Supplement to "FLOWER: Representing Flow in ER Diagrams to Understand Data Preprocessing": Algorithm

1 Algorithm

In this supplement, we suggest a general algorithm suitable for flow analysis of source code. This algorithm is compatible with the primary languages for data analysis today, which tend to be imperative and object-oriented, but can be modified to accommodate other paradigms.

Conceptually, this algorithm transforms imperative statements into stateful operations that may be parsed into directed provenance graphs. These graphs can then be easily analyzed and transformed for further presentation, such as with FLOWER. The algorithm consists of *Flow* and *State* objects, from which relationships may be observed:

Given: *States*, where a State is an object containing data representing the conceptual state of an entity in the ER model at a specific phase within the pipeline.

$$S := \{name, operation, parents, writes, reads\}$$

Where *name* is the text of the code statement producing it, *operation* is the name of the operation (such as a function call) that produces it, *parents* is a set of the parent states the State is derived from, $reads := \{R_1, R_2 ...\}$ is a set of resources (such as file name strings) that may would be read in its construction, and $writes := \{W_1, W_2 ...\}$ is a set of resources that it may be written to. A State can only be derived from a read operation or another State, so there will not exist any state that does not have an external resource in its ancestry.

Given: A *Flow* F, containing a set of $statements := \{T_1, T_2 ...\}$ in a given imperative language, a set of $initial := \{S_1, S_2 ...\}$ States, and initially empty sets of reads and writes

$$F := \{initial, statements, reads := \{\}, writes := \{\}\}$$

Where a Flow can be used to describe anywhere a sequence of statements may exist with an optional set of initial states. In Python or R, scopes like modules and function calls are Flows, where a function's arguments form its initial states. For object methods, the owning object itself may be an initial state, as might be (separately) its attributes. The following operations are available on a State in a Flow:

- Read: A State may be constructed from one or more read operations in a single statement, which are put in its list of reads. This is the only type of State which might have no parent States. The operation for constructing this state will be a *read*.

- Update: States cannot be modified directly by operations within a Flow; Instead, each modification to a state results in a new State whose parents are the modified state along with any other states included in the modification operation. The operation for this state will be the operation used in the update. The new State replaces the old State within the controlling Flow.
- Assign: A State may be Assigned to another State label while retaining its own, without modification. Anywhere the original or assigned label is updated, the state referred to by both is updated and replaced according to the update operation above. This includes States belonging to other Flows: If a State in a Flow's initial state is assigned or updated, the State is updated and replaced there as well as in the current Flow. For languages with copy operations, copying is treated as an update, not an assign.
- Write: A State may be written out to one or more resources. These resources are added to the *write* list in the State.

We also define *opaque* and *transparent* operations. An opaque operation is one that does not have behavior known to the algorithm. A transparent operation is one that does have known behavior, such as having the source available or an operation otherwise known by the algorithm ahead of time to have defined behavior when operating on a State. Reads and writes are necessarily transparent as they must be known to the algorithm to be defined as read or write.

A Flow must first have its statements normalized into a list of *assignment expressions* of the form

$$Expr := \{\{S_1, S_2...\}, operation, \{E_1, E_2...\}\}$$

where S is one or more labels of a state to be updated (or created, if the state is not already present in the Flow), *operation* is the name of the operation used in this statement, and E is one or more expressions that may be parent States or non-State values.

The algorithm for generating the graph of a Flow iterates over each statement, first preprocessing it into an assignment expression and then processing it to update the flow's States.

Statements are preprocessed given the following rules:

- Object methods are transformed into function calls, with the owner object as an initial State for the method if the owner is a State.

$$\{\{S_3\}, S_1.foo, \{S_2\}\} \rightarrow \{\{S_3\}, S_1.foo, \{S_1, S_2\}\}$$

- Non-assignment opaque functions, following imperative convention, are assumed to augment the first argument.

$$\{\{\}, op, \{S_1, S_2, S_3\}\} \rightarrow \{\{S_1\}, op, \{S_1, S_2, S_3\}\}$$

- Nested expressions containing states are decomposed into temporary states and analyzed prior to the main expression.

$$\{\{S_4\}, foo, \{bar(baz(S_1, S_2), S_3\}\} \rightarrow \{\{T_1\}, baz, \{S_1, S_2\}\}; \{\{T_2\}, bar, \{T_1\}\};$$
$$\{\{S_3\}, foo, \{T_2, S_3\}\}$$

Next, the expression $\{NewStates, op, SourceStates\}$ is processed.

– If *op* is a transparent, non-read, non-write function, create a new Flow using the arguments to *op* as the initial States. If op is an object method, include any known states in its attributes as initial states. Recursively analyze the nested Flow (skipping recursive function calls to avoid the halting problem), then replace the right side of the expression with any states it returns.
– If *op* is "read": Create NewStates and add them to F. The resource read is also recorded in F.
– Else if *op* is "write": Record resource written to in F.
– Else if a state exists on the left or right of the expression, update NewStates into F, creating or replacing with the new States as needed. Non-state values may be recorded in the new State's *operation* for purposes such as attribute inference.

Throughout this process, a tool may also associate States with types and attributes (such as keys) inferred from transparent functions in each statement.

Because we only care about general flow and cannot necessarily determine runtime behavior, control flow structures such as conditionals and loops are treated as inline statements. Assuming code correctness, all branches of the code are anticipated to be reached at some point, so we assume that State transformations in a branch should be captured regardless of whether conditions would cause it to be so. This approach gives the algorithm linear time complexity O(n), where n is the sum of statements in the Flow and calls to nested Flows. An extension to the algorithm may be appended that, when a state is modified within a conditional, it is given a new identity associated with that branch. This strategy can then distinguish branching behavior, if desired.

At this point, F will contain the final State nodes which form a graph pointing backward to the root read ancestor. On a macro level the inputs and outputs may simply be used to connect the Flow analyzed, such as a script modifying some files, to the rest of the entities in a FLOWER diagram. For a more detailed view, a tool might also compress the State graph into a number of "interesting" nodes (discussed in Section **??**) to display as sub-entities. This detailed view may lead to a better understanding of a system, as in pipelines where multiple files that are input and output may not necessarily be related inside of the pipeline itself.

This algorithm supports the ability to be extended for known systems by the notion of *patterns*. A tool may be given a set of patterns to define its transparent functions, reads, and writes. When a new system of functions must be accounted for, be it a new library or a project-specific behavior, additional patterns may be added to the tool using what is known about the new system.