, сами типы в программе описываются парами из объявления класса и соответствующей подстановки, определённой на параметрах этого класса:

```
OwnershipClassType : OwnerClassDecl ::=
    <Inner:OwnerClassDecl> <Substitution:Substitution>;
```

Важно заметить, что определённые таким образом типы не объявляются явно, а создаются в процессе проверки и вывода типов как вспомогательные элементы. Тем не менее, мы вводим их для улучшения производительности и сохраняем в качестве элементов отображения `OwnershipClassTypes`.

3.3. Связывание имён владельцев

Каждая ссылка на владельца в программе должна связываться с соответствующим объявлением владельца. Для этого был введён интерфейс `OwnerDeclaration`:

```
interface OwnerDeclaration { ... }
OwnerVariable implements OwnerDeclaration;
World implements OwnerDeclaration;
ThisOwner implements OwnerDeclaration;
UnknownOwner implements OwnerDeclaration;
```

Можно заметить, что `World`, `ThisOwner`, `UnknownOwner` являются одновременно и ссылками, и объявлениями владельцев. На практике это не приводит к неожиданным коллизиям в силу того, что при реализации мы придерживаемся следующих инвариантов:

- все `this`-ссылки на владельца внутри класса связываются с одним и тем же экземпляром `ThisOwner`, специфичным для данного класса;
- все `world`- и «?»-ссылки на владельцев связываются с одними и теми же уникальными экземплярами `World` и `UnknownOwner`, специфичными для данной программы.

Продемонстрируем на примере, как комбинирование «ленивых» синтезированных и унаследованных атрибутов позволяет обеспечить данный инвариант. Метод `decl()` определяется как «ленивый» синтезированный атрибут для всех ссылок на владельцев для связывания их с соответствующими объявлениями:

```
syn lazy OwnerDeclaration Owner.decl();
eq ThisOwner.decl() = lookupThisOwner();
eq World.decl() = world();
```

Для `this`-ссылок на владельцев синтезированный атрибут связывания вызывает унаследованный атрибут `lookupThisOwner()`, определённый в терминах объявлений классов и интерфейсов:

```
inh OwnerDeclaration ThisOwner.lookupThisOwner();

eq Program.getChild().lookupThisOwner() = new BadOwnerVariable();
eq ClassDecl.getChild().lookupThisOwner() = thisOwner();

lazy syn OwnerDeclaration TypeDecl.thisOwner() {
  ThisOwner t = new ThisOwner();
  t.setParent(this);
  return t;
}
```

Благодаря механизму «прокидывания» значений унаследованных атрибутов через вершины абстрактного синтаксического дерева, для которых они не объявлены (inherited attribute propagation [19]), вызов метода `lookupThisOwner()` из экземпляра класса `ThisOwner` будет обработан в ближайшем (согласно иерархии в AST) определении класса или интерфейса `TypeDecl`. Если же этот

метод будет вызван выше, то он будет обработан экземпляром класса `Program`, который вернёт `BadOwnerVariable`. Так как атрибут `thisOwner()` определён как «ленивый» (`lazy`), значение, возвращаемое им, будет *уникально* для каждого конкретного определения класса.

В случае с `world`-ссылками соответствующий атрибут `world()`, возвращающий экземпляр класса `World`, реализуется для класса `Program`, а доступ к нему предоставляется всем вершинам AST посредством «прокидывания»:

```
syn lazy World Program.world() = new World();
eq Program.getChild().world() = world();
inh World ASTNode.world();
```

Реализация связывания прочих ссылок на владельцов с их объявлениями аналогична реализации связывания переменных в JastAddJ.

3.4. Расширение проверки типов

Принимая во внимание изменённую структуру типов в нашем языке по сравнению с изначальной спецификацией Java, необходимо изменить логику присваивания и проверки типов таким образом, чтобы вычислялись необходимые подстановки. Благодаря механизму переопределения атрибутов и аспектной модели JastAdd уточнения в логике присваивания типов могут быть сделаны без изменения кода исходной реализации компилятора JastAddJ. Рассмотрим следующий пример:

```
class C<p, q> {
  void foo()  {
    C<p, q> c = this;
  }
}
```

Очевидным образом тип `this`-ссылки в данном фрагмента должен равняться не C (каким бы он был согласно спецификации Java 1.4), а $C\langle\sigma\rangle$, где $\sigma = \{p \mapsto \mathtt{p},\ q \mapsto \mathtt{q}\}$. Изначальная реализация присваивания типа `this`-ссылке в JastAddJ реализована в аспекте `TypeAnalysis` посредством комбинации синтезированных и унаследованных атрибутов следующим образом:

```
syn lazy TypeDecl ThisAccess.decl() =
    isQualified() ? qualifier().type() : hostType();
syn ThisAccess.type() = decl();
```

При этом `hostType()` является унаследованным атрибутом, определяемым в терминах «ближайшего» определения класса или интерфейса `TypeDecl`:

```
inh TypeDecl Expr.hostType();
syn TypeDecl TypeDecl.hostType() = this;
```

Мы переопределяем логику присваивания типа `this`-ссылке путём *уточнения* (`refine`) аспекта `TypeAnalysis`:

```
1 refine TypeAnalysis eq ThisAccess.type() {
2   TypeDecl original = TypeAnalysis.ThisAccess.type();
3    if (original instanceof OwnerTypeDecl) {
4      return original.lookupOwnershipType(original.getSubstitution());
5    } else {
6      return original;
7    }
8  }
```

В строке 1 приведённого примера указывается, что данный атрибут является *уточнением* атрибута, уже описанного в аспекте `TypeAnalysis`. В строке 2 исходный тип `this`-ссылки вычисляется путём вызова оригинального атрибута `TypeAnalysis.ThisAccess.type()`. Если данный класс является параметризованным владельцем, то в строке 4 производится поиск соответствующего типа посредством вызова атрибута `lookupOwnershipType()`. В противном случае возвращается исходный класс как тип.

Синтезированный атрибут `lookupOwnershipType()` заслуживает отдельного упоминания. Экспериментальным путём было выяснено, что большинство типов, встречающихся в процессе анализа, может быть переиспользовано, и не обязательно вычислять их каждый раз путём композиции подстановок. Гораздо проще кэшировать их в виде элементов-детей в соответствующих определениях классов и интерфейсов (вспомним AST-вершину `OwnerhipClassTypes` из *раздела 3.2*). Следующий фрагмент кода иллюстрирует создание нового типа владения и его кэширование в

вершине соответствующего объявления класса. Важно отметить, что метод `getOwnershipClassTypes()` был сгенерирован автоматически, согласно спецификации AST, описанной в *разделе 3.2.*

```
syn lazy TypeDecl TypeDecl.lookupOwnershipType(Substitution subst);

eq OwnerClassDecl.lookupOwnershipType(Substitution subst) {
  // попытка найти закэшированный тип
  OwnershipClassType type = getOwnershipClassTypes().get(subst);
  if (type != null) return type;

  // создание нового типа
  OwnershipClassType ot = ...

 // кэширование созданного типа
 getOwnershipClassTypes().put(ot.getSubstitution(), ot);
 return ot;
}
```

Аналогичным образом уточняются реализации атрибутов для вывода типов других выражений, а также для сравнения типов на соответствие согласно спецификации гибридных типов владения [31].

3.5. Зависимые параметры владения

В случае гибридных типов владения недостаток статической информации о владельцах может быть частично возмещён за счёт отслеживания зависимостей между владельцами объектов, на которые указывают неизменяемые ссылки. Для этой цели мы вводим понятие *зависимых владельцев*, которые служат для сохранения статической информации о равенстве тех или иных объектов-владельцев при том, что сами эти владельцы на этапе компиляции могут быть неизвестны. В качестве пояснения рассмотрим следующий фрагмент кода:

```
1 class E<P> { D myD = ... }
2
3 class D<owner> {
4   E<owner> e;
5   void use(D<owner> arg) { ... }
6   void exploit(E<owner> arg) { this.e = arg; }
7   void test(E e) {
8     final D d = e.myD; // неявно, d:D<d^D.owner>
```

```
 9    d.use(d); // приведение типа не нужно
10    d.exploit(e); // необходимо приведение типа
11  }
12 }
```

Класс `E` объявляет поле типа `D`, однако информация о владельце объекта, на который ссылается поле `myD`, утеряна, потому что соответствующая аннотация владения опущена в строке 1 данного примера. Вследствие этого владелец объекта, на который ссылается переменная `d` в строке 9 статически, неизвестен. Тем не менее в силу того, что переменная `d` объявлена неизменяемой, несложно видеть, что владелец `d` соответствует ожидаемому статическому владельцу параметра метода `d.use()`, а стало быть, динамическая проверка типа не требуется (что не так в случае следующего вызова метода `d.exploit(e)`). Это знание сохранено благодаря присваиванию типа $\texttt{D<d}^{\texttt{D.owner}}\texttt{>}$ переменной `d`. В данном случае $\texttt{d}^{\texttt{D.owner}}$ является статически известным зависимым владельцем, поскольку он зависит от объекта, на который ссылается неизменяемая переменная `d`. Индекс `D.owner` ссылается на конкретный параметр владения статически известного типа `D` переменной `d` [5].

На практике существует проблема того, что *любое* выражение в программе может иметь тип, содержащий некоторое количество зависимых владельцев, которые в дальнейшем могут быть использованы компилятором для добавления динамических проверок в код. Для того чтобы точно определить, какие зависимые владельцы могут использоваться, и не сохранять при этом информацию о всех, мы реализовали простой алгоритм статического анализа, который для каждого выражения вычисляет множество зависимых владельцев, которые используются препроцессором при проверке его типа. Данный алгоритм было бы просто реализовать посредством шаблона объектно-ориентированного программирования Visitor, потому что он обходит все вершины некоторого поддерева AST [18]. Однако для этой же цели JastAdd уже предоставляет механизм собирательных атрибутов. Реализация алгоритма анализа, собирающего информацию о множестве реально используемых зависимых владельцев, представлена в примере 4.

[5] Зависимые владельцы также реализуют интерфейс `OwnerDeclaration`, но они не присутствуют в тексте программы в явном виде, поэтому мы опустили их описание в разделе 3.2 для того, чтобы сохранить простоту изложения.

Пример 4. Сбор информации о зависимых параметрах владения посредством собирательных атрибутов

```
coll Collection<OwnerDeclaration> MethodDecl.usedDependentOwners()
  [new HashSet<OwnerDeclaration>()]
  with addAll
  root MethodDecl;

Expr contributes usedDependentOwners()
  to MethodDecl.usedDependentOwners()
  for enclosingBodyDecl();

syn Collection<OwnerDeclaration> Expr.usedDependentOwners() { ... }
```

Данный алгоритм анализа реализуется собирающим атрибутом `MethodDecl.usedDependentOwners()`, который определён для каждого метода `MethodDecl`. Сбор информации происходит путём вызова метода `addAll()` для добавления элементов в объект типа `HashSet<OwnerDeclaration>()`, изначально создаваемый пустым. Добавление элементов происходит в каждой вершине AST типа `Expr`. Множество вершин типа `Expr`, участвующих в анализе данного метода, синтаксически ограничено самим этим методом, на что указывает ограничение региона сбора информации значением атрибута `enclosingBodyDecl()`.

3.6. Сообщения об ошибках и трансляция, основанная на типах владения

Сообщения об ошибках, основанные на проверке типов владения, встраиваются в исходную реализацию компилятора JastAddJ путём уточнения императивно определённых атрибутов. При этом переиспользуется логика оригинального аспекта для проверки ошибок `ErrorCheck`:

```
refine ErrorCheck public void ASTNode.collectErrors() {
  ownersSpecificChecks();
  typeCheckWithOwners();
  ErrorCheck.ASTNode.collectErrors();
}
```

Сообщения подразделяются на две категории: ошибки проверки типов (*errors*) и предупреждения (*warnings*). Последние, как пра-

вило, указывают на невозможность статической проверки соответствия типов, что характерно для гибридного подхода:

```
public void Expr.ownersSpecificChecks() {
  TypeDecl selfType = type();
  TypeDecl expected = expectedOwnType();
  if (!selfType.conformsTo(expected)) {
  warning("The type " + selfType.fullName() +
      " of the expression " + this.toString() +
      " does not statically conform to its expected type " +
      expected.fullName());
}}
```

Данные предупреждения соответствуют динамическим проверкам, которые будут добавлены при трансляции, а значит, в перспективе и возможным ошибкам исполнения, вызванным нарушением инварианта владения. Эти данные уже на этапе компиляции могут быть эффективно использованы для анализа и устранения возможных несоответствий при добавлении аннотаций владения в код.

Разработанный препроцессор производит как проверку типовой корректности согласно спецификации гибридных типов владения, так и трансляцию, основанную на выведенных типах. Результатом работы препроцессора являются Java-файлы с добавленными в определения классов полями владельцев, динамическими приведениями типов и проверками инвариантов владения. Данные файлы могут быть скомпилированы при помощи любого Java-компилятора, поддерживающего стандарт Java 1.4.

4. Эксперименты

В данном разделе приведены результаты использования разработанного препроцессора для добавления аннотаций владения в код стандартной библиотеки Java Collection Framework из JDK 1.4.2.

Бóльшая часть традиционных классов коллекций обладает структурой, схожей со структурой класса `List`, приведённой в примере 1. Предполагая, что объекты классов реализации типа `Link` не должны покидать контекста оперирующей с ними коллекции, мы попытались проаннотировать классы `LinkedList`, `TreeMap` и `Iterator` типами владения. Нашей целью было ответить на следующие вопросы.

1. Каков необходимый минимум аннотаций владения для обеспечения инварианта владения путём динамических проверок?
2. Какова цена (в терминах времени исполнения программы) операций обновления и выборки из коллекций, чья реализация была проаннотирована?
3. Сколько необходимо аннотаций, чтобы обеспечить статическую типовую корректность в отношении инварианта владения?

Для обеспечения инварианта владения путём динамических проверок нам понадобилось добавить 17 аннотаций для класса `codeLinkedList` и 15 аннотаций для класса `TreeMap`. При этом в среднем операции обновления структуры данных замедлились в 1,5–1,7 раза по сравнению с оригинальной реализацией за счёт проверок инварианта при каждой операции обновления.

Понадобилось добавить ещё 17 аннотаций в классы библиотеки для того, чтобы обеспечить статическую типовую корректность класса `LinkedList`. При этом большую помощь оказали предупреждения препроцессора, описанные в разделе 3.6. Очевидно, что в этом случае время обновления структуры данных `LinkedList` не отличается от результатов для исходной реализации.

К сожалению, в силу ограниченности выразительности выбранной системы типов владения, не удалось произвести полную миграцию класса `TreeMap`. Во многом это обусловлено наличием в реализации *статических* методов, осуществляющих создание внутренних элементов типа `Link`. В этом случае невозможно определить истинного владельца объектов данного класса, и по умолчанию он считается равным наиболее глобальному владельцу `world`.

5. Обзор существующих подходов

В данном разделе мы приводим обзор существующих подходов по расширению компиляторов языка Java и добавлению в язык более выразительных систем типов.

Термин «подключаемые системы типов» (pluggable type systems) был впервые введён Гиладом Браха (Gilad Bracha) и обозначает в той или иной мере все виды типовых аннотаций, позволяющих за-

давать и статически проверять различные свойства программ, описываемые в терминах данных систем типов как корректные. При этом важной особенностью подключаемых систем типов является тот факт, что они не изменяют семантику исполнения программы, а только позволяют удостовериться в том, что программа удовлетворяет конкретным свойствам. В качестве примера классической подключаемой системы типов можно привести `@NonNull`-аннотации, позволяющие избежать ошибки исполнения, возникающей при попытке разыменовать ссылку, указывающую на `null` [16]. Семейство различных систем типов владения является другим примером подключаемых типов.

Существует ряд инструментов, позволяющих реализовать подключаемые системы типов «поверх» основной системы типов языка Java. В большинстве подходов для этого используется механизм аннотаций, введённый в стандарте Java 1.5. К данному семейству инструментов относятся JavaCop [3] и Checker Framework [28]. Последний инструмент был успешно применён для реализации системы типов владения и аннотации большей части библиотеки JDK [35].

Данные инструменты не подходили для решения задачи внедрения в Java *гибридных* типов владения, так как описываемая в нашей работе система не является *подключаемой* в строгом смысле и требует не только дополнения исходного алгоритма проверки типов, но и расширения компилятора для реализации трансляции на основе выведенных типов и дополнения кода динамическими проверками инварианта. Вследствие перечисленных отличий нами был выбран JastAdd [20] как инструмент, позволяющий реализовать полноценный компилятор, а не только препроцессор проверки типов.

Аналогичным по выразительности инструментом для построения компиляторов является Polyglot [27]. В отличие от JastAdd Polyglot ориентирован на реализацию языков, являющихся расширением Java. Кроме того, отсутствие аспектной модели и декларативных средств работы с атрибутными грамматиками делает переиспользование компонентов, реализованных в Polyglot, более трудоёмким.

Результатом, наиболее близким к описанному в данной работе, является реализация *non-null*-типов [16] в среде JastAdd [14]. Тем не менее описываемая авторами задача не требует расширения компилятора алгоритмом трансляции, дополнения алгоритма свя-

зывания имён, использования собирательных атрибутов и работы с подстановками.

Для реализации гибридных типов владения для полной спецификации языка Java версии 1.4 нами был адаптирован ряд концепций из недавних работ по типам владения [4, 7, 8, 29]. Это было сделано главным образом для того, чтобы добиться плавной интеграции идеи инкапсуляции уровня объектов с концепцией объектного программирования как Iterator, а также чтобы сделать возможной поддержку локальных внутренних классов в Java.

Заключение

В данной статье мы описали детали реализации гибридных типов владения для языка Java в среде разработки компиляторов JastAdd. Выбранный нами подход основан на декларативном описании атрибутных грамматик для расширения логики компилятора новыми алгоритмами анализа и процедурами для генерации кода. В статье были изложены основы гибридных систем типов владения и концепция атрибутных грамматик в терминологии выбранного инструмента реализации. В данной работе мы также предоставили отчёт о внедрении типов владения в исходный код библиотеки Java Collections Framework. Насколько нам известно, это первый опыт использования атрибутных грамматик для реализации гибридной системы типов. Предлагаемое решение не требует изменения оригинальной реализации компилятора, а также может быть дополнено новыми алгоритмами статического и динамического анализа, надстроенными «поверх» типов владения.

Список литературы

[1] *Сафонов В. О.* ASPECT.NET — инструмент аспектно-ориентированного программирования для разработки надежных и безопасных программ // Компьютерные инструменты в образовании. № 5. 2007. С. 3–13.

[2] *Трифанов В. Ю.* Динамическое обнаружение гонок в Java-программах с помощью векторных часов // Системное программирование. Вып. 5: Сб. статей / под ред. А. Н. Терехова, Д. Ю. Булычева. СПб.: Изд-во СПбГУ, 2010. С. 95–116.

[3] *Andreae C., Noble J., Markstrum S. , Millstein T.* A Framework for Implementing Pluggable Type Systems. // Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Portland, Oregon, USA, 2006. P. 57–74.

[4] *Boyapati C., Liskov B., Shrira L.* Ownership Types for Object Encapsulation. In G. Morrisett, Editor, POPL '03: Proceedings of the 30^{th} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, 2003. P. 213–223.

[5] *Boyapati C., Rinard M.* A Parameterized Type System for Race-Free Java Programs. In J. Vlissides, editor, OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tampa. Florida, USA, 2001. P. 56–69.

[6] *Boyapati C., Salcianu A., Beebee W., Rinard M.* Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In R. Gupta, editor, PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 2003. P. 324–337.

[7] *Cameron N., Noble J.* Encoding Ownership Types in Java. In J. Vitek, editor, Proceedings of the 48^{th} International Conference on Objects, Models, Components, Patterns (TOOLS 2010). Berlin, Heidelberg, Springer-Verlag. 2010. P. 271–290.

[8] *Clarke D., Drossopoulou S.* Ownership, Encapsulation and the Disjointness of Type and Effect. In S. Matsuoka, Editor, OOPSLA '02: Proceedings of the 17^{th} ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Seattle, Washington, USA, 2002. P. 292–310.

[9] *Clarke D. G.* Object Ownership and Containment. PhD Thesis, University of New South Wales. New South Wales, Australia, 2001. 231 p.

[10] *Clarke D. G., Potter J. M., Noble J.* Ownership Types for Flexible Alias Protection. In C. Chambers, editor, OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Vancouver, British Columbia, Canada, 1998. P. 48–64.

[11] *Cook W. R.,* editor. OOPSLA '06: Proceedings of the 21^{st} ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. Portland, Oregon, USA, 2006. 497 p.

[12] *Dietl W., Ernst M. D., Müller P.* Tunable Universe Type Inference. Technical Report 659, Department of Computer Science, ETH Zurich. Dec. 2009. 27 p.

[13] *Ekman T., Hedin G.* Rewritable Reference Attributed Grammars. In M. Odersky, editor, ECOOP 2004: Proceedings of the 18th European Conference on Object-Oriented Programming. Oslo, Norway, 2004. P. 144–169.

[14] *Ekman T., Hedin G.* Pluggable Checking and Inferencing of Nonnull Types for Java // Journal of Object Technology. 6(9). Oct. 2007. P. 455–475.

[15] *Ekman T., Hedin G.* The JastAdd Extensible Java Compiler. In D. F. Bacon, C. V. Lopes, and G. L. Steele, Jr., editors, OOPSLA '07: Proceedings of the 22^{nd} Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. Montreal, Quebec, Canada, 2007. ACM. P. 1–18.

[16] *Fähndrich M., Leino K. R. M.* Declaring and Checking Non-Null Types in an Object-Oriented Language. In G. L. Steele Jr., editor, OOPSLA '03: Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim. California, USA, 2003. P. 302–312.

[17] *Flanagan C.* Hybrid Type Checking. In S. Peyton Jones, Editor, POPL '06: Proceedings of the 33^{rd} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Charleston, South Carolina, Jan. 2006. P. 245–256.

[18] *Gamma E., Helm R., Johnson R., Vlissides J.* Design Patterns. Addison-Wesley; Boston; MA, 1995. 395 p.

[19] *Hedin G.* Reference Manual for JastAdd, 2011. 43 p. URL: http://jastadd.org/ web/documentation/reference-manual.php

[20] *Hedin G., Magnusson E.* Jastadd: an Aspect-Oriented Compiler Construction System. Sci. Comput. Program. Vol. 47. 2003. P. 37–58.

[21] *Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G.* An Overview of AspectJ. In J. L. Knudsen, Editor, ECOOP '01: Proceedings of the 15^{th} European Conference on Object-Oriented Programming. London, UK, 2001. P. 327–353.

[22] *Knuth D. E.* Semantics of Context-Free Languages. Theory of Computing Systems. Vol. 2(2). June 1968. P. 127–145.

[23] *Lengauer T., Tarjan R. E.* A Fast Algorithm for Finding Dominators in a Flowgraph. ACM Trans. Program. Lang. Syst. Vol. 1. January 1979. P. 121–141.

[24] *Milner R.* A Theory of Type Polymorphism in Programming. J. Comput. Syst. Sci. Vol. 17. 1978. P. 348–375.

[25] *Muchnick S. S., Jones N. D.*, editors. Program Flow Analysis: Theory and Applications. Prentice-Hall, 1981. 448 p.

[26] *Müller P.* VSTTE 2005: Verified Software: Theories, Tools, Experiments: Zurich, Switzerland, October 10–13, 2005, Revised Selected Papers and Discussions. chapter Reasoning about Object Structures Using Ownership. Springer-Verlag; Berlin; Heidelberg, 2008. P. 93–104.

[27] *Nystrom N., Clarkson M. R., Myers A. C.* Polyglot: an Extensible Compiler Framework for Java. In G. Hedin, editor, CC '03: Proceedings of the 12th International Conference on Compiler Construction. Warsaw, Poland; Springer-Verlag, 2003. P. 138–152.

[28] *Papi M. M., Ali M., Correa T. L. jr., Perkins J. H., Ernst M. D.* Practical Pluggable Types for Java. In ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis. Seattle, WA, USA, July 22–24. 2008. P. 201–212.

[29] *Potanin A., Noble J., Clarke D., Biddle R.* Generic Ownership for Generic Java. // Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland, Oregon, USA, 2006. P. 311–324.

[30] *Sergey I., Clarke D.* Dominance Analysis via Ownership Types and Abstract Interpretation. Technical Report, Katholieke Universiteit Leuven, 2011. 18 p. URL: http://people.cs.kuleuven.be/ ilya.sergey/ownership-cfa/vmcai-submission.pdf

[31] *Sergey I., Clarke D.* Gradual Ownership Types. Technical Report, Katholieke Universiteit Leuven, 2011. 40 p. URL: http://people.cs.kuleuven.
be/ ilya.sergey/gradual/gradual-ownership-report.pdf

[32] *Sergey I. Clarke D.* Towards Gradual Ownership Types. In IWACO '11: Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, Lancaster, UK, 2011. To be appeared.

[33] *Siek J., Taha W.* Gradual Typing for Functional Languages. In em Scheme and Functional Programming Workshop. Portland, Oregon, USA, 2006. P. 81–92.

[34] *Siek J., Taha W.* Gradual Typing for Objects. In E. Ernst, Editor, ECOOP 2007: Proceedings of the 21st European Conference on Object-Oriented Programming, Berlin. Germany, 2007. P. 2–27.

[35] *Zibin Y., Potanin A., Li P., Ali M., and Ernst M. D.* Ownership and Immutability in Generic Java. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. Reno/Tahoe, Nevada, USA, 2010. P. 598–617.