

**AUTOMATICALLY CONVERTING A PROGRAM WRITTEN IN A
PROCEDURAL PROGRAMMING LANGUAGE INTO A DATAFLOW GRAPH
AND RELATED SYSTEMS AND METHODS**

BACKGROUND

[0001] Procedural programming is a paradigm in which programs are written as a series of explicit commands for a processor to carry out in order. Sub-sequences of commands can be grouped and called as desired from other parts of the program and thereby extend the range of commands the rest of the program can execute. These sub-sequences are sometimes referred to as subroutines, functions or subprograms, depending on the particular programming language being employed.

[0002] Dataflow programming, in contrast, emphasizes the flow of data through a sequence of operations through programs called dataflow graphs that define operations and how data flows between the operations. Dataflow graphs are generally configured to obtain data records from one or more sources, perform operations on the data records in a sequence, and produce some kind of output. Typically the parts of the dataflow graph that perform operations on data records are called components, and the connections between the components are called flows.

[0003] A difference between programs written in procedural languages and dataflow graphs is that multiple operations can be performed simultaneously in a dataflow graph because data records reaching earlier components in the dataflow graph (“upstream” components) can be operated on while other data records have already propagated to later components in the dataflow graph (“downstream” components). This phenomenon is a type of parallelism sometimes called pipeline parallelism. In this way, a dataflow graph can be thought of like a series of workers in an assembly line, with the components representing the workers, whereas a program written in a procedural language is more like a single worker completing a series of tasks, where each task is completed before the next is begun.

SUMMARY

[0004] According to some aspects, a non-transitory computer readable medium is provided comprising instructions that, when executed by at least one processor, perform a method of converting a program into a dataflow graph, wherein the program comprises data operations and a control flow indicating a sequence of execution of the data operations, the method comprising analyzing the program using the at least one processor and using a grammar stored by the at least one computer readable medium, said analyzing comprising identifying, based on the grammar, a plurality of sequential steps of the program, wherein each step of the plurality of steps comprises one or more data operations of the program, generating a catalog storing control flow relationships between steps of the plurality of identified steps of the program, for each of the identified plurality of steps of the program, based on data operations of the step and based on the grammar, identifying one of a plurality of dataflow graph component types as corresponding to the step, thereby identifying a plurality of dataflow graph components each corresponding to a respective step of the plurality of steps of the program, and configuring the plurality of dataflow graph components based on respective corresponding steps of the plurality of steps of the program and based on the grammar, thereby producing a plurality of configured dataflow graph components, and generating, using the at least one processor, a dataflow graph comprising the plurality of configured dataflow graph components, and one or more flows representing flows of data records between the one or more dataflow graph components, wherein the one or more dataflow graph components are connected via the one or more flows in an order according to the generated catalog.

[0005] According to some aspects, a computer-implemented method is provided of automatically converting a program into a dataflow graph, wherein the program comprises data operations and a control flow indicating a sequence of execution of the data operations, the method comprising analyzing the program using the at least one processor and using a grammar stored by the at least one computer readable medium, said analyzing comprising identifying, based on the grammar, a plurality of sequential steps of the program, wherein each step of the plurality of steps comprises one or more data operations of the program, generating a catalog storing control flow relationships

between steps of the plurality of identified steps of the program, for each of the identified plurality of steps of the program, based on data operations of the step and based on the grammar, identifying one of a plurality of dataflow graph component types as corresponding to the step, thereby identifying a plurality of dataflow graph components each corresponding to a respective step of the plurality of steps of the program, and configuring the plurality of dataflow graph components based on respective corresponding steps of the plurality of steps of the program and based on the grammar, thereby producing a plurality of configured dataflow graph components, and generating, using the at least one processor, a dataflow graph comprising the plurality of configured dataflow graph components, and one or more flows representing flows of data records between the one or more dataflow graph components, wherein the one or more dataflow graph components are connected via the one or more flows in an order according to the generated catalog.

[0006] According to some aspects, a system for automatically converting a program into a dataflow graph is provided, wherein the program comprises data operations and a control flow indicating a sequence of execution of the data operations, the system comprising means for analyzing the program using the at least one processor and using a grammar stored by the at least one computer readable medium, said analyzing comprising identifying, based on the grammar, a plurality of sequential steps of the program, wherein each step of the plurality of steps comprises one or more data operations of the program, generating a catalog storing control flow relationships between steps of the plurality of identified steps of the program, for each of the identified plurality of steps of the program, based on data operations of the step and based on the grammar, identifying one of a plurality of dataflow graph component types as corresponding to the step, thereby identifying a plurality of dataflow graph components each corresponding to a respective step of the plurality of steps of the program, and configuring the plurality of dataflow graph components based on respective corresponding steps of the plurality of steps of the program and based on the grammar, thereby producing a plurality of configured dataflow graph components, and means for generating a dataflow graph, the dataflow graph comprising the plurality of configured dataflow graph components, and one or more flows representing flows of data records between the one or more dataflow graph components, wherein the one or more dataflow

graph components are connected via the one or more flows in an order according to the generated catalog.

[0007] The foregoing apparatus and method embodiments may be implemented with any suitable combination of aspects, features, and acts described above or in further detail below. These and other aspects, embodiments, and features of the present teachings can be more fully understood from the following description in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF DRAWINGS

[0008] Various aspects and embodiments will be described with reference to the following figures. It should be appreciated that the figures are not necessarily drawn to scale. In the drawings, each identical or nearly identical component that is illustrated in various figures is represented by a like numeral. For purposes of clarity, not every component may be labeled in every drawing.

[0009] FIG. 1 depicts an illustrative dataflow graph, according to some embodiments;

[0010] FIG. 2 depicts a flowchart of a method of automatically converting a program written in a procedural language into a dataflow graph, according to some embodiments;

[0011] FIG. 3 is a flowchart of a method of automatically converting a program written in a procedural language into a dataflow graph, according to some embodiments;

[0012] FIG. 4A is an illustrative SAS program, according to some embodiments;

[0013] FIG. 4B depicts an illustrative dataflow graph generated according to the program shown in FIG. 4A, according to some embodiments;

[0014] FIG. 4C depicts the illustrative dataflow graph of FIG. 4B with dataflow graph components annotated with corresponding steps of the SAS program, according to some embodiments;

[0015] FIG. 4D illustrates details of converting a first step of the program shown in FIG. 4A into a first component of the dataflow graph shown in FIG. 4B, according to some embodiments;

[0016] FIG. 4E illustrates an example of parsing a step of the program shown in FIG. 4A with a grammar to identify the step and control flow relationships between the step and other steps, select a corresponding dataflow graph component, and configure the selected dataflow graph component, according to some embodiments;

[0017] FIG. 4F illustrates details of converting a second step of the program shown in FIG. 4A into a second component of the dataflow graph shown in FIG. 4B, according to some embodiments; and

[0018] FIG. 5 illustrates an example of a computing system environment on which aspects of the invention may be implemented.

DETAILED DESCRIPTION

[0019] The inventors have recognized and appreciated that the functionality of a data processing system may be increased via a computer-executed tool that converts programs, written in a procedural language, into components that may be executed as a dataflow graph. In particular, the computer-executed tool may identify types of dataflow graph components as corresponding to portions of a program so that the behavior of each portion may be reproduced by a suitable portion of a dataflow graph. In some embodiments, the graph generated from a program written in a procedural programming language may support various forms of parallelism, such as pipeline parallelism and/or component parallelism, even though the program written in the procedural language may have lacked such parallelism. In some embodiments, parallelism may be achieved by parsing the program based on a grammar and identifying control flow relationships between data operations performed by the program. Furthermore, types of dataflow graph components may be identified using the grammar as corresponding to particular data operations (or groups of data operations) of the program. A dataflow graph may be generated to comprise the identified components, which may be connected together with flows in an order according to the identified control flow relationships between data operations.

[0020] There can be a number of benefits to processing data using dataflow programming compared with procedural programming. In particular, as discussed above, a dataflow graph may naturally provide for parallel processing through pipeline

parallelism, making dataflow programming a natural choice for processing datasets where particular operations are repeated on many pieces of data, such as data records of a dataset. An additional type of parallelism afforded by dataflow graphs is component parallelism, which is produced when different branches of the dataflow graph are operating at the same time. For instance, data records from a single data source may be output along different flows, and different operations performed along the different branches at the same time.

[0021] Despite the advantages that can be afforded by dataflow graphs, some organizations (e.g., businesses) may nonetheless rely on programs written in a procedural language for data processing. In some cases, an organization may use dataflow graphs in part whilst still leveraging programs written in a procedural language. This approach typically involves executing dataflow graphs instead of the former programs written in the procedural language whilst including components in the dataflow graphs that execute, external to the dataflow graph, some of the code of the programs written in the procedural language. In this manner, the dataflow graph is being executed as a “wrapper” of sorts, and while the dataflow graph is ultimately the program that is executed by the system, a great deal of the functionality of the program may still be executed externally by the procedural language.

[0022] One drawback with this approach may be that the dataflow graph language may have no ability to understand what is happening within a component that executes external code written in a procedural language. Another useful feature of dataflow graphs is the ability to trace data lineage through a single dataflow graphs or through multiple dataflow graphs to gain an understand of where data is coming from and going to within the operations of an organization. A component that executes external code may, however, be akin to a “black box” to the dataflow graph software with respect to data lineage, because the dataflow graph software may not be able to trace data lineage through the component since the operations the component performs are unknown to the software. Moreover, this approach may inhibit pipeline parallelism because the external code may need to be fully executed before downstream components may execute.

[0023] As discussed above, the inventor has recognized and appreciated that the functionality of a data processing system may be increased via a computer-executed tool

that converts programs, written in a procedural language, into components that may be executed as a dataflow graph. These techniques may be applied to generate native dataflow graph code throughout the generated dataflow graph so that data lineage can be tracked through the dataflow graph. The natural efficiency benefits of dataflow graphs, such as component parallelism and/or pipeline parallelism as discussed above, may be naturally provided in a generated dataflow graph. The inventor has further recognized and appreciated techniques for implementing such a tool based on a grammar, as discussed further below.

[0024] According to some embodiments, a system converting a program written in a procedural language into a dataflow graph may execute a parser to parse the text of the program. In some embodiments, such parsing may identify operations performed on data by the program and may further identify the type of such operations. Based on the type of data operation being performed by the program, a particular type of dataflow graph component able to perform the same operation may be identified by accessing a data structure that has been constructed to associate particular operations with particular dataflow graph components. For example, if language within the program is identified by the parser as performing a merge between data, the data structure may identify a dataflow graph component that joins together flows of data records as corresponding to a merge operation. Such a parser may also, in some embodiments, determine an execution order of data operations as executed by the program. When generating a dataflow graph based on the program, flows between components representing the data operations may be generated and the components connected together via the flows in an order based on the determined execution order.

[0025] Following below are more detailed descriptions of various concepts related to, and embodiments of, techniques for automatically converting programs written in procedural languages into dataflow graphs. It should be appreciated that various aspects described herein may be implemented in any of numerous ways. Examples of specific implementations are provided herein for illustrative purposes only. In addition, the various aspects described in the embodiments below may be used alone or in any combination, and are not limited to the combinations explicitly described herein.

[0026] FIG. 1 depicts an illustrative dataflow graph, according to some embodiments. Dataflow graph 100 is depicted in FIG. 1 to illustrate some of the above-described features of a dataflow graph. A dataflow graph is a program that can be represented as a visual flow of data records through components and flows. The components, as referred to herein, are parts of the dataflow graph that perform operations on data records, and the connections between the components are referred to herein as flows.

[0027] Generally speaking, a dataflow graph is produced through arranging components and flows within a graphical user interface. FIG. 1 illustrates an example of such a view of a dataflow graph, wherein the rectangular boxes components and the lines between them each represent a flow. Dataflow graphs are a convenient way to write programs that operate on data record because the flow of data can be visualized when viewing the dataflow graph. Once a dataflow is ready to be executed, it may be compiled then executed by a suitable computing system.

[0028] In the example of FIG. 1, three data inputs 102, 104 and 106 are accessed, manipulated and combined in a manner to be described below, to produce two data outputs 122 and 124. A data input may be provided from various sources, including from a file stored on a computer readable medium, from a collection of files, or from a database table. In the example of FIG. 1, the data inputs 102, 104 and 106 are depicted in a manner often used to depict single data files stored on a computer readable medium.

[0029] In the example of FIG. 1, input data 102 is read and operated on by filter component 108, which outputs data records that meet a particular logical expression. For example, if input data 102 included a data field specifying a country code, filter 108 might be configured to output only those data records that have a value of that data field of "US". Data records output from the filter 108 are input to sort component 110, which sorts the data records into an order dictated by a selected sort "key," which identifies one or more data fields according to which the records are to be ordered. The sorted data records are output to the join component 120.

[0030] In addition, input data 104 is read and operated on by transform component 112, which manipulates the input data records according to transform code stored within (or otherwise accessed by) the transform component 112. A transform may, for

example, add or remove data fields for each data record and/or may manipulate values of data fields in producing output data. Note that, in the example of FIG. 1, the filter component 108 and the transform component 112 exhibit component parallelism. That is, data records from input data 102 may be operated on by the filter 108 at the same time as data records from input data 104 are operated on by the transform 112. The component parallelism may, in some cases, be exploited by executing the filter 108 and transform 112 components using different processors, and in some cases using different computing systems.

[0031] In the example of FIG. 1, join component 116 receives data records output from transform 112 and output from filter component 114 (which operates as discussed above in relation to filter 108). The join component may be configured to produce a single set of data records from two sets of data records by matching records from each set according to a key which identifies a data field from each set of input data records. Transform code stored within (or otherwise accessed by) the join component 116 may further dictate how to construct outgoing data records from the join component based on data field values contained within matching records from the two sets of input data records.

[0032] The data records output from the join component 116 are replicated to produce two identical sets of data records by replicate component 118. One of these sets of data records is stored as output data 124, whereas the other set is joined with the data records output from sort 110 and the result stored as output data 122.

[0033] In general, components in a dataflow graph are configured to some extent to define their function. In the case of the filter and transform components in the example of FIG. 1, for instance, the components are configured through transform code which dictates how the records are selected or transformed, respectively. In some cases, a transform component may also define an output record format that differs from the record format of input data records. The sort component may be configured by specifying a sort key and in some cases, by specifying how the sort is to be performed (e.g., ascending versus descending values). In each of these cases, some manner of configuration is applied to describe how the component should operate on data records. The data that describes such configuration may be collectively referred to herein as

“configuration data.” Transform code, key values, record formats, etc. are therefore each a type of configuration data. The replicate component in the example of FIG. 1 may, in at least some cases, function without defined configuration data because the component is designed only to replicate a data flow and may not be otherwise configurable.

[0034] FIG. 2 depicts a flowchart of a method of automatically converting a program written in a procedural language into a dataflow graph, according to some embodiments. As discussed above, according to some aspects of the present application, the functionality of a data processing system may be increased via a computer-executed tool that converts programs, written in a procedural language, into components that may be executed as a dataflow graph. Method 200 illustrates how such a tool may be operated to perform such a conversion by converting program 201 into dataflow graph 208.

[0035] In the example of FIG. 2, a program 201, which is written in a procedural language, is provided as input to a conversion engine 205, which comprises one or more computing devices executing the tool discussed above. The program 201 may be written in any procedural language, as the techniques described herein are not limited to any particular language. Examples of suitable languages include Statistical Analysis System (SAS), BASIC, Fortran, Java, Mathematica, MATLAB, Pascal, Perl, Python, R, Unix shell, and Visual Basic.

[0036] During execution by the conversion engine, the tool may utilize a predefined grammar 206. The grammar 206 may provide various definitions relied upon by the tool executed by the conversion engine 205 that allow the tool to parse the program 201 and generate the dataflow graph 208. These definitions provided by the grammar 206 may indicate to the tool how to interpret words and symbols within the program 201, how to identify data operations (or groups of data operations) within the program as having functionality corresponding to a particular type of dataflow graph component, how to generate dataflow graph components, how to identify data operations (or groups of data operations) within the program as relating to configuration data for an identified type of dataflow graph component, how to configure dataflow graph components that correspond to data operations within the program, and/or how to determine the control flow of data operations within the program.

[0037] As used herein, “control flow” refers to an order, consistent with the program, in which individual data operations are performed. For instance, a program may perform operations C, D, and E and may be configured such that operation C is performed before either of operation D or operation E. This dependency on the relative ordering of operations C, D, and E is one example of control flow.

[0038] In the example of FIG. 2, data produced by the tool parsing the program may be stored in catalog 207, which may comprise any suitable data structure in which aspects such as determined control flow relationships of data operations in the program may be stored. The catalog may be accessed by the conversion engine during subsequent parsing operations as described further below.

[0039] As referred to herein, a “data operation” for a program refers to a portion of the program that, when executed, modifies the execution environment of the program. Data operations may include, for instance, the modification of the value of one or more variables or other data that exist within the execution environment. It will be appreciated that the modification of a value may itself comprise one or more data operations. For instance, when a command in a procedural program sets the value of a variable to a value equal to the result of a calculation, each part of the calculation may represent a separate data operation, as does the end result of storing the calculated value in the variable. As such, in at least some cases, a single command of the program may represent multiple data operations because various changes to the execution environment occur during execution of the command.

[0040] According to some embodiments, the conversion engine 205 may generate one or more components and flows of a dataflow graph 208 as corresponding to a sequence of data operations within the program 201. In some cases, generating a component of dataflow graph 208 may include identifying a type of component from amongst a plurality of available dataflow graph components. In some cases, generating a component of dataflow graph 208 may also include generating configuration data for an identified component, which may comprise defining parameter or other configurable aspects of the component based on data operations within the program. For example, instances of illustrative dataflow graph transform code 209 shown in FIG. 2 may be

generated by the tool executed by the conversion engine and associated with the identified component.

[0041] According to some embodiments, the tool executed by conversion engine 205 may generate flows between identified components of the dataflow graph 208 based on determined control flow relationships of data operations within the program. For instance, where the tool identifies first and second components for particular data operations (or sets of data operations), the tool may also identify that the data operation(s) associated with the first component occur earlier within the control flow than the data operation(s) associated with the second component. As a result, the tool may generate one or more flows connecting the first and second components so that the second component is executed downstream the dataflow graph from the first component (that is, data flows out of the first component earlier in the sequence of components than data flowing into the second component).

[0042] The net result of the process performed by the conversion engine 205, based on the grammar 206, is to generate the dataflow graph 208, which is a dataflow graph representation of the program 201. This relationship between a program and an equivalent dataflow graph is illustrated in FIG. 2, and in subsequent drawings, by a zig-zag arrow.

[0043] FIG. 3 is a flowchart of a method of automatically converting a program written in a procedural language into a dataflow graph, according to some embodiments. Method 300 is an illustrative process in which a program 301 written in a procedural language is automatically converted into a dataflow graph. Method 300 may be performed, for example, by the tool executed by conversion engine 205 shown in FIG. 2 and discussed above.

[0044] Method 300 begins with act 302 in which the program 301 is separated into steps based on a grammar 303. A tool performing method 300 may rely, as with the example of grammar 206 in FIG. 2, on a grammar 303 to indicate, among other things, how to interpret words and symbols within a program and how to identify data operations (or groups of data operations) within the program as having functionality corresponding to a particular type of dataflow graph component. In particular, the tool may parse the program 301 using the grammar 303 to identify one or more data

operations in act 302. Each group of one or more data operations may be termed a “step,” so that the data operations of the program are separated into a sequence of steps wherein each step contains one or more data operations performed by the program.

[0045] In act 304, the tool executing method 300 may store or otherwise record information identifying the recognized steps in catalog 305. The catalog produced in act 304 may identify steps of the program and the control flow relationships of said steps, and may further identify the one or more data operations of the program 301 that correspond to each step.

[0046] In act 306, the tool executing method 300 selects types of dataflow graph components that correspond to the identified steps of the program (and which may have been stored in the catalog in act 304). The type of dataflow graph component selected may be based on parsing the data operations of an identified step using grammar 303 and determining, based on said parsing, which type of dataflow graph component corresponds to the data operations of the step. As discussed above, the grammar 303 may be configured to relate particular types of dataflow graph components to particular language within the program, so that a dataflow graph component that performs the same function as a step in which the language appears may be generated appropriately.

[0047] In some embodiments, a preconfigured library of dataflow graph component types may be available to the tool so that identifying a dataflow graph component comprises selecting one of the component types from said library (e.g., a component of type “sort” or of type “filter” as discussed in relation to FIG. 1). In some embodiments, an instance of a dataflow graph component of the selected component type may be generated that has a corresponding function as a step of the program.

[0048] In act 308, the tool executing method 300 may configure dataflow graph components selected in act 306. Act 308 may accordingly comprise generating configuration data for one or more of the components selected in act 306. The tool may access a grammar to determine how to configure a given dataflow graph component based on the data operations present in the corresponding step of the program. As discussed above, data that describes the configuration of a dataflow graph component is referred to herein as “configuration data.” Act 308 may thereby comprise generating values for operational parameters of a component and/or generating transform code for a

component. It may be appreciated that some of the components identified in act 306 may not require configuration data and, as a result, configuration data may not be generated for all of the identified components.

[0049] In act 310, the tool executing method 300 may generate a dataflow graph comprising the identified components configured, where applicable, by act 308 (e.g., comprising the generated configuration data), and with flows connecting the components in a sequence according to the identified control flow relationships of the steps of the program. In some embodiments, act 310 may comprise recording the already-identified and configured components along with generated flows connecting the components, to a suitable computer readable storage medium. For example, a dataflow graph may be stored as a data file that includes data describing or otherwise identifying dataflow graph components, flows and configuration data for the components, and act 310 may comprise recording said data file. In some embodiments, such a data file may be arranged so as to be compiled by another program before execution. In some embodiments, the generated dataflow graph may be produced as an executable data file, in which case act 310 may comprise a step of compiling the dataflow graph according to the identified dataflow graph components, flows and configuration data for the components. Act 310 may comprise generating flows so as to connect the dataflow graph components in an ordering according to the control flow relationships identified by the catalog.

[0050] According to some embodiments, program 301 may comprise one or more procedures (which may also be referred to as routines, functions, subroutines or macros) and the tool executing method 300 may identify such procedures when parsing the program using a grammar. For instance, the tool may identify one or more procedure calls and/or may recognize procedure definitions. In some embodiments, the tool executing method 300 may generate a reusable group of dataflow graph components, sometimes called a “subgraph,” to represent the data operations of an identified procedure. In some cases, this subgraph may be stored in a data file that may be accessed by the generated dataflow graph. This may allow the generated dataflow graph to reuse the same set of components in a similar manner to how the program 301 may make repeated procedure calls and thereby reuse the same section of code. The subgraph

may represent an entire identified step of the program, or may instead represent a portion of an identified step of the program.

[0051] In cases where the tool executing method 300 generates a subgraph to represent a procedure of program 301, the procedure may utilize one or more values as inputs to the procedure. In such cases, the generated subgraph may be parameterized so that the behavior of the dataflow graph components in the subgraph behave differently based on the value of the subgraph parameters. For instance, branches of components may be enabled or disabled within the subgraph based on the value of a component.

[0052] According to some embodiments, program 301 may comprise, or otherwise function based on, one or more global variables, which are values defined in the execution environment and that may alter the way in which data operations in the program are performed. In such cases, the tool executing method 300 may generate one or more dataflow graph parameters, which are parameters set at the dataflow graph level whose values may be accessed by one or more components of the dataflow graph to dictate the component's behavior. As an illustrative example, the program 301 may be written to initialize a global variable having a value of a country code and to perform subsequent processing of data dependent on said value. A generated dataflow graph may be defined with a graph parameter to define the country code and components of the dataflow graph may utilize this parameter definition in transform code or otherwise.

[0053] According to some embodiments, the tool executing method 300 may identify comments within program 301. Comments are sections of a program that are not executed and are typically identified using a delimiter at the beginning or the beginning and end of the comment section. In some cases, the tool executing method 300 may identify such comments within the program and copy those comments into relevant locations within the generated dataflow graph. Comments may, for instance, be identified by the tool as being related to particular data operations within the program based on proximity of the comment text to the data operations. Such comments may be placed within transform code of a component corresponding to those data operations such that any information within those comments may be preserved in the resulting dataflow graph.

[0054] According to some embodiments, the tool executing method 300 may identify and preserve whitespace within program 301 when generating the dataflow graph. As referred to herein, “whitespace” refers to any character or series of characters that represent horizontal or vertical space in typography within the program, the most common example being a “space” character. In some cases, the tool executing method 300 may identify such whitespace within a data operation of the program and preserve the presence of the whitespace when generating transform code (or other configuration data) for the data operation within a component of the dataflow graph.

[0055] FIGs. 4A-4F provide an illustrative example of converting a program written in the procedural language SAS into a dataflow graph, according to some embodiments. FIG. 4A depicts an illustrative SAS program, which performs a series of operations on two input data files and produces an output data file. As discussed above, a computer-executed tool may be executed to convert the illustrated program into components that may be executed as a dataflow graph.

[0056] According to some embodiments, when executed and provided program 400 as input, the tool may identify four steps of the program, labeled 401a, 401b, 401c and 401d. As discussed above in relation to act 302 of FIG. 3, a tool may access a grammar to identify data operations or groups of data operations that correspond to steps of the program. In the illustrative program 400, these steps are configured to perform the following actions. Step 401a reads a data file called “FILE_A”, parses its contents into three data fields and creates an additional data field (ORDER_FULL_NAME) from the contents of the three data fields. Step 401b reads a data file called “FILE_B”, parses its contents into three data fields and creates an additional data field (ORDER_FULL_NAME) from the contents of the three data fields. Step 401c merges the results of steps 401a and 401b. Step 401d filters out any data records that do not have a particular value (“US”) in the first two characters of the ORDER_FULL_NAME data field.

[0057] According to some embodiments, the tool being executed may identify the contents of these portions of the program 400 as corresponding to operations that may be performed by components of a dataflow graph. In particular, the tool may recognize the contents of the portions of code labeled 401a and 401b as each performing a

transformation of a read data file. Further, the tool may recognize the contents of the portion of code labeled 401c as performing a join operation, and the contents of the portion of code labeled 401d as performing a filter operation. Moreover, the tool may identify control flow relationships of these steps by recognizing that step 401c utilizes the result of steps 401a and 401b (“MERGE DATASET_A DATASET_B”) and that step 401d utilizes the result of step 401c (“SET DATASET_C”). As a result, the components and flows shown in FIG. 4B may be identified. The tool may also generate suitable configuration data for components 406, 408, 410 and 412. As a final step, the tool may then produce a dataflow graph arranged as shown in FIG. 4B and that includes the generated configuration data.

[0058] FIG. 4C illustrates FIG. 4B with annotations associated with components 406, 408, 410 and 412 that show respective steps of the program 400 shown in FIG. 4A. Procedural language code that is equivalent to a component of the dataflow graph 450 is shown connected to said component in FIG. 4C with a gray zig-zag arrow. It will be appreciated, therefore, that the annotated sections of program 400 are not verbatim stored by or otherwise executed by respective components in dataflow graph 450. Rather, the components of the dataflow graph 450 perform functionality that is equivalent to the associated sections of the program.

[0059] FIG. 4D illustrates details of converting a first step of the program shown in FIG. 4A into a first component of the dataflow graph shown in FIGs. 4B-4C, according to some embodiments. To further illustrate the process of configuring a component by generating configuration data for the component by the tool, FIG. 4D shows step 401a from FIG. 4A, and illustrates how the resulting section of a dataflow graph may be generated.

[0060] In the example of FIG. 4D, a flow 405 is generated to connect the identified transform component 406 to the input file FILE_A, as a result of the data operation “INFILE FILE_A” in the section 401a of the program, which indicates that the program is to read FILE_A. In addition, the transform component 406 is configured to read a sequence of bytes from the input file as dictated by the program’s data operation “INPUT ORDER_NAME \$10 EXT \$4 CHANGE_FLAG \$1,” which indicates that ten bytes are to be read as a value for the data field ORDER_NAME, the next four bytes

are to be read as a value for the data field EXT, and the next byte is to be read as a value for the data field CHANGE_FLAG. The component 406 may be configured to continue to read bytes from the input FILE_A, assuming there are additional bytes; in this manner, each block of 15 bytes in the input file becomes a portion of successive data records.

[0061] In addition, the component 406 is configured with transform code 451 to generate an additional output data field by concatenating the three field values read from the input file with underscore characters as dictated by the data operation defining ORDER_FULL_NAME in the program section 401a. The component 406 may be configured to produce data records on flow 407 according to the illustrated output record format 452. The tool generating the dataflow graph may generate the record format 452 by interpreting the program section 401a using a grammar, to identify the types and lengths of the data fields being produced by this step of the program. In this case, the lengths of the ORDER_NAME, EXT and CHANGE_FLAG fields may be inferred from the manner in which they are read from the input file, and the length of the ORDER_FULL_NAME field may be inferred as a result of concatenating the 15 total characters of the three other fields with two underscore characters.

[0062] FIGs. 4E-4F illustrate details of converting a second step of the program shown in FIG. 4A into a second component of the dataflow graph shown in FIGs. 4B-4C, according to some embodiments. FIG. 4E reproduces the content of step 401c from FIG. 4A and depicts three different parsing operations of the step using a grammar 480.

[0063] In a first instance in FIG. 4E, the illustrated data operations of the program 481 are parsed using grammar 480 to identify these data operations as a step, namely step 401c described above. The tool may recognize, using the grammar 480, the data operations 481 as corresponding to the step based on the highlighted language that: a) identifies the data operations beginning with "DATA.." and ending with "...RUN;" as defining a step; and b) recognizes that the step depends on "DATASET_A" and "DATASET_B" and that this therefore implicates a particular control flow relationship between this step and other portions of the program.

[0064] In a second instance in FIG. 4E, the tool may parse the illustrated data operations of the program 482 (which may now have been identified as corresponding to a step 401c) using the grammar 480 to determine a type of dataflow graph component

that corresponds to the function of the step in the program. In the example of FIG. 4E, the highlighted keyword “MERGE” may be identified using the grammar as corresponding to a dataflow graph component of the “JOIN” type.

[0065] In a third instance in FIG. 4E, the tool may parse the illustrated data operations of the program 482 (which may now have been identified as corresponding to a step 401c) using the grammar 480 to determine how to configure an instance of the type of dataflow graph component selected in the second instance of parsing described above. In the example of FIG. 4E, the highlighted language “BY ORDER NAME;” may be identified using the grammar as indicating that the instance of the “JOIN” dataflow graph component should be configured to use a field called “ORDER NAME” as a key field for the join operation.

[0066] Accordingly, FIG. 4E provides one example of how a grammar 480 may produce these three different types of information regarding a portion of a program through parsing of that portion. While the parsing is described above in terms of three separate instances of parsing, it will be appreciated that the tool may utilize grammar 480 to parse the portion of the program in any number of ways that each parse any suitable parts of the portion of the program in any order, and that the above is provided merely as an illustrative process.

[0067] In view of the above-described parsing of step 401c, FIG. 4F illustrates how the resulting section of a dataflow graph may be generated. In the example of FIG. 4F, flows 407 and 409 provide data records having the illustrated input record formats from the two upstream transform components 406 and 408. In parsing the step 401c of the program, the tool may identify that a join component is appropriate as a result of the “MERGE” command in the program, and may further generate configuration data for the join component that specify a join key 462 of ORDER_NAME as a field to use to match up incoming records for the join. As a result of the join, the only records that may be output from component 410 along flow 411 may be those for which there were records with the same ORDER_NAME value present in both of the incoming sets of data records from the transform components.

[0068] FIG. 5 illustrates an example of a suitable computing system environment 500 on which the technology described herein may be implemented. The computing

system environment 500 is only one example of a suitable computing environment and is not intended to suggest any limitation as to the scope of use or functionality of the technology described herein. Neither should the computing environment 500 be interpreted as having any dependency or requirement relating to any one or combination of components illustrated in the exemplary operating environment 500.

[0069] The technology described herein is operational with numerous other general purpose or special purpose computing system environments or configurations. Examples of well-known computing systems, environments, and/or configurations that may be suitable for use with the technology described herein include, but are not limited to, personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, set top boxes, programmable consumer electronics, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like.

[0070] The computing environment may execute computer-executable instructions, such as program modules. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The technology described herein may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote computer storage media including memory storage devices.

[0071] With reference to FIG. 5, an exemplary system for implementing the technology described herein includes a general purpose computing device in the form of a computer 510. Components of computer 510 may include, but are not limited to, a processing unit 520, a system memory 530, and a system bus 521 that couples various system components including the system memory to the processing unit 520. The system bus 521 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. By way of example, and not limitation, such architectures include Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced

ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus also known as Mezzanine bus.

[0072] Computer 510 typically includes a variety of computer readable media. Computer readable media can be any available media that can be accessed by computer 510 and includes both volatile and nonvolatile media, removable and non-removable media. By way of example, and not limitation, computer readable media may comprise computer storage media and communication media. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by computer 510. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term “modulated data signal” means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared and other wireless media. Combinations of the any of the above should also be included within the scope of computer readable media.

[0073] The system memory 530 includes computer storage media in the form of volatile and/or nonvolatile memory such as read only memory (ROM) 531 and random access memory (RAM) 532. A basic input/output system 533 (BIOS), containing the basic routines that help to transfer information between elements within computer 510, such as during start-up, is typically stored in ROM 531. RAM 532 typically contains data and/or program modules that are immediately accessible to and/or presently being operated on by processing unit 520. By way of example, and not limitation, FIG. 5

illustrates operating system 534, application programs 535, other program modules 536, and program data 537.

[0074] The computer 510 may also include other removable/non-removable, volatile/nonvolatile computer storage media. By way of example only, FIG. 5 illustrates a hard disk drive 541 that reads from or writes to non-removable, nonvolatile magnetic media, a flash drive 551 that reads from or writes to a removable, nonvolatile memory 552 such as flash memory, and an optical disk drive 555 that reads from or writes to a removable, nonvolatile optical disk 556 such as a CD ROM or other optical media. Other removable/non-removable, volatile/nonvolatile computer storage media that can be used in the exemplary operating environment include, but are not limited to, magnetic tape cassettes, flash memory cards, digital versatile disks, digital video tape, solid state RAM, solid state ROM, and the like. The hard disk drive 541 is typically connected to the system bus 521 through a non-removable memory interface such as interface 540, and magnetic disk drive 551 and optical disk drive 555 are typically connected to the system bus 521 by a removable memory interface, such as interface 550.

[0075] The drives and their associated computer storage media discussed above and illustrated in FIG. 5, provide storage of computer readable instructions, data structures, program modules and other data for the computer 510. In FIG. 5, for example, hard disk drive 541 is illustrated as storing operating system 544, application programs 545, other program modules 546, and program data 547. Note that these components can either be the same as or different from operating system 534, application programs 535, other program modules 536, and program data 537. Operating system 544, application programs 545, other program modules 546, and program data 547 are given different numbers here to illustrate that, at a minimum, they are different copies. A user may enter commands and information into the computer 510 through input devices such as a keyboard 562 and pointing device 561, commonly referred to as a mouse, trackball or touch pad. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 520 through a user input interface 560 that is coupled to the system bus, but may be connected by other interface and bus structures, such as a parallel port, game port or a universal serial bus (USB). A monitor 591 or

other type of display device is also connected to the system bus 521 via an interface, such as a video interface 590. In addition to the monitor, computers may also include other peripheral output devices such as speakers 597 and printer 596, which may be connected through an output peripheral interface 595.

[0076] The computer 510 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 580. The remote computer 580 may be a personal computer, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the computer 510, although only a memory storage device 581 has been illustrated in FIG. 5. The logical connections depicted in FIG. 5 include a local area network (LAN) 571 and a wide area network (WAN) 573, but may also include other networks. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

[0077] When used in a LAN networking environment, the computer 510 is connected to the LAN 571 through a network interface or adapter 570. When used in a WAN networking environment, the computer 510 typically includes a modem 572 or other means for establishing communications over the WAN 573, such as the Internet. The modem 572, which may be internal or external, may be connected to the system bus 521 via the user input interface 560, or other appropriate mechanism. In a networked environment, program modules depicted relative to the computer 510, or portions thereof, may be stored in the remote memory storage device. By way of example, and not limitation, FIG. 5 illustrates remote application programs 585 as residing on memory device 581. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

[0078] Having thus described several aspects of at least one embodiment of this invention, it is to be appreciated that various alterations, modifications, and improvements will readily occur to those skilled in the art.

[0079] Such alterations, modifications, and improvements are intended to be part of this disclosure, and are intended to be within the spirit and scope of the invention. Further, though advantages of the present invention are indicated, it should be

appreciated that not every embodiment of the technology described herein will include every described advantage. Some embodiments may not implement any features described as advantageous herein and in some instances one or more of the described features may be implemented to achieve further embodiments. Accordingly, the foregoing description and drawings are by way of example only.

[0080] The above-described embodiments of the technology described herein can be implemented in any of numerous ways. For example, the embodiments may be implemented using hardware, software or a combination thereof. When implemented in software, the software code can be executed on any suitable processor or collection of processors, whether provided in a single computer or distributed among multiple computers. Such processors may be implemented as integrated circuits, with one or more processors in an integrated circuit component, including commercially available integrated circuit components known in the art by names such as CPU chips, GPU chips, microprocessor, microcontroller, or co-processor. Alternatively, a processor may be implemented in custom circuitry, such as an ASIC, or semicustom circuitry resulting from configuring a programmable logic device. As yet a further alternative, a processor may be a portion of a larger circuit or semiconductor device, whether commercially available, semi-custom or custom. As a specific example, some commercially available microprocessors have multiple cores such that one or a subset of those cores may constitute a processor. However, a processor may be implemented using circuitry in any suitable format.

[0081] Further, it should be appreciated that a computer may be embodied in any of a number of forms, such as a rack-mounted computer, a desktop computer, a laptop computer, or a tablet computer. Additionally, a computer may be embedded in a device not generally regarded as a computer but with suitable processing capabilities, including a Personal Digital Assistant (PDA), a smart phone or any other suitable portable or fixed electronic device.

[0082] Also, a computer may have one or more input and output devices. These devices can be used, among other things, to present a user interface. Examples of output devices that can be used to provide a user interface include printers or display screens for visual presentation of output and speakers or other sound generating devices for audible

presentation of output. Examples of input devices that can be used for a user interface include keyboards, and pointing devices, such as mice, touch pads, and digitizing tablets. As another example, a computer may receive input information through speech recognition or in other audible format.

[0083] Such computers may be interconnected by one or more networks in any suitable form, including as a local area network or a wide area network, such as an enterprise network or the Internet. Such networks may be based on any suitable technology and may operate according to any suitable protocol and may include wireless networks, wired networks or fiber optic networks.

[0084] Also, the various methods or processes outlined herein may be coded as software that is executable on one or more processors that employ any one of a variety of operating systems or platforms. Additionally, such software may be written using any of a number of suitable programming languages and/or programming or scripting tools, and also may be compiled as executable machine language code or intermediate code that is executed on a framework or virtual machine.

[0085] In this respect, the invention may be embodied as a computer readable storage medium (or multiple computer readable media) (e.g., a computer memory, one or more floppy discs, compact discs (CD), optical discs, digital video disks (DVD), magnetic tapes, flash memories, circuit configurations in Field Programmable Gate Arrays or other semiconductor devices, or other tangible computer storage medium) encoded with one or more programs that, when executed on one or more computers or other processors, perform methods that implement the various embodiments of the invention discussed above. As is apparent from the foregoing examples, a computer readable storage medium may retain information for a sufficient time to provide computer-executable instructions in a non-transitory form. Such a computer readable storage medium or media can be transportable, such that the program or programs stored thereon can be loaded onto one or more different computers or other processors to implement various aspects of the present invention as discussed above. As used herein, the term "computer-readable storage medium" encompasses only a non-transitory computer-readable medium that can be considered to be a manufacture (i.e., article of manufacture) or a machine. Alternatively or additionally, the invention may be

embodied as a computer readable medium other than a computer-readable storage medium, such as a propagating signal.

[0086] The terms “program” or “software,” when used herein, are used in a generic sense to refer to any type of computer code or set of computer-executable instructions that can be employed to program a computer or other processor to implement various aspects of the present invention as discussed above. Additionally, it should be appreciated that according to one aspect of this embodiment, one or more computer programs that when executed perform methods of the present invention need not reside on a single computer or processor, but may be distributed in a modular fashion amongst a number of different computers or processors to implement various aspects of the present invention.

[0087] Computer-executable instructions may be in many forms, such as program modules, executed by one or more computers or other devices. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Typically the functionality of the program modules may be combined or distributed as desired in various embodiments.

[0088] Also, data structures may be stored in computer-readable media in any suitable form. For simplicity of illustration, data structures may be shown to have fields that are related through location in the data structure. Such relationships may likewise be achieved by assigning storage for the fields with locations in a computer-readable medium that conveys relationship between the fields. However, any suitable mechanism may be used to establish a relationship between information in fields of a data structure, including through the use of pointers, tags or other mechanisms that establish relationship between data elements.

[0089] Various aspects of the present invention may be used alone, in combination, or in a variety of arrangements not specifically discussed in the embodiments described in the foregoing and is therefore not limited in its application to the details and arrangement of components set forth in the foregoing description or illustrated in the drawings. For example, aspects described in one embodiment may be combined in any manner with aspects described in other embodiments.

[0090] Also, the invention may be embodied as a method, of which an example has been provided. The acts performed as part of the method may be ordered in any suitable way. Accordingly, embodiments may be constructed in which acts are performed in an order different than illustrated, which may include performing some acts simultaneously, even though shown as sequential acts in illustrative embodiments.

[0091] Further, some actions are described as taken by a “user.” It should be appreciated that a “user” need not be a single individual, and that in some embodiments, actions attributable to a “user” may be performed by a team of individuals and/or an individual in combination with computer-assisted tools or other mechanisms.

[0092] Use of ordinal terms such as “first,” “second,” “third,” etc., in the claims to modify a claim element does not by itself connote any priority, precedence, or order of one claim element over another or the temporal order in which acts of a method are performed, but are used merely as labels to distinguish one claim element having a certain name from another element having a same name (but for use of the ordinal term) to distinguish the claim elements.

[0093] Also, the phraseology and terminology used herein is for the purpose of description and should not be regarded as limiting. The use of “including,” “comprising,” or “having,” “containing,” “involving,” and variations thereof herein, is meant to encompass the items listed thereafter and equivalents thereof as well as additional items.