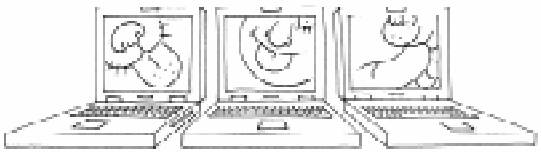
Конопко Кирилл Сергеевич

HELGINS: УНИВЕРСАЛЬНЫЙ ЯЗЫК ДЛЯ НАПИСАНИЯ АНАЛИЗАТОРОВ ТИПОВ

1. ЯЗЫКООРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



За время, прошедшее с появления на свет самых первых языков программирования, языки впитали в себя и отразили в своём дизайне множество различных подходов к написанию программ. Любой такой подход или парадигма программирования изобреталась для того, чтобы облегчить задачу программисту, разбивая его задачу на более мелкие подзадачи (иными словами, производя декомпозицию). Так, в структурном программировании появились циклы и процедуры, что стало большим шагом вперёд по сравнению с более ранними программами, представлявшими собой однородные наборы из команд и условных и безусловных переходов. В объектно-ориентированном программировании появились классы объектов, содержащие в себе данные и методы работы с ними – ещё один уровень декомпозиции, шаг вперёд по сравнению с однородным набором процедур. Такой шаблон (патерн) как MVC (Model–view–controller, «Модель–представление–поведение») отделил представление данных от их непосредственного содержания.

Возможность декомпозиции большой и сложной задачи на более мелкие является важнейшим условием для успешного решения этой задачи. Декомпозиция выявляет внутреннюю структуру задачи, облегчая её понимание. Адекватное отображение структуры решаемой задачи на структуру составных частей создаваемой программы позволяет создать успешное разделение труда, при котором каждый программист трудится над своей частью, даже не слишком мешая при этом другим коллегам. Первый приходящий в голову пример: создание компилятора. При анализе задачи создания компилятора можно выявить, что компилятор должен состоять, например, из следующих подсистем: синтаксический анализатор, семантический анализатор и генератор кода. Каждая из подсистем реализуется своим собственным набором классов, каждый такой набор классов может писаться отдельным программистом достаточно независимо от остальных подсистем компилятора. И зачастую задача по созданию программного средства, в силу своей природы, естественным образом разбивается на подзадачи, лежащие в различных предметных областях. В каждой такой предметной области существуют свои собственные формальные теории, свои методы, свои специфические трудности. Та же упомянутая выше задача написания компилятора являет собой совокупность подзадач из трёх различных областей, причём методы и формализмы, используемые для построения синтаксического анализатора, совершенно не годятся для создания семантического анализатора, и наоборот, а также ни

те, ни другие не нужны при разработке генератора кода. Итак, достаточно сложная программа состоит из подсистем, каждая из которых находится в своей предметной области, и в каждой такой предметной области есть всегда свои собственные, специфические сущности и специфические способы их описания.

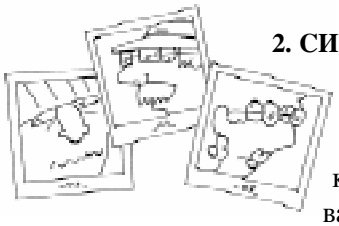
Каждая из этих подсистем может быть затем запрограммирована на языке общего назначения, таком как, например, Java. И тут есть одна неприятность. Языки общего назначения в некотором смысле похожи на язык для машины Тьюринга. Да, на них действительно можно написать всё, что угодно. Но языки общего назначения не специализированы ни под какую конкретную область и ни под какую конкретную задачу, которую надо решить, – они даже чересчур универсальны, и это плохо. Проблема эта в современных языках решается весьма удовлетворительно путём создания программных библиотек. Хорошие библиотеки имеют удобный интерфейс программирования приложений (Application Programming Interface – API). Но даже удобный API – это всё же не язык: выразительные возможности API ограничены выразительными возможностями того языка, на котором он написан. И разумеется, использование API, корректное с точки зрения языка, на котором написана библиотека, но нарушающее внутреннюю логику предметной области, для которой создавалась библиотека, никакой средой разработки за ошибку не посчитается. Проверки же на уровне логики базового языка – это сравнительно низкоуровневые проверки, и они выявят лишь сравнительно низкоуровневые ошибки. Язык же вырастает из API библиотеки только тогда, когда внутренняя логика предметной области выражена явно и эта логика поддерживается соответствующими средствами. Самое важное: логика предметной области должна поддерживаться средой разработки. Понятно, что для разработчика программы было бы лучше, если бы каждая предметная область была представлена в его программе полноценным языком программирования, а не библиотекой.

Тут мы подходим к идее языкоориентированного программирования. Что же это такое? Языкоориентированным программированием называется такой способ программирования, при котором решаемая задача разбивается на подзадачи, для решения каждой из которых используется свой язык, максимально естественно описывающий её предметную область. Часто может оказаться так, что подходящего языка программирования для данной области ещё не существует, и поэтому такой язык на ходу создаётся самим разработчиком программы. Чтобы такой подход работал, нужно средство для достаточно быстрого создания языков.

Оказывается, представление языка в виде грамматики для таких целей не подходит. Мало того, что язык, кроме грамматики, состоит ещё из дополнительных условий на корректность программы (система типов, видимость переменных) и, разумеется, из семантики, то есть описания того, какие действия обозначает та или иная языковая конструкция. К тому же, такое грамматическое, текстовое представление языка неудобно для его дальнейшего расширения и для взаимодействия с другими языками. Для языкоориентированного программирования удобнее всего представлять программу в виде дерева, а синтаксис языка – это описание структуры этого дерева. Отдельно описывается грамматика, семантика, система типов и прочие аспекты языка, если они нужны.

Как же писать такую программу? С одной стороны, программа является не текстом, а деревом. С другой стороны, графически представлять дерево неудобно – нет ничего хуже, чем программирование мышкой. Что же делать? Решение было найдено: представлять на экране отдельные элементы дерева в виде обычных текстовых полей и новые элементы дерева создавать с клавиатуры. При этом процесс редактирования выглядит почти так же, как редактирование обычной программы в текстовом редакторе. Например, если я имею узел «2», то есть целочисленную константу, и нажимаю после неё «+», то этот узел дерева трансформируется в узел «2 + _», то есть оператор сложения, у которого первым опе-

рандом та самая константа «2», а второй пока пуст. Редактор знает, как отображать дерево на экране и в ответ на какие клавиши и каким образом дерево нужно перестраивать. Таким образом, в описание языка входит, помимо его абстрактного синтаксиса, то есть структуры деревьев, ещё и описание того, как эти деревья надо на экране рисовать. Третьей составляющей описания языка является его семантика – то есть описание того, как выполнять программы на этом языке. Это может быть или интерпретатор, или генератор машинного кода, или это может быть генератор кода на каком-нибудь языке общего назначения, таком как Java, который потом уже обрабатывается своим, уже существующим компилятором. Итак, три необходимых аспекта языка мы назвали: это структура, представление и семантика. Если же мы создаём статически типизированный язык, то нам к тому же необходимо описать ещё и его систему типов.



2. СИСТЕМЫ ТИПОВ

Важной частью многих языков программирования является система типов. А что это такое система типов? Любой язык программирования предполагает наличие правил построения корректных языковых конструкций. В свою очередь, многие конструкции состоят в конечном итоге из выражений. Основной характеристикой выражения является значение этого выражения. Можно рассматривать выполнение программы как последовательное вычисление значений тех выражений, которые встречаются в программе. Значение выражения характеризуется его типом. А что же такое тип? Тип выражения говорит о том, какого рода объектом является его значение. Именно от типа зависит, каким переменным можно присваивать выражение, в каком контексте использовать, какие операции можно над ним производить. Например, строки нельзя делить друг на друга, а вещественные числа можно. Нельзя умножать строку на вещественное число. Также во многих языках программирования, как правило, нельзя присвоить переменной типа «целое число» выражение, имеющее тип «строка». И так далее. Говорят, что подобные операции – использование выражений некоторого типа там, где этот тип использовать нельзя, – являются типово некорректными. А система типов – это и есть те правила, по которым выражениям программы сопоставляются некие типы и проверяется типовая корректность.

Проверка типовой корректности программы может производиться либо транслятором (компилятором) во время трансляции программы в исполняемый код, либо же во время исполнения программы. В последнем случае типовые некорректности порождают ошибки времени выполнения. Языки, в которых проверка типовой корректности производится во время трансляции, называются языками со статической типизацией, а те, что проверяют корректность типов во время выполнения программы – языками с динамической типизацией. Языки с динамической типизацией не очень интересны с точки зрения их системы типов: весь алгоритм типизации в таких языках сводится к тому, чтобы посмотреть на уже вычисленные значения аргументов той операции, которая сейчас будет над ними произведена, и, если нужно, выдать ошибку. Поэтому в дальнейшем мы будем говорить только о статически типизированных языках программирования.

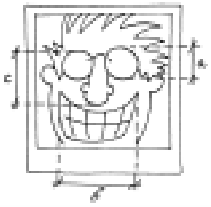
Статическая типизация позволяет программисту выявить типовые ошибки ещё на этапе компиляции, не прибегая для этого к отладке программы. Но вычисление типов полезно не только для обнаружения ошибок. Системы типов широко используются для поддержки важных возможностей сред разработки.

Программисты давно уже не пишут программы в обычных текстовых редакторах, чтобы потом скомпилировать её отдельной программой-компилятором. Программисты пишут программы в интегрированных средах разработки, включающих в себя и ре-

дактор исходного кода программы, и компилятор, и отладчик, и многие другие полезные инструменты. Редактор любой современной интегрированной среды разработки просто обязан обладать возможностью автодополнения (autocompletion). *Автодополнение* – это возможность по запросу программиста автоматически дописать имя используемой переменной или функции. Например, пусть в нашей программе где-то определена переменная с именем *variable*, и ниже мы пишем что-нибудь вроде «2 / *vari*». Если мы после этого нажмём определённую комбинацию клавиш, редактор сам за нас допишет имя переменной, и выражение примет вид «2 / *variable*». Автодополнение особенно полезно, когда в программе используется много переменных с длинными именами – без автодополнения программисту пришлось бы каждый раз вводить имя целиком, а так достаточно ввести только две-три первые буквы, для того чтобы использовать переменную. Разумеется, хорошим стилем программирования является именно использование длинных – «говорящих» – имён, описывающих предназначение переменной, функции или метода, а поэтому механизм автодополнения является совершенно необходимой частью современных интегрированных сред разработки. Ну, хорошо, скажете вы, но при чём же здесь система типов? Давайте рассмотрим такую ситуацию. Где-то в программе определены две переменные, одна определена как «double *variable1*», а вторая – как «String *variable2*». Мы снова написали «2 / *vari*» и хотим вызвать автодополнение, чтобы «*vari*» дополнилось до *variable1*. Но в данном случае простой механизм автодополнения нам не допишет имени, а предложит два варианта – *variable1* и *variable2*. И сколько бы мы не написали начальных букв имени *variable1*, автодополнение всё равно будет предлагать нам два варианта, пока мы не напишем все символы имени. С другой стороны, понятно, что использование второго предложенного варианта – переменной *variable2* – является в этом месте некорректным, поскольку нельзя делить число, в данном случае 2, на строку, каковой и является *variable2*. Более умный механизм мог бы предлагать при автодополнении не все видимые переменные, а только те, которые подходят по типу в данном месте. Разумеется, здесь и работает система типов.

Кроме того, система типов играет важную роль при так называемых рефакторингах, то есть изменениях структуры программы, не меняющих её поведения. Большинство современных сред разработки поддерживают автоматические рефакторинги. Пожалуй, простейшим из автоматических рефакторингов является introduce variable, то есть ввести новую переменную. Суть этого рефакторинга состоит в том, что некоторое выражение заменяется на новую переменную, определение которой создаётся выше в тексте программы, и там же этой переменной присваивается то выражение, которое на неё заменилось. Для примера рассмотрим такой кусок программы: «x = 239 + 13 + 42;». Пусть мы хотим выделить «13 + 42» в новую переменную с именем i. Среда разработки, поддерживающая данный рефакторинг, превратит наш кусок программы в такой: «int i = 13 + 42; x = 239 + i;». Имя новой переменной, естественно, вводится программистом руками. А вот чтобы понять, какой тип проставить в декларации созданной переменной (в данном случае – int), надо знать тип выражения, которое выделили в эту переменную. Опять работает система типов!

Итак, типы нужны при проверке и подсветке ошибок, при автодополнении, при использовании автоматических рефакторингов. Ясно, что система типов является важной частью типизированного языка программирования, не менее важной, чем его синтаксис.

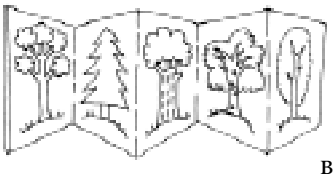


3. ЯЗЫК ДЛЯ ОПИСАНИЯ СИСТЕМ ТИПОВ

Мы рассмотрели в общих чертах языкоориентированное программирование и выяснили, что язык описывается несколькими аспектами: структурой, внешним представлени-

ем, семантикой и проч., в частности, системой типов. Можно рассматривать описания различных аспектов языка как различные предметные области. Таким образом, мы получаем ряд предметных областей, в которых работает разработчик языка. Совершенно естественным является в данном случае придумать набор языков, описывающих эти предметные области. Тогда у нас будет язык для описания структуры, язык для создания редактора, язык для описания семантики, язык для описания системы типов, то есть специальные языки для описания языков. И в самом деле, в систему Jetbrains MPS, на примере которой мы и будем рассматривать языкоориентированное программирование, входит набор специальных предопределённых языков, на которых пользователь описывает уже свои собственные языки. И нет ничего удивительного в том, что эти языки для описания языков (метаязыки) написаны сами на себе!

В данной статье будет детально рассмотрена одна из предметных областей описания языков, а именно – система типов. В системе Jetbrains MPS для описания систем типов служит язык под названием HELGINS (High-Level Equation-based Language for Gathering Information from Node Structure). Этот язык – простое и достаточно удобное средство для написания анализаторов типов к программам на языках, описанных в формате Jetbrains MPS.



3.1. НЕМНОГО ПРО MPS

Расскажу немного о том, как выглядит программа на MPS. Программа представляет собой лес, то есть совокупность деревьев. Дерево состоит из узлов различных видов. Понятие «вид узла» весьма похоже на привычное понятие «грамматический символ». Вид узла характеризует набор его свойств или полей (таких как имя у переменной, строка у строковой константы и т. п.), узлы каких видов могут быть его детьми, какие у него могут быть ссылки. Сами виды узлов относятся к структуре языка и описываются на метаязыке структуры. В системе MPS вид узла принято называть словом «концепт». Вот пример описания концепта:

```
concept VariableDeclaration
properties:
name : string
children:
initializer 0..1 Expression
type        1    Type
```

здесь написано, что концепт *VariableDeclaration* (декларация переменной) содержит в себе свойство «имя» типа «строка» (name : string), а также имеет одного (1) ребёнка, чей концепт – это *Type*, и может иметь, а может и не иметь (0..1) одного ребёнка, чей концепт – это *Expression* (выражение). Дети помечены соответствующими ролями: один ребёнок называется *initializer* (инициализатор), другой – *type* (тип). Расмотренное описание годится, например, для декларации переменной в языке Java: у неё должен быть тип, при том она может быть или не быть проинициализирована в момент декларации.

Рассмотрим ещё пример: описание концепта «использование переменной» (листинг 1).

Когда программист где-нибудь в своей программе использует переменную, он имеет в виду, что использует именно ту самую переменную, которую он определил выше с таким же именем, и современная среда разработки это обычно понимает: например, позволяет, находясь на использовании, переходить на декларацию переменной. Когда программа представляется не текстом, а изначально в виде дерева, тогда в узле «использование переменной» достаточно иметь ссылку на соответствующую декларацию. Что и описано в данном случае: узел кон-

Листинг 1

```
concept VariableReference
references :
variableDeclaration 1 VariableDeclaration
```

Листинг 2

```
RULE typeOf_StringLiteral
APPLICABLE FOR concept = StringLiteral as sl
DO {   TYPEOF (sl) :==: < String > ; }
```

цепт *VariableReference* (ссылка на переменную) имеет одну (1) ссылку (reference) на узел, чей концепт – это *VariableDeclaration* (описанный выше). Ссылка помечена ролью «*variableDeclaration*».

3.2. ПРАВИЛА ВЫВОДА ТИПОВ В HELGINS

Итак, мы вкратце рассмотрели структуру программы на МПС и теперь перейдём непосредственно к языку HELGINS. Задача анализатора типов – сопоставить некоторым узлам в дереве программы их типы. Соответственно, на HELGINS пишутся правила, по которым эти типы сопоставляются узлам.

Самое простое правило – это присвоить всем узлам некоторого концепта один и тот же заранее известный тип. В языке Java так обстоит дело, например, со строковыми литералами («hello!»), булевскими (true, false) и целочисленными константами (2, 3, 9) – у них всегда тип String, boolean, int соответственно. На языке HELGINS это записывается так (см. листинг 2).

Данное правило гласит, что оно применимо к любому узлу, чей концепт – это *StringLiteral*, и для всех таких узлов утверждает, что их тип TYPEOF (*sl*) равен *String*. Таинственные угловые скобочки обозначают, что значение любого синтаксического узла, стоящего внутри них, есть он сам, то есть его синтаксическое дерево. Пояснить эту довольно туманную фразу поможет несложная табличка (см. таблицу 1).

Другой достаточно часто встречающийся вид правила – это правило, которое декларирует равенство типов двух различных узлов. Например, тип использования переменной должен быть равен типу её объявления (декларации). Это можно записать вот так (см листинг 3).

Здесь мы видим, что правило утверждает равенство типа использования переменной TYPEOF (*varRef*) и типа декларации переменной TYPEOF (*varRef.variableDeclaration*). Мы помним, что у *VariableReference* есть одна ссылка, помеченная ролью *variableDeclaration* – вот здесь мы к ней и обращаемся!

Однако одними равенствами нам при проверке типов не обойтись. Очень часто требуется сказать, что тип такого-то узла должен быть не строго равен какому-то типу, а должен быть одним из его подтипов. Например, в операции присваивания тип правой части должен быть (нестрогим) подтипом левой части. В Java допустимы такие

Таблица 1

Выражение	Значение
3 + 2	число 5
< 3 + 2 >	узел, чей концепт – это *PlusExpression*, с двумя детьми – целочисленными константами 2 и 3
< *String* >	узел, чей концепт – это *ClassType*, со ссылкой на класс *String*

Замечание. String – это не выражение, значения не имеет.

Листинг 3

```
RULE typeOf_VariableReference
APPLICABLE FOR concept = VariableReference as varRef
DO { TYPEOF(varRef) :==: TYPEOF (varRef.variableDeclaration) ; }
```

Листинг 4

```
RULE typeOf_VariableDeclaration
APPLICABLE FOR concept = VariableDeclaration as varDecl
DO {
  TYPEOF (varDecl) :==: varDecl.type ;
  if (varDecl.initializer != null) {
     TYPEOF (varDecl.initializer) :<=: TYPEOF (varDecl) ;
  }
}
```

присваивания: *Object o* = «hello»; здесь тип правой части – *String*, а левой – *Object*, но поскольку *String* является подтипом *Object*, то всё нормально. Таким образом, кроме равенств, нам потребуются ещё и неравенства, которые мы сейчас и рассмотрим на примере правила для декларации переменной.

Первая строчка внутри тела правила уже должна быть понятной. Тип декларации переменной равен тому типу, который в ней указан (переменная «int *a*» имеет тип int, «*String* s» имеет тип *String* и т. п.). Далее проверяется, есть ли у переменной инициализатор (*varDecl.initializer* != null) мы помним, что у узла с концептом *VariableDeclaration* ребёнка с ролью *initializer* может и не быть, см. выше). И если инициализатор всё-таки есть, то начинается самое интересное: создаётся неравенство, утверждающее, что тип инициализатора TYPEOF (*varDecl.initializer*) должен быть нестрогим подтипом (:<=:) типа самой переменной TYPEOF (*varDecl*). Теперь, если мы захотим написать *String s* = 239, система типов «выругается», потому что тип инициализатора, то есть int, не является подтипом типа переменной, то есть *String*. Выражение же *Object o* = «hello» будет вполне законно, так как тип инициализатора, то есть *String*, является подтипом *Object*.

Типовые неравенства позволяют очень красиво и коротко записывать некоторые утверждения про типы. Например, в C, Java и многих C-подобных языках есть такое выражение, как тернарный оператор. Он содержит три аргумента, первый из них булевского типа, и если его значение – истина, то результатом всего выражения является первый аргумент, а иначе – второй. Записывается он так: *c* ? *t* : *f*, где *c*, *t* и *f* – какие-то выражения. Внимание, вопрос: каков тип всего тернарного оператора? Во-первых, его тип должен быть надтипом второго аргумента, так как он может вернуть свой второй аргумент. Во-вторых, по тем же причинам его тип должен быть надтипом третьего аргумента. Правильным ответом является следующий: тип всего тернарного оператора должен быть наименьшим общим надтипом типов его второго и третьего аргументов. А теперь смотрите, как просто правило для тернарного оператора записывается на HELGINS (листинг 5), где *condition*, *ifTrue* и *ifFalse* – название первого, второго и третьего аргумента соответственно. Заметьте, мы просто сформулировали вполне логичную и понятную вещь: тип тернарного оператора больше или равен типу его второго и третьего аргумента. После этого анализатор типов сам вычис-

Листинг 5

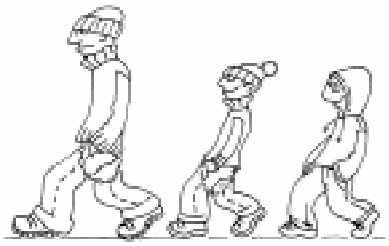
```
RULE typeOf_TernaryOperator
APPLICABLE FOR concept = TernaryOperator as ternaryOp
DO {
  TYPEOF (ternaryOp.condition) :<=: < boolean > ;
  TYPEOF (ternaryOp) :>=: TYPEOF (ternaryOp.ifTrue) ;
  TYPEOF (ternaryOp) :>=: TYPEOF (ternaryOp.ifFalse) ;
}
```

Листинг 6

```
SUBTYPING RULE int_extends_long
APPLICABLE FOR concept = IntegerType as intType
DO {   return < long > ; }
```

ляет тип *ternaryOp* как наименьший общий надтип их обоих.

3.3. ПРАВИЛА НАСЛЕДОВАНИЯ

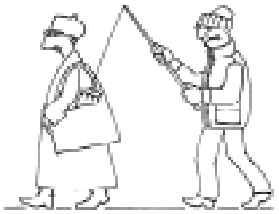


Вот мы тут рассуждаем – подтипы, надтипы... А как анализатор типов узнаёт, кто чей надтип? Откуда он берёт эту информацию? От разработчика системы типов берёт. Есть в HELGINS специальные правила, которые используются не для того, чтобы присваивать узлам дерева типы, а для того, чтобы выяснить, кто чей надтип или подтип. Это – правила наследования. Работают они очень просто: для каждого типа они указывают его непосредственный надтип, и далее по свойству транзитивности наследования анализатор понимает, является ли один тип «больше» другого или же нет (когда мы говорим, что один тип «больше» другого, мы имеем в виду, что первый тип есть надтип другого). Типичный пример правила наследования приведен в листинге 6.

Здесь записано, что правило применимо ко всем узлам (а типы – это тоже обычные узлы MPSовского дерева), чей концепт *IntegerType*, и что для всех таких узлов их непосредственным надтипом является тип *long*. Если у нас будет подобное же правило ещё и для для *short* и *int* (гласящее, что *short* «меньше» *int*), то на вопрос, «является ли *short* подтипом *long*», анализатор типов ответит утвердительно.

Замечу, что в секции APPLICABLE FOR как правил наследования, так и правил вывода может находиться и более сложное условие, нежели просто указание концепта. Там может стоять шаблон, с которым сравнивается узел при выяснении того, применимо ли данное правило к узлу или же нет. Рассмотрим, например, следующее правило. Оно выражает тот факт (см. листинг 7), что в Java, начиная с версии 1.5, переменным типа int можно присваивать значения типа Integer (равно как и в обратную сторону, но данное правило про это не говорит).

Внутри странного вида обратных угловых скобок находится шаблон, с которым надо сравнивать узел. Таким образом, данное правило применимо не ко всем узлам, у которых концепт *ClassType*, что было бы весьма неправильно, а только к тем из них, которые ссылаются на класс Integer, что даёт именно тот эффект, который требуется.



3.4. МЕХАНИЗМ РАБОТЫ

Мы познакомились с тем, как писать анализаторы типов на языке HELGINS. Теперь интересно было бы узнать, а как же он работает? Как, используя эти правила, анализатор типов присваивает узлам тот тип, который нужен?

В целом, картина такая: анализатор идёт от корня дерева к его листьям поиском в ширину. При этом для каждого встретив-

Листинг 7

```
SUBTYPING RULE Integer_extends_int
APPLICABLE FOR > Integer < as integerType
DO {   return < int > ; }
```

шегося узла он применяет все правила, которые к нему применимы. Во время обхода дерева, правилами создаётся некоторое количество уравнений и неравенств.

Например, уже рассмотренное нами правило для *VariableReference* (см.выше) создаёт для каждого узла, к которому было применено, по уравнению: TYPEOF (*varRef*) :==: TYPEOF (*varRef.variableDeclaration*). В момент создания уравнения как тип использования переменной TYPEOF (*varRef*), так и тип её декларации TYPEOF (*varRef.variableDeclaration*) может быть неизвестен. В таких случаях в уравнение вместо конкретных типов входят так называемые типовые переменные. Так что в данном случае как в правой, так и в левой части уравнения может оказаться типовая переменная (и на самом деле окажется!).

Итак, во время проверки типов в каждый момент времени в анализаторе находится некоторое количество уравнений и неравенств, в которые могут входить как конкретные типы, так и типовые переменные. И эту систему уравнений и неравенств надо как-то решать.

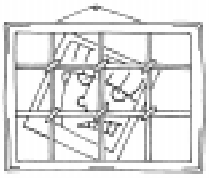
Разберёмся сначала с уравнениями. Если в уравнение входят два конкретных типа, то они сравниваются, и если они не совпадают, то система типов выводит сообщение об ошибке типизации, а если совпадают, то уравнение просто выкидывается. Также типы могут совпадать частично. Имеется в виду тот случай, когда какой-то узел внутри одного типа соответствует типовой переменной внутри другого типа, а в остальном типы совпадают – например *List< T >* и *List < String >*, где *T* – типовая переменная. В таком случае исходное уравнение выкидывается, а для таких детей создаются новые уравнения, в нашем случае *T* :==: *String*.

Если же в уравнении в одной из частей стоит типовая переменная, то решается оно так: во все уравнения и неравенства, где эта переменная используется, подставляется другая часть уравнения, то есть то, чему эта переменная равна. Описанный алгоритм достаточно известен и называется алгоритмом Робинсона или алгоритмом «W».

Теперь про неравенства. Если в неравенство входят два конкретных типа, то проверяется, является ли один тип подтипом другого, и если нет, то система типов выводит сообщение об ошибке типизации, а если является, то уравнение просто выкидывается. Также может случиться, что среди надтипов одного типа находится тип, лишь частично совпадающий с другим. Тогда, как и в случае с уравнениями, создаются уравнения для детей. Например, при решении неравенства *List< String >* :<=: *Iterable< T >* создастся уравнение *String* :==: *T*. Если же в неравенство в одну из частей входит переменная, то неравенство сохраняется внутри анализатора до тех пор, пока обе части неравенства не станут в результате подстановок переменных конкретными типами.

Может случиться и так, что когда анализатор уже обошёл всё дерево, в некоторых неравенствах правой или левой частью всё ещё является типовая переменная. Тогда анализатор делает следующее: в ситуации, когда *T* :<=: *type*, где *T* переменная, а *type* – конкретный тип, вместо *T* везде подставляется *type*. В ситуации же, когда *type1* :<=: *T*, *type2* :<=: *T*,.. *typeN* :<=: *T*, вместо *T* подставляется наименьший общий надтип типов *type1*, *type2*,.. *typeN*. Такая несимметричность объясняется тем, что, в отличие от наименьшего общего надтипа, наибольший общий подтип считается весьма трудоёмко, почти всегда не существует и практически нигде не нужен.

Таков в общих чертах тот механизм, который превращает сформулированные разработчиком системы типов правила в работающий анализатор типов.



4. ЗАКЛЮЧЕНИЕ

Итак, мы рассмотрели язык для написания анализаторов типов: язык HELGINS, поговорили про то, зачем и кому он нужен, обсудили пользу языкоориентированного подхода к программированию, посмотрели, как писать системы типов на HELGINS, и даже немного кос-

нулись реализации. Основными достоинствами языка HELGINS являются его краткость – правило занимает обычно 3–4 строчки, если не считать скобочек; простота – не требуется особого умственного усердия и дополнительных знаний, чтобы начать на нём писать анализаторы типов; и выразительность – программист выражает именно свои мысли про то, каковы у какого узла должны быть типы, а не лезет разбираться в сложной машинерии алгоритмов вывода. Разработчику системы типов не требуется писать свой алгоритм вывода каждый раз, когда он разрабатывает новый язык – всё уже за него один раз написано, ему нужно просто нарастить «мясо» своих собственных правил на универсальный «скелет», обеспечивающий вывод.

Платой за такую универсальность и основным существенным недостатком HELGINS является то, что написанные на нём анализаторы типов пока уступают в производительности анализаторам, написанным с нуля руками под конкретный язык. Впрочем, это не очень страшно, ибо производительность всё равно вполне удовлетворительная.

Главное, что HELGINS позволяет быстро и удобно писать системы типов, что помогает быстро и удобно создавать подходящие языки. Подходящие языки позволяют создавать программы быстрее и удобнее. А когда программист может воплощать свои мысли быстро и удобно, тогда и работа доставляет ему удовольствие. Это и есть самое главное.

Конопко Кирилл Сергеевич,
аспирант кафедры Системного программирования математико-механического факультета СПбГУ.