УДК 004.4

КОМПЛЕКСЫ В ЛЯПАСЕ[1]

В. О. Сафонов, Д. А. Стефанцов

Национальный исследовательский Томский государственный университет, г. Томск, Россия

Описаны уточнения, которые внесены в правила выполнения операций над комплексами в языке программирования ЛЯПАС. Цель этих нововведений — обеспечить более удобную и безопасную работу с комплексами. Представлена реализация новых правил в модуле транслятора.

Ключевые слова: *язык программирования ЛЯПАС, операции над комплексами, транслятор.*

DOI 10.17223/20710410/38/8

COMPLEXES IN LYAPAS

V. O. Safonov, D. A. Stefantsov

National Research Tomsk State University, Tomsk, Russia

E-mail: vsaffonov.1115@gmail.com, d.a.stephantsov@gmail.com

The changes in semantics of operations over the data structures called complexes in the LYaPAS programming language are discussed. The modifications include resizing the complexes, passing complexes that are created by a callee to the caller, and compile-time error reporting due to modifying operations being applied to input complexes. The goal is to make the work with complexes more convenient and less error prone, which is assumed to have a positive impact on the security of the programs written using the new language constructs. The implementation of the new semantics in the module of the translator is demonstrated.

Keywords: *LYaPAS, data structures, complex, translator.*

Введение

Отечественный язык программирования ЛЯПАС [1] возрождается с целью иметь высокопроизводительную доверенную вычислительную систему для создания доверенного ПО, разработки и исследования криптографических алгоритмов, безопасного управления сетевым оборудованием, критически важными объектами и технологическими процессами. Главные компоненты создаваемой системы — транслятор с ЛЯПАСа и операционная система (ОС) ЛЯПАС.

Модульный транслятор с ЛЯПАСа [2, 3] разрабатывается как вспомогательное средство для создания транслятора, работающего под управлением ОС ЛЯПАС [4]. *Модули* реализуют последовательные фазы процесса трансляции программы на ЛЯПАСе в программу на машинном языке. Результат работы каждого из модулей представляется на выходном языке соответствующего модуля. Данная работа посвящена

[1]Работа поддержана грантом РФФИ, проект № 17-01-00354.

разработке и реализации модуля для работы с комплексами. Далее этот модуль и его язык называются *комплексояз*. Описание этого языка можно найти в [5].

Комплекс — структура данных в виде набора равных по размеру элементов, расположенных в памяти непосредственно друг за другом. Комплекс имеет потенциальный и текущий размеры, которые называются *ёмкостью* и *мощностью* соответственно. Обращение к элементу за пределами комплекса приводит к аварийному завершению программы.

Комплекс, как и переменную, можно передать в подпрограмму. В текущей версии ЛЯПАСа [1] эта передача происходит по ссылке. Изменения комплекса (как входного, так и выходного) в подпрограмме видны в вызывающей программе.

Комплексы в существующей версии ЛЯПАСа обладают некоторыми недостатками, например, их ёмкость нельзя изменять динамически; проверка выхода за границы выполняется по ёмкости, а не по мощности. Деление на входные и выходные комплексы является формальным. За неизменность входного комплекса отвечает программист, транслятор не накладывает ограничения на работу с этим комплексом. Перенос такой ответственности с программиста на транслятор позволит внести в программы на ЛЯПАСе больше дисциплины. Для решения некоторых задач необходимо иметь возможность увеличивать размеры уже созданного комплекса, например, если нужно поместить в комплекс строку, которая считывается с клавиатуры, то заранее не известно, как много памяти потребуется.

В данной работе вносятся уточнения в правила выполнения в ЛЯПАСе операций над комплексами, обеспечивающие более удобную и безопасную работу с комплексами, и описываются их реализации в модуле транслятора.

Модуль транслирует операции работы с комплексами в набор операций более низкого уровня, часть которых присутствуют в предыдущих промежуточных языках. Первая часть новых операций — это операции захвата и освобождения памяти. Операция захвата присутствует в двух модификациях, одна из которых захватывает память с некоторым запасом для оптимизации дальнейшего расширения комплекса. Вторая часть — операции работы со строками и символами: размещение строки в памяти, чтение символа с консоли и запись символа в консоль.

Модульный транслятор поддерживает вывод ошибок трансляции и ошибок во время выполнения программ. На этапе трансляции обнаруживаются логические ошибки использования комплексов, например модификация входного комплекса в подпрограмме. К типичным ошибкам во время выполнения относятся выход за мощность комплекса и нехватка памяти для захвата. Аварийное завершение программы сопровождается сообщением, которое содержит информацию об ошибке и название операции.

1. Операции над комплексами в ЛЯПАСе

1.1. Динамическое изменение ёмкости

В ЛЯПАС вводится возможность изменять ёмкость комплекса. Увеличение ёмкости влечёт захват участка памяти с бо́льшим размером, копирование элементов из старого участка памяти в новый, а затем освобождение старого участка. Для сокращения количества таких операций широко используется подход увеличения ёмкости с некоторым запасом [6]. Для выбора оптимальной ёмкости будем использовать неравенство $m \leqslant k^i$, где m — ёмкость, запрашиваемая пользователем, k — некоторый коэффициент больше единицы. После нахождения минимального i, для которого неравенство истинно, используем k^i в качестве оптимальной ёмкости.

Пусть n — текущая ёмкость комплекса, тогда запись значения m в ёмкость комплекса обладает следующими свойствами:

1) ёмкость не меняет свое значение, если $m \leqslant n$;
2) ёмкости будет присвоено значение не меньше m.

Из первого свойства следует, что запись в ёмкость комплекса никогда не уменьшает её. Для сокращения размера занимаемой комплексом памяти можно воспользоваться операцией сокращения ёмкости [1].

1.2. Правила передачи комплексов в подпрограммы

Необходимо обеспечить неизменность входных комплексов на этапе трансляции программы. Назовём операции над комплексами, которые могут менять мощность, ёмкость или элементы комплекса, операциями записи. Оставшиеся операции над комплексами назовём операциями чтения. Запишем следующие правила:

1) к входному комплексу можно применять только операции чтения, операции записи запрещены;
2) к выходному и входо-выходному комплексам можно применять операции записи и чтения.

Будем выдавать ошибку трансляции, если программа содержит операции, запрещённые по отношению к данному типу комплексов.

1.3. Создание комплексов с помощью подпрограмм

Иногда возникает потребность создавать комплекс внутри подпрограммы, а в вызывающей программе использовать этот комплекс. Например, такой подпрограммой может быть построение последовательности (длины n) чисел Фибоначчи:

```
1 fibonacci(n/L1)
2 n⇒S1⇒Q1 ↑(n=0)2 0⇒L1.0⇒j ↑(n=1)2 1⇒L1.1⇒k 2⇒i
3 §1 ↑(i=n)2 L1j+L1k⇒L1i Δi Δk Δj →1
4 §2 **
```

В текущей версии ЛЯПАСа мы можем использовать такую подпрограмму только с созданным заранее комплексом:

```
1 main()
2 @+L1(10)
3 *fibonacci(10/L1)
4 **
```

В таком случае при создании комплекса можно ошибочно задать ёмкость больше или меньше, чем требуется функции `fibonacci`. Такая ошибка не критична, но может привести к лишним выделениям памяти.

Рассмотрим следующую функцию; она создаёт комплекс случайного размера и заполняет его числами 1, 2, ...:

```
1 makerandom(f,t/L1)
2 t-f⇒d X;d+f⇒S1⇒Q1 Oi
3 §1 ↑(i=Q1)2 i+1⇒L1i Δi →1
4 §2 **
```

Во время вызова подпрограммы недоступна информация о том, как много элементов будет содержать комплекс, это определяется внутри подпрограммы. Предлагается

разрешить передачу в подпрограмму ранее не созданного комплекса в качестве выходного. В этом случае будет создан комплекс с нулевой ёмкостью, а подпрограмма расширит его ёмкость до нужного размера:

```
1 main()
2 *fibonacci(10/L1)
3 *makerandom(100,4000/L2)
4 **
```

2. Программная реализация

Операции над комплексами транслируются в набор операций низкого уровня. В некоторых случаях количество операций превышает норму и ведёт к «разбуханию» кода [7]. В результате программа может не поместиться на устройствах с ограниченным количеством оперативной памяти, что приведёт к дополнительному обмену с жёстким диском, а также уменьшит коэффициент попадания команд в кэш процессора L1. Всё это может отрицательно повлиять на производительность программ. Предлагается операции над комплексами, которые транслируются в большой набор операций низкого уровня, заменять на вызов подпрограмм, которые назовём внутренними подпрограммами комплексояза. Список внутренних подпрограмм определяется экспериментальным путём и зависит от реализации транслятора, в частности рекомендуется оставить этот список пустым для устройств без дефицита оперативной памяти.

2.1. Разбиение на подзадачи

Работу комплексояза можно разделить на два этапа: проверку корректности входной программы и её трансляцию. Первый этап служит для выявления ошибок транслируемой программы и включает в себя две подзадачи; первая из них является общей для всех модулей транслятора — валидация операций и их операндов, вторая подзадача специфична для комплексояза и отвечает за соблюдение правил выполнения операций над комплексами. Второй этап состоит из подзадач трансляции операндов, а также трансляции операций над комплексами в операции более низкого уровня.

Для иллюстрации рассмотрим небольшой отрывок из входной программы комплексояза:

```
1 definition f1, a / F1
2 move L2[0], a
3 move 10, 15
4 read_complex F1
```

Первая подзадача первого этапа выявит, что третья строка содержит ошибку в первом операнде, так как операция `move` не может писать значения в константу. Вторая подзадача сообщит о попытке чтения из выходного комплекса в четвертой строке. Модуль комплексояз заканчивает работу на первом этапе, если найдена хотя бы одна ошибка, результатом является список найденных ошибок. После исправлений ошибок программа выглядит следующим образом:

```
1 definition f1, a, F1 /
2 move L2[0], a
3 read_complex F1
```

Программа после трансляции операндов:

```
1 definition f1, a, F1 /
2 move 8byte L2_buffer[0], a
3 read_complex F1
```

Программа после финального шага трансляции операций над комплексами:

```
1 definition f1, a, F1 /
2 move 8byte L2_buffer[0], a
3 move t1, 8byte F1_struct[0]
4 label 1
5 compare t1, 8byte F1_struct[1]
6 jump_⩾ 3
7 read_char 1byte F1_buffer[t1]
8 compare 1byte F1_buffer[t1], 10
9 jump_eq 2
10 inc t1
11 jump 1
12 label 2
13 inc t1
14 label 3
15 move 8byte F1_struct[0], t1
```

2.2. Вспомогательный язык для записи правил трансляции промежуточных языков

Правило трансляции записывается в следующем виде:

```
1 op <args>
2 =>
3 op_1 <args>
4 ...
5 op_n <args>
```

Здесь `op` — транслируемая операция; `op_1`, … , `op_n` — список результирующих операций. Количество аргументов для каждой из операций не ограничено.

Операция `op` стоит из двух частей: тип операции и её значение. Если у операции есть тип, но отсутствует значение, то записывается только первая часть; например метка записывается следующим образом: `label`.

Типы аргументов

1) Аргумент {name_of_arg} сохраняет исходный тип:

```
1 cmd/store {arg}
2 =>
3 cmd/move {arg}, "acc"
```

Какой бы тип ни был у аргумента команды `store`, например строка или число, в результирующей команде `move` этот тип сохранится.

2) Аргумент <name_of_arg> позволяет выбрать результирующий тип:

```
1 cmd/some "<int>"
2 =>
3 cmd/some <int>, "<int>"
```

В этом случае происходит прямая подстановка, поэтому первый аргумент будет иметь целочисленный тип, а второй — строковый. Можно использовать более интересные подстановки:

```
1 cmd/swap_comp_el "<complex>", "<int:1>", "<int:2>"
2 =>
3 cmd/move "swaptemp", "<complex>[<int:1>]"
4 cmd/move "<complex>[<int:1>]", "<complex>[<int:2>]"
5 cmd/move "<complex>[<int:2>]", "swaptemp"
```

3) Вспомогательные аргументы, которые могут быть использованы только в результирующих операциях:

- <free_label $= N$> — подставить число, равное сумме номера первой свободной метки и N. Например, если метки 1, 2, 3 и 4 используются в процедуре, то <free_lable=2> будет транслирован в 7;
- <free_var $= N$> — подставить строку, которая получается в результате конкатенации двух строк: первая часть — t, вторая — сумма номера первой свободной переменной и числа N. Например, <free_var=3> будет транслироваться в t5 при условии, что в процедуре уже присутствуют переменные t1 и t2.

Пример:

```
1 cmd/clear_complex "<complex>"
2 =>
3 cmd/move "<free_var=1>", 0
4 label <free_label=1>
5 cmd/compare "<complex_cardinality>", "<free_var=1>"
6 cmd/jump_eq <free_label=2>
7 cmd/move "<complex_cell=<free_var=1>>", 0
8 cmd/inc "<free_var=1>"
9 cmd/jump <free_label=1>
10 label <free_label=2>
```

Параметризация аргументов

В аргументе могут присутствовать дополнительные параметры, например id аргумента. Это удобно использовать, если у аргументов одинаковые имена. Пример:

```
1 cmd/swap {variable:1}, {variable:2}
2 =>
3 cmd/move "swaptemp", {variable:1}
4 cmd/move {variable:1}, {variable:2}
5 cmd/move {variable:2}, "swaptemp"
```

2.3. Архитектура

Рассмотрим этап инициализации на диаграмме последовательности (рис. 1).

Для реализации класса `Translator` [8] используется паттерн «шаблонный метод» [9], что позволяет зафиксировать общее поведение алгоритма инициализации и оставляет возможность пользователям класса переопределить метод, который возвращает правила трансляции. Связка классов `CmdBuilder` [10] и `ArgBuilder` [11] рассматривается как паттерн «команда» [9], где `CmdBulder` — это класс, который хранит коллекцию `ArgBuilder`, которая позволяет сконструировать необходимый набор аргументов. Рабочий цикл транслятора приведён на рис. 2.

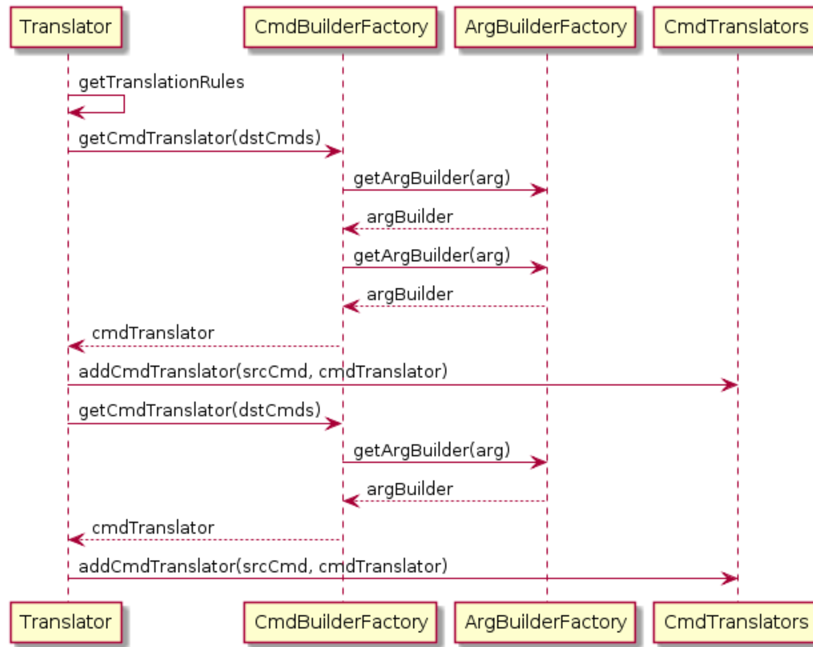

Рис. 1

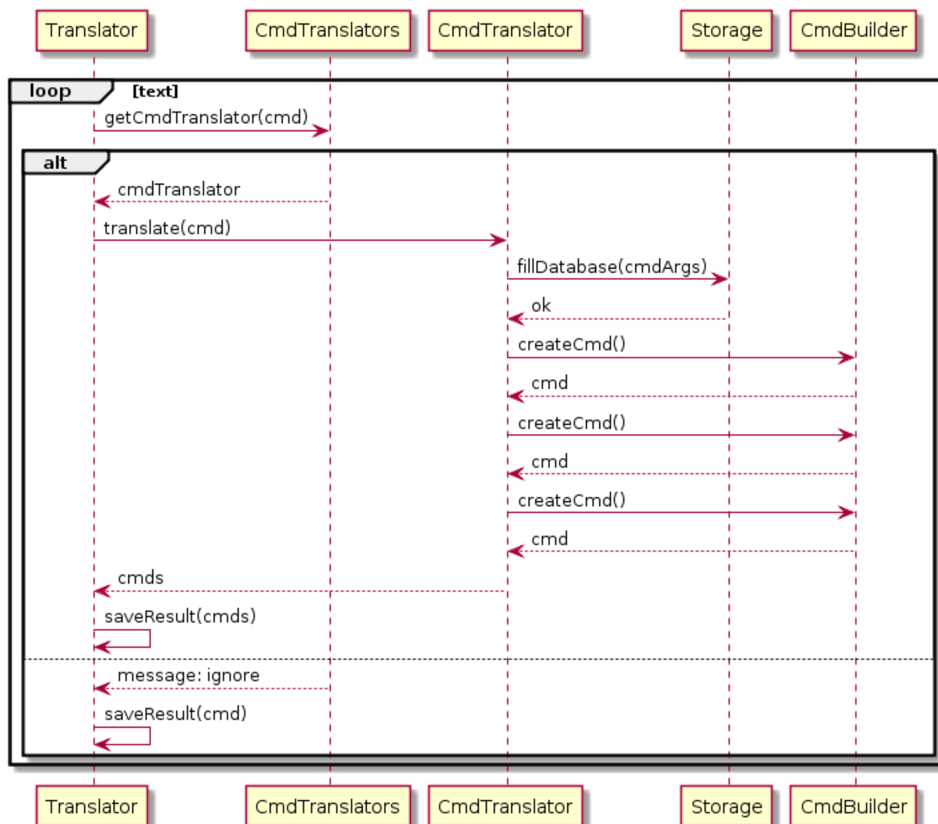

Рис. 2

Заключение

В качестве языка программирования для реализации модуля комплексояза выбран язык C++. Он удовлетворяет основным требованиям скорости разработки, а также

является достаточно популярным, что позволяет привлечь к разработке транслятора сторонних программистов. С переходом модульного транслятора на Open Source [3] последний пункт является особенно важным.

В язык ЛЯПАС внесены следующие доработки и изменения:

1) разработан механизм динамического изменения ёмкости комплексов, что позволяет расширять или уменьшать размеры комплексов в зависимости от нужд программиста;
2) разработан механизм неявного создания выходного комплекса во время вызова подпрограммы;
3) разработан механизм вывода ошибок трансляции и ошибок времени выполнения для комплексов;
4) реализована проверка на неизменность входных комплексов на этапе трансляции;
5) реализована проверка выхода за пределы комплекса по мощности.

ЛИТЕРАТУРА

1. *Агибалов Г. П., Липский В. Б., Панкратова И. А.* О криптографическом расширении и его реализации для русского языка программирования // Прикладная дискретная математика. 2013. №3. С. 93–104.
2. *Стефанцов Д. А., Сафонов В. О., Першин В. В. и др.* Модульный транслятор с языка ЛЯПАС // Прикладная дискретная математика. Приложение. 2016. №8. С. 122–126.
3. `https://github.com/tsu-iscd/lyapas-lcc` — LYaPAS Compiler Chain. 2017.
4. *Томских П. А., Стефанцов Д. А.* Разработка операционной системы на языке ЛЯПАС // Прикладная дискретная математика. Приложение. 2015. №8. С. 134–135.
5. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/doc/cyaz.md` — LYaPAS Cyaz Documentation. 2017.
6. `http://en.cppreference.com/w/cpp/container/vector/reserve` — std::vector::reserve. 2017.
7. *Meyers S.* Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Professional, 2005. 297 p.
8. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/sources/libs/translation_module/src/include/translation_module/translation_module.h` — LYaPAS class Translator. 2017.
9. *Gamma E., Helm R., Johnson R., et al.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.
10. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/sources/libs/translation_module/src/include/translation_module/cmd_builder.h` — LYaPAS class CmdBuilder. 2017.
11. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/sources/libs/translation_module/src/include/translation_module/arg_builders.h` — LYaPAS class ArgBuilder. 2017.

REFERENCES

1. *Agibalov G. P., Lipskiy V. B., and Pankratova I. A.* O kriptograficheskom rasshirenii i ego realizatsii dlya russkogo yazyka programmirovaniya [Cryptographic extension and its implementation for Russian programming language]. Prikladnaya Diskretnaya Matematika, 2013, no. 3, pp. 93–104. (in Russian)

2. *Stefantsov D. A., Safonov V. O., Pershin V. V., et al.* Modul'nyy translyator s yazyka LYaPAS [Modular translator from LYaPAS]. Prikladnaya Diskretnaya Matematika. Prilozhenie, 2016, no. 8, pp. 122–126. (in Russian)
3. `https://github.com/tsu-iscd/lyapas-lcc` — LYaPAS Compiler Chain, 2017.
4. *Tomskikh P. A. and Stefantsov D. A.* Razrabotka operatsionnoy sistemy na yazyke LYaPAS [The development of an operating system in LYaPAS]. Prikladnaya Diskretnaya Matematika. Prilozhenie, 2015, no. 8, pp. 134–135. (in Russian)
5. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/doc/cyaz.md` — LYaPAS Cyaz Documentation, 2017.
6. `http://en.cppreference.com/w/cpp/container/vector/reserve` — std::vector::reserve, 2017.
7. *Meyers S.* Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Professional, 2005. 297 p.
8. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/sources/libs/translation_module/src/include/translation_module/translation_module.h` — LYaPAS class Translator, 2017.
9. *Gamma E., Helm R., Johnson R., et al.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.
10. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/sources/libs/translation_module/src/include/translation_module/cmd_builder.h` — LYaPAS class CmdBuilder, 2017.
11. `https://github.com/tsu-iscd/lyapas-lcc/blob/73b21bcd5f674bc6762a379bc32f71f61ee51164/sources/libs/translation_module/src/include/translation_module/arg_builders.h` — LYaPAS class ArgBuilder, 2017.