COMPARISON OF OPPORTUNITY MODULARIZATION IN FUNCTIONAL AND IMPERATIVE PROGRAMMING STYLES

I. N. Skopin

Institute of Computational Mathematics and Mathematical Geophysics SB RAS,
630090, Novosibirsk, Russia

The article "Comparison of opportunity modularization in functional and imperative programming styles" discusses the issues of expressiveness of the language means proposed for the two programming styles. The solution to these issues is relevant in connection with the preparation for the transition in the near future to the efficient use of ultra-high power computing equipment: the programmer has the opportunity to build calculations using a very large number of processors and cores.

Today, interest in functional computing has grown very much, and a dangerous trend, very typical for the development of computer science, has appeared: one can consider an unconventional functional style as a panacea for all programming problems in the imperative style update. In this regard, *one of the main goals of the article* being proposed is to dispel the myth of the universality of the functional style and find for it, as well as for the imperative style, an adequate place among the methodological approaches to solving programming problems.

This goal is specified as a comparison of means of supporting imperative and functional modularization. The basis of this comparison is the assertion that the gain in the transition from the traditional style to the new is not in what you have to give up, but in a new quality, which the new style gives in comparison with the old one (see D. Hughes article "Why Functional Programming Matters"). With regard to the transition from an imperative to a functional model of computing, this means an answer to the question of language programming tools that provide new qualities of modularity. First of all, these are tools that allow one to implement lazy calculations and handling functions of high orders, as well as a mechanism for eliminating re-counting, called memoization.

The approach to achieving the goals is based on the use of formally defined operations of an abstract computer in the execution of individual constructions of an imperative and functional program. This allows somebody to consider language constructs as templates for connecting parts of a program. As a result, it is possible to uniformly describe the development and use of modules for both the imperative and functional model of computing. One of the main consequences of this approach is the ability to accurately specify the boundaries of the adequate applicability of imperative and functional modularization, as well as the conditions for the correct joint combined use of the means of these two styles. The proposed method is in good agreement with the concept of mixed computing, in that aspect, which back in 1977 A. P. Ershov quite accurately called the essence of translation.

The exposition of imperative and functional modularization completes the list of losses that a programmer should not forget about when switching from using a traditional model of calculations to a functional one:

— Memory passivity cannot be expressed in a functional style.

— The notion of states of a computational process, which in many cases give its natural decomposition, is lost in a functional language.

— The concept of the context of calculations in a functional language becomes significantly narrower than when working in an imperative style: for and it is important that the contexts are organized hierarchically.

— Managing the ordering of computations over time is contrary to functionality.

All these losses are somehow connected with the concept of common (but not global!) For different data modules for which a functional language cannot offer adequate means of expression. This explains the success of mixed modularity, when the functional parts of the program are invoked for execution within the framework of the operating environment in which they are executed as modules independent of the environment.

Key words: modularization, imperative programming, functional programming, computing model, abstract calculator, memoization, style of programming.

References

1. Asanovic K. et al. The landscape of parallel computing research: a view from Berkeley. Technical Report No. UCB/EECS-2006-183. Berkeley: University of California, EECS Department, December 18, 2006. [Electron res.]: `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf` (access date: 24.05.2019).

2. Backus J. Can Programming be Liberated from von Neumann style? A Functional Style and its Algebra of Programs. Comm. ACM, 21, 1978.

3. McCarthy. 91 function — Wikipedia. [Electron res.]: `https://en.wikipedia.org/wiki/McCarthy\_91\_function\#Knuth.27s\_generalization` (access data: 24.05.2019).

4. Gorodnjaja L. V. Fundamentals of functional programming. Lectures. Moscow: Internet University of Information Technology, 2004. ISBN 5-9556-0008-6.

5. Hughes J. Why Functional Programming Matters // Computer Journal. N 32 (2). 1989.

6. Nepejivoda N. N., Skopin I. N. Foundations of Programming. Moscow-Izhevsk: RChD, 2003.

7. Dahl O. J., Dijkstra E. W., Hoare C. A. R. Structured Programming. Moscow: Mir, 1975.

8. Liskov B., Guttag J. V. Abstraction and Specification in Program Development (MIT Electrical Engineering and Computer Science). The MIT Press, 1986, ISBN-10: 0262121123.

9. Stroustrup B. What is Object-Oriented Programming? // IEEE Software. 1988. V. 5 (3).

10. Kosvichenko A. Zachem ge nugna virtyalizacija? [Electron res.]: `https://habrahabr.ru/post/91503/` (access data: 24.05.2019).

11. Sylvan S. Why does Haskell matter? [Electron res.]: `http://www.dtek.chalmers.se/~sylvan/haskell/why_does_haskell_matter.html` (access data: 24.05.2019).

12. Ershov A. P. O sushchnosti translyacii. Preprint N 6, Novosibirsk: VC SO AN SSSR, 1977.

13. N. Winstanley. What the hell are Monads? 1999. [Electron res.]: `http://www.abercrombiegroup.co.uk/~noel/research/monads.html` (access data: 24.05.2019).

14. Keene S. E. Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. Addison-Wesley (Reading, Massachusetts, 1989).

15. Skopin I. N. An Approach to the Construction of Robust Systems of Interacting Processes // In: Parallel PROGRAMMING: Practical Aspects, Models and Current Limitations. NOVA science publishers. Series: Mathematics Research Developments. Editor: M. S. Tarkov. 2014, ISBN: 978-1-63321-957-1.

16. Cejitin G. S. Na puty k sborochnomu Programmirovaniju // Programmirovanie. 1990. N 1. P. 78–99.

О ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ И МОДУЛЬНОСТИ

И. Н. Скопин

Институт вычислительной математики и математической геофизики СО РАН,
630090, Новосибирск, Россия

УДК 004
DOI: 10.24411/2073-0667-2019-00008

В связи с грядущим переходом к использованию экзафлопсных вычислителей, актуальна разработка методов программирования для нетрадиционных моделей вычислений, допускающих выполнение на очень большом числе процессоров. В этом плане весьма перспективной представляется функциональная модель, возможности которой по отношению к модуляризации программ, обсуждаются в сопоставлении с модульностью в императивных языках. Показана несостоятельность претензии на универсальность как функционального, так и императивного стилей — каждый из них имеет свою область адекватного применения.

Ключевые слова: модуляризация, императивное программирование, функциональное программирование, модель вычислений, абстрактный вычислитель, мемоизация, стиль программирования.

Введение. Современный этап развития программирования характеризуется нарастающей потребностью подготовки к переходу в недалеком будущем к эффективному использованию сверхвысоких мощностей вычислительной техники. Этот переход требует пересмотра большинства сложившихся решений в сфере высокопроизводительных вычислений и поддерживающего их системного программирования. Он связан с необходимостью преодоления комплекса технологических, а также социальных проблем, с формированием новой вычислительной парадигмы, обусловленной тем, что в распоряжении программиста появляется возможность строить расчеты, использующие очень большое число процессоров и ядер.

Задачи, возникающие при подготовке к эре экзафлопсных вычислений, т. е. квинтиллион (10^{18}) операций в секунду, рельефно представлены в отчете группы исследователей университета Беркли [1]. Среди них методы разработки алгоритмов, рассчитанных на параллельное выполнение, рассматриваются как наиболее актуальные. Эти методы включают распараллеливание последовательных программ, в частности, при автоматической генерации параллельного кода, а также конструирование программ, параллелизм которых задается на уровне модели вычислений. Среди последних особое место занимают разработки, связанные с нетрадиционными вычислительными моделями и, в первую очередь, с функциональной моделью, которую еще в середине семидесятых годов Джон Бэкус в своей тьюринговской лекции провозгласил в качестве перспективной альтернативы фон Неймановской модели. [2]. Он последовательно указывал на ограниченность этой модели и, что существенно, на следствия ее ограниченности, проявляющиеся в языках программирования.

Сегодня интерес к функциональным вычислениям очень вырос. Стали появляться новые функциональные языки, которые наряду с Лиспом заняли свое место в арсенале инструментов программирования. Одновременно накапливается опыт, а вместе с ним — типовые приемы программирования в функциональном стиле. Как следствие, стало возможным глубокое сопоставление императивных и функциональных стилей. Теперь уже недостаточно простых примеров вроде функции F91[1], подтверждающих выразительности нетрадиционных языков. Появляются иллюстрации, которые показывают, что функциональные языки обладают такими качествами, которые оказываются полезными в такой, казалось бы, выигрышной для традиционных стилей прикладной области как вычислительные алгоритмы. В первую очередь это возможность разработки алгоритмов с высокой степенью параллелизма, которая в связи с грядущим переходом к экзафлопсным вычислениям наверняка станет и уже становится одним из решающих факторов роста реального применения функциональных вычислений.

Отмечая привлекательность и несомненные достоинства функциональной модели, нельзя не указать на то, что все чаще высказываются претензии функционального стиля на универсальность, а это плохой симптом, который означает только то, что недостаточно четко определены границы применимости этого нетрадиционного стиля. В предлагаемой работе мы намерены несколько развеять эйфорию, обусловленную преимуществами функциональности по сравнению с традиционными стилями программирования. В первую очередь это касается задачи конструирования программы как модульной системы, допускающей построение из автономных частей, каждая из которых реализует определенный аспект вычислений.

Модуляризация как важнейший аспект технологий программирования во многом перекликается с таким не менее важным фактором успешности развития отрасли как преподавание. В упомянутой выше лекции Бэкуса среди проблем перехода к более выразительным моделям вычислений отмечалось, что сегодня активно трудится огромная армия программистов, которые привыкли думать в императивном стиле и уже не в состоянии переучиваться. Обучая, они готовят своих последователей, также неспособных к программированию в новых стилях. Эта проблема не является характерной только для информатики и программирования: преодоление привычек и поведенческих стереотипов всегда требует определенных усилий, а без уверенности в том, что переход к новому даст существенные преимущества, необходимая трата сил и средств рассматривается как расточительство. В результате — неспособность выйти за круг сложившихся понятий и, что также существенно, используемых инструментов.

Задачи преподавания программирования с использованием перспективных моделей вычислений естественным образом подразделяются на определение учебного материала, т. е. того, *что* является предметом обучения, и построение эффективной с дидактической точки зрения методики, т. е. того, как *как* этот предмет должен быть представлен в одной или нескольких учебных дисциплинах. Понятно, что без определения предмета говорить о методике обучения бессмысленно.

[1] Функцию $F91$ предложили Зохар Мана, Амир Пнуэли и Джон Маккарти еще в 1970 году [3] для проверки качества методов автоматической верификации. Она определяется для целых чисел следующим образом:

$$F91(n) = \text{ if } n > 100 \text{ then } n - 10 \text{ else } F91(F91(n + 11))$$

Для всех $n \leqslant 100 F91(n)$ равна 91, а последовательность $101, 102, \ldots$ отображает в $91, 92, \ldots$

Следуя только что представленному утверждению, в данной работе мы говорим только о том, что нужно знать при переходе к активному использованию функционального стиля в программировании, оставляя задачи разработки методик преподавания для дальнейших исследований. Более того, не претендуя на систематическое изложение средств функционального программирования, мы показываем лишь те особенности оперирования этого стиля, которые определяют его возможности модуляризации программ (систематическое изложение средств функционального стиля можно найти в работе [4]).

1. Мотивация. Одним из поводов для данной работы послужила статья Джона Хьюза „Сильные стороны функционального программирования“ [5][2], в которой правильно говорится, что выигрыш от нового стиля не в том, от чего приходится отказываться, если сравнивать его с традиционным программированием, а в новом качестве, которое дает функциональность. Далее приводится аналогия ситуации со структурным программированием, преимущества которого не в отказе от оператора **go to**, а в доведении идеи модульности до практического использования. Чтобы не уклоняться от темы, не будем оценивать это мнение автора, заметив вскользь, что своя модульность достижима и на уровне Фортрана, и она хорошо работает на своем месте [6]. Более интересно обсудить функциональные средства, которые Хьюз провозглашает как развитие модульности программ, недостижимое в любом императивом стиле. К ним относятся возможность оперирования функциями высших типов и ленивые вычисления.

Указанные возможности Хьюз уподобляет клею в столярных работах. Если бы не было клея, то, к примеру, стул пришлось бы вырезать из цельного куска дерева, что неизмеримо сложнее, чем предварительная подготовка деталей, которые затем собираются правильным способом. Так же точно применяются приемы комбинирования функций и ленивые вычисления, чтобы получать функциональные программы из заготовок. В статье приводятся убедительные примеры того, как это работает, как за счет комбинирования получается существенное ускорение и вычислительных итеративных, и переборных алгоритмов. В частности, с помощью разделения того, что в императивном стиле было бы названо заголовком и телом цикла, можно эффективно оперировать потенциально бесконечными структурами. Соответствующие приемы можно корректно сочетать, и в результате получается очень выразительное программирование.

Что касается наглядности, то здесь можно поспорить: иногда она достигается, но в ряде случаев комбинации запутывают читателя функциональной программы настолько, что с трудом удается увидеть, как получаются требуемые результаты. Быть может, это связано с непривычностью стиля, с не очень очевидной нотацией, но здесь хотелось бы обсудить другой вопрос, который имеет прямое отношение к методам разделения задач на подзадачи. Речь о том, можно ли упомянутые приемы считать средствами модульности? Для ответа стоит обратиться к тому определению модуля, которое является достаточно общим (оно не должно зависеть от стиля программирования) и интуитивно понятным.

Вполне правомерно отнести модульность к средствам абстрагирования. Эта точка зрения соответствует взгляду на модульность со стороны структурного программирования [7], концепции абстрактных типов данных [8], а также объектно-ориентированного программирования [9]. Она отражает реальную потребность практики конструирования больших программных систем: при использовании модуля знать о том, как он реализован, не только ненужно, но и вредно, а при реализации модуля не нужно знать ничего об ис-

[2]Статья Хьюза в оригинале называется „Why Functional Programming Matters“.

пользовании, кроме спецификаций, фиксирующих решаемую задачу. Такая модульность предполагает, в частности, осуществимость подмены реализации.

Приведенная трактовка сразу же выводит модули на уровень программных текстов, причем содержательно осмысленных. Только для таких текстов, а не для отдельных конструкций языка, можно говорить об их спецификациях, в соответствии с которыми они составляются и используются. Если обратиться к метафоре клея для производства надежного стула, необходимо, чтобы составляющие его детали были бы хорошо подогнаны друг к другу, а это уже зависимость компонент, т. е. налицо нарушение главного принципа модульности.

Как для модулей, смысл которых сводится к выполнению отдельной процедуры или функции, так и в случае модулей, которые объединяют в себе, к примеру, средства целой оконной системы, существенным слагаемым качества является интерфейс. Не будет большим преувеличением утверждение о том, что развитие модульности есть развитие способов задания интерфейса. Что же касается выразительности описания алгоритмов, то в аспекте модульности она вторична по отношению к понятности и достаточности интерфейса для разделения уровней реализации и использования. Средства, которые предлагаются функциональным стилем программирования, в этом отношении ничем не помогают. Их можно и нужно рассматривать как способ выразительного задания реализаций, и не стоит приписывать им качества, на которые сама по себе модель вычислений претендовать не должна.

2. Модули как шаблоны для соединения частей программного текста. Текстовая трактовка модулей, на которой мы настаиваем, апеллирует к статике. В то же время, клей функционального стиля — способ динамического *соединения частей* или *склеивания* их. Именно за счет этого и можно манипулировать понятием потенциально бесконечных функциональных структур, части которых образуются по мере необходимости их для обработки. Действительно, по сравнению с функциональными императивные средства склеивания частей очень бедны: есть базовые действия и несколько управляющих конструкций, семантика которых определена операционально, есть шаблоны, которые используются для задания конструкций вызовов процедур и императивных функций, и все. Эта бедность связана с установкой на статичность вычислений, которая принимается для императивного языка на концептуальном уровне.

Императивное соединение частей наследует способ управления машины фон Неймана, который по сути своей является статическим, привязываемым к конкретным адресам. Относительная адресация — это просто использование двух ступеней обращения к вполне статичным относительно друг друга адресам. Косвенная адресация хотя и дает возможность привнесения динамики в управление, но только посредством планирования вычисления адреса. И трудности оперирования с указателями обусловлены именно тем, что изначально бедные возможности оперирования с адресами приходится приспосабливать к потребности динамического доступа. Пассивность памяти и активность лишь одного элемента модели (процессора) требует решения многочисленных задач использования адресов в разных смыслах, что на уровне модели препятствует выразительной реализации динамики.

Уже из этого следует, что склеивание частей функциональной программы имеет все основания быть более выразительным по сравнению с императивным соединением. Однако, когда для описания обработки данных достаточны статические или сводимые к статическим средства, преимущества в выразительности не проявляются, а накладные расходы

на реализацию динамического выбора соединений возрастают. Это условие делает явными границы применения функционального стиля для современных архитектур, вместе с тем, имея все основания предполагать от перспективных архитектур ослабление указанного недостатка и, как следствие, расширение границ адекватного применения функциональности. В связи с этим уместно указать на то, что и в рамках сложившихся архитектур вычислительных систем предлагаются решения, которые повышают выразительность программирования. Имеются в виду средства виртуализации, которые на уровне приложения позволяют не особенно заботиться об экономии процессорных ресурсов [10].

Если же обратиться к соединениям частей императивной и функциональной программ без привязки к смыслу действий, которые они выполняют, то оба варианта вполне могут быть выражены в одних и тех же понятиях, описывающих поведение абстрактного вычислителя соответствующего языка.

Например, условный оператор

```
if <условие> then <действие T> else <действие F>
```

можно рассматривать как шаблон для соединения трех языковых конструкций, которые естественно считать формальными параметрами этого шаблона. Оперирование с конкретизацией такого шаблона распадается на *извлечение параметров*:

(1) `<условие>` — произвольное выражение, вырабатывающее логическое значение, и
(2) `<действие T>`,
(3) `<действие F>` — произвольные операторы.

и *выполнение семантических действий*, предписываемых для данной конструкции.

Точно также обстоит дело с функциональными соединениями. Например, для произвольной функции, определенной для двух аргументов, можно задать шаблон вычисления ее для списка с помощью соотношения[3]:

```
reduce f x () = x
reduce f x (a : l) = f a (reduce f x) l
```

Здесь заданы три аргумента функции reduce:

(1) `f` — произвольная функция;
(2) `x`, — значение, резервируемое для результата вычислений над пустым списком;
(3) `(<элементы списка>)` — произвольный список элементов, у которого выделены и соответствующим образом обозначены голова (a) и хвост (l).

Эти аргументы служат параметрами шаблона, которые, как и в императивном случае, должны *извлекаться* для *выполнения семантических действий* (которые определяются выражениями после знака равенства).

[3]Здесь и далее для изображения списков и других средств функционального программирования мы используем нотацию языка Haskell [11], который сегодня претендует на роль стандарта в области чисто функциональных языков.

В частности, конкретизация f как операции (*) и x — как константы 1 при оставлении третьего параметра свободным приводят к функции перемножения всех элементов списка, для которой можно задать шаблон-определение уже с одним лишь параметром:

```
product = reduce (*) 1
```

Разница императивных и функциональных шаблонов соединения частей заключается в следующем:

1) Подобные шаблоны в императивном случае заданы заранее, тогда как в функциональном языке они могут произвольным образом комбинироваться для получения новых шаблонов (как это сделано для функции product).

2) В функциональном соотношении можно различать ситуации, когда требуются разные шаблоны в зависимости от того, какие параметры поставляются. В примере параметр-список содержит или не содержит элементов, и в зависимости от этого определяются разные вычисления. Это сродни императивному определению условного оператора сразу для двух вариантов: с **else** и без **else**.

3) Различается дисциплина взаимодействия извлечения параметров и выполнения семантики. В императивном случае считается, что извлечение не зависит от выполнения и, например, может быть проведено отдельно, тогда как в функциональном стиле они взаимозависимы (см., в частности, п.2).

4) В функциональном стиле активизация оперирования шаблоном не предполагает ожидание готовности всех его параметров. Вместо этого абстрактный вычислитель активизирует шаблон, когда его результат требуется для других действий. Поэтому действие с параметром начнет выполняться только тогда, когда этого требует некоторое вычисление, приостановленное из-за невозможности выполняться без получения этого параметра. Это соглашение называется принципом *ленивых вычислений*. Вообще говоря, он противоречит фон Неймановской однородности (пассивной) памяти, которая не в состоянии передавать информацию о том, готовы для вычислений данные в ячейке или нет. В результате при реализации ленивых вычислений в императивном языке приходится специально заботиться об определении моментов вычислений, когда наступает необходимость их выполнения.

В примере мы намеренно обратились к стандартно заданному условному оператору как шаблону-соединению. Конечно, более показательно было бы сопоставление функциональных шаблонов с вычислением императивного вызова процедуры, особенно в случае, когда язык допускает идентификацию вызываемой процедуры с помощью ее сигнатуры (а не только по имени), что часто называют полиморфизмом. Здесь соответствие оказалось бы более полной аналогией, поскольку алгоритм процедуры правомерно рассматривать в качестве определения того, какая семантика предписывается выполнению конструкции вызова процедуры. Но даже для стандартных императивных соединений конструкций, которые никто никогда и не думал называть модулями, видно, что содержательная семантика (определяемая алгоритмом задачи), как в императивном, так и в функциональном языке задается отдельно от правил извлечения семантики вызова и передачи ее на выполнение.

Стоит заметить, что для ранних языков программирования, которые разрабатывались, когда не были осознаны преимущества ленивых вычислений, они все-таки неявно планировались. Так, передача параметра по наименованию в языке Алгол 60 может рассматриваться как способ откладывания вычисления фактического параметра до того момента, когда вычисления тела процедуры не потребуют этого. Разработчики языка не пожелали обратить внимание на трудности, возникающие в данной модели вычислений, когда

есть неоднократное обращение к параметру, в результате чего при повторном вычислении фактического параметра может получаться другое значение. Эти трудности обусловлены сохранением значений в памяти и возможностью ничем не ограниченного использования хранимых значений, т. е. опять-таки базовыми принципами модели фон Неймана.

3. Извлечение и оперирование как основа соединения шаблонов. Чтобы убедиться в соответствии того, что должно происходить при соединении частей в императивных и функциональных конструкциях, рассмотрим приведенные примеры более подробно. Это рассмотрение исходит из выделения в вычислениях конструкций двух составляющих:

— извлечение конструкции и ее параметров — операция extract, и

— собственно вычисление как организация оперирования результатами вычисления параметров — операция evaluate.

Для условного оператора вычисление evaluate проводится в контексте действий, которые (вне примера) привели к его извлечению и передаче абстрактному вычислителю для работы. Этот контекст есть ничто иное как реализация запроса к обстановке, из которой извлекается очередной оператор. Точно по такой же схеме может выполняться вычисление функции product или reduce, извлекаемой и передаваемой для работы абстрактному вычислителю, но уже функциональному.

Для условного оператора выполняются следующие шаги:

`1. extract / <условный оператор> -> <условие>`

Эта нотация означает запрос к обстановке в контексте конструкции <условный оператор> с целью извлечения ее первого параметра. Если <условный оператор> уже извлечен, то вследствие синтаксиса конструкции результат активации операции extract обязательно будет получен.

`2. evaluate <условие>`

Возможны два варианта результата этого вычисления, которые приводят к различным продолжениям процесса.

Результат шага 2 есть True:

`3.1. extract / <условный оператор> -> <действие T>`

`4.1. evaluate <действие T>`

Результат шага 2 есть False:

`3.2. extract / <условный оператор> -> <действие F>`

`4.2. evaluate <действие F>`

Взаимодействие абстрактного вычислителя (операция evaluate) и программы, реализующей запросы к обстановке, (операция extract) может быть организовано многими способами и, в частности, с помощью ленивых вычислений, когда extract активируется в точности в те моменты, в которые результат запроса необходим абстрактному вычислителю (в точном соответствии с трактовкой ленивых вычислений в функциональном программировании).

Если бы мы захотели строить транслятор, то пришлось бы определить еще одну операцию, которая бы генерировала объектный код в качестве остаточной программы смешанного вычисления evaluate и extract при задержанных данных. Этот путь приводит к концепции и теории смешанных вычислений в том их аспекте, который еще в 1977 году А. П. Ершов весьма точно назвал сущностью трансляции [12].

Перейдем к соединениям частей при вычислении функций product и reduce. Как и в предыдущем случае считается, что функция уже извлечена для вычислений вне примера и передана абстрактному вычислителю. Но теперь для выполнения нужных действий необ-

ходимо дополнить семантику операции извлечения с тем, чтобы она работала с программно определяемыми фрагментами. Наиболее естественно это делается с помощью приема, который можно назвать *атрибутированием имен*. Суть его в том, что при обработке определения имени ему в качестве атрибута приписывается обозначаемое тело, в данном случае — задаваемой функции (точнее, не само тело, а некий результат его обработки, что несущественно для приводимой иллюстрации). Тогда операция extract, примененная к имени, должна извлекать для активизации тело определения функции из соответствующего атрибута имени. Некоторые детали этого процесса, связанные с параметрами, мы опускаем. При желании атрибутирование имен можно считать механизмами макрогенерации, образования элементов списка свойств Лиспа, реализации поддержки императивного вызова процедуры и др. Все эти и другие варианты используются достаточно давно, причем они практически не зависят от модели вычислений языка.

С использованием сделанного уточнения вычисление функции product можно описать следующим образом:

```
1. extract / product -> <определение product> = reduce (*) 1
```

В результате выполнения этого шага оказывается необходимым вычислять функцию reduce. Это содержание следующего шага:

```
2. evaluate reduce (*) 1
```

В рамках этого вычисления необходимо извлечение определения reduce:

```
2.1. extract / reduce ->
(<определение reduce 1> <определение reduce 2>)=
f x () = x
f x (a : l) = f a (reduce f x) l
```

Без привлечения аргументов разделить варианты невозможно. Следуя принципу ленивых вычислений, решение этого вопроса откладывается. Таким образом, для альтернативных вычислений evaluate может установить лишь фактическое *связывание параметров с аргументами*. f связывается с операцией (*), а x — с константой 1:

```
2.2. evaluate
(<определение reduce 1> <определение reduce 2>) (*) 1 =
(*) 1 () = 1
(*) 1 (a : l) = (*) a (reduce (*) 1) l
```

Есть определенная свобода в реализации этого шага, поскольку совсем не обязательно (а часто и вредно) производить явную подстановку и переписывание соотношений. Фактически нужно лишь обеспечить осуществимость дальнейших действий так, чтобы их результаты оказались идентичными эффекту, который достигается выполнением их при предварительно сделанной подстановке.

Следующий шаг — попытка извлечения третьего аргумента reduce:

```
2.3. extract / reduce -> (<элементы списка>)
```

Это действие будет заблокировано, или приостановлено, если перед шагом 1 извлечение product выполнено в контексте, который не содержит третьего аргумента. На ситуацию можно посмотреть по-другому, операционально: данный шаг осуществляет *запрос к обстановке*, и до тех пор, пока запрос не будет выполнен, продолжение действий блокируется.

Когда список получен, осуществляется выбор вариантов, который реализуется как совместное вычисление обоих определений функции reduce:

```
2.4. evaluate
```

`(<определение reduce 1> <определение reduce 2>)`
`(<элементы списка>)`

Совместность вычисления вариантов — принципиальный момент этого шага. Она означает, что к моменту блокировки или завершения этого вычисления (см. ниже) оба варианта независимо должны быть готовы к дальнейшим действиям: к запросам к обстановке или к передаче результатов для внешнего использования. При этом должно быть гарантировано, что вычисление одного варианта никак не влияет на вычисление другого. Как раз это условие противоречит операциональной модели вычислений императивного языка, в которой допустимо, что независимые процессы обращаются к общей памяти для записи значений.

Некоторые из совместно вычисляемых вариантов могут оказаться невыполнимыми (как в нашем случае: условия связывания параметров с аргументами разграничивают выполнимость двух ветвей вычислений). Для них должно быть обеспечено завершение без результата, т. е. соответствующие ветви вычислений ликвидируются. При корректно построенном функциональном вычислении, в конечном счете, всегда должен остаться единственный вариант, который дает результат.

Возможны два варианта результата, которые приводят к различным продолжениям процесса:

Результат шага 2.4 есть связывание третьего параметра с пустым списком:

`2.5.1. extract / reduce -> x`
`2.6.1. evaluate 1`

Поскольку x связан с 1, результатом вычислений оказывается константа 1.

Результат шага 2.4 есть связывание третьего параметра с непустым списком, что, в свою очередь, приводит к связыванию a с головой этого списка, а l — с хвостом:

`2.5.2. extract / reduce -> f a (reduce f x) l`
`2.6.2. evaluate (*) a (reduce (*) 1) l`

Поскольку f связан с (*), здесь произошло извлечение нужной для product операции с соответствующими операндами. Корректность получения нужных a и l гарантирована результатом шага 2.4.

Таким образом, в зависимости от третьего аргумента функции product обеспечена одна из трех возможностей вычислений:

— *блокировка*, если не выполнен п. 2.3;

— *завершение с результатом* 1, если выполнен п. 2.6.1;

— *продолжение вычисления*, т. е. выполнение действий, связанных с вычислением (*) a (reduce (*) 1) l, если выполнен п. 2.6.2.

Третья возможность приводит к активизации рекурсивного вычисления reduce, что влечет за собой выстраивание применений шаблонов для соединения частей функциональной программы.

4. Функции высших типов и ленивые вычисления. Любопытно, что и императивная семантика допускает определение шаблонов для соединения, простирающееся на иерархию шаблонов. Однако в императивном случае такое определение привело бы к необходимости довольно сложных соглашений о взаимодействии контекстов, которые трудны не только для понимания, но и для проверки корректности программ (один из моментов такого рода мы уже отмечали: это совместность вычисления вариантов). Возможно, по этим причинам иерархически построенные шаблоны в императивных языках не применяются — все ограничивается процедурами и иногда в дополнение к ним макрогенерацией

(чаще в урезанном виде). Как следствие, для таких языков не получается концептуально непротиворечиво обеспечить возможность программирования с функциями высших типов.

Избавление языка от средств и конструкций, препятствующих разумному использованию функций высших типов, по сути дела и выводит императивный язык в ту сферу, в которой хорошо работают функциональные языки. Но, как совершенно справедливо замечает Хьюз, сильная сторона альтернативного стиля не в ограничениях, а в новых возможностях, которые становятся достижимыми, в частности, за счет этих ограничений.

Сами по себе ленивые вычисления с определенным успехом давно применяются и в императивных языках. В частности, так называемая подстановка параметра по необходимости есть ничто иное, как отголосок концепции ленивых вычислений. И это вполне хорошо работает на своем месте (иллюстрации тому можно найти в [6]). Однако, говоря о ленивых вычислениях в традиционных языках, нельзя обойти вниманием, что здесь явно должно определяться время, когда необходимость выполнения вычисления наступает. Не вдаваясь в детали, достаточно заметить, что решать вопрос о наступлении необходимости на основе только локальных данных о процессе чаще всего невозможно. В функциональном языке просто нет места для подобных проблем: единственный (!) момент, когда нужно активизировать любые вычисления, — тот, в который активная функция потребует результат этих вычислений.

По отдельности трудности функций высших типов и ленивые вычисления для императивной модели, хотя и серьезны, но не непреодолимы. Но вот их сочетание просто невыразимо в этой модели из-за зависимости результатов вычислений от порядка выполнения действий. Это обстоятельство выделяет функциональные языки как поле, на котором за счет более богатых средств склеивания частей программы можно получать выгоду.

5. Потери, связанные с функциональными вычислениями. Функциональные возможности склеивания частей, безусловно, повышают уровень средств абстракции, предоставляемых программисту. Так, с помощью функции reduce можно без труда составлять любые функции, работающие по заданному шаблону распространения функции f от двух аргументов до применения ее ко всем элементам списка (подобно тому, как из операции умножения (*) была получена функция product).

Можно забыть о том, как устроена функция reduce и считать, что, имея список

```
(a : b : ...: c)
```

мы строим выражение

```
x 'f' a 'f' b 'f' ...'f' c
```

(в языке Haskell, нотации которого мы придерживаемся, обрамленное кавычками имя функции от двух аргументов превращает ее в инфиксную бинарную операцию), просто ставя на первое место x и заменяя двоеточия (разделитель элементов списка) обращением к f (в примере с product это приводит к выражению 1*a*b*...*c).

Стоит обратить внимание, что если для product следует учитывать, что последовательность умножений надо прервать, когда появляется нулевой сомножитель, то для этого не потребуется переписывать функцию reduce (и знать, как устроена). Достаточно определить функцию cut0, которая строит по произвольному списку копию исходного списка без последующих за нулем элементов, и скомбинировать product и cut0:

```
product . cut0
```

Это работает, причем за счет ленивых вычислений получится не „*сначала* построить список, *потом* перемножать элементы“, а именно то, что надо: последовательная генера-

ция, совмещенная с перемножением. Однако тут же видно и ограничение: не получается избавиться от лишних умножений, предшествующих появлению нуля, если мы хотим реализовывать product при помощи reduce и не применять предварительную генерацию списка из одного нулевого элемента.

Другие еще более выразительные примеры того же ряда приводит Хьюз, правда, не указывая на ограничения.

Но стоит ли в связи с этим reduce и product считать модулями, разделяющими уровни реализации и использования? Ответ простой: это модульность того же порядка, что и в условном операторе по отношению к его реализации с помощью машинных команд условного и безусловного перехода или в использовании символических адресов в ассемблере по отношению к реальным адресам. И единственное преимущество функциональных средств соединения частей перед императивными — заметный рост выразительности. Но это преимущество имеет и оборотную сторону, обусловленную тем, что теряется при переходе к чистой функциональности. К потерям относятся следующие свойства императивной модели:

— Пассивность памяти нельзя выразить в функциональном стиле. Это означает, что, например, работа с массивами однородной информации, со статическими структурами, обработка которых не определяется в рамках регуляризированных обходов деревьев, с другими подобными типами данных оказывается затруднительной. По этой же причине затрудняется использование побочных эффектов вычислений;

— Понятие состояний вычислительного процесса, которые во многих случаях дают естественную его декомпозицию (а значит, и соответствующую модульность) теряется в функциональном языке. Оно и понятно в силу глобальности по своей сути понятия состояния[4];

— Понятие контекста вычислений становится существенно более узким, нежели при работе в императивном стиле. Для функциональности важно, чтобы контексты были организованы иерархически. Сегодня иерархии контекстов связываются с понятием структурности и, в частности, со структурной организацией действий и данных. Это очень полезные случаи, без которых, пожалуй, не обойтись ни в одном сложно организованном вычислительном процессе. Но есть достаточное число ситуаций, когда выстраивать контексты в виде иерархий было бы весьма неудобно;

— Управление упорядочиванием вычислений во времени противоречит функциональности. И это приводит к затруднениям при решении довольно многих задач, например, при имитационном моделировании.

Если допустить, что претензии функционального стиля на универсальность обоснованы, то все это и многое другое пришлось бы выбросить из арсенала средств и методов программиста в угоду новому очередному „самому мощному и универсальному“ подходу к организации вычислений.

[4]При обсуждении проблем функционального стиля иногда говорят о состояниях. В частности, с их помощью пытаются решать давнюю проблему организации упорядоченного ввода/вывода в функциональной программе. Однако это не императивные состояния конечного автомата, с которыми можно связывать определенные действия, допускающие оформление в виде модулей, а лишь средство разграничения вычислений общих функций в разных ситуациях. С помощью таких состояний можно имитировать время, упорядоченность действий, получать другие полезные эффекты. Есть первый опыт определения специальных языковых конструкций, предназначенных для оформления типовых приемов программирования с использованием функциональных состояний — так называемые монады [13].

Заключение. Возвращаясь к вопросам модульности, приходится признать, что претензии на более развитую модульность императивного стиля не оправдываются. Сам по себе стиль не может быть более или менее модульным, чем другой. Но совершенно понятно, что для каждого стиля программирования нужна своя модульность, которая позволит абстрагироваться от реализационных деталей в рамках своей модели вычислений. Это понятно для известных и применяемых сегодня императивных стилей: для программирования от состояний, структурного программирования, событийного программирования и др. В рамках такого разграничения формируются правила и регламенты, которым должны подчиняться модули.

Опыт императивных стилей показывает, что при помощи развитых средств модульности в программных комплексах вполне уживаются части, отвечающие разным, порою несовместным стилям. Но что касается функционального стиля, то приходится констатировать, что средства его модульности еще очень неразвиты. Опыт промышленной разработки программных систем, реализованных в этом стиле, слишком мал. Работы по аппаратной поддержке функциональных моделей вычислений не вышли на уровень технологии. Наконец, как следствие предыдущего, весьма бедный спектр функциональных языков программирования — главные причины того, что сегодня о функциональной модульности можно говорить лишь на уровне гипотез и экспериментов.

В качестве примера успешной попытки реализовать поддержку реальной модульности в функциональном стиле можно указать на систему программирования CLOS (Common Lisp Object System) [14], которая претендует на то, чтобы называться объектно-ориентированным Лиспом. Здесь, как в любой объектной среде, имеются средства группировки свойств, функций вокруг функционального аналога объекта. Можно говорить и о классах таких объектов, и о многом другом, что заимствовано из императивных объектных языков. Видно стремление разработчиков системы предоставить программисту развитые и достаточно привычные средства абстракции. Как следствие, программы на CLOS'е довольно легко читать и понимать.

Тем не менее, опыт использования этой системы не позволяет говорить о том, что сформированы общепринятые понятия модульной декомпозиции функционального языка. Среди заметных препятствий для адекватной модульности на базе CLOS не последнее место занимает эклектичность системы.

Нетрудно заметить, что все потери функционального стиля по сравнению с императивным программированием так или иначе связаны с понятием общих (но не глобальных![5]) для разных модулей данных, для которых он не может предложить адекватные средства выражения. Это объясняет успешность модульности, когда функциональные части программы вызываются для выполнения в рамках операционной среды, в которой они оформляются как независимые от окружения модули. Это же обусловливает затруднения модуляризации, в которой императивно заданные фрагменты пытаются прямолинейно встроить в функциональную среду программу. Адекватность операционных средств описания глобальности не противоречит выделению локальных фрагментов, которые легко обособить. Но если захотеть пойти дальше и попытаться задать в императивном модуле

[5]Разграничение понятий общности и глобальности данных сводится к следующему. Глобальные данные принадлежат единому для некоторых модулей контексту обезличено — они в принципе доступны для всех модулей контекста, тогда как общие данные назначаются для коммуникаций между вполне определенными модулями и только для них. Мотивация введения этих понятий, а также понятия всеобщности (доступности для использования всем, но не для генерации и изменения) приведена в [15].

функциональной программы две или более программные единицы, объединенные общим контекстом, — а ведь именно это и определяет свои преимущества императивного стиля, то встанут обычные проблемы синхронизации и согласования императивных вычислений с общими данными, характерные для этого стиля. Таким образом, единственной задачей императивного модуля, непротиворечиво встраиваемого в функциональную систему, может стать вычисление обособленных функций, связанных с окружением лишь получением аргументов и передачей в качестве результатов вычисленных значений. Но в этой задаче нет ничего нового: именно так реализуются в операционном окружении все базовые атомарные функции любой функциональной системы программирования.

В данной работе мы говорили преимущественно о базовых средствах языков, обеспечивающих модуляризацию программ при использовании функционального и императивного стилей. Они определяют возможности, и ограничения, проявляющиеся в форматах модулей сборочного конструирования программ. Эта тема достаточно подробно освящена в статье родоначальника концепции сборочного программирования Г. С. Цейтина [16].

Список литературы

1. Asanovic K. et al. The landscape of parallel computing research: a view from Berkeley. Technical Report No. UCB/EECS-2006-183. Berkeley: University of California, EECS Department, December 18, 2006. URL: `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf` (дата обращения: 24.05.2019).

2. Backus J. Can Programming be Liberated from von Neumann style? A Functional Style and its Algebra of Programs. Comm. ACM, 21, 1978.

3. McCarthy 91 function Wikipedia. URL: `https://en.wikipedia.org/wiki/McCarthy_91_function#Knuth.27s_generalization` (дата обращения: 24.05.2019).

4. Городняя Л. В. Основы функционального программирования. Курс лекций. М.: Интернет-университет информационных технологий, 2004. ISBN 5-9556-0008-6.

5. Hughes J. Why Functional Programming Matters. Computer Journal, 32 (2), 1989. Русский перевод: Дехтяренко И. А. Сильные стороны функционального программирования. URL: `http://www.softcraft.ru/paradigm/fp/whyfp.shtml` (дата обращения: 24.05.2019).

6. Непейвода Н. Н., Скопин И. Н. Основания программирования. Москва–Ижевск: РХД, 2003 г.

7. Дал У.-И., Дейкстра Э., Хоор К. Структурное программирование // Пер с англ. М.: Мир, 1975.

8. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.

9. Stroustrup B. What is Object-Oriented Programming? IEEE Software. 1988. V. 5 (3).

10. Косивченко. А. Зачем же нужна виртуализация? URL: `https://habrahabr.ru/post/91503/` (дата обращения: 24.05.2019).

11. Sylvan S. Why does Haskell matter? URL: `http://www.dtek.chalmers.se/~sylvan/haskell/why_does_haskell_matter.html` (дата обращения: 24.05.2019).

12. Ершов А. П. О сущности трансляции. Препринт № 6, Новосибирск: ВЦ СО АН СССР, 1977.

13. Winstanley N. What the hell are Monads? 1999, URL: `http://www.abercrombiegroup.co.uk/~noel/research/monads.html` (дата обращения: 24.05.2019).

14. Keene S. E. Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. Addison-Wesley (Reading, Massachusetts, 1989).

15. Skopin I. N. An Approach to the Construction of Robust Systems of Interacting Processes // In: Parallel PROGRAMMING: Practical Aspects, Models and Current Limitations. NOVA science publishers. Series: Mathematics Research Developments. Editor: M. S. Tarkov. 2014, ISBN: 978-1-63321-957-1.

16. Цейтин Г. С. На пути к сборочному программированию. Программирование. 1990. № 1, С. 78–99.

Скопин Игорь Николаевич — старший научный сотрудник Института вычислительной математики и математической геофизики (ИВМиМГ) СО РАН, тел: +7 983 126-26-86; e-mail: iskophin@gmail.com.

Интерес **Игоря Николаевича** к проблематике разработки языков программирования и к конструированию компиляторов восходит к 1968-70 годам, когда он был студентом механико-математического факультета Новосибирского государственного университета (НГУ) и проходил производственную практику в Новосибирском филиале ИТМ и ВТ АН СССР (ныне Новосибирский институт программных систем). В этой организации он принимал участие в разработке ряда систем программирования и инструментов. В 1974–76 годах он руководил группой специалистов, которая конструировала специализированную систему поддержки обработки текстовой информации. Эта работа послужила основой кандидатской диссертации «Функциональное структурирование текстовой информации, его реализация и приложения» (научный руководитель А. П. Ершов), которую И. Н. Скопин защитил в 1981 году.

В 1986 году И. Н. Скопин получил приглашение заведовать лабораторией в Институте информатики и вычислительной техники Академии педагогических наук СССР (после реорганизации — Институт компьютерных систем в обучении). В качестве цели работы в этой лаборатории были обозначены исследования и разработка базовых программных средств поддержки преподавания. Проблемы образовательной информатики пришлось решать в период становления института, а потому одной из существенных задач стала организация команды программистов, способной эффективно работать в новой прикладной области. Результативности деятельности И. Н. Скопина в этой области способствовало его преподавание в НГУ.

Среди других наиболее значимых мест работы И. Н. Скопина следует указать Новосибирский Центр исследований и разработки корпорации Интел, где он участвовал в решении задач, связанных с пакетом математических программ MKL.

В настоящее время И. Н. Скопин является сотрудником Лаборатории вычислительной физики.

Igor Skopin — Senior Researcher of Institute of Computational Mathematics and Mathematical Geophysics (ICMMG) SB RAS, tel: +7 983 126-26-86; e-mail: iskophin@gmail.com.

The interest of **Igor Skopin** in problems of programming languages developing and in the construction of compilers dates back to 1968–70, when he was a student in the Mechanics and Mathematics Department of Novosibirsk State University (NSU). He had practices at the Novosibirsk branch of Lebedev Institute of Precision Mechanics and Computer Engineering with Academy of Sciences of the USSR (now it is called as the Novosibirsk Institute of Software Systems), where he took part in the development of programming systems and tools. In 1974-76 he led a team of programmers who designed a specialized support system for processing text information. This work served as the basis for his dissertation “Functional structuring of textual information, its implementation and applications” (supervisor A. P. Ershov), which Skopin defended 1981.

Дата поступления — 14.08.2019