Notes on data structures within DynComp for C

For a good understanding of the motivation, general algorithm, and an implementation overview you should read Philip Guo’s “A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs”, MIT Master’s Thesis, May 2006; especially Chapter 4: “DynComp: A Tool for Dynamic Inference of Abstract Types”. Similar background information and a few more implementation details appear in Robert Rudd’s “An Improved Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs”, MIT Master’s Thesis, Jan. 2010: especially Chapter 3: “Dyncomp, a dynamic abstract type inference tool for C and C++ programs”. Copies of both of these papers are located <WHERE? I have pdfs I should add to repo>.

It is important to understand the difference between Value and Variable tags and their data structures (more details below).

(It is also important when reading code to recognize the subtle naming difference of ‘val’ vs. ‘var’.)

A good overview of the use of the union-find (UF) data structure to manipulate disjoint sets is <https://en.wikipedia.org/wiki/Disjoint-set_data_structure>. Dyncomp uses a Disjoint-set forests data structure (as described in the article). Each node in the forest is an uf\_object that contains three items: a parent pointer, an unsigned integer rank and the actual set member, an unsigned integer referred to as a ‘tag’. As noted in the article, an element of each set is selected to represent the set; DynComp uses the term ‘leader’ for this representative. It is equal to the tag value of the root of the set in question. DynComp uses both the union-by-rank and path-compression optimizations.

In DynComp, the union-find routines are located in kvasir/union\_find.[ch] and are shared by both the Value tag data structures and the Variable tag data structures. (details below) Given a tag ‘X’, the union-find routine uf\_find is used to locate the leader of the set containing ‘X’. As elements are added to a set, the leader may change.

Value Tags

There is only one set of value tags. The DynComp runtime maintains a counter for the next available value tag. It is incremented each time a value is generated during the user’s program execution. When the DynComp runtime detects a program value being generated, it creates a new singleton set in the UF object data structure that contains the new value tag associated with that program value. A value tag uniquely identifies each and every value generated during a program’s execution. If at some point during the program the value ‘6’ is generated it might be assigned the value tag of ‘1234’. (The numeric value of the value tag has no relationship to the value it represents.) If at some later point in the execution another value ‘6’ is generated it will be assigned a different tag, perhaps ‘5678’. As there is a value tag for every value ever stored in any program location, the number of value tags can get quite large; on the order of 10,000,000 is not uncommon. (There are value tags generated that are never used, but that is not germane to this discussion. Also, I have not described how value tags get assigned to locations. The vast majority of the code in Kvasir and Dyncomp is required for this task. It is done via instrumentation of the user’s code – we will ignore this process for this discussion.)

Value tags are stored in the primary\_tag\_map[], indexed by address.

Value union-find data is stored in the primary\_val\_uf\_object\_map[], indexed by value tag.

Neither of these data structures should be directly referenced – various getters and setters are located in kavsir/dyncomp\_main.[ch].

At any point during the execution of the user’s program, the address of a program memory location may be used to as an index into the primary\_tag\_map to look up the value tag that represents the current program value stored in that memory location. Calling uf\_find with this value tag will return the UF object (located in primary\_val\_uf\_object\_map) that is the leader of the value set containing that tag. If at a later point in the program’s execution that memory location is assigned a different program value, the value tag for that new value is stored in the primary\_tag\_map. Note that the UF object containing the previous value tag for that memory location is not discarded. It might be a live value still accessible from another variable or it might represent a value that occurred during program execution and is a member of a set of values that were assigned to a program variable. (This is explained in more detail in the Variable Tags section below.) If a UF object is not referenced from a memory location or from a variable set, it may be garbage collected if additional tag values are needed.

The principal purpose of the DynComp dynamic comparability analysis tool is to perform dynamic type inference to group variables into comparability sets. All the variables in a comparability set belong to the same “abstract type” of data that the programmer likely intended to represent. DynComp uses a union-find data structure (the primary\_val\_uf\_object\_map) to capture the interactions between program locations. Let’s consider a simple example: the expression ‘A + B’. The arithmetic operator ‘+’ implies that A and B should be placed in the same union-find set. When the DynComp runtime detects such an interaction, it locates the UF object associated with the current value of A, then uses that to find the UF object that represents the leader of the value set containing A. It does the same process for B. Finally, it places A and B in the same comparability set by calling the uf\_union routine with the two UF leader items as arguments. The uf\_union routine will combine the two value sets into one and, in the process, choose one of the two arguments to be the new leader and return that item to represent the value of the expression.

Variable Tags

There are multiple sets of variable tags. There is a separate set associated with the entry and exit of every function in the user program. Each of these locations is referred to as a ‘program point’ or ppt. Usually, the entry sets are not used, so this document will just refer to a function’s variable tags without identification of entry/exit. Not all variables are tracked: only parameters, globals, and the pseudo-variable ‘return’ (used to represent the return value of a function). So while the memory location containing a local variable will have a current value tag to be used in the calculation of expression value tags, the local variable and its associated value tag(s) will never be included in a function’s variable sets.

Each time the user’s program enters or exits a function, DynComp updates its variable union-find sets based on the union-find sets of the current values of its variables. The aim is to produce a union-find set that contains all the different values observed for each variable. At the end of the user’s program execution, any variables with the same variable tag are members of the same comparability set.

The file kvasir/kvasir\_main.h contains the definitions of the various union-find data structures used to track variable interactions. Comments in this file give a good description of each of the data items. A brief recap:

var\_tags – variable analogue of primary\_tag\_map. An array indexed by variable number, giving the variable’s tag.

var\_uf\_map – variable analogue of primary\_val\_uf\_object\_map. A hash table with a variable tag as its key, giving the leader of the tag’s union-find set.

At each program point, for each variable *v* being tracked, DynComp finds the current value tag assigned to the memory location of *v*. It then uses uf\_find to obtain the leader of the associated value set. A potential source of confusion at this point is that DynComp then copies this *value* tag to be the *variable* tag for the current observation of *v*. DynComp then searches var\_uf\_map for a UF object with this variable tag. If not found, it creates a new singleton object in the map with this variable tag. The next step is to call uf\_union with the UF object just found (or created) and with the UF object associated with *v*’s variable tag in the var\_tags array. (Note that if the variable’s current value matches one of the previously observed values, the uf\_union would result in no changes.) Finally, the variable tag returned from uf\_union is assigned back to *v*’s entry in var\_tags.

The algorithm described in the previous paragraph leads to the following state: The var\_tags array for a program point contains a variable tag for each variable that may be observed at that program point. This variable tag is the leader (as of the last observation) of a set of variable tags located in the var\_uf\_map. Each member of this set represents a distinct set of values. At least one element of the set was observed (for a particular variable or variables) at some program point. The value set represented by a particular member of the variable set may be located by treating the variable tag value as if it were a value tag and looking up the value set in the primary\_val\_uf\_object\_map.

During program execution, as value interactions occur, the leader of a value set might change. This can occur if the value currently marked as the leader of a value set appears in an expression (post the last ppt observation) and some other item in that expression is chosen to be the new leader of the updated value set. Hence, one of the tasks during a program point observation is to check to see if the value tag associated with a variable set entry is no longer the leader. As there is an association between the leader of a value set and the variable tag that represents that set, we must update the variable set to reflect this change in the value set leader.



The diagram above shows a variable with two elements in its variable set. This means that over the course of (perhaps) several calls to this function, this particular variable has been observed with two distinct values. (The blue dotted lines represent the how of tag for the leader of a Value set is reused as a member of a Variable set.)

Note:

It is important to point out a problem with the DynComp algorithm as described in the two papers noted at the beginning of this discussion. As part of the observation process described in the previous paragraph, the papers claim that you only need to check for changes in the value set associated with the leader of the variable set for a particular variable. This is incorrect. Remember, there is nothing special about the leader of a set, it is just a shorthand token for referring to the entire set. In fact, you must check every member of the variable set to see if any of its associated value sets have changed.