Глава из диссертации „Using Oberon's Active Objects for Language Interoperability and Compilation“

Patrik Reali *

2003 г.

Критика языка Oberon

В данном разделе обсуждается опыт, полученный во время использования языка Active Oberon. Мы обращаем внимание на слабые места языка и предлагаем некоторые улучшения. Многие проблемы унаследованы от языка Oberon, т.к. Active Oberon является его обратно-совместимым расширением, и мы не могли устранить их.

Мы считаем, что язык Oberon недоопределён. Оригинальное сообщение о языке [7] чрезвычайно короткое, автор полагается на здравый смысл и интуицию читателя. В этом есть определенное преимущество — язык проще изучить (меньше новых правил, используются известные правила), но, с другой стороны, в языке остаются „белые пятна“.

Сообщение определяет синтаксис и семантику языка. Синтаксис является основой языка, показывает как конструировать программы, в то время как семантика придает смысл программе. Нашу интепретацию сообщения о языке можно выразить следующими словами: *что разрешено синтаксисом и не запрещено семантикой*. Если синтаксис разрешает и семантика не запрещает, то это является допустимым.

Хотя список критических замечаний выглядит длинным, большинство из них являются проблемой только для создателей компилятора и не представляют проблемы для пользователей языка. В сравнении с критикой других языков [2, 5] наши замечания кому-то покажутся скуд-

*Перевод Андреева М. В.

ными, но они могут помочь лучше понять язык. Мы делаем вывод, что Oberon спроектирован очень хорошо.

Если не указано явно, далее термин Oberon относится одновременоо как к языку Oberon, так и к языку Active Oberon.

1 Синтаксис

1.1 Точка с запятой

Использование точки запятой в языке непоследовательно. Определения завершаются ею, в то же время для операторов она служит разделителем. Oberon часто критикуют за отличие трактовки точки с запятой от широко распространенных языков, поэтому, как минимум, следует обрабатывать её непротиворечивым способом.

Мы предлагаем везде использовать точку с запятой как разделитель, как это обычно ассоциируется с Oberon. На практике это дело вкуса.

1.2 Неоднозначный тип константы

Символьные константы и односимвольные строки имеют одну и ту же нотацию ("x"). Это является проблемой для компилятора, т.к. тип константы зависит от контекста, в котором она используется. Когда она передаётся как фактический параметр в `ARRAY OF SYSTEM.BYTE` или используется приведение типа при помощи `SYSTEM.VAL`, тип константы не может быть определён.

Есть три возможных решения данной проблемы: 1) использовать различную нотацию для символьных констант и строк, 2) сделать константу `CHAR` совместимой со строками, и 3) сделать односимвольные строки совместимыми с константами `CHAR`. Первое решение самое лучшее, но выглядит непрактичным, т.к. придется переписывать большое количество программ; мы считаем, что "x" должно трактоваться как строка, и что строки длины 1 должны быть совместимы с `CHAR`.

1.3 Неоднозначные правила вывода в грамматике

Грамматика Oberon содержит неоднозначные правила вывода, для правильного разбора которых требуются семантические знания:

qualified identifiers Парсер не может отличить квалифицированный идентификатор `M.p` и поле записи `r.f` в правиле `designator` [1]

[1] Эта проблема уже существовала в Modula [6]

type casts `x(y)` в правиле `term` может трактоваться как приведение типа значения `x` к типу `y`, либо как вызов функции `x` с параметром `y`.

factor `CharConstant` и `string` начинаются с одного и того же разделителя. Обычно сканер успешно обрабатывает данный случай, но для односимвольных строк конфликт остаётся.

constant expressions Правило `ConstExpression` требует семантическую информацию; его разбор не однозначен.

Данная нечистота дизайна является следствием использования нисходящего однопроходного парсера, в котором синтаксический и семантический анализы выполняются одновременно; семантическая информация доступна и подходящее правило может быть сразу выбрано. Тем не менее, это может быть проблемой в случае разделения синтаксического и семантического анализов, либо когда используются генераторы парсера подобные CoCo или Yacc.

2 Семантика

2.1 Числовые константы и свёртывание констант

Тип числовой константы равен минимальному типу, которому число принадлежит [7]. Тип результата арифметической операции равен типу операнда, который включает типы всех остальных операндов (за исключением деления). Функция приведения типа имее чётко определённую семантику.

Данные семантические правила достаточно прозрачны и имеют смысл для операций над переменными; в случае с константами легко получить неверный результат: например, объявим буфер размера `64 * 1024`, компилятор должен вычислить значение `0` или выдать ошибку переполнения; тип первого операнда `SHORTINT`, тип второго `INTEGER`, таким образом тип результата должен быть `INTEGER`; а так как значение результата слишком велико для этого типа, оно должно либо обнулиться, либо должна быть выдана ошибка переполнения. Это совершенно не то, что ожидает программист.

Применяя те же правила к функциям приведения типа очевидно, что `LONG` и `SHORT` изменяют тип операнда. Это означает, что предыдущее выражение должно быть записано как `64 * LONG(1024)`.

В языке есть правило `ConstExpression`, которое применяется при определении констант и размеров массивов; значение этих выражений используется во время компиляции, поэтому требуется свёртывание констант. Мы исследовали согласованность этих правил в различных компиляторах Oberon [2]: все из них применяют оптимизацию свертки констант в определенной степени; они вычисляют значение константного выражения во время компиляции и в коде используют только результат. После выполнения свёртки нет необходимости приспосабливать компилятор к семантике языка.

В ooc [3] все числовые значения нетипизированы, тип назначается только когда константа используется в неконстантном выражении. Данная стратегия позволяет выполнять арифметические операции независимым от типа способом [3]; приведение типа теперь эквивалентно тождественной функции и не оказывает влияние на значения: запись в память произвольной константы при помощи полиморфной функции `SYSTEM.PUT` трудная задача, т.к. тип константы зависит от её значения и `SYSTEM.PUT` вынуждена подстраиваться; как вариант, константу можно записать в переменную, а затем перемнную записать в память.

OP2 и Paco применяют другой подход. Числовые константы типизируются, но арифметические операции выполняются в наибольшем типе; тип результата назначается в зависимости от его значения. Это позволяет избежать переполнения при свёртке констант [4], но необходимо использовать функции приведения типа констант для создания низкоуровневых программ.

Есть четыре возможных решения этой проблемы: 1) сохранить текущую семантику языка и распространить её на константные выражения; 2) правило `ConstExpression` должно вычисляться вне системы типов; 3) явно указывать тип констант; 4) оставить в системе только один целочисленный тип.

Мы убеждены, что (1) будет сбивать с толку: результат может не совпасть с тем, что ожидает пользователь [5]. Предложение (2) вводит две различных семантики для одного синтаксиса, и соответственно тоже вводит в заблуждение, несмотря на то, что его выбрали для свёртки констант в компиляторах ETH. Предложение (3) может решить проблему практичным способом; многие другие языки добавляют информацию о типе к константам; Oberon делает это в некоторых случаях: для шестнадцатеричных символьных констант, для шестнадцатеричных целочис-

[2] OP2, Paco, oo2c, и Wirth's ARM компилятор

[3] На практике константы выражаются в наибольшем числовом типе `LONGINT`

[4] Ошибка всё равно возникает, если значение не помещается в `LONGINT`

[5] Запись `64 * LONG(1024)` не практична

ленных констант и для вещественных констант. Оно решает проблему элегантным способом, т.к. прекрасно вписывается в семантику языка. Предложение (4) тоже привлекательно, оно упрощает язык, компилятор и устраняет проблему; язык Oberon-0, который использовался Виртом в лекциях по конструированию компиляторов, полагается только на типы `INTEGER` и `BOOLEAN`, что достаточно для многих программ. Недостаток может проявиться только в низкоуровневом программировании, где оборудование требует значения определенного размера. Отображение структуры оборудования напрямую в стуктуру Oberon значительно усложнится, регистры с размером меньше чем машинное слово должны будут трактоваться специальным способом. Возможное решение в замене полиморфных функций `PUT` и `GET` модуля `SYSTEM` на не полиморфные функции (`PUT8, PUT16, PUT32, GET8, GET16, GET32`). Мы уже вводили такие функции для уменьшения путаницы, созданной неявно типизированными константами при вызове полиморфных функций; опыт работы с ними был весьма удовлетворительный.

Для сравнения, Java использует другой подход: любая арифметическая операция использует тип `int` [6]; сумма двух значений типа `short` будет иметь тип `int`. Для присвоения переменной меньшего типа нужно воспользоваться приведением типа. Это устраняет элегантным способом проблему для Java, но Oberon активно использует все три целочисленных типа, и данное решение сделало бы несостоятельным большое количество существующего кода.

2.2 Область определения и блоки

В Oberon идентификатор может указывать только на один объект в области своего определения. Это может ввести в заблуждение, т.к. область определения не соответсвует блоку: она простирается от самого определения до конца текущего блока. Это означает, что внутри блока идентификатор может иметь два значения: между началом блока и определением (наследуется от внешнего блока), и другое значение от начала определения до конца блока.

Определение указателей, в частности, подчёркивает данную проблему: если тип `T` определён как `POINTER TO T1`, тогда `T1` может быть определён текстуально позже в области определения `T` [7].

[6] Есть одно исключение: если хотя бы один оператор имеет тип `long`, то все операции будут использовать тип `long`

[7] Это открывает ещё одну дыру, т.к. в области определения могут быть вложенные блоки и `T1` может быть определён в одном из них; это явно не запрещено

В случае, когда `T1` определён дважды, один раз во внешней области и один раз после `T` в том же блоке, сообщение о языке не уточняет, какое определение является более приоритетным. В нашей трактовке определения одного и того же блока должны иметь больший приоритет; опыт показывает, что большинство компиляторов Oberon реализуют другой вариант. Reali [4] первым сообщил о данной проблеме и привел соответствующий пример.

В Active Oberon удалось решить проблему путём расширения области определения до целого блока.

2.3 Обработка ошибок и исключения

Ошибки и исключения — белые пятна в сообщение о языке: их описание часто недоопределено или вообще отсутствует. Очевидно, разыменование указателя со значением `NIL` должно приводить к возникновению исключения времени выполнения, и компиляция семантически некорректных программ должна отслеживаться компилятором.

В некоторых случаях в отчёте определяется, что операция некорректна, но не говорится является определение корректным или нет, и может ли ошибка проявиться во время исполнения или только во время компиляции.

Например, в массиве доступны только *элементы с индексом от 0 до ДЛИНА МАССИВА - 1*. Это означает, что массивы отрицательного и нулевого размеров бесполезны, т.к. невозможно обратиться к их элементам (и не имеют смысла). Семантика языка не запрещает массивы отрицательного размера, и мы делаем вывод, что они должны быть допустимы.

Различные компиляторы по разному трактуют эту возможность, делая программы непереносимыми. OP2 и Paco сообщают об ошибке во время компиляции, а oo2c добавляет исключение времени выполнения в код.

Мы нашли различия в трактовке следующих элементов (в языке они не определены):

- определение массивов нулевой или отрицательной длины
- размещение в памяти массивов нулевой или отрицательной длины
- `ASSERT(FALSE)`
- обнаружение и обработка ошибок переполнения

3 Дизайн

3.1 Правила видимости

Active Oberon принуждает к более объектно-ориентированному стилю программирования чем Oberon; это воздействует на правила видимости и делает правила экспорта, ориентированные на модули, отчасти неадекватными.

Из-за наследования поля и методы суперкласса видимы без указания модуля, в котором они определены. Другая проблема в том, что все процедуры модуля могут изменять значения полей объекта, хотя это должно быть позволено только методам этого объекта.

Active Oberon нуждается в новых модификаторах видимости, которые делают поля и методы объекта видимыми только в подклассах.

Несмотря на то, что в нём нет острой необходимости, модификатор только для чтения, добавленный в Oberon-2, доказал свою эффективность. Reali [4] рекомендует его использовать, и объясняет в деталях причины его переноса в Active Oberon.

3.2 Инициализаторы Active Oberon

На данный момент есть две нотации инициализаторов Active Oberon, одна предложена в этой главе, другая — в Oberon.NET [1], где все методы, названные `NEW`, автоматически становятся инициализаторами.

В нашем варианте каждый метод, помеченный "&", является инициализатором; так мы исключаем перегрузку методов из языка. Таким образом, каждый инициализатор должен иметь уникальное имя внутри класса. С другой стороны, инициализаторы могут использоваться как обычные методы.

В Oberon.NET все инициализаторы именуются "`NEW`". Что легко читается, и нет конфликтов, т.к. в языке нет подклассов. Проблема возникает при использовании `NEW`, предопределённой в языке процедуры, которая имеет другое назначение: это мешает явно вызвать инициализатор, и требует сделать исключение в правилах видимости, определяя встроенный `NEW` всегда видимым, иначе он будет перекрыт инициализатором.

Т.к. Active Oberon использует наследование, нотация Oberon.NET потребует внести в язык множество исключений, что создаст неопределенность и несовместимость.

Мы считаем, что инициализаторы должны быть легко опознаваемы (и иметь предопределённое имя), и должна быть возможность вызвать

их как обычные методы. Подход, подобный подходу в Java и C# , когда имя инициализатора совпадает с именем класса, кажется неплохой идеей.

3.3 Встроенные типы и константы

Встроенные в Oberon типы и константы — предопределённые идентификаторы. Только ключевое слово `NIL` является исключением. Как любой идентификатор, предопределённый идентификатор может быть перекрыт локальным определением; программист может переопределить его. Это весьма необычно и очень опасно. Вирт уже понял это, но по некоторым причинам воздержался от перевода таких идентификаторов в ключевые слова.

Вот пример, на который ссылается Вирт, показывающий разрушительные последствия такого решения:

```
CONST TRUE = FALSE;

VAR b: BOOLEAN;

BEGIN
  b := FALSE;
  IF b THEN ... END; (* условие ложно *)
  IF b = TRUE THEN ... END; (* условие истинно *)
END
```

Мы считаем, что встроенные типы и константы должны быть ключевыми словами языка.

4 Другие предложения

4.1 Разделение SYSTEM

Модуль `SYSTEM` традиционно используется как хранилище небезопасных и низкоуровневых функций, встраиваемых компилятором в код. Мы считаем, что было бы выгодно разделить этот модуль на две части: модуль `SYSTEM` или `UNSAFE`, содержащий небезопасные, но переносимые функции, и модуль, содержащий функциональность, доступную только на специфичной платформе, который должем называться именем платформы (например, `INTEL, MAC, UNIX`).

Это бы упростило переносимость компилятора и кросс-платформенную компиляцию, т.к. каждый back-end определял бы свой низкоуровневый модуль вместо переопределения всего `SYSTEM`, как это происходит сейчас. Так же это упростило бы переносимость системы — можно определить, какие модули небезопасные, а какие непереносимые.

Список литературы

[1] J. Gutknecht. Active Oberon for .NET. http://www.oberon.ethz.ch/oberon.net/whitepaper/, June 2001.

[2] B. Kernighan. Why Pascal is not my Favourite Programming Language. Technical Report 100, Bell Labs, 1983.

[3] ooc, Optimizing Oberon-2 Compiler. http://ooc.sourceforge.net.

[4] P. Reali. Structuring a Compiler with Active Objects. In J. Gutknecht and W.Weck, editors, Proceedings of JMLC, volume 1897 of LNCS, pages 250–262, Zurich, Switzerland, 2000. Springer.

[5] H. Thimbleby. A Critique of Java. Software - Practice and Experience, 29(5):457–478, 1999.

[6] N.Wirth. MODULA : A Language for Modular Multiprogramming. Software Practice and Experience, 7:3–35, 1977.

[7] N. Wirth. The Programming Language Oberon. Software Practice and Experience, 18(7):671–690, July 1988.