Процедурный сшитый код для обеспечения нисходящей разработки структурированных программ в ДССП для троичной машины

А.А. Бурцев

кандидат физико-математических наук

Аннотация: В НИЛ троичной информатики ВМК МГУ созданы троичная виртуальная машина ТВМ и кросс-система ДССП-ТВМ разработки программ для неё на языке ДССП-Т – троичном варианте языка ДССП, а также диалоговый интерпретатор ДССП/ТВМ, функционирующий на ТВМ как резидентное ПО. В статье объясняется, как решена проблема обеспечения нисходящей разработки структурированных программ при реализации кросс-компилятора и диалогового интерпретатора ДССП для ТВМ.

Ключевые слова: структурированное программирование, сшитый код, нисходящая разработка, троичный компьютер, ДССП.

Введение

В НИЛ троичной информатики ВМК создан программный комплекс ТВМ (Троичная Виртуальная Машина) [1], имитирующий функционирование современного варианта троичного процессора двухстековой архитектуры с поддержкой структурированного программирования на уровне машинных команд, аналогичной той, что была обеспечена в троичной ЦМ “Сетунь-70”.

Для программирования ТВМ сначала была построена кросс-система ДССП-ТВМ [2], позволяющая создавать программы для ТВМ на языке ДССП-Т с помощью кросс-компилятора, а затем был создан диалоговый интерпретатор ДССП/ТВМ [3], способный функционировать на троичной машине (ТВМ) в качестве её резидентной системы программирования.

Чтобы обеспечить возможности для структурированного программирования в полном объёме, при создании этих систем потребовалось решать проблему обеспечения нисходящей разработки программ. Результаты решения этой проблемы применительно к ДССП для ТВМ и являются предметом данной статьи.

1. Структурированное программирование как метод разработки программ и характеризующие его принципы

Вводя в употребление термин «структурированное программирование» (“structured programming”) Эдсгер Дейкстра предлагал [4] называть им такой метод разработки программы, в результате которого создаваемая программа приобретает свойственную ей (как правило, древовидную) структуру, отражающую динамическую сущность заложенного в неё алгоритма. И ведь действительно, если такая структура сама характеризует динамику поведения программы, т.е. порядок следования в ней ветвлений, циклов, вызовов подпрограмм, а также их иерархию, и явно отражена в её исходном тексте, то такую структурированную программу становится намного легче понять, модифицировать и сопровождать. В результате это и позволяет, в конечном счёте, радикально решить проблему снижения трудоёмкости программирования, т.е. существенно снизить затраты труда и облегчить работу не только одного программиста, но и всего программистского коллектива в целом

Согласно замыслу автора этого термина Дейкстры структурировать программу означает:

1) разрабатывать её путём пошагового уточнения (step wise refinement) и нисходящего анализа согласно так называемому принципу «сверху-вниз» (top-down);

2) выделять обособленные смысловые её части в отдельные программные компоненты – модули;

3) использовать для управления ходом исполнения программы только известные хорошо понимаемые управляющие конструкции (структуры).

Применительно к языкам программирования того времени для оформления отдельных модулей предлагалось использовать подпрограммы (процедуры и функции), а для последовательного сочленения и организации ветвлений и циклов было предложено использовать только такие управляющие конструкции, которые соответствуют правилу «один вход - один выход»: **begin…end, if…then…else, while…do…, repeat…until.**

Существенно, что из набора средств, рекомендованных к употреблению в программе для управления ходом её исполнения, Дейкстра предложил исключить оператор перехода (**goto**), объясняя это тем, что его применение может существенно повредить структуру программы и ухудшить тем самым её читабельность.

Излишне заострив на этом внимание (например, в своей статье “Goto considered harmful”), Дейкстра спровоцировал полемику среди ведущих программистов того времени по вопросу целесообразности отказа от оператора перехода.

Некоторые защитники оператора **goto** (например, Дональд Кнутт [5]) приводили примеры программ, структура которых, по их мнению, наоборот, ухудшалась, когда не разрешалось использовать этот оператор. Как правило, в приводимых программах требова-

лось обеспечить обработку чрезвычайных ситуаций с прекращением исполнения всей программы или какой-то её обособленной части в случае возникновения подобной ситуации. А в рекомендованном Дейкстрой наборе подходящей управляющей конструкции (на тот момент) не было, так что для обеспечения экстренного выхода (из программы или отдельной её компоненты) действительно приходилось употреблять оператор перехода или его разновидности (**exit**, **return**), либо усложнять программу, предусматривая излишние дополнительные проверки по окончании отдельных её участков, ветвлений и циклов, при исполнении которых подобная чрезвычайная ситуация могла возникнуть.

Но, к счастью, такая полемика оказалась даже полезной. В результате она привела к появлению в языках программирования специальных структурированных управляющих конструкций для обработки так называемых исключительных ситуаций, например, в форме защищённого оператора: **begin-except-end**.

К сожалению, излишний акцент на запрете оператора перехода допустил возможность упрощенно трактовать структурированное программмирование всего лишь как «программирование без goto». А такая упрощённая трактовка выхолащивает весь глубокий смысл, заложенный в этот термин его основоположником. Появились даже публикации, в которых нисходящая разработка (top-down) программы и структурированное программирование рассматривались как два разных отдельных метода [6, п 2.14–2.16].

Ещё большее сожаление вызывает недопонимание истинного смысла термина “structured programming”, когда встречаешь в русско-язычных изданиях его не совсем верный перевод в виде фразы «структурное программирование», что, увы, конечно же, не отражает полностью его истинный смысл. Ведь английское слово “structured” означает не прилагательное от слова «структура» (“structure”), а причастие, образованное от глагола “to structure”, что в переводе означает «структурировать», т.е. придавать структуру подлежащему воздействию объекту, а в нашем конкретном случае, программе.

Поэтому ещё раз подчеркнём, что структурированное программирование как метод разработки компьютерных программ, предложенный Дейкстрой, подразумевает совокупное соблюдение трёх принципов:

1) пошаговая нисходящая разработка программы;

2) дробление программы на обособленные компоненты - модули;

3) употребление только структурированных управляющих конструкций при составлении программы.

2. Языки для поддержки структурированного программирования

Успешность применения структурированного программирования как метода разработки программ во многом зависит того, какую поддержку применению этого метода могут оказать используемые для разработки языки и системы программирования.

Среди широко распространённых к тому времени (на момент публикации известных заметок Дейкстры о структурированном программировании) языков, пожалуй, лишь Алгол и ПЛ/1 хоть в какой-то мере могли быть признаны пригодными для составления структурированных программ. Эти языки предоставляли достаточно полный ассортимент управляющих конструкций, рекомендованных Дейкстрой. Но при этом они и никак не запрещали программировать «неструктурированно», ибо допускали возможность употребления оператора перехода **goto** без каких-либо ограничений.

А вот применять подпрограммы в этих языках оказывалось не совсем выгодным. Во-первых, по соображениям эффективности: вызов подпрограммы (особенно в ПЛ/1) сопровождался значительными накладными расходами, что приводило к существенному замедлению программы. А во-вторых, из-за досадных неудобств, связанных с оформлением исходного текста программы: объявление подпрограммы требовалось поместить в текст программы заранее, по крайней мере, до её первого вызова.

Всё это в совокупности приводило к тому, что программисты, как правило, стали оформлять в виде подпрограмм лишь те фрагменты алгоритма, которые приходилось бы записывать повторно, т.е. оформлять подпрограммы (процедуры и функции) считалось целесообразным ради сокращения исходного текста всей программы.

А чтобы отразить задуманную структуру алгоритма в исходном тексте программы, старались использовать так называемую «лесенку»: одинаковыми отступами от начала строки отмечали начало и конец соответствующих управляющих конструкций, вложенных друг в друга при составлении текста программы. Более того, именно наличие такой «лесенки» стало впоследствии считаться чуть ли не главным признаком хорошо структурированной программы.

Употребление «лесенки», конечно же, способствует пониманию структуры программы, но только в том случае, когда эта «лесенка» видна вся сразу целиком. В том же случае, когда текст программы не удаётся разместить на одной странице листа бумаги (при печати) или целиком в окне на одном экране (при просмотре программы на мониторе), «лесенка» уже не помогает увидеть созданную структуру всей программы. В таких случаях разбиение всей программы на отдельные модули (процедуры и функции) становится просто необходимым. При этом (в идеале) хорошо бы потребовать, чтобы размер отдельного модуля (подпрограммы) был таким, чтобы текст каждого модуля умещался целиком в просматриваемом окне экрана или странице печати.

И хотя впоследствии было создано много языков с учётом потребностей структурированного программирования (Паскаль, Алгол-68, Си, Forth, Модула, Модула-2, Ада и др.), следует признать, что лишь немногие из них поощряют выработку строгого стиля структуризации программ и позволяют отразить в исходном тексте создаваемой программы применение нисходящего (top-down) метода её разработки.

Почти во всех широко распространённых языках (Паскаль, Си, Forth и выросших из них многочисленных разновидностях) требуется сначала объявить подпрограмму (всю целиком или хотя бы характеризующий её заголовок), а уж потом разрешается употреблять её вызов. Это досадное ограничение затрудняет

сочинение исходного текста программы в полном соответствии с планом построения программы, выработанным путём нисходящего анализа (по принципу «сверху-вниз»).

В самом деле, чтобы лучше быть понятной человеку, всю программу надо изображать по принципу «сверху-вниз», т.е. сначала представить её головную часть, а уж затем, спускаясь вниз, охарактеризовать детали реализации отдельных её частей. Поскольку программа представляет собой текст, предназначенный для чтения слева направо и сверху вниз, значит, сначала в этом тексте следует разместить тело основной (головной) процедуры программы, а затем уже тела вызываемых в нём подпрограмм. И далее так же следует поступить с каждой подпрограммой: сначала разместить в тексте её тело, а следом за ним уже помещать объявления (и тела) тех процедур, которые использует в своём определении эта подпрограмма.

Но составленный таким способом исходный текст не способен принять транслятор (компилятор или интерпретатор) используемого языка. Вот поэтому и приходится для имеющегося транслятора представлять программу в несколько изменённом, перевёрнутом виде, т.е. образно говоря ставить программу «с ног на голову». При этом приходится сначала разместить в исходном тексте объявления всех используемых подпрограмм, а потом уже помещать в текст тела процедур, их вызывающих. И конечно же, при таком вынужденном способе записи программы основное тело программы окажется, увы, не в начале, как хотелось бы, а только в самом конце исходного текста программы.

Возникает очевидный вопрос: почему трансляторы столь несовершенны, почему не могут «прочитать» и проанализировать программу, записанную в том же естественном порядке («сверху-вниз»), как она создавалась методом пошагового нисходящего уточнения?

Для ответа на такой вопрос необходимо глубже изучить проблемы построения трансляторов. Оказывается, чтобы «научить» транслятор воспринимать программу, сочинённую и записанную методом «сверху-вниз», необходимо решить (при его создании) так называемую «проблему неопределённых ссылок вперёд».

Эта проблема заключается в следующем. Допустим, в программе определяется тело процедуры P, содержащее вызовы ещё неопределённых процедур P1, P2 … Pn. Чтобы сформировать код тела процедуры P, необходимо знать адреса размещения всех вызываемых процедур Pi (i=1…n), но эти адреса станут известны позднее, после того, как транслятор, проанализировав их определения, сформирует для них код и разместит их тела (в памяти или в выходном файле).

Проще всего такая проблема решается, если транслятору разрешается повторно прочитать исходный текст анализируемой программы, т.е. в итоге прочитать его дважды. Такой двухпроходной транслятор на первом проходе фиксирует для каждой процедуры, где она будет размещена, и запоминает её адрес, а на втором проходе уже формирует тела всех процедур, расставляя в них команды вызова подпрограмм с адресами, намеченными на первом проходе.

Однако, необходимость повторного прохода существенно ограничивает возможности применения транслятора. Это не только замедляет его работу, но и в ряде случаев делает её практически невозможной (например, если большой файл читается с устройства последовательного доступа). Поэтому в целях эффективности их применения трансляторы стараются создавать однопроходными.

В однопроходном трансляторе для разрешения «проблемы ссылок вперёд» требуется для каждой процедуры, которая определяется позже, чем встречается её вызов, сначала формировать список тех позиций кода программы, где требуется поместить ссылку на неё (в команде вызова такой процедуры), а затем после того, как будет обработано объявление этой процедуры и сформировано и размещено её тело, потребуется проставить ссылки на неё во всех тех позициях кода, которые были запомнены в сформированном для неё списке. Создание таких списков, конечно же, существенно усложняет однопроходной транслятор и требует дополнительных расходов памяти.

Проблема ещё более усложняется, если такая позже определяемая процедура наделяется параметрами. Чтобы знать заранее, какие у неё параметры, трансляторы как раз и «требуют» предварительно задать хотя бы её заголовок, с тем, чтобы знать, какие команды для подготовки нужных параметров поместить в формируемый код программы непосредственно перед самой командой вызова этой процедуры.

Видимо, все эти возникающие сложности, связанные с разрешением «проблемы ссылок вперёд», и служат причиной, почему не всякий транслятор наделяют способностью «понимать» программу, составленную методом «сверху-вниз» и потому имеющую «ссылки вперёд» на процедуры, определяемые позже их вызова.

3. ДССП и сшитый код

Диалоговая система структурированного программирования (ДССП) была создана [7] в начале 80-х годов XX века в проблемной лаборатории ЭВМ на факультете ВМК МГУ под руководством Брусенцова Н.П. ДССП была призвана существенно облегчить разработку ПО для широкого класса малых цифровых машин – мини- и микрокомпьютеров.

ДССП создавалась в качестве программного эмулятора новой двоичной цифровой минимашины, в которой предполагалось воплотить ценные идеи двухстековой архитектуры и структурированных команд управления, унаследованные от модернизированной троичной ЦМ «Сетунь-70».

ДССП относится к классу языков, построенных на основе интерпретации так называемого сшитого (threaded) кода. К такому же классу относится и язык Форт, известный своей словарной организацией и интерактивным режимом работы.

ДССП создавалась для поддержки структурированной разработки программ и потому способствует соблюдению всех трёх принципов такой разработки. Этим она выгодно отличается от других языков того же класса, в том числе и от Форта.

ДССП не только воспринимает написанную «сверху-вниз» структурированную программу, но ещё и стимулирует дробление такой программы на более мелкие процедуры. Для организации ветвлений и цик-

лов в ДССП предлагается использовать команды условного вызова процедуры и многократно повторяющегося вызова процедуры, принуждая тем самым оформлять тела ветвлений и циклов в виде отдельных процедур.

В ДССП каждая из команд ветвлений вместо перехода по телу сшитого кода (как это обычно делается в системе Форт) по сути осуществляет вызов процедуры слова, выбранного по условию, а каждая команда цикла организует многократный вызов процедуры слова, пока такой вызов допускается условием цикла. Таким образом, в ДССП поддержка структурированного программирования осуществляется непосредственно на уровне сшитого кода аналогично тому, как такая поддержка была обеспечена в ЦМ «Сетунь-70» на уровне машинных команд.

Определение нового слова в ДССП и в языке Форт синтаксически выглядят одинаково:

: P P1 P2 ... Pn ; [определение слова с именем P]

Однако Форт допускает подобное определение, только если все слова P1 P2 ... Pn уже были определены и стали известны системе. А ДССП допускает, что в этой последовательности слов P1 P2 ... Pn могут встречаться в том числе и такие слова, которые еще не были определены и предполагается, что будут определены впоследствии. Именно благодаря такой особенности ДССП и позволяет разрабатывать программу методом нисходящего анализа: сначала определить главное слово, затем слова следующего уровня, из которых оно определяется, и так постепенно спуститься до детального определения самых простых слов.

При создании ДССП в качестве основной требовалось, прежде всего, решить «проблему неопределённых ссылок вперёд». Далее рассмотрим, как решалась эта проблема в версиях ДССП для двоичных машин.

Но прежде вспомним простой приём создания списка позиций употребления неопределённых ссылок, который уже не раз применялся для решения подобной проблемы. Для сохранения списка адресов таких позиций все эти позиции связывались в цепной список так, что в каждой из них помещалась ссылка на предыдущую позицию, где встречался вызов той же не определённой пока процедуры. А начальная ссылка такого списка, указывающая на позицию последнего вызова этой процедуры, помещалась в специальном заголовке таблицы неопределённых процедур. Как только встречалось определение этой процедуры и становился известен адрес её размещения, требовалось пройти по сформированному для неё списку позиций и проставить во всех этих позициях только что назначенный для неё адрес.

Такой приём, в частности, применялся в однопроходном ассемблере ЦМ «Сетунь-70». Почему же этим известным приёмом нельзя было воспользоваться при создании ДССП?

Дело в том, что ДССП предлагает составлять и отлаживать программу в режиме диалога и допускает, что создаваемая программа может быть запущена на пробное исполнение, будучи ещё не до конца реализованной. В ней могут встречаться вызовы слов–процедур, которые ещё не были определены (предполагается, что они будут определены поздне).

Поэтому необходимо позаботиться о том, чтобы при попытке вызова такой не определённой пока процедуры диалоговая система смогла бы продолжить своё нормальное функционирование, сообщив лишь о встреченном ею вызове неопределённой процедуры. А это значит, в тех позициях, где остаются неразрешённые ссылки, требуется на некоторое время поместить особые команды для вызова пустой процедуры–заглушки, а ещё лучше — встроенного отладчика (если он имеется), который бы позволил в режиме диалога исполнить вместо вызванной процедуры разумную последовательность заменяющих её действий.

Вот почему при создании интерпретатора ДССП для двоичных машин не представлялось возможным применить описанный выше прием, применённый в ассемблере для ЦМ «Сетунь-70»: ведь чтобы построить такой цепной список, потребовалось бы разместить на таких позициях ссылки для его продолжения, а они могли бы помешать нормальному функционированию интерпретатора.

Для решения «проблемы неопределённых ссылок вперёд» в первой версии ДССП (ДССП-НЦ [8]) был специально предложен новый вид сшитого кода, получивший название «сшитый код двойной косвенности». Чтобы объяснить его суть, сначала познакомимся с разновидностями [9] сшитого кода, которые до этого применялись при построении Форт-подобных систем.

1. *Процедурный сшитый код* (*Subroutine Threaded Code* — STC). Так согласно принятой классификации (см. [9]) называют код, состоящий из машинных команд вызова подпрограмм (процедур):

call adrP1; call adrP2; … call adrPn;

который непосредственно может исполняться базовой машиной, т.е. той, на которой функционирует интерпретатор Форт-подобной системы.

2. *Прямой сшитый код* (*Direct Threaded Code* — DTC). Такой код, в основном, состоит только из адресов вызываемых подпрограмм:

IntProlog; adrP1; adrP2; … adrPn; adrRet;

Но впереди у него помещается код-пролог, который как раз и организует (запускает) необходимую последовательность вызовов процедур, задаваемыми этими адресами. В конце списка адресов помещается адрес особой процедуры, которая завершает последовательность таких вызовов.

Команда вызова подпрограммы, как правило, складывается из двух частей: кода команды и адреса подпрограммы. Поэтому хранить программу в DTC-коде оказывается выгодным, так как для её размещения потребуется значительно меньший объём памяти.

3. *Косвенный сшитый код* (*Indirect Threaded Code* — ITC). В таком коде размещаются ссылки на тела, в первой позиции которых задаётся либо адрес процедуры, реализующей слово-примитив, либо адрес процедуры, которая запускает интерпретацию нового уровня вызовов процедур из стоящего следом списка ссылок:

ptrInt; ptrP1; ptrP2; … ptrPn; ptrRet

В отличие от DTC-кода, который начинает исполняться как машинный код, для ITC-кода предполагается, что он начинает исполняться интерпретатором.

4. *Косвенный знаковый сшитый код* (*Indirect Token Threaded Code* — ITTC). В отличие от ITC-кода в этом виде кода вместо обычных ссылок применяются ин-

дексы, по которым из специально заготовленной таблицы выбираются адреса подпрограмм, подлежащих вызову:

indInt; indP1; indP2; … indPn; indRet

Индексы – это числа небольшого диапазона, поэтому список из них занимает гораздо меньше места в памяти, чем список ссылок, состоящий из обычных машинных адресов.

Сшитый код двойной косвенности строится аналогично ITTC-коду, только для ссылки на вызываемую процедуру вместо индексов используются обычные указатели на помещённые в словарь специальные заголовки слов, в первой позиции которых задаётся ссылка на тело реализующей это слово процедуры, либо ссылка на процедуру вызова встроенного отладчика в случае, когда определение для такого слова ещё не встретилось. При последующем определении такого слова, потребуется исправить лишь ссылку на тело этого слова в его заголовке, и ничего не потребуется изменять в тех позициях сформированного кода ДССП-программы, где встречался вызов этого слова, поскольку в таких позициях всегда проставляется указатель на заголовок этого слова. А этот указатель получает значение только однажды и далее уже не меняется. Оно фиксируется в момент, когда в словарь добавляется заголовок этого слова, если оно опознано как новое, неизвестное ранее (т.е. не найденное в словаре). Такой момент возникает либо когда это слово определяется (т.е. задаётся его тело), либо когда оно впервые употребляется (вызывается) в теле другого слова.

Реализованный на микрокомпьютере «НЦ-03Д» интерпретатор ДССП-НЦ, чтобы вызвать исполнение очередного слова, на которое ссылался текущий указатель по телу сшитого кода двойной косвенности, вынужден был потратить 7 машинных команд вместо 5-ти, если бы он был реализован для косвенного сшитого кода (как для Форта). Таким образом, возможность программирования «сверху-вниз» достигалась в ДССП-НЦ ценой лишь незначительного снижения производительности интерпретатора.

Но в последующей версии ДССП-80 для микрокомпьютеров унифицированной архитектуры (PDP-11) снижение производительности обещало уже быть более весомым. Ведь основной цикл внутреннего интерпретатора удавалось реализовать всего одной командой базовой машины (JSR @R5+), если для внутреннего представления ДССП-программы использовать прямой сшитый код. Поэтому ради эффективности интерпретатора было предложено решить «проблему неопределённых ссылок вперёд» другим способом.

Подробно этот способ описан в [10]. Основная идея этого решения заключается в том, что в словаре вместе с заголовком неопределённого слова хранится в форме особой таблицы список адресов тех позиций тела формируемого кода ДССП-программы, где встречался вызов этого неопределённого слова. В самих же этих позициях временно помещается ссылка на встроенный отладчик. А когда встречается определение этого слова, то ссылка на тело этого слова проставляется во всех позициях, запомненных в такой таблице, а сама таблица расформировывается.

Конечно же, формирование и хранение такой таблице в заголовке каждого слова существенно усложняет структуру ДССП-словаря. Но с таким усложнением пришлось согласиться ради быстродействия ДССП-интерпретатора. В дальнейшем словарь такой сложной структуры использовался во всех последующих версиях ДССП, созданных для различных операционных сред и разнообразных архитектур двоичных машин.

4. Обеспечение нисходящей разработки в ДССП для ТВМ

При разработке ДССП-ТВМ предстояло решить проблему, какую же разновидность сшитого кода применять для внутреннего представления, чтобы ТВМ могла напрямую (т.е. без специального интерпретатора) вызывать процедуры по встречающимся в этом коде адресам. Для решения этой проблемы было предложено применить в ТВМ такую кодировку команд, чтобы значение адреса воспринималось ею и как команда вызова процедуры с этого адреса. В результате для внутреннего представления программ в ДССП-ТВМ стал использоваться так называемый процедурный сшитый код (STC согласно классификации, приведённой в [9]), который может непосредственно исполняться троичной машиной.

Кросс-компилятор формирует такой код на языке ассемблера ТВМ. В нём для каждого слова-примитива ДССП-ядра размещается одна машинная команда, которая реализует действие этого примитива. Для тех примитивов, действие которых невозможно исполнить одной машинной командой, заготовлены ассемблерные процедуры, их реализующие, а в сшитый код вставляется адрес (или команда вызова) соответствующей ей процедуры. Совокупность таких ассемблерных процедур образует ассемблерное ядро, которое добавляется к каждой откомпилированной ДССП-программе.

Поскольку кросс-компилятор ДССП-ТВМ формирует код ДССП-программы на ассемблере ТВМ, то он тем самым просто перекладывает решение проблемы «неопределённых ссылок вперёд» на транслятор языка ассемблера, который разрешает неопределённые ссылки путём осуществления двойного прохода текстового файла, подаваемого ему на обработку. Интерпретатор же ДССП/ТВМ должен решать такую проблему самостоятельно.

Ранее были рассмотрены возможные варианты решения данной проблемы в версиях ДССП, реализованных для двоичных машин: либо путём применения для внутреннего представления ДССП-программы сшитого кода двойной косвенности (в ДССП-НЦ]), либо за счёт усложнения словаря путём хранения в нём для каждого встреченного (но не определённого ещё) слова специальных таблиц, содержащих адреса позиций тел, в которых было использовано это слово.

Но такие варианты решения этой проблемы при создании интерпретатора ДССП/ТВМ потребовали бы серьёзной переделки и самого ДССП-компилятора, и построенного для него ассемблерного ядра. Ведь интерпретатор должен уметь работать с тем же строением словаря и с тем внутренним представлением ДССП-программ, которые поддерживаются кросс-

компилятором и ассемблерным ядром системы разработки ДССП-ТВМ.

В ходе разработки интерпретатора ДССП/ТВМ удалось найти иное решение данной проблемы, не потребовавшее каких-либо модификаций ни словаря, ни внутреннего представления ДССП-программы. Рассмотрим предлагаемый способ разрешения неопределённых ссылок и объясним, почему он сделан именно таким.

Для каждого встреченного неопределённого слова ДССП-интерпретатор должен уметь, по крайней мере, запоминать адреса всех позиций в телах, где такое слово было вызвано. А встретив определение этого слова, проставить во всех запомненных позициях ссылку на сформированное тело этого слова.

Для сохранения адресов таких позиций можно связать все эти позиции в цепной список, записав в каждой из них ссылку на предыдущую позицию, где встречался вызов этого неопределённого слова. А начальную ссылку такого списка, указывающую на позицию последнего вызова этого слова, можно разместить в заголовке неопределённого слова (в поле команды).

Такой известный приём построения списка позиций употребления неопределённых слов нами ранее уже рассматривался. Существенный недостаток такого приёма сохранения списка позиций проявляется при попытке запускать в среде интерпретатора ДССП-программу, содержащую вызовы неопределённых слов. Такие вызовы в прежних реализациях ДССП-интерпретаторов могли бы вызывать непредсказуемые последствия, поэтому разработчики ДССП вынуждены были отказываться от такого привлекательного приёма сохранения списка позиций.

Но в создаваемом интерпретаторе ДССП/ТВМ описанный приём сохранения списка позиций можно выгодно использовать, если поставить в конце созданного списка (т.е. в той позиции тела, где неопределённое слово встретилось первый раз) адрес специальной процедуры, назначенной для общей реакции на вызов неопределённого слова (см. рис. 1, вариант А).

Воспользуемся тем, что ТВМ воспринимает 26-тритный адрес как команду вызова процедуры с этого адреса. Значит, при вызове такой команды с одной из позиций, используемой в построенном цепном списке, произойдет череда вызовов процедур по адресам из этого списка и, в конечном итоге, будет вызвана назначенная спецпроцедура реакции.

Предполагается, что в такой процедуре реакции интерпретатор может предложить разработчику ДССП-программы выполнить действия, имитирующие эффект исполнения не определённой пока процедуры, и затем вернуться к продолжению прерванной программы. (В дальнейшем такая спецпроцедура реакции, возможно, станет компонентой специализированного отладчика в составе интерпретатора).

Поэтому спецпроцедура реакции должна, во-первых, уметь распознать, какое неопределённое слово было вызвано, т.е. получить адрес его словарного заголовка, и во-вторых, обеспечить продолжение прерванной программы с той точки, где было вызвано неопределённое слово.

По адресу возврата из спецпроцедуры реакции можно узнать адрес позиции, откуда она была вызвана. В этой позиции должен завершаться цепной список, построенный для вызванного неопределённого слова. Далее потребуется осуществить обход словаря и для каждого неопределённого слова (помеченного флагом U) проверить, завершается ли его цепной список именно в этой позиции. Так можно узнать, какое неопределённое слово было вызвано.

Но к сожалению, в ДССП не во всех случаях по адресу возврата можно правильно определить позицию, откуда была вызвана процедура (у нас это спецпроцедура реакции). Например, её нельзя однозначно определить, если процедура вызывалась управляющими командами: BR+ BRS BR DW EXEC .

Чтобы обеспечить возможность правильного определения позиции конца цепного списка по адресу возврата из спецпроцедуры, потребуем, чтобы он заканчивался на позиции, где поставлена команда явного вызова спецпроцедуры реакции на неопределённое слово. Такая дополнительная позиция требуется для каждого неопределённого слова на время работы с его цепным списком. Для таких позиций будем выделять память динамически, используя для этой цели свободные (нулевые) ячейки специально организованной таблицы, размещённой в памяти так, чтобы не мешать формированию тела и словаря ДССП-программы (см. рис. 1 вариант Б).

При определении тела неопределённого слова, когда во все позиции его цепного списка заносится адрес формируемого тела, сам цепной список расформировывается, а отведённая ему ячейка спецтаблицы освобождается (она помечается нулевым значением) и может быть распределена снова.

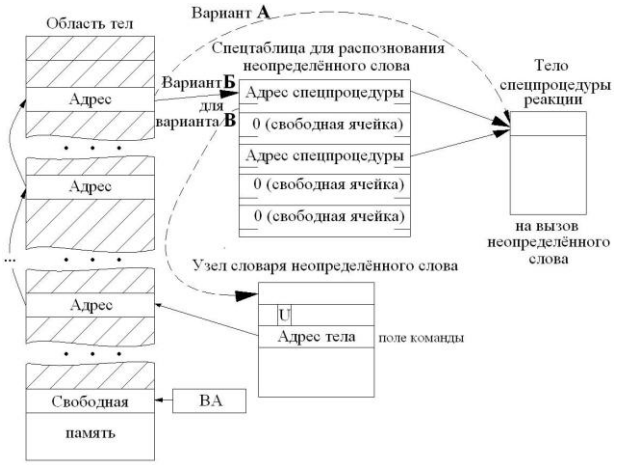

рис. 1. Схемы вариантов решения проблемы неопределённых слов.

Предложенный вариант (Б) позволяет не только однозначно распознать, какое неопределённое слово было вызвано, но и вернуться из спецпроцедуры реакции к продолжению прерванной программы. Заметим, что для обеспечения правильного возврата (именно в ту точку, где было вызвано неопределённое слово), процедура реакции должна будет очистить стек возвратов от “лишних” адресов, накопившихся в нём в результате промежуточных вызовов процедур по адресам из цепного списка.

В данном варианте для распознавания неопределённого слова требуется осуществлять обход словаря и

цепных списков. Можно значительно ускорить процесс поиска неопределённого слова, если в спецтаблице рядом с позицией конца цепного списка сохранять ссылку на заголовок неопределённого слова, для которого этот список построен (см. рис. 1 с вариантом В). Но тогда потребуется усложнить процедуру передвижения словаря, т.к. ей придётся ещё и корректировать такие ссылки, сохраняемые в спецтаблице.

Если даже в качестве таких ссылок на узлы словаря использовать не абсолютные, а относительные адреса, то их всё равно пришлось бы корректировать: если не при передвижении словаря, так при его очистке, выполняемой операцией CLEAR .

Усложнение операций передвижения и очистки словаря может замедлять регулярную работу интерпретатора по построению ДССП-программы. А длительный поиск неопределённого слова может проявиться лишь в виде паузы при реакции интерпретатора на вызов неопределённого слова. Поскольку такая реакция, как правило, сама предусматривает приостановку исполнения ДССП-программы с выходом в режим диалога, то данная пауза вряд ли будет заметна. Поэтому в настоящее время для решения проблемы неопределённых слов в интерпретаторе ДССП/ТВМ используется вариант Б, изображённый на рисунке 1 как основной.

7. Заключение

Для полноценной поддержки структурированного программирования необходимо не только предусмотреть в языке программирования управляющие конструкции для структурированного кодирования создаваемой программы, но и обеспечить возможность её пошаговой нисходящей разработки. ДССП является одной из немногих оригинальных систем, которая обеспечивает такую поддержку.

При реализациях ДССП для двоичных машин проблема обеспечения нисходящей разработки требовала либо усложнения строения сшитого кода, либо усложнения структуры словаря. Но при реализации ДССП для троичной машины (ТВМ) удалось найти такое решение проблемы, которое позволило и упростить ДССП-словарь, и применить для внутреннего представления ДССП-программ такой вариант процедурного сшитого кода, который может напрямую непосредственно исполняться созданной виртуальной троичной машиной.

Subroutine threaded code to support top-down development of structured programs in DSSP for ternary computer

A.A. Burtsev

Abstract: In research laboratory of ternary informatics leaded by Brusentsov N. P. at the Computer Science (CS) department of the Moscow State University (MSU) the ternary virtual machine (TVM) and cross-system (DSSP-TVM) for development of programs for it using the DSSP-T language are created. Ternary virtual machine (TVM) is a simulator of ternary computer, which architecture has two stacks (data stack and control stack) and machine commands for structured programming like «Setun-70». Two variants of Dialogue System of Structured Programming (DSSP) for TVM have been constructed. DSSP-TVM lets it possible to create DSSP-program for TVM by means of cross-compiler. DSSP/TVM is a dialogue interpreter, which can run on TVM as its resident soft-ware. Programming language DSSP-T used in both systems is a ternary version of language of the DSSP.

The main problem during realization of the DSSP-TVM cross-compiler and the DSSP/TVM dialogue interpreter is how to support top-down development of structured program. Variants of solving this problem in DSSP for TVM are explained.

Keywords: structured programming, threaded code, top-down development, ternary computer, DSSP.

Литература

1. Сидоров С.А., Владимирова Ю.С. Троичная виртуальная машина // Программные системы и инструменты. Тематический сборник №12, М.: Изд-во факультета ВМиК МГУ,2011. С.46-55.

2. Бурцев А.А., Рамиль Альварес Х. Кросс-система разработки программ на языке ДССП для троичной виртуальной машины // Программные системы и инструменты. Тематический сборник №12, М.: Изд-во факультета ВМиК МГУ,2011. С.183-193.

3. Бурцев А. А., Бурцев М. А. ДССП для троичной виртуальной машины // Труды НИИСИ РАН, т. 2, № 1.— М.: Изд-во НИИСИ РАН, 2012.— С. 73–82.

4. Дал У., Дейкстра Э., Хоор К. Структур(ирован)ное программирование.— М.: Изд-во Мир, 1975.— 247 с.

5. Knuth D. Structured programming with GOTO statements // ACM Computer Survey, 1974, v.6, p.261-301.

6. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ.— М.: Изд-во Мир, 1981.

7. Бурцев А.А., Сидоров С.А. История создания и развития ДССП: от "Сетуни-70" до троичной виртуальной машины. // Вторая Международная конференция "Развитие вычислительной техники и её программного обеспечения в России и странах бывшего СССР" SORUCOM-2011 (12-16 сентября 2011 г., г. Великий Новгород, Россия): Труды конференции. В.Новгород: Изд-во НовГУ, 2011. с. 83-88.

8. Захаров В. Б., Златкус Г. В., Руднев И. А., Сидоров С. А. Реализация диалоговой системы структурированного программирования на микрокомпьютере «Электроника НЦ-03Д» // Архитектура и программное оснащение цифровых систем.— М.: Изд-во МГУ, 1984, с. 10–17.

9. Ritter T., Walker G. Varieties of threaded code for language implementation // BYTE, 1980, v. 5, No. 9, p.206.

10. Сидоров С. А. Программирование сверху вниз и организация словаря ДССП // Вопросы кибернетики. Сборник статей / ред. В. Б. Бетелин. М.: Изд-во НИИСИ, 1999.— С. 32–44.