(12) UK Patent Application (19) GB (11) 2503589 (13) A

(43) Date of A Publication 01.01.2014

(21) Application No: 1314580.0

(22) Date of Filing: 14.08.2013

(71) Applicant(s):
Micro Focus IP Development Ltd
The Lawn, 22-30 Old Bath Road, NEWBURY, Berkshire, RG14 1QN, United Kingdom

(72) Inventor(s):
Jeremy Wright

(74) Agent and/or Address for Service:
EIP
Fairfax House, 15 Fulwood Place, LONDON, WC1V 6HU, United Kingdom

(51) INT CL:
G06F 9/45 (2006.01)

(56) Documents Cited:

US 20120096444 A1	**US 20110307507 A1**
US 20060200811 A1	**US 20040154009 A1**

(58) Field of Search:
INT CL **G06F**
Other:
EPODOC,WPI,TXTE,INSPEC,INTERNET,XPESP,XPI3E, XPIPCOM

(54) Title of the Invention: **Processing for application program deployment**
Abstract Title: **Compiling an application program written in legacy source code according to stack or legacy semantics based on equivalence**

(57) A computer system (1200) prepares an application program (1210) comprising legacy source code written in, for example, COBOL for deployment into a processing environment (1260). The system has a control flow analyser unit (1242) operable to determine equivalence in control flow between source code if compiled using stack semantics and if compiled legacy semantics. Determination of equivalence involves generating and inspecting a directed call graph to identify strongly connected components. In the absence of such, tuples comprising ranges of functions, end points of functions and reaching endpoints of direct/indirect predecessor nodes are created and a union of the range and the reaching end point calculated. If the union is NULL and there are no strongly connected components then equivalence exists. The system has a compiler unit (1243) to compile the legacy source code using stack based semantics if equivalence is determined and using legacy semantics if non-equivalence is determined. This compilation may be to intermediate code such as Java Bytecode or common intermediate language (CIL) code.

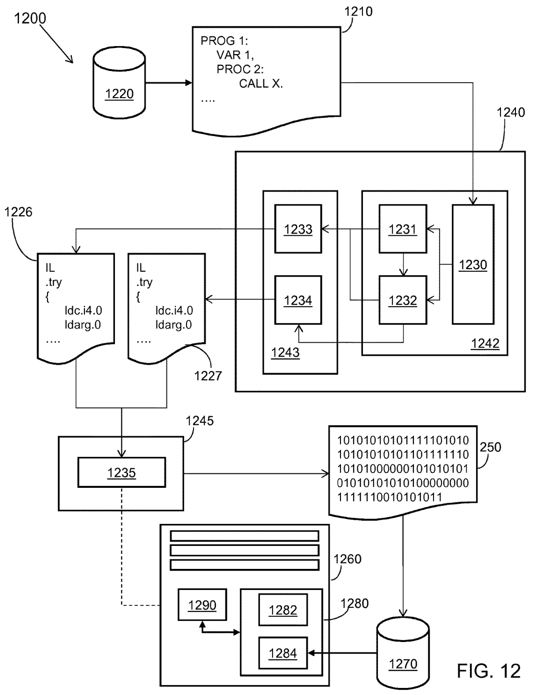


FIG. 12

GB 2503589 A

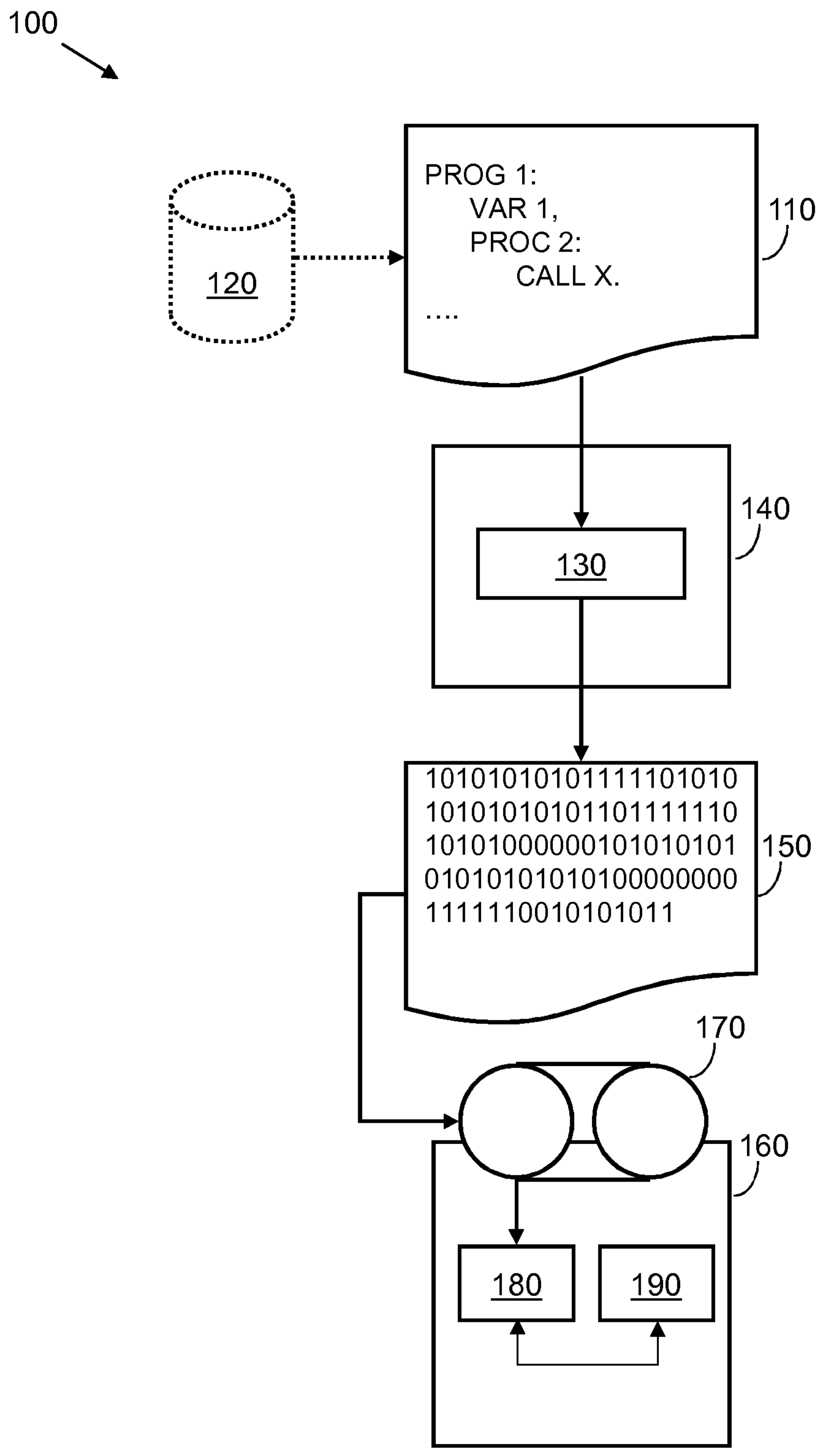

100
120
PROG 1:
VAR 1,
PROC 2:
CALL X.
....
110
140
130
1010101010101111101010
1010101010101101111110
1010100000010101010101
0101010101010100000000
1111110010101011
150
170
160
180
190

FIG. 1

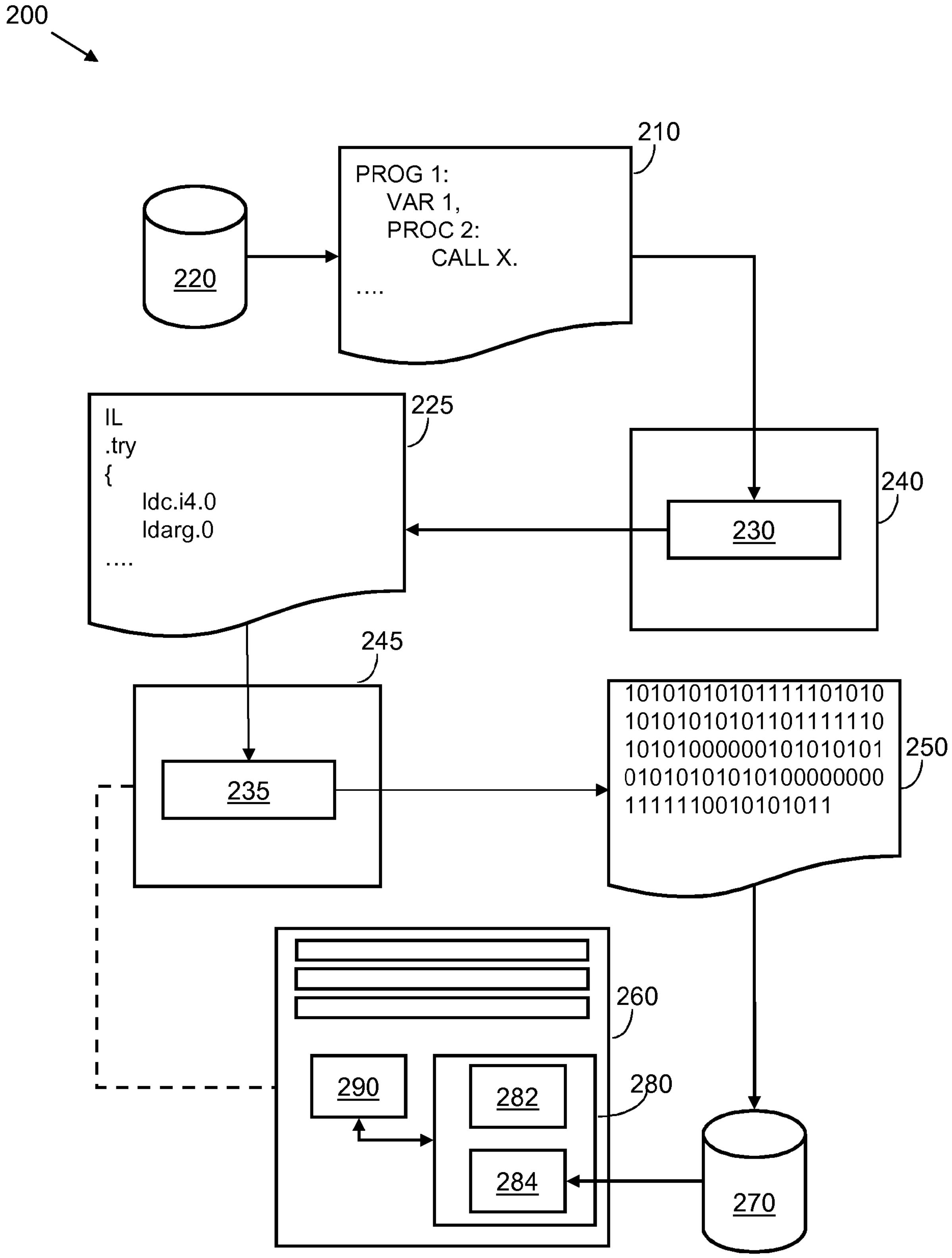
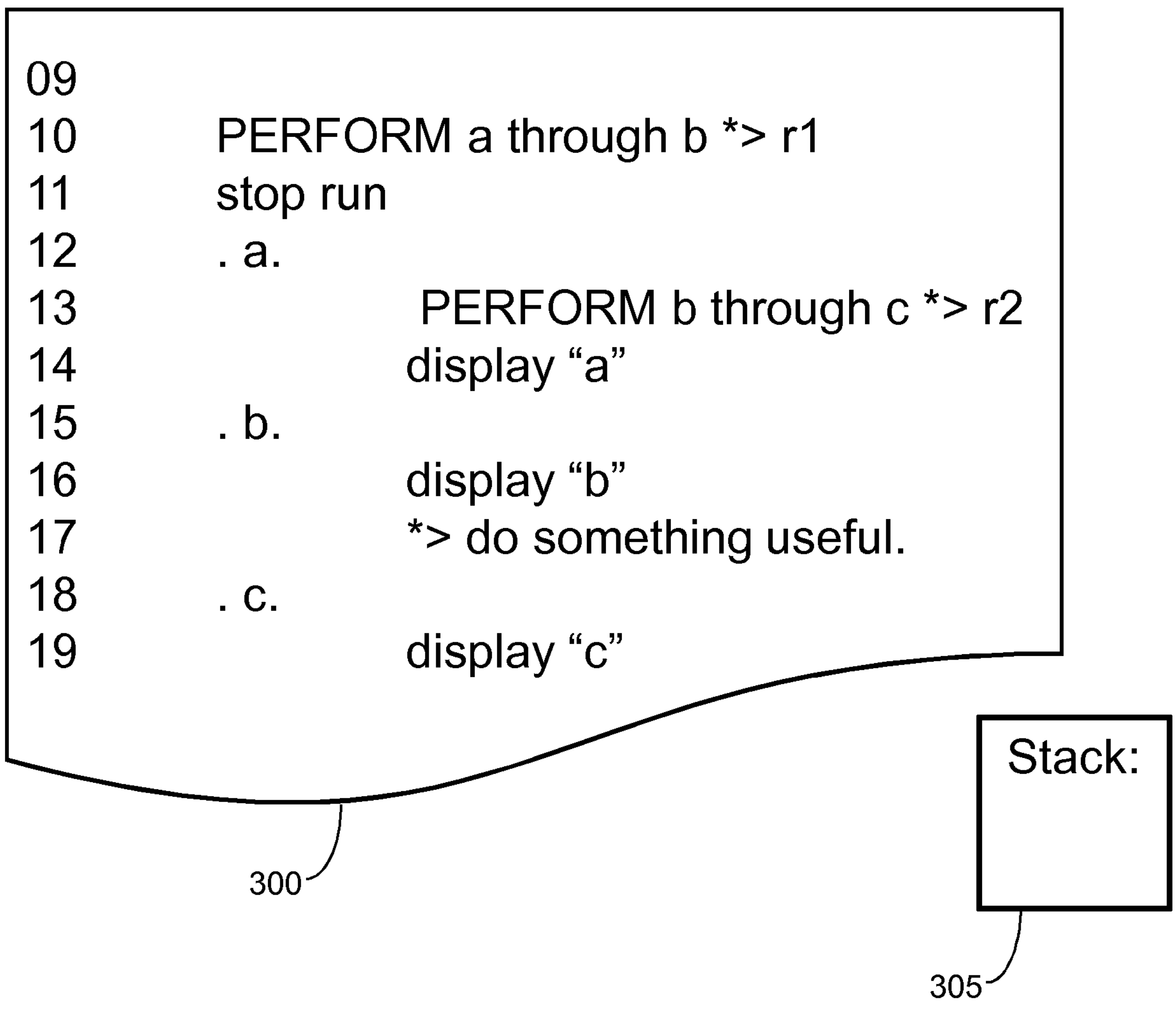

200
210
PROG 1:
VAR 1,
PROC 2:
CALL X.
....
220
225
IL
.try
{
ldc.i4.0
ldarg.0
....
230
240
245
235
250
10101010101111101010
10101010101101111110
10101000000101010101
01010101010100000000
1111100101010101
260
290
280
282
284
270

FIG. 2



```

        PERFORM a through b *> r1
        stop run
        . a.
                 PERFORM b through c *> r2
                display “a”
        . b.
                display “b”
                *> do something useful.
        . c.
                display “c”
```

FIG. 3

Line	Stack Control	Stack Contents	Active End Point	Display
9				
10	Push b, r1	(b, r1)	End of b, line 17	
12		(b, r1)	End of b, line 17	
13	Push c, r2	(c, r2) (b, r1)	End of c, line 19	
15		(c, r2) (b, r1)	End of c, line 19	
16		(c, r2) (b, r1)	End of c, line 19	b
17		(c, r2) (b, r1)	End of c, line 19	
18		(c, r2) (b, r1)	End of c, line 19	
19		(c, r2) (b, r1)	End of c, line 19	c
14	Pop c, r2	(b, r1)	End of b, line 17	a
15		(b, r1)	End of b, line 17	
16		(b, r1)	End of b, line 17	b
17	Pop b, r1		End of b, line 17	
11				

FIG. 4

Line	Exit Bucket Control	Exit Bucket Contents	Active End Points	Output
9				
10	Add b, r1	(b, r1)	End of b, line 17	
12		(b, r1)	End of b, line 17	
13	Add c, r2	(b, r1) (c, r2)	End of b, line 17 End of c, line 19	
15		(b, r1) (c, r2)	End of b, line 17 End of c, line 19	
16		(b, r1) (c, r2)	End of b, line 17 End of c, line 19	b
17	Clear b	(c, r2)	End of b, line 17 End of c, line 19	
11				

FIG. 5

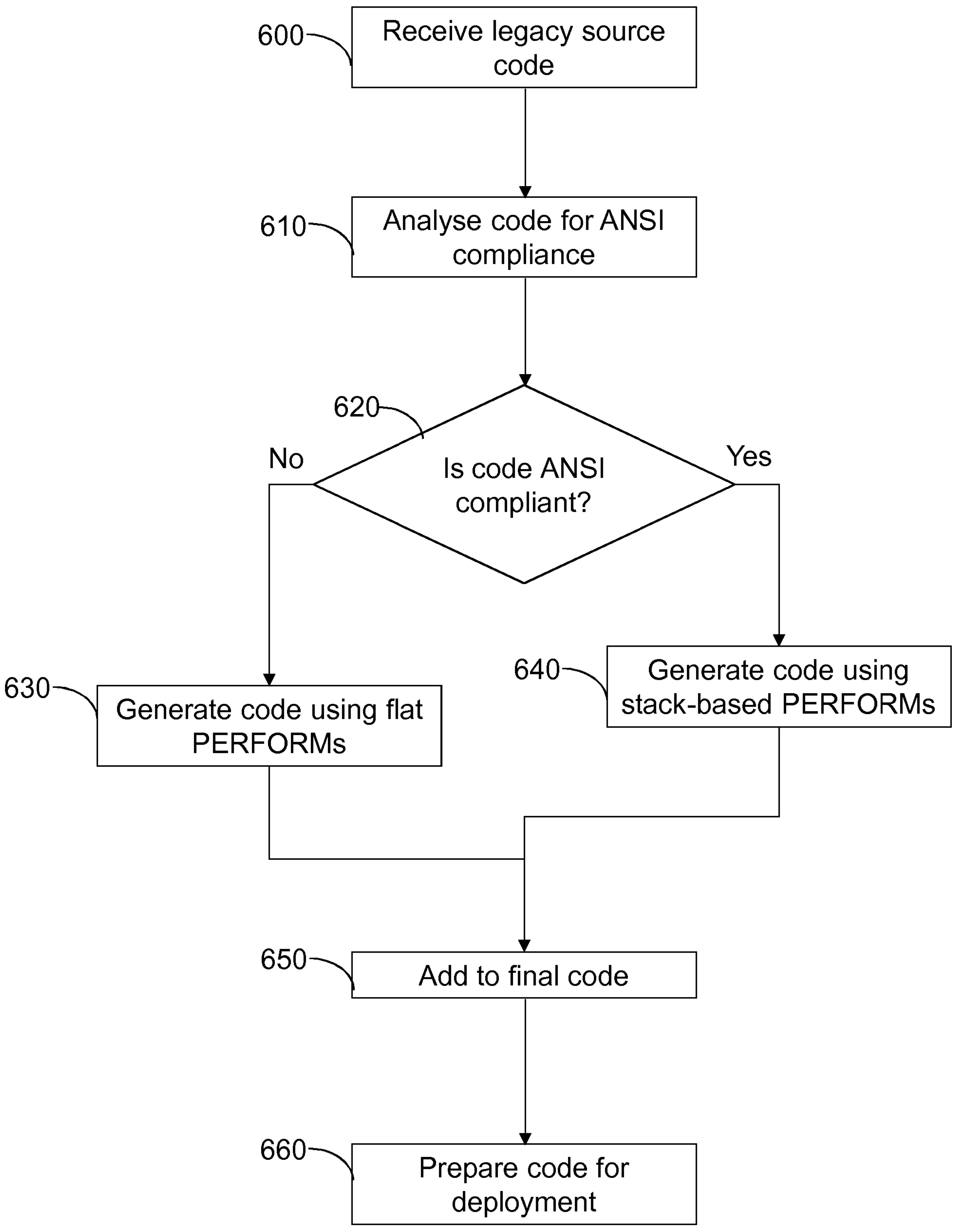

600
Receive legacy source code
610
Analyse code for ANSI compliance
620
Is code ANSI compliant?
No
Yes
630
Generate code using flat PERFORMs
640
Generate code using stack-based PERFORMs
650
Add to final code
660
Prepare code for deployment

FIG. 6

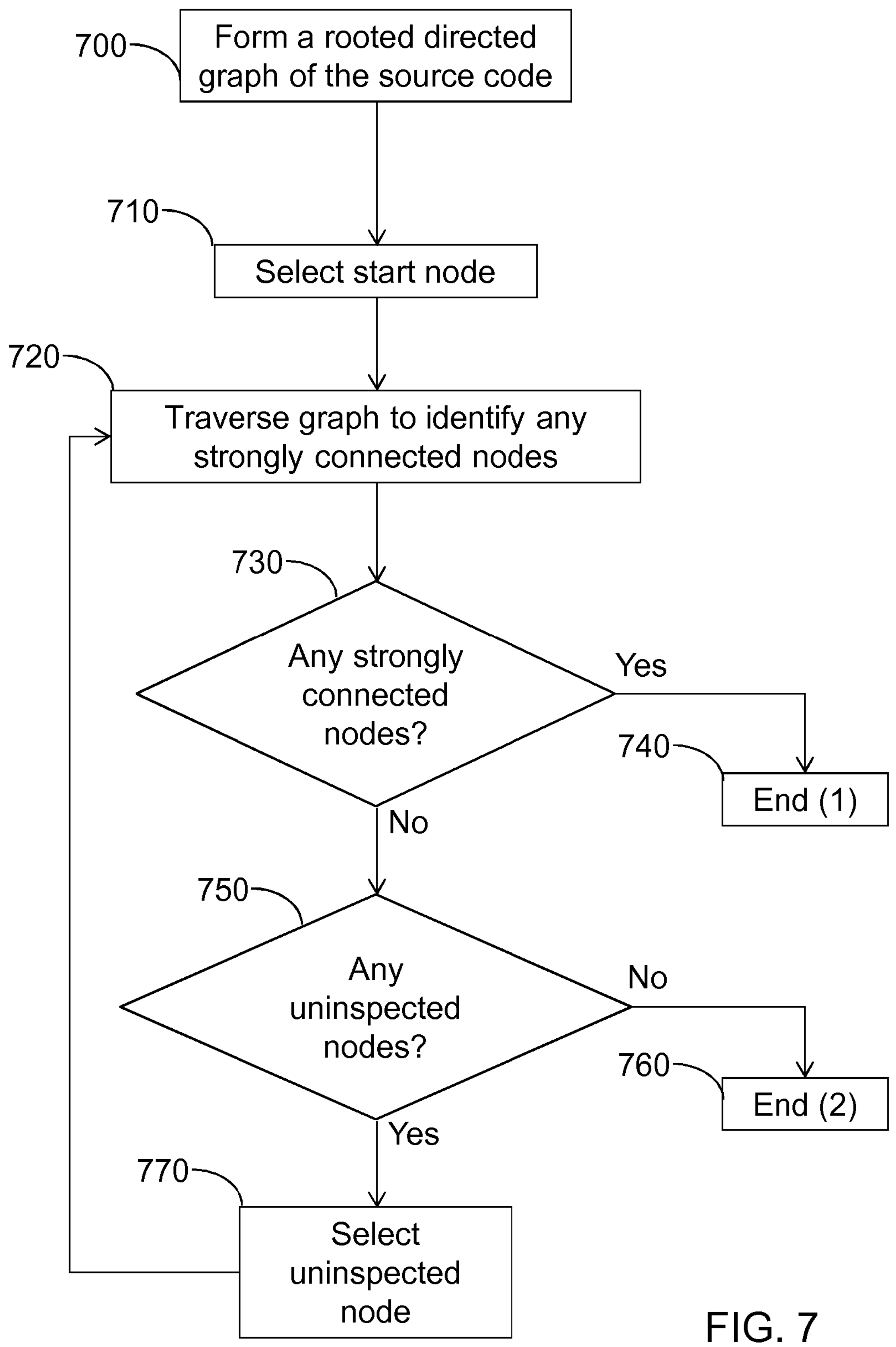

700
Form a rooted directed graph of the source code
710
Select start node
720
Traverse graph to identify any strongly connected nodes
730
Any strongly connected nodes?
Yes
740
End (1)
No
750
Any uninspected nodes?
No
760
End (2)
Yes
770
Select uninspected node

FIG. 7

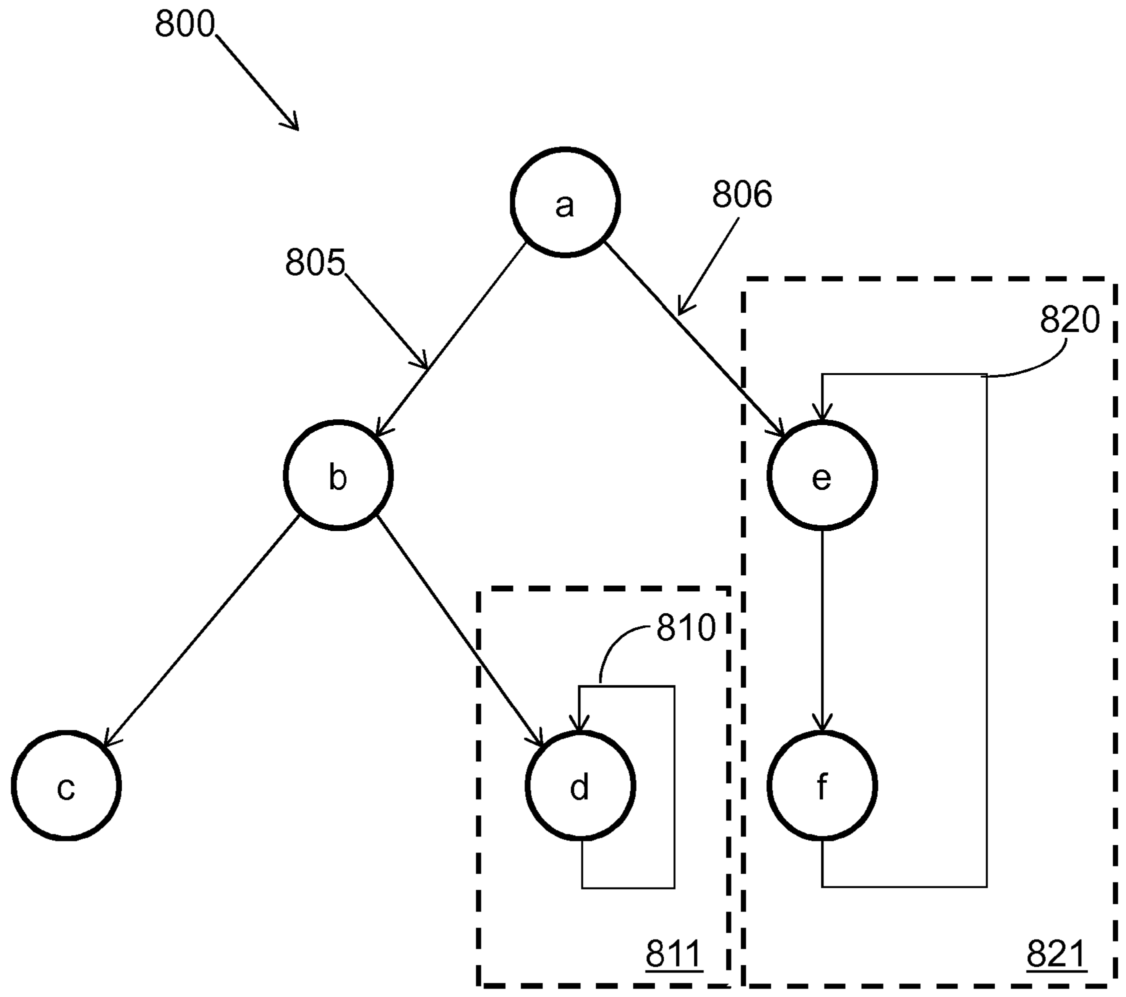


FIG. 8

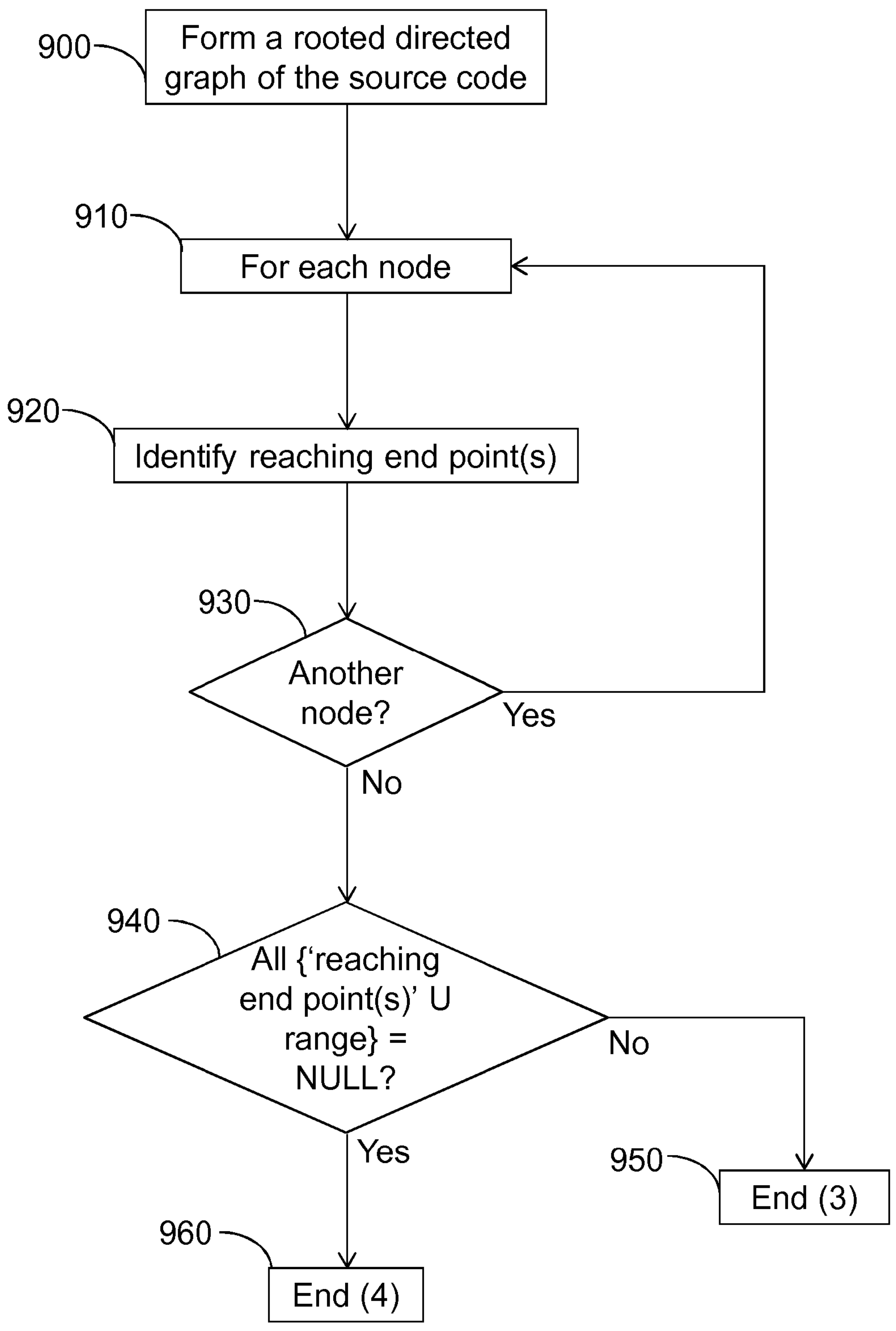

900
Form a rooted directed graph of the source code
910
For each node
920
Identify reaching end point(s)
930
Another node?
Yes
No
940
All {‘reaching end point(s)’ U range} = NULL?
No
Yes
950
End (3)
960
End (4)

FIG. 9

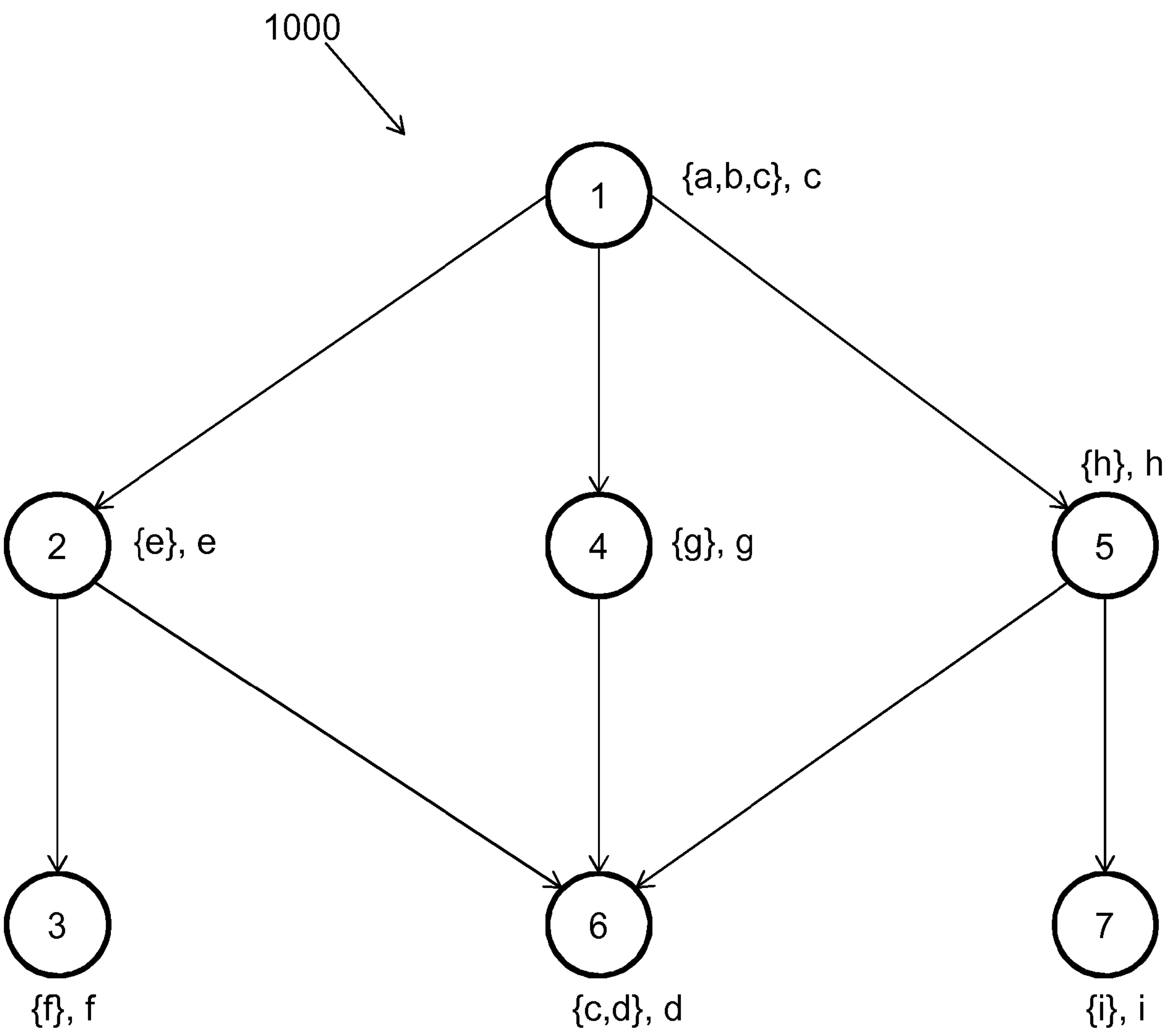


FIG. 10

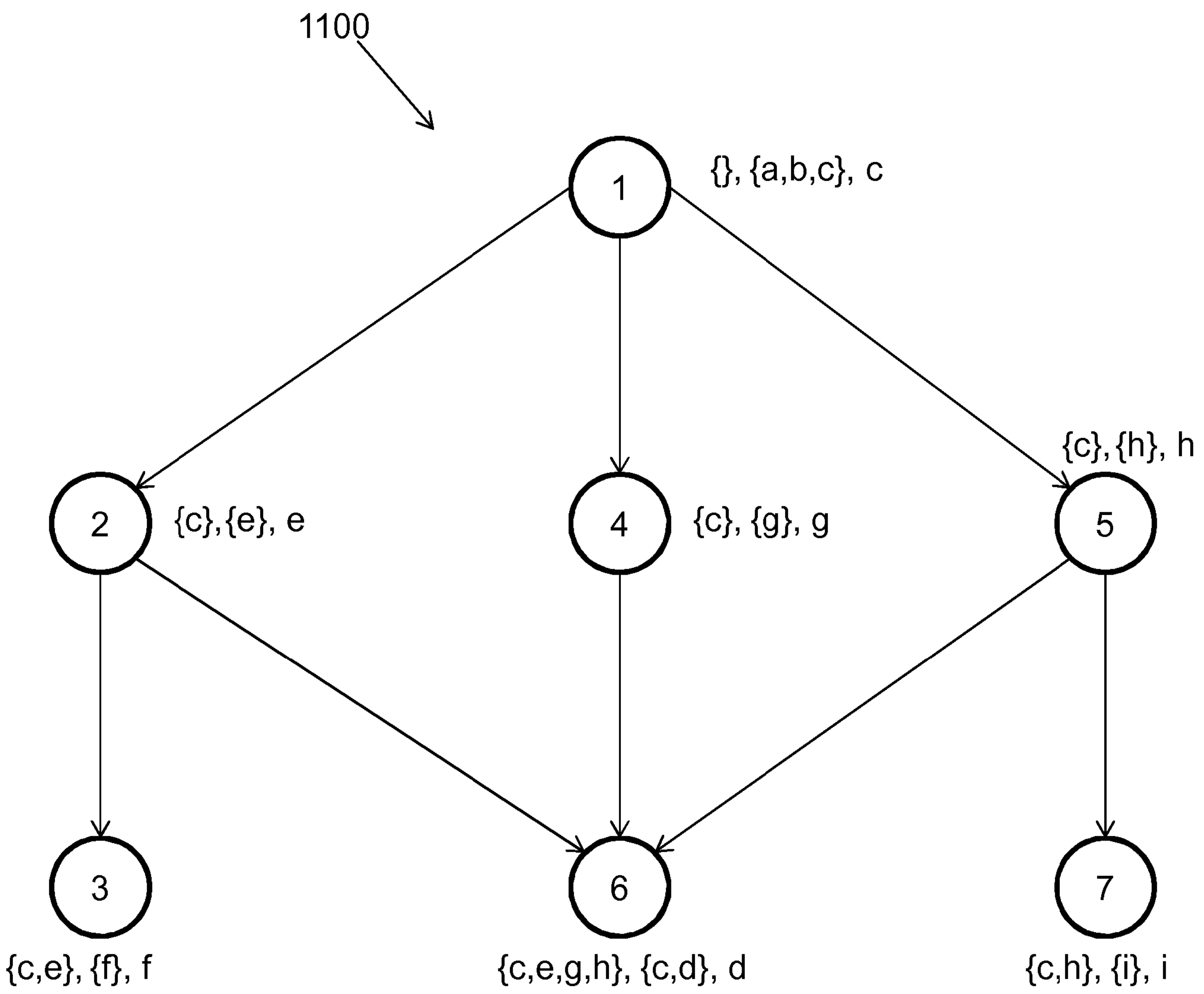
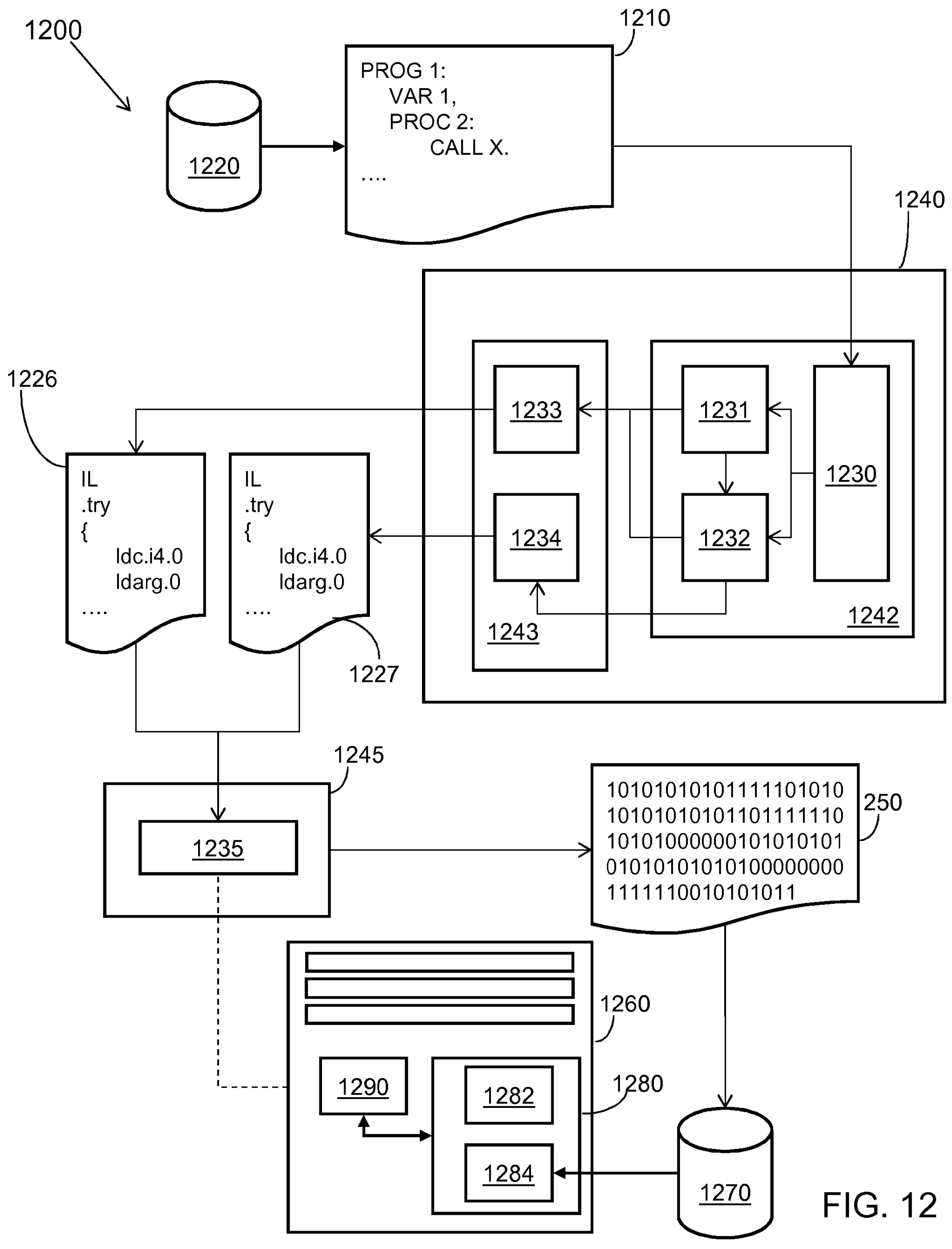


FIG. 11



1200
1220
1210
PROG 1:
VAR 1,
PROC 2:
CALL X.
....
1240
1226
IL
.try
{
ldc.i4.0
ldarg.0
....
IL
.try
{
ldc.i4.0
ldarg.0
....
1227
1233
1234
1231
1232
1230
1242
1243
1245
1235
250
1010101010101111101010
1010101010110111111110
1010100000010101010101
0101010101010000000000
111111001010101011
1260
1290
1282
1284
1280
1270

FIG. 12

PROCESSING FOR APPLICATION PROGRAM DEPLOYMENT

Technical Field

The present invention relates to one or more of preparation and deployment of an application program into a processing environment.

Background

There is often an inherent tension in computing technology between the pace of change of computer architectures and the need for stable, reliable solutions that operate for decades. Before personal computers entered the market in the 1980s and 1990s, the concept of "computing" as a technical field was associated with large-scale, centralised "mainframe" computing systems. These systems were often cabinet or room-sized and were used for mission-critical tasks in large organisations, scientific institutions and government. In the 1960s and 1970s many mainframe computing systems were manufactured and supplied by International Business Machines Corporation (IBM®). For example, well-known systems included the IBM 700/7000 series, and those based on the System/360 and System/370 computing architectures. Many features that are now taken for granted, such as transistor then complementary metal-oxide-semiconductor (CMOS) central processing units (CPUs), 8-bit bytes, 32-bit words, hardware floating point support and 32-bit words were introduced slowly over decades in these systems.

In parallel with the development of modern computing architectures based on mainframe innovations, there was a need to better control computing operations. Although difficult to comprehend in the modern era, the System/360 computing architecture was one of the first computers in wide use to include dedicated hardware provisions for the use of an operating system. Before that time, mainframe systems required computer program code to be manually initiated, often on punched cards. Early mainframe systems were controlled and programmed using system-specific machine code. Over time, mnemonic assembly languages developed, which added support for macro instructions. This then led into the development of the first system-independent programming languages, languages such as FORTRAN, LISP, COBOL, PL/1, PL/M and ALGOL 60. Grace Hopper, an early computing pioneer, developed the first compiler and later the COBOL language. Much of the work in this era paved for way for technical aspects of digital computing that are considered fundamental today, such as subroutines, formula translation, relative addressing, the linking loader, code optimization and symbolic manipulation.

System-independent programming languages, and the compiler technology that allowed system-independent program code to be deployed as machine code to specific hardware architectures, greatly improved the portability of computer programs, benefiting the control of computing devices regardless of specific program content or function. It also enabled a computer program to continue to control a computer system as a computer architecture evolved. Hardware features could be added to the computer system or architecture without requiring a rewrite of computer program code.

The success of system-independent programming languages brings its own set of technical problems. As computer programs could survive upheaval in physical architectures, resources could be invested in building stable, mission-critical computing functions. These functions could be designed to be operated for decades, human lifetimes even. For example, over the years many control, data recordal and file handling functions were developed, embodied in computer program code, and became the foundation of modern engineering and production systems. As more functionality was added, this was often achieved by appending additional computer program code to existing computer programs. Moreover, if program code was superseded, it was not uncommon to add new, replacement code and leave the old code in-situ, just in case it is invoked by another part of the program. At the turn of the modern Internet era in the twenty-first century, many organisations were reliant on computer program code originally written and developed in the 1960s and 1970s.

As hardware developments have accumulated and incorporated global networking aspects, modern computer systems are required to operate in an environment quite different from the early eras of mainframe computing. However, as described above, organisations, entities and control systems are deeply reliant on legacy computer applications and program code, which were created in programming languages that date from these earlier eras. Migration is one option but may not be entirely possible. Legacy computer programs can run to millions of lines of computer program code, contributed by thousands of different programmers over tens of years.

Early system-independent computing languages were often sequential in nature, making it difficult to identify and extract individual functions and often requiring a so-called "big bang" migration where all computing functions are transferred to a new implementation at once. Even though they are often rigorously tested, these tests cannot reliably cover all of the complexity of the legacy computer program. For implementations in hospitals, civil infrastructure, and aviation, as just a few examples, the risk of failure may be too great to attempt migration. This then requires aligning the technical benefits of modern hardware systems with computer program code that may pre-date the implementing computer engineers.

Summary

Aspects of the present invention are set out in the appended claims.

Further aspects and embodiments will become apparent from the following description, claims and drawings.

Further features and advantages of the invention will become apparent from the following description of preferred embodiments of the invention, given by way of example only, which is made with reference to the accompanying drawings.

Brief Description of the Drawings

Figure 1 is a schematic diagram showing a first set of system components for controlling a computing device according to an example;

Figure 2 is a schematic diagram showing a second set of system components for controlling a computing device according to an example;

Figure 3 is an illustrative source code program written in a system-independent programming language;

Figure 4 is table illustrating control flow parameters of the source code program of Figure 3, when executed using stack based PERFORM rules;

Figure 5 is table illustrating alternative runtime parameters of the source code program of Figure 3, when executed using OS/VS COBOL PERFORM rules;

Figure 6 is a flow diagram of a high level method according to embodiments;

Figure 7 is a flow diagram of a directed graph generation procedure according to embodiments;

Figure 8 is a diagram of a directed graph including instances of self- and mutual recursion;

Figure 9 is a flow diagram of a reaching end point detection procedure according to embodiments;

Figure 10 is a diagram of a directed graph including showing for each node a range and an end point;

Figure 11 is a diagram of the directed graph of Figure 10 including for each node any reaching end point(s); and

Figure 12 is a schematic diagram showing a set of system components for controlling a computing device according to embodiments.

Detailed Description

Certain examples described herein provide technical improvements when deploying computer programs, which were written in a legacy programming language intended for execution in a legacy computer system, into a modern processing (or execution) environment. For example, computer program code that was originally written in a computer programming language for use with legacy mainframe computer systems, such as those based on the IBM System/360 computing architecture, may be optimised for use with modern computing architectures. In this sense the optimisation is independent of clever programming techniques in the computer program itself; rather it is provided by adapting the control instructions that are used to operate the underlying computer hardware, i.e. a more efficient set of control instructions are provided that are independent of high-level program function. Optimisation may involve improving the speed at which a computer implements a program for a fixed CPU speed and/or making more efficient use of working memory, for example by reducing the amount of memory used and/or optimising memory read/write operations, and/or by not compiling unreachable (i.e. superseded) code. Hence, the examples described herein make a computing system implementing any machine code that results from said examples work better, faster or more reliably in terms of its performance, the computing system is better controlled to provide measurable, deep-level improvements.

It is also possible to make use of technical functionality that was never available on the legacy mainframe systems, for example modern Internet and networking functions that may be employed and linked in a suitable processing environment. These advantages may be realised without need to alter the original legacy computer program code. This is of benefit in mission critical computing systems, for example in health-care or transportation, where it is not possible to modify the computer program code for fear of failure or error. As the original legacy program is not modified, a technical contribution provided by certain described examples is not provided within the computer program as such; indeed in mission critical computing systems functionality of the computer program code must not change. The examples do not relate to a better way to write a program; rather they provide a measurable improvement as compared with fundamental programming conventions.

Certain examples described herein are arranged to be implemented on one or more computing systems, for example a system comprising one or more processors and a memory, such as a random access memory. The examples described herein do not comprise a method that is implemented manually or in the mind, indeed this would be impossible. Examples herein operate in both so-called native and managed processing environments, as will be described.

Figure 1 shows a first set of system components 100 for controlling a computer system 160 according to an example. Source code 110, e.g. data in the form of command statements and/or variable declarations, comprises computer program code for a computer program written in a system-independent legacy programming language. For example, the language may comprise one of FORTRAN, LISP, COBOL, PL/1, PL/M and ALGOL 60. Source code 110 may be retrieved from a computer-readable storage medium 120. Source code 110 is typically human-readable. To control the computer system 160, the source code 110 must be converted to a machine-readable version. This is performed by a compiler unit 130 of a computer device 140. The compiler unit 130 takes source code 110 and converts it into object code 150. The object code comprises a translation of the source code 110 that may be read and executed by the computer system 160. The command statements and/or variable declarations in the source code 110 are converted into control instructions specific to the computer system 160, i.e. instructions that may be interpreted by the specific hardware components of the computer system. The object code 150 is typically not human-readable, e.g. Figure 1 shows a binary version of the control instructions. It is sometimes referred to as a binary, machine or executable code. In this case, the object code 150 is stored on a computer-readable storage medium 170. When the computer program is to be implemented the object code 150 is loaded from the computer-readable storage medium 170 into a memory 180 of the computer system 160. The memory 180 is typically a random access memory. During execution, control instructions as defined in the object code 150 are processed by one or more processors 190 of the computer system 160. Each processor typically has a different set of suitable control instructions. An instruction may control operations such as, among others, load, jump, and arithmetic logic unit commands.

The arrangement of Figure 1 is one example of a native processing environment (or, simply, a 'native environment'), wherein source code is compiled to native code, i.e. object code 150 for the computer system 160. Legacy mainframe computer systems from the 1960s, 70s and 80s were typically designed as native computing environments. In such mainframes, the memory and the one or more processors may be limited when compared to modern computer systems. For example, a state-of-the-art system used by the National Aeronautics and Space Administration (NASA) and installed in 1969 had a central processing unit cycle time of 60 nanoseconds (e.g. a speed of 16MHz) and a memory cycle time of 780 nanoseconds (e.g. around 1.3MHz). The memory size was 2MB. Nowadays, although invariably employing different computer architectures (rather than mainframe computers), operating systems and programming languages, native environments as generally depicted in in Figure 1 are still used. For example, a respective computer system 160 nowadays is more likely to comprise a modern server computing apparatus (i.e. modern relative to a mainframe computer system) operating under a Windows (TM), UNIX (TM) or LINUX (TM) operating system, or even a virtualised operating system, which offers far more programming flexibility and the ability to access modern network services and the Internet.

In contrast to a native environment, Figure 2 shows an arrangement that may be referred to as a 'managed processing environment' (or, simply, a 'managed environment'). In this case, source code 210 is compiled by a first compiler unit 230 of a first computer device 240. The first compiler unit 230 converts the source code

210 to intermediate code 225. Intermediate code is system-independent (e.g. processor and computer architecture independent). It is of a lower level than source code 210, i.e. its instructions are not typically human-readable and have a closer mapping to system-specific instruction sets. Intermediate code 225 may comprise common intermediate language (CIL) code, as defined by the common language infrastructure (CLI) standard and implemented as part of a Microsoft® .NET or Mono framework, or Java® Bytecode. Intermediate code 225 in the form of common intermediate language code may be assembled into a form of bytecode called a CLI assembly. This may be performed by the first compiler unit 230. In Figure 2 the intermediate code 225 is further compiled by a second compiler unit 235 of a second computer device 245 into object code 250. The first and second computer devices and/or the first and second compiler units may, in some cases, form part of the same device or unit. The second compiler unit 235 may comprise a just-in-time and/or ahead-of-time compiler unit: in the former case, the intermediate code 225 is converted into object code 250 at runtime, this may occur in a piece-meal or line-by-line (i.e. interpreted) fashion; in the latter case, the intermediate code 225 is converted into object code 250 before execution. In this latter case, the object code 250 may be stored in a computer-readable medium 270 before execution. In both cases, the object code is similar to object code 150 and at least a portion of object code 284 is loaded into a memory 280 of computer system 260 for execution by at least one processor 290.

In Figure 2, computer system 260 may comprise a modern server computing device (i.e. modern relative to a mainframe computer system). By compiling to intermediate code 225, source code 210 may be optimised to make use of advanced features of the computer system 260. For example, one or more of security, multi-threading and memory management may be improved by making use of run-time management services 282. Run-time management services 282 are implemented by system libraries that comprise executable code – they may form part of an execution or runtime environment. This executable code is loaded into memory 280 along with at least a portion of the object code 284 during execution. In one case, run-time management services 282 may comprise the second compiler unit 235 in the form of a just-in-time compiler, wherein the second computer device 245 comprises the computer system 260 (as indicated by the dashed line). In this case, the second compiler unit 235 may comprise a virtual machine in the form of a common language runtime in the .NET framework or a Java Virtual Machine (JVM). The virtual machine may then handle exceptions, garbage collection (i.e. efficient use of space in memory 280), variable type safety (i.e. a discrepancy between data types) and thread management (i.e. control of machine code being processed by the at least one processor 290). These functions typically will not have been available on a legacy mainframe system, or at least may have been implemented in an inefficient manner due to the age of the hardware of the mainframe system. Even in a case where intermediate code 225 is compiled ahead-of-time to object code 250, the same functions of the run-time management services 282 may be implemented by suitable code conversion from intermediate code to object code (e.g. a given intermediate code statement may be compiled in a particular manner for a particular server computing device to make use of hardware functionality of said device using appropriate machine code functions).

Certain embodiments described herein may be implemented in either a native environment as described in relation to Figure 1 or a managed environment as described in relation to Figure 2. In the former case, source code is compiled to object code for execution within a runtime environment; in the latter case, source code may be compiled to intermediate code for execution by a just-in-time compiler (e.g. a virtual machine) within a runtime environment. In both cases, the output may be referred to as executable code, as the code may be executed directly or by way of the just-in-time compiler. Compilation may be performed on a first computing device for execution within a runtime environment installed on a second computing device, wherein compilation prepares control instructions based on the source code that may make use of functions provided by the runtime environment. As such compilation is described as a preparation step for subsequent deployment into a runtime environment that generates a concrete output – executable code to control a computing device.

Embodiments herein aim to address certain issues that arise when porting legacy applications, specifically source code, for example OS/VS COBOL source code intended for execution on an IBM mainframe, to a modern native or managed processing (e.g. a.NET or Java) environment. While the description herein focuses on OS/VS COBOL, this is for convenience only and it will be appreciated that the principles broadly apply to other legacy COBOL variants, such as (but not limited to) RM COBOL, COBOL/370, Enterprise COBOL, OS/390 COBOL and VS COBOL II.

In particular, embodiments relate to COBOL functions and, more particularly, to COBOL PERFORM statements, which are key program constructs within the COBOL programming language. A PERFORM statement is an instruction or a command that transfers control (e.g., explicitly) to one or more instructions defined to be a PERFORM 'range' and returns control (e.g., implicitly), when an 'end point' of the PERFORM range is reached, to the next executable instruction after execution of the instructions called by the PERFORM statement is finished. As such, a PERFORM statement transfers the control flow to the next instruction in sequence, immediately following the termination of the PERFORM statement. PERFORM statements are used to develop COBOL code modules, each of which can be called using a PERFORM statement. According to embodiments herein, a COBOL source code program (or part thereof) has an entry point and is made up of a plurality of PERFORM ranges (determined by respective PERFORM statements). In a typical COBOL program one PERFORM range may be configured to call one or more other PERFORM ranges to build a program structure.

It will be appreciated that the following examples *mutatis mutandis* can apply equally to any similar constructs in other high level languages.

Like some other mainstream programming languages, COBOL has an associated ANSI standard (74, 85), which among other things specifies the syntax and semantics of COBOL. Its purpose is to promote a high degree of machine independence to permit the use of COBOL on a variety of data processing systems. However, while the standard defined the behaviour of legal PERFORM cases, it did not specify the behaviour of illegal cases, and different vendors chose different implementations, which, in particular, differed in behaviour for the illegal cases. This in effect means that although COBOL is intended to be a high-level, system independent programming language, it is not safe, for example, to take code written for a legacy IBM environment and expect it to behave in exactly the same way, for example, in a legacy RM environment. Moreover, with COBOL and similar languages, there is typically a relatively high degree of inherent programming flexibility, and developers have an increased opportunity to create applications that execute properly while not being

ANSI-compliant. This can lead to code that may be optimised for one processing environment and care must be taken when porting it to another processing environment.

Under legacy OS/VS COBOL, for example, during compilation, for every paragraph or section that ends a PERFORM range a so-called ‘exit bucket’ is allocated and created, where an exit bucket comprises a small block of code. At the end of the respective paragraph or section an indirect jump via the exit bucket is compiled. The exit buckets are all initialised to point to the next statement, so that before any PERFORM statements are executed control flow will be sequential. The points to jump to are loaded at runtime into a number of registers. For each PERFORM statement a so-called ‘save bucket’ is allocated and created, which also comprises a small block of code. The code for the PERFORM statement is then to save the exit bucket (corresponding to the end of the PERFORM range) and then set it to point to the return point, again by setting various registers, and jump to the start of the performed code. The return point follows immediately, where code is compiled to restore the exit bucket from the save bucket.

In the following table, the left hand column illustrates the structure of an exemplary PERFORM statement and the right hand column illustrates how the PERFORM statement may be implemented using save and exit buckets (where & = address of, * = indirect):

	MOVE EXITB TO SAVE1
	MOVE &RETURN1 TO EXITB
PERFORM A THRU B	GO TO A
	RETURN1.
	MOVE SAVE1 TO EXITB
A.	A.
*> Statements in paragraph A	*> Statements in paragraph A
B.	B.
*> Statements in paragraph B	*> Statements in paragraph B
	GO TO *EXITB
C.	C.

As can be seen in the foregoing table, the PERFORM statement is converted into two MOVE statements, a GO TO statement, a label, and a final MOVE statement (that being the restore of EXITB from SAVE1). The PERFORM statement exit at the end of B becomes a GO TO via the exit bucket (EXITB).

It can be seen that a form of partial recursion is permitted, in that more than one PERFORM statement referring to the same exit point may be being executed at once. However full recursion is not possible – i.e. the same PERFORM statement may not be executed more than once simultaneously - since there is only one (level of) save bucket and so the original value of the exit bucket would be lost and could never be restored. Consequently, the program control flow would be permanently changed, and a loop would likely result. The use of save and exit buckets in this way means that with OS/VS PERFORM statements all the exit points of the PERFORM statements currently being executed are 'active' simultaneously, and if control reaches any of them a respective return jump will occur. In other words, there can be no strict nesting of PERFORM statements, and they can be returned from in arbitrary order. This leads to a compiled code that is a single monolithic (or ‘flat’) program, in which PERFORM statements are, in effect, converted to a sequential code including a series of stateless redirection commands employing the save and exit bucket constructs.

While the OS/VS form of compiled code may be suited to the processing architecture of legacy IBM mainframe computers, it is typically not suitable for modern native and managed processing environments for a number of reasons. For instance, modern processing environments often employ stack based processing, for example, in which functions are nested and only the end point of a most-recently called function (as opposed to all end points) is active at any one time, and managed environments are typically object orientated, and optimally employ small program fragments, or methods, which typically have a relatively small maximum size, for example a maximum of 64K bytes for JVM Bytecode. Such environments are consequently optimised for stack based processing of object orientated source code programs. The code fragment 300 in Figure 3 will now be used to illustrate the difference between the behaviour of a PERFORM statement using stack based semantics (i.e. which are ideal for a managed environment) and the behaviour of an OS/VS PERFORM statement using OS/VS semantics.

Stack Based PERFORM

The control flow of the code shown in Figure 3, stack control, stack contents, the active end points and output are illustrated in the table in Figure 4. As can be seen, the control flow progresses from line 9 with a PERFORM statement in line 10 specifying a range ‘a through b’, which causes redirection to ‘a’ and an associated push of data (b, r1) onto the stack 305, such that there is an active end point at the end of ‘b’. The first statement in ‘a’ is another PERFORM statement specifying a range ‘b through c’, which causes redirection to ‘b’ and an associated push of data (c, r2) onto the top of the stack 305, such that there is a new active end point at the end of ‘c’ (effectively temporarily de-activating the previous end point). The first statement in ‘b’ causes an output “b” and after line 17 (i.e. the now deactivated initial end point) executes the program simply continues by executing ‘c’, with the output of “c”. That is the end of ‘c’ and, since the active endpoint has been reached, the respective data is popped from the top of the stack 305 (the active endpoint reverting to the end of ‘b’) and the program redirects back to line 14, causing an output “a”, followed by another output “b”, after which the active endpoint has been reached and the respective data is popped from the stack, followed by redirection to line 11 and the end of the program. The result is that the program executes, with the output “bcab”.

As has been demonstrated, there is only ever one active endpoint in this stack based program control flow.

Flat (OS/VS) Based PERFORM

The control flow of the code in Figure 3, exit bucket control, exit bucket contents, the active end points and output are illustrated in the table in Figure 5.

Although the control flow starts the same as for the stack based PERFORM, significantly, when the second PERFORM is called (line 13), because of the way exit buckets are constructed (as explained above), both end points (i.e. end of ‘b’ and end of ‘c’) remain active. Consequently, and in contrast with the foregoing stack based example, when execution reaches line 17, the end point of b is recognised, the program jumps back to line 11 and the program ends. The output is only “b”. Paragraphs c, a and b (repeated) are completely omitted from the control flow.

The foregoing simple examples illustrate clearly why it is not a trivial task to cross-compile a legacy source code program, for example written in OS/VS COBOL, into object code, or intermediate code, optimised for execution in a modern processing environment, without following exactly the respective legacy semantics.

As used herein ‘semantics’ means the processes a specific processing environment follows (or is expected to follow) when executing a program in a specific language intended for that environment. By ‘legacy semantics’ we mean a compilation configured to maintain compatibility with the expected control flow that was defined (explicitly and/or implicitly) by the structure and arrangement of the associated source code, which was written in a legacy programming language, e.g. OS/VS COBOL, RM COBOL, etc. and intended for execution by a legacy computer system. For example, in the case of OS/VS, the legacy semantics require a respective compiler to construct for PERFORM statements exit buckets and save buckets (as described above), thereby rendering all end points active all the time and resulting in a flat program.

In practice, typically, known mainstream compilers are adapted to compile legacy source code into object code or intermediate code by following legacy semantics, since, to do otherwise (as has been illustrated) risks occurrence of missed code, erroneous control flow and/or infinite loops, to name a few potential issues. Unfortunately, such compilers tend to create large, unwieldy and slow-running intermediate code, which is not optimised for a respective modern processing environment. In particular, and by way of example, following legacy semantics, an OS/VS COBOL program may result in object code or intermediate code having one function per entry point, and a single monolithic function (encompassing potentially many PERFORM ranges) for the body of the program. Moreover, steps therefore have to be taken, when compiling intermediate code into object code, to break the single monolithic function up artificially into small portions, for example, to comply with maximum byte code size restrictions, for example a maximum of 64K bytes of JVM Bytecode.

In contrast, source code that is suitable for optimisation for a modern processing environment can be compiled into object code or intermediate code using stack based semantics. By 'stack based semantics' we mean the compilation approach and expected control flow that is required (explicitly and/or implicitly) for compatibility with environments that employ stack based processing. In such cases, the code is compiled so that only one end point is ever active (it being determined by the data that is on the top of the stack at any particular time) at any one time. Moreover, source code that is compiled using stack based semantics may naturally, beneficially achieve one function (or method) per entry point and one function (or method) for each PERFORM range, thereby rendering the resulting intermediate code ideally suited for efficient stack based processing by a respective object orientated runtime environment.

It will be apparent from the foregoing that compilation of legacy source code using stack based semantics may result in object code or intermediate code instructions that have a different control flow to the same source code when compiled using legacy semantics. Not least, legacy source code that is compiled using legacy semantics tends to be far longer (in terms of the number of instructions) and less efficient during execution.

In arriving at embodiments herein, it has been appreciated that, despite the technical control flow problems that can arise if legacy source code is compiled using stack based semantics, in particular instances, it is feasible to compile using stack based semantics a legacy source code program, for example written in OS/VS COBOL, into an object code or intermediate code that is optimised for a modern processing environment, with no risk of damaging the control flow of the compiled program. It is emphasised that the control flow of the original source code application program is not changed – indeed it is imperative that the control flow is not changed (at least at the functional level); and it is purely the manner of compilation which is changed in order to optimise the resulting code for execution within a modern processing environment. Legacy application program source code instructions, which can be optimised in this way, have been found to execute anywhere from five times to more than ten times faster than the same legacy source code when compiled using legacy semantics. Such efficiency gains, which result from having shorter, more efficient code, which takes advantage of stack based processing control flows, can be significant even if only a small portion of legacy code can be optimised in this way; bearing in mind legacy applications can comprise many programs, which individually or collectively can run into millions of lines of executable code.

The approach adopted according to embodiments is to characterise automatically the expected control flow of a legacy (e.g. OS/VS) COBOL source code program as part of the code deployment process, using selected control flow analyses, to determine whether the control flow of the legacy source code will be the same irrespective of whether legacy semantics or stack based semantics are applied. Use of such analyses has been found to be a practical and efficient alternative, for example, to running highly time-consuming, brute-force simulations or emulations to determine the complete runtime control flow behaviour of legacy code that is compiled using stack based semantics. In effect, it has been appreciated that the control flow analyses can be used to determine equivalence in control flow between legacy, flat PERFORM statements and stack based PERFORM statements. In general, such equivalence means that an application, whether written with flat PERFORM statements or stack based

PERFORM statements, will behave in the same way (i.e. will exhibit the same control flow), irrespective of whether the application is compiled using legacy semantics or stack based semantics.

Moreover, it has been appreciated that the particular analyses employed in the following examples are broadly indicative of whether the PERFORM statements in the OS/VS source code are ANSI compliant. If the code is determined to be ANSI compliant, flat PERFORM statements and stack-based PERFORM statements will behave in an equivalent fashion if complied using stack based rules; where use of stack based semantics desirably leads to smaller, faster and generally more efficient intermediate code, that is optimised for a modern processing environment, as has been described.

Two analyses have been selected: (1) a determination whether a PERFORM call graph includes recursion; and (2) a determination whether a called PERFORM range includes an end point of a calling PERFORM range. Given the teaching herein, other analyses may be devised and/or performed.

A high level process for analysing control flow in legacy COBOL source code will now be described with reference to the flow diagram in Figure 6. In a first step, 600, the legacy source code program (which may be a standalone program of part of a larger program) is received or loaded from storage. In step 610, the legacy source code is processed using the analyses to establish if flat PERFORM statements and stack-based PERFORM statements will behave in an equivalent fashion if complied using stack based semantics (i.e. indicating ANSI compliance), in a manner to be described. If, in step 620, the code is determined not to be ANSI compliant, in step 630, it is compiled into object code or intermediate code that is compatible with the respective legacy native system, using legacy semantics, and not optimised for execution in a modern processing environment. If, in step 620, the code is determined to be ANSI compliant, in step 640, it is compiled into object code or intermediate code that is optimised for execution in a modern processing environment, using stack based semantics. In either event, in step 650, the code that is produced is added to a final code (if the code was part of a larger program). Finally, the code is finalised for deployment into a respective processing environment, e.g. output in an executable object or intermediate form. The finalised code may then be executed by the computer device performing the analysis, for example in a test runtime environment, or otherwise communicated to a further computer device for execution. In the latter case, the further computer device has a runtime environment that is configured to execute the compiled code. In one case, the entity performing the analysis may be different from the entity executing the finalised code, e.g. the former may comprise a system developer and the latter an end-user. As such the analysis may be performed ahead of, and independently from, subsequent execution. For example, object or intermediate code may be packaged upon and/or copied to a computer-readable storage medium that is then accessed by a further computer device. In one case, object or intermediate code may be communicated over a network and stored within a storage device of the further computer device (e.g. stored on a hard disk or flash-memory drive). In use, the object or intermediate code may then be retrieved from storage and loaded into memory ready for execution by one or more processors.

The legacy source code is analysed for ANSI compliance in a manner that will now be described.

PERFORM recursion

According to embodiments herein, a first kind of control flow analysis that is performed determines whether the PERFORM structure of a program exhibits any recursion. Recursion herein can mean either self-recursion (i.e. a PERFORM range calls itself) or mutual recursion (i.e. more than one PERFORM range forms a recursive loop). The presence of either kind of recursion indicates that the associated source code is not ANSI compliant and that stack PERFORM statements cannot be guaranteed to behave in the same way as flat PERFORM statements when compiled using stack based semantics.

Embodiments herein conveniently adapt Tarjan's Algorithm to determine the presence of recursion in the PERORM structure. Tarjan's Algorithm is known and identifies 'Strongly Connected Components' in a directed graph. Strongly Connected Component (SCC) is a standard Computer Science term. For example, it is defined in compiler textbooks such as "Advanced compiler design and implementation", Steven S. Muchnick, ISBN 1-55860-320-4. A SCC exists in a directed call graph comprising a set of nodes if it is possible to start from and return to a node in the graph, directly or indirectly, by following only the directed arcs between nodes of the graph in the directions indicated. Nodes within a common SCC can be referred to as being strongly connected. A SCC in a 'PERFORM graph' (which will be referred to herein as a 'directed call graph' or simply a 'call graph') indicates recursion, either self or mutual. In order for a legacy program to be ANSI compliant, there must be no recursion, and therefore no SCC.

The process will now be described with reference to the flow diagram in Figure 7 and an exemplary directed call graph in Figure 8. The process may be performed by a computer program, for example written in C++ or C# and deployed on the same system 240 as the first compiler 230 of Figure 2.

In a first step 700, the process parses the source code to identify PERFORM ranges, determine for each PERFORM range which other PERFORM ranges can be reached (i.e. whether PERFORM ranges are 'called') and generate a directed call graph dataset comprising data elements representing a directed call graph 800 of the source code program. In Figure 8, for example, each node of the call graph (a-f) represents a PERFORM range; a PERFORM range at the end of an arc (i.e. at the end of an arrow) is reached from (or called by) the PERFORM range at the beginning of the arc. For example, in Figure 8, PERFORM range 'a', by virtue of arc 805, calls PERFORM range 'b' and, by virtue of arc 806, calls PERFORM range 'e'; PERFORM range 'b' calls PERFORM ranges 'c' and 'd'; PERFORM range 'd' calls itself; PERFORM range 'e' calls PERFORM range 'f'; and PERFORM range 'f' calls PERFORM range 'e'. The terms 'PERFORM range' and 'node' may be used interchangeably herein.

In a next step, 710, the directed call graph dataset acting as input is inspected, a first node is selected (assume node 'a') and, in step 720, the call graph is traversed (i.e. the arcs are followed) to determine whether it is possible to traverse the graph and find a way back to the selected node 'a'. In a next step, 730, if any inspected nodes are determined to be strongly connected (i.e. there is recursion or mutual recursion) the process ends in step 740 "End (1)". If the inspected node(s) is (or are) not determined to be strongly connected, in step 750, a determination is made of whether there are any remaining nodes, which have not yet been inspected. If there are no more nodes then the process ends in step 760 "End (2)". Otherwise, an uninspected node is selected in step 770 and the process jumps to step 720 until all nodes have been inspected. More generally, the traversal of the graph continues until any strongly connected nodes are discovered or until there are no further nodes to inspect.

In this example it can be seen that there is no path back to ‘a’ and so ‘a’ is not strongly connected. Likewise, ‘b’ and ‘c’ are not strongly connected. However, it can also be seen that ‘d’ can be reached from itself, so is strongly connected (i.e. ‘c’ exhibits self-recursion and so is a SCC 811). Likewise, it is possible to get back to ‘e’ via ‘f’, so ‘e’ and ‘f’ are considered to be strongly connected (i.e. ‘e’ and ‘f’ exhibit mutual recursion and form a SCC 821).

With reference to Figure 8, it is apparent that had there been no strongly connected nodes (i.e. had directed arcs 810 and 820 not been present), the process would not have found any strongly connected nodes and the process would have ended at End (2).

Embodiments herein determine that the presence of any self- or mutual recursion implies that the code is not ANSI compliant and that stack based PERFORM statements in the legacy source code are not guaranteed to behave in the same way as flat PERFORM statements if compiled using stack based semantics. If this is the case, the code has to be compiled into object code or intermediate code that includes flat PERFORM statements using legacy semantics (as exemplified above with regard to Figure 5). In contrast, if no recursion is found (i.e. End (2) prevails), then the process undertakes the next control flow assessment.

PERFORM Reaching End Points

According to embodiments herein, a second kind of control flow analysis that is performed determines whether the program has any PERFORM ranges that include a ‘reaching end point’. As used herein, a ‘reaching end point’ for a called PERFORM range is an endpoint of a calling PERFORM range; as exemplified by line 17 in the exemplary code in Figure 3. The calling PERFORM range can be the immediate (i.e. direct) predecessor or any direct or indirect predecessor of an immediate predecessor. If a called PERFORM range includes an endpoint of a calling PERFORM range, whether that end point causes redirection or not differs depending on whether the code is flat or stack-based. The second kind of flow control analysis may be performed by a computer program, for example written in C++ or C# and deployed on the same system 240 as the first compiler 230 of Figure 2.

The process will now be described with reference to the flow diagram in Figure 9 and the exemplary directed call graphs in Figure 10 and Figure 11.

In a first step, 900, a directed call graph of the source code is generated as before. Indeed, the same directed call graph dataset that was previously generated may be used without the need to re-generate it. In any event, Figure 10 illustrates another directed call graph 1000 representative of the source code program. In Figure 10, each node of the graph (1-7) as before represents a PERFORM range and a PERFORM range at the end of an arc is reached from (or called by) the PERFORM range at the beginning of the arc. The graph in Figure 10 contains no recursion (i.e. no loops, or the process would not have reached this stage). More specifically, in Figure 10, PERFORM range '1' calls PERFORM ranges '2', '4' and '5'; PERFORM range '2' calls PERFORM range '3'; PERFORM range '4' calls PERFORM range '6'; and PERFORM range '5' calls PERFORM range '7'. In this example, it can be seen that PERFORM range '6' is called by each of PERFORM ranges '2', '4' and '5'. As shown, each node in the call graph is accompanied by a tuple including a first field indicating the PERFORM range of the respective node (for example identifying the paragraphs that reside within the range) and a second field indicating the end point of the PERFORM range. For example: PERFORM range '1' is accompanied by a tuple including a range {a,b,c} and an endpoint 'c'; PERFORM range '2' has a range {e} and an endpoint 'e'; PERFORM range '3' has a range {f} and an endpoint 'f'; PERFORM range '4' has a range {g} and an endpoint 'g'; PERFORM range '5' has a range {h} and an endpoint 'h'; PERFORM range '6' has a range {c,d} and an endpoint 'd' and PERFORM range '7' has a range {i} and an endpoint 'i'.

Next, in step 910, for each node, the process inspects the directed call graph dataset and generates a third field for each tuple, in step 920, indicating the relevant reaching end point(s) for the respective node. For example, for node '1', the third field is {}, since there are no calling (or predecessor) PERFORM ranges. For node '2', however, the third field is {c}, representing the end point of the calling node '1'. The reaching end points for each node are illustrated as an additional leading field in the modified graph in Figure 11 (though, the order of the fields in the tuple is not important). In Figure 11, the reaching end points for node ‘3’ are {c,e} (i.e. the end points of the two preceding nodes); the reaching end point for node ‘4’ is {c}, the reaching end point for node ‘5’ is {c}, the reaching end points for node ‘6’ are {c,e,g,h} (i.e. the end points of nodes ‘1’, ‘2’, ‘4’ and ‘5’) and the reaching end points for node ‘7’ are {c,h}. The process is repeated, in step 930, until there are no more nodes to inspect.

In a next step, 940, a determination is made as to whether within each tuple the union of the range and the reaching end point(s) of any node is not NULL. If the union is not NULL (i.e. there is an overlap in reaching end point(s) and range for at least one node), then the process ends at End (3) in step 950, since the respective PERFORM range is found to include an end point of a predecessor PERFORM range, which means flat and stack based PERFORM statements would not behave equivalently when compiled using stack based semantics. Alternatively, if the unions are all NULL, the process ends at End (4) in step 960.

Embodiments herein determine that if all unions are NULL (i.e. End (4) is reached), then the program is ANSI compliant and stack based PERFORM statements in the associated legacy source code are expected to behave in the same way as flat PERFORM statements when complied using stack based semantics. If this is the case, the code can be compiled into object code or intermediate code that includes using stack based semantics. In contrast, if at least one union is not NULL, (i.e. End (3) prevails), then the code (which is thereby deemed not to be ANSI compliant) has to be compiled into object code or intermediate code that includes flat PERFORM statements, using respective legacy semantics, as explained above.

It has been appreciated that the PERFORM ‘reaching end points’ process can be completed very efficiently, since, at the point of running the process it has already been established that there are no recursive loops. This means that the process can be treated as a standard ‘forward flow’ problem and the nodes represented in the directed call graph dataset can be processed in Reverse Post Order, such that the respective analyses can be conducted in a single pass of the dataset. In other words, additional complexity in the form of additional nodes causes only a linear increase in processing time.

With reference again to the flow diagram in Figure 6, it is apparent that two different compilation processes are available for use in dependence upon whether the source code has been deemed to be ANSI compliant or not. The resulting object code or intermediate code will comprise different instructions and have a different control flow depending upon which compilation process is selected. In either case, the output of the respective compilation process is object code or intermediate code suitable for deployment onto and execution by a suitable modern, stack based processing environment.

A system 1200 for performing embodiments herein will now be described with reference to the diagram in Figure 12. While the system 1200 in this example relates to the preparation of executable code and subsequent deployment of an application program into an exemplary managed processing environment, it will be appreciated that the same principles apply to the preparation and deployment of an application program into a native processing environment, whereby object code rather than intermediate code is generated at a first compilation stage. Although the example of Figure 12 shows both preparation and deployment processes, these may be performed separately as described above.

In Figure 12 system components according to embodiments herein is illustrated. Source code 1210 is loaded from computer readable storage 1220 for compilation. In the example of Figure 12, compilation is first into intermediate code; however similar components may also be used for compilation directly into object code. Before compilation, the source code 1210 is analysed by a control flow analyser unit 1242, of computer system 1200, comprising a directed call graph generator unit 1230, configured to parse the source code 1210 and generate a directed call graph dataset representing the control flow of said code. The directed call graph generator unit 1230 is configured to feed the directed call graph dataset to a recursion detector unit 1231 and to a reaching end point detector unit 1232. Alternatively, the directed call graph generator may store a dataset representing the directed graph to a computer readable storage medium (not shown) from which the recursion detector unit 1231 and the reaching end point detector unit 1232 can read the data as necessary. In any event, according to the present embodiments, the directed call graph generator unit 1230 triggers the recursion detection unit 1231 once the directed call graph has been generated.

The recursion detector unit 1231 evaluates the directed call graph (as described above) to determine whether the source code 1210 contains any loops indicative of recursion. In this example, if any loops are detected the recursion detector unit 1231 triggers a legacy compiler unit 1233 of an intermediate code compiler unit 1243, which is adapted to compile the source code 1210 into intermediate code 1226 using legacy semantics. In a native compilation case (not shown), a legacy compiler unit may form part of an adapted compiler unit such as compiler unit 130 in Figure 1 for compilation directly to object code. If the recursion detector unit 1231 determines that the code contains no loops indicative of recursion, it instead triggers the reaching end point detector unit 1232 to determine whether the code contains any reaching end points. If the code is determined to contain any reaching end points, the reaching end point detector unit 1232 triggers the legacy compiler unit 1233, which is adapted to compile the code 1210 into intermediate code 1226 using legacy semantics. If the code is determined not to contain any reaching end points, it is deemed to be ANSI compliant and the reaching end point detector unit 1232 triggers a stack based compiler unit 1234, of the intermediate code compiler unit 1243, which is adapted to compile the code 1210 into intermediate code 1227 using stack based semantics. As before, in a native compilation case (not shown), legacy compiler unit 1233 and stack based compiler unit 1234 are arranged to perform the same function but to output object code rather than intermediate code.

It will be appreciated that there are many different ways in which the foregoing analyses and compilation can be implemented and that the particular approach described is merely one exemplary approach.

For example, in practice, legacy code compiler unit 1233 and stack based compiler unit 1234 may be separate units within the system 1240 or may be a single unit which is configured to apply different compilation semantics depending upon whether the code is determined to be ANSI compliant or not.

In any event, in practice, although both are shown, only one of the intermediate codes, 1226 or 1227, is generated and stored in computer readable storage (not shown), depending on which compiler produces the intermediate code.

The control flow analyser unit 1242 and the intermediate code compiler unit 1243 may form different units (as shown) or may form separate functional elements within a larger unit, apparatus or system.

In a managed environment, the intermediate code is system-independent (e.g. processor and computer architecture independent), but is generated for a specific runtime environment. That is, the code, although system-independent, would not be expected to run in a different runtime environment. For example, CIL could not run on the JVM, and, similarly, JVM Bytecode could not run on the .NET CLI. The intermediate code 1226/1227 comprises CIL code, as defined by the CLI standard and implemented as part of a Microsoft® .NET or Mono framework, or Java® bytecode (depending on the target environment). Additionally, or separately, compilation may insert function calls to libraries that form part of the runtime environment. As such, a specific runtime environment is required to execute object or intermediate code generated by the described examples; the object or intermediate code would not run on a different runtime environment.

As described above, if, instead, the system 1200 had been configured to deploy an application program into a modern native processing environment, native code equivalents of the legacy compiler unit 1233 and stack based compiler unit 1234 would have been configured to generate object code, rather than intermediate code, for a respective native processing environment. Nevertheless, as with the equivalent intermediate code, the resulting object code would comprise different instructions and have a different control flow depending upon which compilation process is selected.

In Figure 12, since this is a managed environment, the intermediate code 1226/1227 is further compiled by a second compiler unit 1235 of a second computer device 1245 into object code 1250. The first and second computer devices and/or the first and second compiler units may, in some cases, form part of the same device or unit. The second compiler unit 1235 may comprise a just-in-time and/or ahead-of-time compiler unit: in the former case, the intermediate code 1226/1227 is converted into object code 1250 at runtime, this may occur in a piece-meal or line-by-line (i.e. interpreted) fashion; in the latter case, the intermediate code 1226/1227 is converted into object code 1250 before execution. In this latter case, the object code 250 may be stored in a computer-readable medium 1270 before execution. At least a portion of object code 1284 is loaded into a memory 1280 of computer system 1260 for execution by at least one processor 290. In a native environment, there is no second compilation; object code prepared by native code equivalents of the legacy compiler unit 1233 and stack based compiler unit 1234 is executed in a similar manner to the object code generated by the second compiler unit as herein described.

Returning to Figure 12, the second compiler unit 1235 is capable of compiling both intermediate code generated using legacy semantics and intermediate code generated using stack based semantics. As has been described, the two kinds of intermediate code will typically have a different control flow and comprise different intermediate code structures (e.g. intermediate code generated using legacy semantics will have save and exit bucket code fragments whereas intermediate code generated using stack based semantics will not). The second compiler unit 1235 may comprise independent compiler units for compiling each kind of intermediate code. In addition, or alternatively, the intermediate code compiler unit 1243 may introduce into the intermediate code a flag or other indication of which form of code has been generated, which may be used by the second compiler unit 1235 to determine how (e.g. which compiler unit to use) to produce the object code. In both native and managed environments, different sets of object code are produced dependent on the analysis performed by a control flow analyser unit and/or the method of Figure 6. Whether or not there is an intermediate compilation stage, object code derived from a compilation unit using legacy semantics is different from object code derived from a compilation unit using stack based semantics, e.g. has different sets of control instructions. As such, object code derived from a compilation unit using legacy semantics makes use a first set of functions in a runtime environment and object code derived from a compilation unit using stack based semantics makes use a second set of functions in a runtime environment, wherein the first and second sets are different. The second set of functions may comprise an additional one or more functions for stack based execution, e.g. using dedicated stack registers in a particular computing architecture, the compilation being configured for said particular computing architecture. Comparatively, the first set of functions may exclude one or more functions for stack based execution and/or include one or more functions for flat processing. For example, in the latter case one or more functions for flat processing may comprise one or more functions for handling large portions of executable code, e.g. large sections of consecutive instructions may require a particular control procedure for memory management. In any case, both the first and second set of functions may be implemented by libraries forming part of a runtime environment. They may relate to differentiated memory management functions. These libraries, when installed and/or registered, may form part of the operating system environment, i.e. be required to execute application programs. In a Windows(TM) system these libraries may comprise dynamic link libraries.

In Figure 12, computer system 1260 comprises a server computing device. In this specific managed environment example, by compiling to intermediate code 1226/1227, source code 1210 may be optimised to make use of run-time management services 1282 associated with the managed environment. Run-time management services 1282 are implemented by system libraries that comprise executable code. This executable code is loaded into memory 1280 along with at least a portion of the object code 1284 during execution. In certain cases, run-time management services 1282 may comprise the second compiler unit 1235 in the form of a just-in-time compiler, wherein the second computer device 1245 comprises the computer system 1260 (as indicated by the dashed line). In this case, the second compiler unit 1235 may comprise a virtual machine in the form of a common language runtime in the .NET framework or a Java virtual machine. The virtual machine may then handle exceptions, garbage collection (i.e. efficient use of space in memory 1280), variable type safety (i.e. a discrepancy between data types) and thread management (i.e. control of machine code being processed by the at least one processor 1290). In other cases, the second compiler unit 1235 may be provided separately, e.g. may form part of an operating system and/or be installed separately. For example, a common language runtime or a Java virtual machine may already be present on a computer system or may be provided by way of independent installation processes. In these other cases, run-time management services may have a similar implementation in a managed environment to those provided for a native environment, e.g. in both cases they may form part of a runtime environment for the successful execution of object code derived from the preparation methods described herein.

In embodiments, the computer systems 1240, 1245 and 1260 may comprise a single system providing each of the aforementioned functions and processes or one or more different computer systems, and it will be appreciated that the arrangement illustrated in Figure 12 is but one possible arrangement. The or each computer system may comprise standard computer hardware, for example, comprising one or more programmable processors, at least one operating system and application software to provide the required functions and processes. The term "deployment" is used herein to describe the process of placing an application program in a suitable form for execution by one or more processors. Deployment may comprise an installation process that copies object or intermediate code to a storage device of a computer system and that configures said code for execution. This configuration may comprise registering the object or intermediate code with an operating system and/or a runtime environment. The phrase "preparing for deployment" is used to describe a compilation process that operates upon source code to produce object or intermediate code suitable for execution. In one case, both preparing for deployment and deployment may form part of a common process. The term "legacy" is used herein to describe features that relate to a computing architecture that is no longer in use. For example, a legacy programming language is a programming language that was configured to operate on a superseded (i.e. legacy) computing architecture. In certain case, the legacy programming language may relate to a specification that has itself been superseded. Even though computer systems have been described as including one or more processors and a memory, these components may be excluded in particular cases.

In certain examples, a method of preparing an application program for deployment into a processing environment is provided. The application program comprises legacy source code instructions representing the application program, written in a legacy system-independent programming language and being for compilation using legacy semantics for a respective legacy native computing environment. In this case the method comprises determining an equivalence in control flow between the source code if compiled using stack based semantics and the source code if compiled legacy semantics and if equivalence is determined, compiling the legacy source code using stack based semantics, for deployment to the processing environment or if non-equivalence is determined, compiling the legacy source code using legacy semantics, for deployment to the processing environment.

Likewise in certain examples, there is provided a computer system to prepare an application program for deployment into a processing environment, the application program comprising legacy source code instructions representing the application program, written in a system-independent programming language and being for compilation using legacy semantics for a respective native computing environment, the system comprising a control flow analyser unit operable to determine an equivalence in control flow between the source code if compiled using stack based semantics and the source code if compiled legacy semantics and a compiler unit communicatively coupled to the control flow analyser unit, the compiler unit comprising a stack based compiler unit, operable if equivalence is determined, to compile the legacy source code using stack based semantics, for deployment to the processing environment and a legacy compiler unit, operable if non-equivalence is determined, to compile the legacy source code using legacy semantics, for deployment to the processing environment.

In certain variations, the legacy source code instructions are written in one of: OS/VS COBOL, RM COBOL, COBOL/370, Enterprise COBOL, OS/390 COBOL and VS COBOL II. A range may comprise one or more instructions that are executable in respect of a COBOL PERFORM statement.

In certain variations, an application program is written in legacy COBOL and equivalence is determined if the legacy COBOL is ANSI compliant. For example, the control flow analyser unit may be operable to determine an equivalence in control flow between the source code if compiled using stack based semantics and the source code if compiled using legacy semantics by determining if the source code instructions are ANSI compliant.

In certain variations, a complier unit comprises an intermediate code compiler unit for generating intermediate code and the system comprises a second compiler unit operable to compile the intermediate code into object code for execution in a runtime environment of a selected managed environment, the object code having a control flow that is determined by the respective control flow of the intermediate code as compiled by the intermediate code compiler unit. In this case there may further be a runtime environment of the managed environment operable to execute the object code according to a control flow that is determined by the respective control flow of the intermediate code as compiled by the intermediate code compiler unit, the runtime environment comprising one or more system libraries comprising code to be called on execution of the object code. The runtime environment may or may not comprise the second compiler unit. A second compiler unit is operable to compile the intermediate code just-in-time for execution thereof by the runtime environment. Similar variations may be applied to the method. These examples and variations are a selection of those described and are not exhaustive.

The above embodiments are to be understood as illustrative examples of the invention. Further embodiments of the invention are envisaged. It is to be understood that any feature described in relation to any one embodiment may be used alone, or in combination with other features described, and may also be used in combination with one or more features of any other of the embodiments, or any combination of any other of the embodiments. Furthermore, equivalents and modifications not described above may also be employed without departing from the scope of the invention, which is defined in the accompanying claims.

Claims

1. A computer system to prepare an application program for deployment into a processing environment, the application program comprising legacy source code instructions representing the application program and having an expected control flow, written in a system-independent programming language and being for compilation using legacy semantics for a respective native computing environment, the system comprising:

a first memory unit to store the legacy source code instructions, the legacy source code instructions comprising a plurality of functions each having a range and at least one function call;

a control flow analyser unit communicatively coupled to the memory unit and being operable to determine an equivalence in control flow of respective compiled code resulting from the source code if compiled using stack based semantics and the source code if compiled using legacy semantics, the control flow analyser unit comprising:

a directed call graph generator unit being operable to parse the legacy source code instructions and generate a directed call graph dataset, the directed call graph dataset comprising data representing nodes of a directed call graph, wherein each node represents the range of a function, and arcs between pairs of nodes, wherein the direction of each arc represents a respective calling and called relationships between the two respective functions;

a recursion detector unit communicatively coupled to the directed call graph generator unit and being operable to inspect the directed call graph dataset and generate a recursion dataset, the recursion dataset comprising data indicating if the directed call graph contains any Strongly Connected Components; and

a reaching end point detector unit communicatively coupled to the recursion detector unit and being responsive to the recursion dataset indicating the absence of any Strongly Connected Components to:

determine from the directed call graph dataset, for each node, a tuple comprising a range of the respective function, an end point of the respective function and any reaching end point of direct or indirect predecessor nodes; and

calculate from the tuple for each node a union over the range and the reaching end point(s); and

a compiler unit communicatively coupled to the control flow analyser unit and comprising:

a stack based compiler unit, operable if equivalence is determined by being responsive to the union for each node comprising a NULL set, to compile the legacy source code into compiled code using stack based semantics, for deployment to the processing environment; and

a legacy compiler unit, operable if non-equivalence is determined by being responsive to the presence of one or more Strongly Connected Components, or to the union for any node comprising a non-NULL set, to compile the legacy source code into compiled code using legacy semantics, for deployment to the processing environment,

wherein, if equivalence is determined, the legacy source code instructions, when compiled into compiled code using stack based semantics, have a first control flow that is the same as the expected control flow, whereas, if non-equivalence is determined, the legacy source code instructions, if compiled into compiled code using stack based semantics would have a second control flow that is different from the expected control flow.

2. A computer system according to claim 1, wherein the compiled code comprises object code for deployment to a native processing environment.

3. A computer system according to claim 1, wherein the complier unit comprises an intermediate code compiler unit for generating intermediate code and the system comprises a second compiler unit operable to compile the intermediate code into object code for execution in a runtime environment of a selected managed environment, the object code having a control flow that is determined by the respective control flow of the intermediate code as compiled by the intermediate code compiler unit.

4. A computer system according to claim 3, further comprising a runtime environment of the managed environment operable to execute the object code according to a control flow that is determined by the respective control flow of the intermediate code as compiled by the intermediate code compiler unit, the runtime environment comprising one or more system libraries comprising code to be called on execution of the object code.

5. A computer system according to claim 4, wherein the runtime environment comprises the second compiler unit.

6. A computer system according to any one of claims 3 to 5, wherein the second compiler unit is operable to compile the intermediate code just-in-time for execution thereof by the runtime environment.

7. A computer system according to any one of the preceding claims, wherein the functions comprise COBOL PERFORM ranges.

8. A computer system according to claim 7, wherein the legacy compiler unit is adapted to compile PERFORM ranges by implementing save buckets and exit buckets.

9. A computer system according to claim 7 or claim 8, wherein the legacy compiler unit is adapted to compile PERFORM ranges so that an end point of each PERFORM range is active during execution.

10. A computer system according to any one of claims 7 to 9, wherein the stack based compiler unit is adapted to compile PERFORM ranges so that only an end point of a most recently called PERFORM range is active during execution.

11. A computer system according to any one of the preceding claims, wherein the legacy source code instructions are written in one of: OS/VS COBOL, RM COBOL, COBOL/370, Enterprise COBOL, OS/390 COBOL and VS COBOL II.

12. A computer system according to any one of the preceding claims, wherein a range comprises one or more instructions that are executable in respect of a COBOL PERFORM statement.

13. A computer system according to any one of the preceding claims, wherein the control flow analyser unit is operable to determine an equivalence in control flow between the source code if compiled using stack based semantics and the source code if compiled using legacy semantics by determining if the source code instructions are ANSI compliant.

14. A computer-implemented method of preparing an application program for deployment, the application program comprising legacy source code instructions, written in a legacy system-independent programming language and intended for compilation and execution in a legacy processing environment, the source code instructions comprising a plurality of functions each having a range and at least one function call, the method comprising:

parsing legacy source code instructions and generating a directed call graph dataset, the directed call graph dataset comprising data representing nodes of a directed call graph, wherein each node represents the range of a function, and arcs between pairs of nodes, wherein the direction of each arc represents a respective calling and called relationships between the two respective functions;

inspecting the directed call graph dataset and generating a recursion dataset, the recursion dataset comprising data indicating if the directed call graph contains any Strongly Connected Components; and

responsive to the recursion dataset indicating the presence of at least one Strongly Connected Component, compiling the legacy source code into compiled code using legacy semantics, for deployment to a processing environment; or

responsive to the recursion dataset indicating the absence of any Strongly Connected Components, determining from the directed call graph dataset, for each node, a tuple comprising a range of the respective function, an end point of the respective function and any reaching end point of direct or indirect predecessor nodes; calculating from the tuple for each node a union over the range and the reaching end point(s); and:

responsive to the union for each node comprising a NULL set, compiling the legacy source code into compiled code using stack based semantics, for deployment to the processing environment; or

responsive to the union for any node comprising a non-NULL set, compiling the legacy source code using legacy semantics, for deployment to the processing environment.

15. A method according to claim 14, comprising generating object code for deployment to a native processing environment.

16. A method according to claim 14, comprising generating intermediate code and subsequently compiling the intermediate code into object code for execution in a runtime environment of a selected managed environment, the object code having a control flow that is determined by the respective control flow of the intermediate code.

17. A method according to claim 16, comprising executing the object code according to a control flow that is determined by the respective control flow of the intermediate code including calling one or more system libraries.

18. A method according to claims 16 or 17, comprising compiling the intermediate code just-in-time for execution thereof by the runtime environment.

19. A method according to any one of the claims 14 to 18, wherein the functions comprise COBOL PERFORM ranges.

20. A method according to claim 19, wherein the PERFORM ranges are compiled using legacy semantics by implementing save buckets and exit buckets.

21. A method according to claim 19 or claim 20, wherein the PERFORM ranges are compiled using legacy semantics so that an end point of each PERFORM range is active during execution.

22. A method according to any one of claims 19 to 21, wherein the PERFORM ranges are compiled using stack based semantics so that only an end point of a most recently called PERFORM range is active during execution.

23. A method according to any one of the claims 14 to 22, wherein the legacy source code instructions are written in one of: OS/VS COBOL, RM COBOL, COBOL/370, Enterprise COBOL, OS/390 COBOL and VS COBOL II.

24. A method according to any one of claims 14 to 23, wherein a range comprises one or more instructions that are executable in respect of a COBOL PERFORM statement.

25. A method of preparing an application program for deployment into a processing environment, the application program comprising legacy source code instructions representing the application program, written in a legacy system-independent programming language and being for compilation using legacy semantics for a respective legacy native computing environment, the method comprising:

determining an equivalence in control flow between the source code if compiled using stack based semantics and the source code if compiled legacy semantics; and

if equivalence is determined, compiling the legacy source code using stack based semantics, for deployment to the processing environment; or

if non-equivalence is determined, compiling the legacy source code using legacy semantics, for deployment to the processing environment.

26. A method according to claim 25, wherein the application program is written in legacy COBOL and equivalence is determined if the legacy COBOL is ANSI compliant.

27. A computer system to prepare an application program for deployment into a processing environment, the application program comprising legacy source code instructions representing the application program, written in a system-independent programming language and being for compilation using legacy semantics for a respective native computing environment, the system comprising:

a control flow analyser unit operable to determine an equivalence in control flow between the source code if compiled using stack based semantics and the source code if compiled legacy semantics;

a compiler unit communicatively coupled to the control flow analyser unit and comprising:

a stack based compiler unit, operable if equivalence is determined, to compile the legacy source code using stack based semantics, for deployment to the processing environment; and

a legacy compiler unit, operable if non-equivalence is determined, to compile the legacy source code using legacy semantics, for deployment to the processing environment.

28. A computer system according to claim 27, wherein the application program is written in legacy COBOL and equivalence is determined if the legacy COBOL is ANSI compliant.

29. A computer system to deploy an application program into a processing environment comprising:

a runtime environment arranged to support execution, by way of at least one processor and a memory, of compiled code generated by a computer system according to claim 1 or claim 27, the runtime environment comprising:

one or more system libraries comprising executable code to implement at least the runtime environment, the one or more system libraries comprising:

a first set of executable code implementing a first set of one or more functions configured to handle a monolithic program, and

a second set of executable code implementing a second set of one or more functions configured to use one or more stack registers,

wherein object code corresponding to code compiled using legacy semantics is arranged to instruct execution of the first set of executable code by way of the at least one processor and object code corresponding to code compiled using stack based semantics is arranged to instruct execution of the second set of executable code by way of the at least one processor.

30. A computer-implemented method to deploy an application program into a processing environment comprising:

loading object code into memory, said object code being derived from compiled code generated by a method according to claim 14 or claim 25;

processing said object code by way of at least one processor communicatively coupled to said memory, said processing comprising:

responsive to said object code corresponding to code compiled using legacy semantics, executing a first set of executable code by way of the at least one processor and the memory, the first set of executable code forming part of one or more system libraries arranged to implement a runtime environment, the first set of executable code implementing a first set of one or more functions configured to handle a monolithic program, and

responsive to said object code corresponding to code compiled using stack based semantics, executing a second set of executable code by way of the at least one processor and the memory, the second set of executable code forming part of one or more system libraries arranged to implement a runtime environment, the second set of executable code implementing a second set of one or more functions configured to use one or more stack registers.

31. A computer program comprising computer program code arranged to, when loaded into system memory and processed by one or more processors, implement the computer-implemented methods of any of claims 14 to 26.

32. A computer-readable storage medium having recorded thereon the computer program of claim 29.

33. A computer system, substantially as hereinbefore described and/or with reference to the drawings in Figures 3 to 12.

34. A computer-implemented method substantially as hereinbefore described and/or with reference to the drawings in Figures 3 to 12.

Amendment to the claims have been filed as follows

Claims

1. A computer system to prepare an application program for deployment into a processing environment, the application program comprising source code instructions representing the application program and having an expected control flow, the application program being written in a system-independent programming language intended for compilation using non-stack-based semantics for a respective native computing environment, the non-stack-based semantics being operable to generate multiple active end points in a program control flow, the system comprising:

a first memory unit to store the source code instructions, the source code instructions comprising a plurality of functions each having a range and at least one function call;

a control flow analyser unit communicatively coupled to the memory unit and being operable to determine an equivalence in control flow of respective compiled code resulting from the source code if compiled using stack-based semantics and the source code if compiled using non-stack-based semantics, stack-based semantics being operable to generate a single active end point in a program control flow, the control flow analyser unit comprising:

a directed call graph generator unit being operable to parse the source code instructions and generate a directed call graph dataset, the directed call graph dataset comprising data representing nodes of a directed call graph, wherein each node represents the range of a function, and arcs between pairs of nodes, wherein the direction of each arc represents a respective calling and called relationships between the two respective functions;

a recursion detector unit communicatively coupled to the directed call graph generator unit and being operable to inspect the directed call graph dataset and generate a recursion dataset, the recursion detector unit being arranged to iteratively select a node as represented in the directed call graph dataset and traverse the directed call graph as represented by the directed call graph dataset by following arcs in said graph, said traversal being used to determine whether there is a path in said graph to the selected node, the recursion dataset comprising data indicating if the directed call graph contains any Strongly

Connected Components, a Strongly Connected Component being indicated when there is a path in said graph to a selected node; and

a reaching end point detector unit communicatively coupled to the recursion detector unit and being responsive to the recursion dataset indicating the absence of any Strongly Connected Components to:

determine from the directed call graph dataset, for each node, a tuple comprising a range of the respective function, an end point of the respective function and any reaching end point of direct or indirect predecessor nodes; and

calculate from the tuple for each node a union over the range and the reaching end point(s); and

a compiler unit communicatively coupled to the control flow analyser unit and comprising:

a stack-based compiler unit, operable if equivalence is determined for the application program by being responsive to the union for each node comprising a NULL set, to compile the source code for the application program into compiled code using stack-based semantics, for deployment to the processing environment; and

a non-stack-based compiler unit, operable if non-equivalence is determined for the application program by being responsive to the presence of one or more Strongly Connected Components, or to the union for any node comprising a non-NULL set, to compile the source code for the application program into compiled code using non-stack-based semantics, for deployment to the processing environment,

wherein, if equivalence is determined for the application program, the source code instructions for the application program, when compiled into compiled code using stack-based semantics, have a first control flow that is the same as the expected control flow, whereas, if non-equivalence is determined for the application program, the source code instructions for the application program, if compiled into compiled code using stack-based semantics would have a second control flow that is different from the expected control flow.

2. A computer system according to claim 1, wherein the compiled code comprises object code for deployment to a native processing environment.

3. A computer system according to claim 1, wherein the complier unit comprises an intermediate code compiler unit for generating intermediate code and the system comprises a second compiler unit operable to compile the intermediate code into object code for execution in a runtime environment of a selected managed environment, the object code having a control flow that is determined by the respective control flow of the intermediate code as compiled by the intermediate code compiler unit.

4. A computer system according to claim 3, further comprising a runtime environment of the managed environment operable to execute the object code according to a control flow that is determined by the respective control flow of the intermediate code as compiled by the intermediate code compiler unit, the runtime environment comprising one or more system libraries comprising code to be called on execution of the object code.

5. A computer system according to claim 4, wherein the runtime environment comprises the second compiler unit.

6. A computer system according to any one of claims 3 to 5, wherein the second compiler unit is operable to compile the intermediate code just-in-time for execution thereof by the runtime environment.

7. A computer system according to any one of the preceding claims, wherein the functions comprise COBOL PERFORM ranges.

8. A computer system according to claim 7, wherein the non-stack-based compiler unit is adapted to compile PERFORM ranges by implementing save buckets and exit buckets.

9. A computer system according to claim 7 or claim 8, wherein the non-stack-based compiler unit is adapted to compile PERFORM ranges so that an end point of each PERFORM range is active during execution.

10. A computer system according to any one of claims 7 to 9, wherein the stack based compiler unit is adapted to compile PERFORM ranges so that only an end point of a most recently called PERFORM range is active during execution.

11. A computer system according to any one of the preceding claims, wherein the source code instructions are written in one of: OS/VS COBOL, RM COBOL, COBOL/370, Enterprise COBOL, OS/390 COBOL and VS COBOL II.

12. A computer system according to any one of the preceding claims, wherein a range comprises one or more instructions that are executable in respect of a COBOL PERFORM statement.

13. A computer-implemented method of preparing an application program for deployment, the application program comprising source code instructions, written in a system-independent programming language intended for compilation using non-stack-based semantics and execution in a processing environment that uses non-stack-based processing, the non-stack-based semantics being operable to generate multiple active end points in a program control flow, the source code instructions comprising a plurality of functions each having a range and at least one function call, the method comprising:

parsing source code instructions for the application program and generating a directed call graph dataset, the directed call graph dataset comprising data representing nodes of a directed call graph, wherein each node represents the range of a function, and arcs between pairs of nodes, wherein the direction of each arc represents a respective calling and called relationships between the two respective functions;

inspecting the directed call graph dataset and generating a recursion dataset, said inspecting comprising iteratively selecting a node as represented in the directed call graph dataset and traversing the directed call graph as represented by the directed call graph dataset by following arcs in said graph, said traversing being used to determine whether there is a path in said graph to the selected node, the recursion dataset comprising data indicating if the directed call graph contains any Strongly Connected Components, a Strongly Connected Component being indicated when there is a path in said graph to a selected node; and

19 11 13

responsive to the recursion dataset indicating the presence of at least one Strongly Connected Component, compiling the source code for the application program into compiled code using non-stack-based semantics, for deployment to a processing environment; or

responsive to the recursion dataset indicating the absence of any Strongly Connected Components, determining from the directed call graph dataset, for each node, a tuple comprising a range of the respective function, an end point of the respective function and any reaching end point of direct or indirect predecessor nodes; calculating from the tuple for each node a union over the range and the reaching end point(s); and:

responsive to the union for each node comprising a NULL set, compiling the source code for the application program into compiled code using stack-based semantics, for deployment to the processing environment, stack-based semantics being operable to generate a single active end point in a program control flow; or

responsive to the union for any node comprising a non-NULL set, compiling the source code for the application program using non-stack-based semantics, for deployment to the processing environment.

14. A method according to claim 13, comprising generating object code for deployment to a native processing environment.

15. A method according to claim 13, comprising generating intermediate code and subsequently compiling the intermediate code into object code for execution in a runtime environment of a selected managed environment, the object code having a control flow that is determined by the respective control flow of the intermediate code.

16. A method according to claim 15, comprising executing the object code according to a control flow that is determined by the respective control flow of the intermediate code including calling one or more system libraries.

17. A method according to claims 15 or 16, comprising compiling the intermediate code just-in-time for execution thereof by the runtime environment.

18. A method according to any one of the claims 13 to 17, wherein the functions comprise COBOL PERFORM ranges.

19. A method according to claim 18, wherein the PERFORM ranges are compiled using non-stack-based semantics by implementing save buckets and exit buckets.

20. A method according to claim 18 or claim 19, wherein the PERFORM ranges are compiled using non-stack-based semantics so that an end point of each PERFORM range is active during execution.

21. A method according to any one of claims 18 to 20, wherein the PERFORM ranges are compiled using stack-based semantics so that only an end point of a most recently called PERFORM range is active during execution.

22. A method according to any one of the claims 13 to 21, wherein the source code instructions are written in one of: OS/VS COBOL, RM COBOL, COBOL/370, Enterprise COBOL, OS/390 COBOL and VS COBOL II.

23. A method according to any one of claims 13 to 22, wherein a range comprises one or more instructions that are executable in respect of a COBOL PERFORM statement.

24. A computer-implemented method to deploy an application program into a processing environment comprising:

loading object code into memory, said object code being derived from compiled code generated by a method according to claim 13;

processing said object code by way of at least one processor communicatively coupled to said memory, said processing comprising:

responsive to said object code corresponding to code compiled using non-stack-based semantics, executing a first set of one or more functions implemented by executable code by way of the at least one processor and the memory, the first set of functions forming part of one or more system libraries arranged to implement a runtime environment, the first set of one or more functions being configured to handle a program using save buckets and exit buckets, and

responsive to said object code corresponding to code compiled using stack-based semantics, executing a second set of one or more functions implemented by executable code by way of the at least one processor and the memory, the second set of functions forming part of one or more system libraries arranged to implement a runtime environment, the second set of one or more functions being configured to use one or more stack registers.

25. A computer program comprising computer program code arranged to, when loaded into system memory and processed by one or more processors, implement the computer-implemented methods of any of claims 13 to 24.

26. A computer-readable storage medium having recorded thereon the computer program of claim 25.

27. A computer system, substantially as hereinbefore described and/or with reference to the drawings in Figures 3 to 12.

28. A computer-implemented method substantially as hereinbefore described and/or with reference to the drawings in Figures 3 to 12.

-

Application No:	GB1314580.0	**Examiner:**	Mr Nikki Dowell
Claims searched:	1 to 28, 31	**Date of search:**	24 September 2013

Patents Act 1977: Search Report under Section 17

Documents considered to be relevant:

Category	Relevant to claims	Identity of document and passage or figure of particular relevance
X	25-28 and 31 at least	US 2006/0200811 A1 (Cheng) see especially paragraphs 0067 to 0098
A	-	US 2004/0154009 A1 (Reynaud) see especially paragraphs 0118 to 0169, 0220, 0221
A	-	US 2011/0307507 A1 (Zhou et al) see whole document
A	-	US 2012/0096444 A1 (Wright et al) see whole document

Categories:

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

Field of Search:

Search of GB, EP, WO & US patent documents classified in the following areas of the UKC^{X} :

Worldwide search of patent documents classified in the following areas of the IPC

G06F

The following online and other databases have been used in the preparation of this search report

EPODOC,WPI,TXTE,INSPEC,INTERNET,XPESP,XPI3E,XPIPCOM

International Classification:

Subclass	Subgroup	Valid From
G06F	0009/45	01/01/2006