Visualizing Memory Graphs

Thomas Zimmermann[1] and Andreas Zeller[2]

[1] Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, 94032 Passau
zimmerth@fmi.uni-passau.de
[2] Universität des Saarlandes, FR Informatik
Lehrstuhl für Softwaretechnik
Postfach 15 11 50, 66041 Saarbrücken
zeller@cs.uni-sb.de

Abstract. To understand the dynamics of a running program, it is often useful to examine its state at specific moments during its execution. We present *memory graphs* as a means to capture and explore program states. A memory graph gives a comprehensive view of all data structures of a program; data items are related by operations like dereferencing, indexing or member access. Although memory graphs are typically too large to be visualized as a whole, one can easily focus on specific aspects using well-known graph operations. For instance, a greatest common subgraph visualizes commonalities and differences between program states.
Keywords: program understanding, debugging aids, diagnostics, data types and structures, graphs

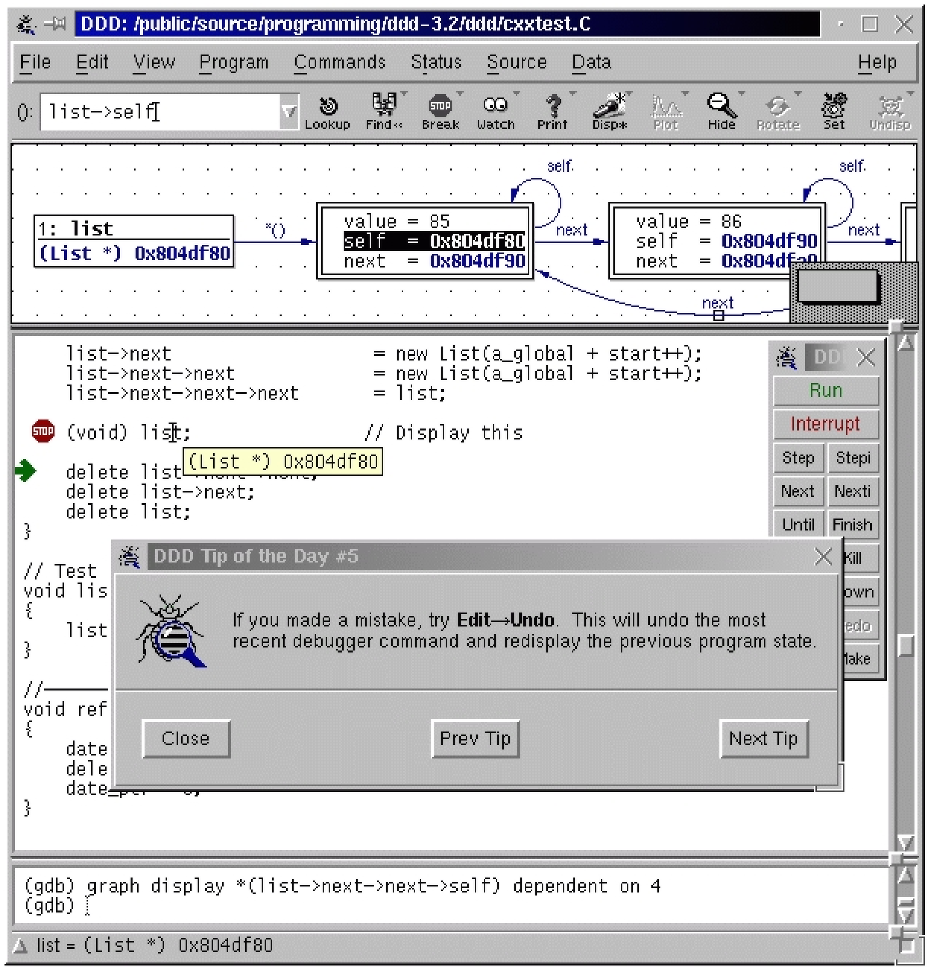
1 A Structured View of Memory

Exploring the state of a program, to view its variables, values, and current execution position, is a typical task in debugging programs. Today's interactive debuggers allow accessing the values of arbitrary variables and printing their values. Typically, values are shown as *texts*. Here's an example output from the GNU debugger GDB:

```
(gdb) print *tree
*tree = {value = 7, _name = 0x8049e88 "Ada", _left = 0x804d7d8,
  _right = 0x0, left_thread = false, right_thread = false,
  date = {day_of_week = Thu, day = 1, month = 1, year = 1970,
   _vptr. = 0x8049f78 ⟨Date virtual table⟩}, static shared = 4711}
(gdb) _
```

Fig. 1. Printing textual data with GDB

Although modern debuggers offer graphical user interfaces instead of GDB's command line, data values are still shown as text. This is useful in the most cases, but fails badly as soon as references and pointers come into play. Consider Figure 1, for example: where does `tree->_left` point to? Of course, one can simply print the dereferenced value. However, a user will never notice if two pointers point to the same address—except by thoroughly checking and comparing pointer values.



```
    list->next                = new List(a_global + start++);
    list->next->next          = new List(a_global + start++);
    list->next->next->next    = list;

    (void) list;             // Display this

    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//
void ref
{
    date
    dele
    date
}
```

(List *) 0x804df80

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

list = (List *) 0x804df80

Fig. 2. The GNU DDD debugger

An alternative to accessing memory in a name/value fashion is to model memory as a *graph*. Each value in memory becomes a vertex, and references between values (i.e. pointers) become edges between these vertices. This view was first explored in the GNU DDD debugger front-end [6], shown in Figure 2.

In DDD, displayed pointer values are dereferenced by a simple mouse click, allowing to unfold arbitrary data structures interactively. DDD automatically detects if multiple pointers pointed to the same address and adjusts its display accordingly. DDD has a major drawback, though: each and every pointer of a data structure must be dereferenced manually. While this allows the programmer to set a focus on specific structures, it is tedious to access, say, the 100th element in a linked list.

To overcome these limitations, we propose *memory graphs* as a basis for accessing and visualizing memory contents. A memory graph captures the program state as a

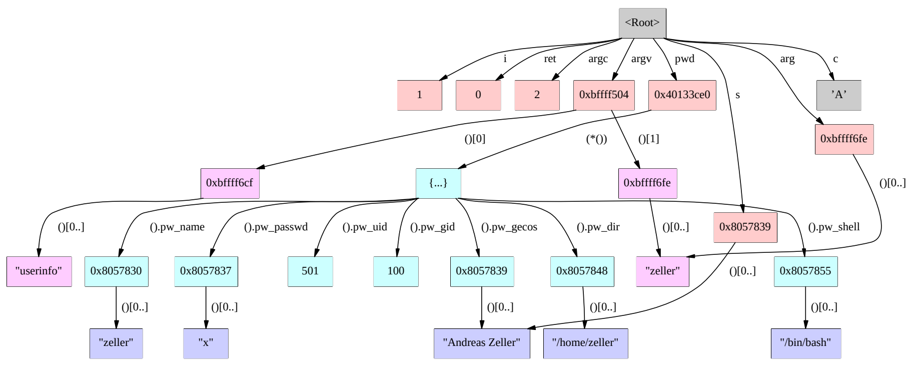


Fig. 3. A simple memory graph

graph, very much like DDD does; however, it is extracted automatically from a program. Since a memory graph encompasses the entire program state, it can be used to answer questions like:

- Are there any pointers pointing to this address?
- How many elements does this data structure have?
- Is this allocated memory block reachable from within my module?
- Did this tree change during the last function call?

In this paper, we show how to extract and visualize memory graphs, sketching the capabilities of future debugging tools.

2 An Example Graph

As a simple example of a memory graph, consider Figure 3. The memory graph shows the state of a program named *userinfo*[1]. *userinfo* takes a UNIX user name as argument and shows the user's full name and e-mail address.

In our case, *userinfo* was invoked with *zeller* as argument. You can see this by following the edge named "*argv*" (the program's arguments). Dereferencing the first element of *argv* (following the edge labeled "()[1]", we find *argv*[1]—the argument "*zeller*".[2]

To fetch the full name, *userinfo* accesses the user database via its *pwd* variable. By dereferencing the link named *pwd* from the top, you find a node named "{. . . }". This is the record *pwd* points to (i.e. $*pwd$). Further descendants include the user id, the group id, the full name of the user and the UNIX user name "*zeller*" as well). You see that the two strings *argv*[1] and $pwd \rightarrow pw\_name$ are disjoint; however, *arg* points to the same string as *argv*[1]. Obviously, such a graph drawing is much more valuable than, say, a table of variables and values.

If you are interested in a formal definition of memory graphs, see Figure 4.

[1] *userinfo* is part of the GNU DDD distribution.

[2] In case you are reading the color version of this document, the different vertex colors indicate different storage areas. Arguments are shown in pink, stack variables in red, heap variables in blue, and static variables in cyan. The character *c* is grey because it resides in a register.

The formal structure of memory graphs

Let $G = (V, E, \mathit{root})$ be a memory graph containing a set V of vertices, a set E of edges, and a dedicated vertex *root*:

Vertices. Each vertex $v \in V$ has the form $v = (\mathit{val}, \mathit{tp}, \mathit{addr})$, standing for a value *val* of type *tp* at memory address *addr*.

As an example, the C declaration

```
int i = 42;
```

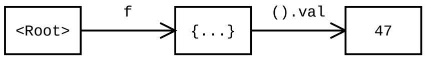
results in a vertex $v_{\texttt{i}} = (42, \text{int}, \mathit{0x1234})$, where *0x1234* is the (hypothetical) memory address of `i`.

Edges. Each edge $e \in E$ has the form $e = (v_1, v_2, \mathit{op})$, where $v_1, v_2 \in V$ are the related vertices. The operation *op* is used in constructing the expression of a vertex (see below).

As an example, the C declaration of the record ("struct") `f`,

```
struct foo { int val; } f = {47};
```

results in two vertices $v_f = (\{\ldots\}, \text{struct foo}, \mathit{0x5678})$ and $v_{f.val} = (47, \text{int}, \mathit{0x9abc})$, as well as an edge $e_{f.val} = (v_f, v_{f.val}, \mathit{op}_{f.val})$ from v_f to $v_{f.val}$:



Root. A memory graph contains a dedicated vertex $\mathit{root} \in V$ that references all base variables of the program. Each vertex in the memory graph is accessible from root.

In the previous examples, `i` and `f` are base variables; thus, the graph contains the edges $e_i = (\mathit{root}, v_i, \mathit{op}_i)$ and $e_f = (\mathit{root}, v_f, \mathit{op}_f)$.

Operations. *Edge operations* construct the name of descendants from their parent's name.

In an edge $e = (v_1, v_2, \mathit{op})$, each operation *op* is a function that takes the expression of v_1 to construct the expression of v_2. We denote functions by $\lambda x.B$—a function that has a formal parameter x and a body B. In our examples, B is simply a string containing x; applying the function returns B where x is replaced by the function argument.

Operations on edges leading from *root* to base variables initially set the name; so $\mathit{op}_i = \lambda x.\texttt{"i"}$ and $\mathit{op}_f = \lambda x.\texttt{"f"}$ hold.

Deeper vertices are constructed based on the name of their parents. For instance, $\mathit{op}_{f.val} = \lambda x.\texttt{"}x\texttt{.val"}$ holds, meaning that to access the name of the descendant, one must append `".val"` to the name of its parent.

In our graph visualizations, the operation body is shown as edge label, with the formal parameter replaced by `"()"` (that is, we use *op*(`"()"`) as label). This is reflected in the figure above.

Names. The following function *name* constructs a name for a vertex v using the operations on the path from v to the root vertex. As there can be several parents (and thus several names), we non-deterministically choose a parent v' of v along with the associated operation *op*:

$$\mathit{name}(v) = \begin{cases} \mathit{op}\big(\mathit{name}(v')\big) \text{ for some } (v', v, \mathit{op}) \in E & \text{if } \exists (v', v, \mathit{op}) \in E \\ \texttt{""} & \text{otherwise (root vertex)} \end{cases}$$

As an example, see how a name for $v_{f.val}$ is found: $\mathit{name}(v_{f.val}) = \mathit{op}_{f.val}(\mathit{name}(v_f)) = \mathit{op}_{f.val}(\mathit{op}_f(\texttt{""})) = \mathit{op}_{f.val}(\texttt{"f"}) = \texttt{"f.val"}$

For details on the construction of memory graphs from data structures, see Figure 7.

Fig. 4. The structure of memory graphs

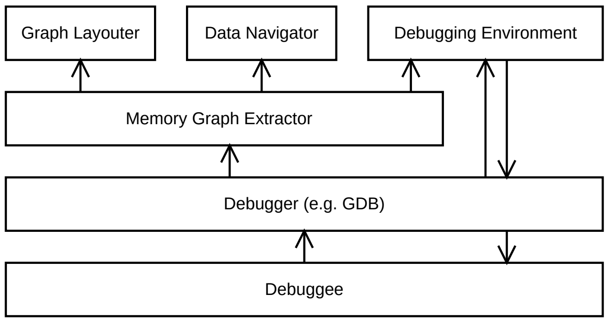


Fig. 5. Memory graphs within a debugging environment

3 Obtaining Memory Graphs

How does one obtain a memory graph? Figure 5 gives a rough sketch. At the bottom is the debuggee, the program to be examined. Its state is accessed via a standard debugger such as GDB. The memory graph extractor queries GDB for variable names, types, sizes, and values. Since GDB is controlled via the command line, the dialogue between memory graph extractor and GDB is actually human-readable, as shown in Figure 6 (bold face stands for GDB commands as generated by the memory graph extractor).

```
...
(gdb) output &(pwd)
(passwd **) 0xbffff478
(gdb) output sizeof(pwd)
4
(gdb) output *(pwd)
{
  pw_name = 0x8057830 "zeller",
  pw_passwd = 0x8057837 "x",
  pw_uid = 501,
  pw_gid = 100,
  pw_gecos = 0x8057839 "Andreas Zeller",
  pw_dir = 0x8057848 "/home/zeller",
  pw_shell = 0x8057855 "/bin/bash"
}
```

```
(gdb) set variable $v17 = pwd
(gdb) output &((*$v17).pw_name)
(char **) 0x40133ce0
(gdb) output sizeof((*$v17).pw_name)
4
(gdb) output (*$v17).pw_name
0x8057830 "zeller"
(gdb) output strlen((*$v17).pw_name) + 1
7
(gdb) output &((*$v17).pw_passwd)
(char **) 0x40133ce4
(gdb) output sizeof((*$v17).pw_passwd)
4
(gdb)
...
```

Fig. 6. Dialogue between memory graph extractor and GDB

You can see how the memory graph extractor queries GDB for the address and size of the *pwd* variable, then, having found it is a pointer, queries the object pointed to by dereferencing *pwd*. The object *pwd* points to is a C struct (a record), so the memory graph extractor goes on querying the addresses, sizes and values of the individual members. Note the usage of an internal GDB variable $\$v_{17}$ here; this is done to avoid the transmission of long expression names (such that we can use, say, $\$v_n \rightarrow value$ instead of $list \rightarrow next \rightarrow next \rightarrow next \rightarrow \cdots \rightarrow value$)

Once the entire graph is extracted, it can be made available for the programmer to display or examine; it can also be shown in a debugging environment where additional manipulations become available.

The formal details of obtaining memory graphs are listed in Figure 7; special caveats about C programs are given in Figure 8.

Unfolding data structures

To obtain a memory graph $G = (V, E, root)$, as formalized in Figure 4, we use the following scheme:

1. Let $unfold(parent, op, G)$ be a procedure (sketched below) that takes the name of a parent expression *parent* and an operation *op* and unfolds the element $op(parent)$, adding new edges and vertices to the memory graph G.
2. Initialize $V = \{root\}$ and $E = \emptyset$.
3. For each base variable *name* in the program, invoke $unfold(root, \lambda x.\texttt{"}name\texttt{"})$.

The *unfold* procedure works as follows. Let $(V, E, root) = G$ be the members of G, let $expr = op(parent)$ be the expression to unfold, let *tp* be the type of *expr*, and let *addr* be its address. The unfolding then depends on the structure of *expr*:

Aliases. If V already has a vertex v' at the same address and with the same type (formally, $\exists v' = (val', tp', addr') \in V \cdot tp = tp' \wedge addr = addr'$), do not unfold *expr* again; however, insert an edge $(parent, v', op)$ to the existing vertex.

As an example, consider the C statements:

```
struct foo f; int *p1; int *p2; p1 = p2 = &f;
```

If `f` has already been unfolded, we do not need to unfold its aliases `*p1` and `*p2`. However, we insert edges from `p1` and `p2` to `f`.

Records. Otherwise, if *expr* is a record containing n members $m_1, m_2, \ldots, m_n$, add a vertex $v = (\{\ldots\}, tp, addr)$ to V, and an edge $(parent, v, op)$ to E. For each $m_i \in \{m_1, m_2, \ldots, m_n\}$, invoke $unfold(expr, \lambda x.\texttt{"}x\texttt{.}m_i\texttt{"}, G)$, unfolding the record members.

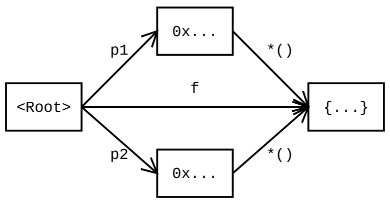
As an example, consider the "Edges" example in Figure 4. Here, the record `f` is created as a vertex and its member `f.val` has been unfolded.

Arrays. Otherwise, if *expr* is an array containing n members $m[0], m[1], \ldots, m[n-1]$, add a vertex $v = ([\ldots], tp, addr)$ to V, and an edge $(parent, v, op)$ to E. For each $i \in \{0, 1, \ldots, n\}$, invoke $unfold(expr, \lambda x.\texttt{"}x\texttt{[}i\texttt{]"}, G)$, unfolding the array elements.

Arrays are handled very much like records, so no example is given.

Pointers. Otherwise, if *expr* is a pointer with address value *val*, add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E. Invoke $unfold(expr, \lambda x.\texttt{"*(}x\texttt{)"}, G)$, unfolding the element *expr* points to (assuming that $*p$ is the dereferenced pointer p),

In the "Aliases" example above, we would end up with the following graph:



Atomic values. Otherwise, *expr* contains an atomic value *val*. Add a vertex $v = (val, tp, addr)$ to V, and an edge $(parent, v, op)$ to E.

As an example, see `f` in the figure above.

For more details on C structures, see Figure 8.

Fig. 7. The construction of memory graphs

Dealing with C data structures

In the programming language C, pointer accesses and type conversions are virtually unlimited, which makes extraction of data structures difficult. Here are some challenges and how we dealt with them.

Invalid pointers. In C, uninitialized pointers can contain arbitrary addresses. A pointer referencing invalid or uninitialized memory can quickly introduce lots of garbage into the memory graph.

To distinguish valid from invalid pointers, we use a *memory map*. Using debugger information, we detect individual memory areas like stack frames, heap areas requested via the *malloc* function, or static memory; a pointer is valid only if it points within a known area.

Dynamic arrays. In C, one can allocate arrays of arbitrary size on the heap via the *malloc* function. While the base address of the array is typically stored in a pointer, C offers no means to find out how many elements were actually allocated; keeping track of the size is left to the discretion of the programmer (and can thus not be inferred by us).

A similar case occurs when a C struct contains arrays that grow beyond its boundaries, as in `struct foo { int member; int array[1]; }` . Although `array` is declared to have only one element, it is actually used as dynamic array, expanding beyond the struct boundaries. Such structs are allocated such that there is sufficient space for both the struct and the desired number of array elements.

To determine the size of a dynamic array, we again use the memory map as described earlier: an array cannot cross the boundaries of its memory area. For instance, if we know the array lies within a memory area of 1000 bytes, the array cannot be longer than 1000 bytes.

Unions. The biggest obstacle in extracting data structures are C *unions*. Unions (also known as variant records) allow multiple types to be stored at the same memory address. Again, keeping track of the actual type is left to the discretion of the programmer; when extracting data structures, this information is not generally available.

To disambiguate unions, we employ a couple of heuristics, such as expanding the individual union members and checking which alternative contains the smallest number of invalid pointers. Another alternative is to search for a *type tag*—an enumeration type within the enclosing struct whose value corresponds to the name of a union member. While such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules—either hand-crafted or inferred from the program.

Strings. A `char` array in C has several usages: It can be used for strings, but is also frequently used as placeholder for other objects. For instance, the *malloc* function returns an `char` array of the desired size; it may be used for strings, but also for other objects.

Generally, we interpret `char` arrays as strings only if no other type claims the space. Thus, if a we have both a `char` array pointer and pointer of another type both pointing to the same area, we use the second pointer for unfolding.

Few of these problems exist in other programming languages. Most languages are far more unambiguous when it comes to interpreting memory contents; in object-oriented languages, unions are obsoleted by dynamic binding.

Fig. 8. Dealing with C data structures

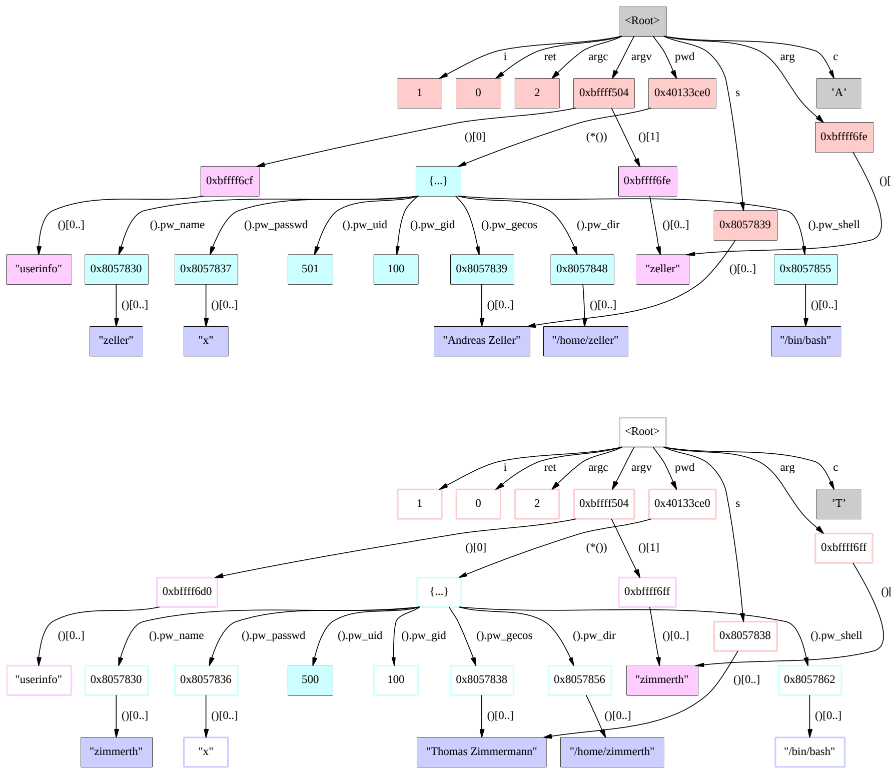


Fig. 9. Finding out what has changed

4 Graph Differences

An important application for memory graphs is *comparing program states*—that is, answering the question “What has changed between these two states?”

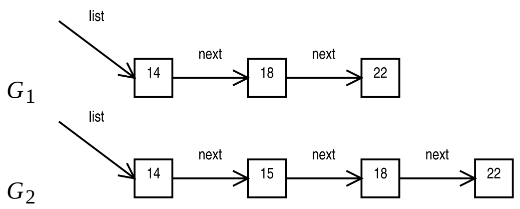
If the state is given in a name/value fashion, comparing states is difficult as soon as pointers come into play. For instance, we might like to invoke the *userinfo* program with a different UNIX user name. In this alternate run, all pointers can have different values (depending on the available memory), but still the same semantics. With a graph abstracting from concrete values, comparing program states becomes a rather simple graph operation—namely, the detection of the greatest common subgraph.

The construction details of the greatest common subgraph is described in Figure 10. In Figure 9, we see the result. The upper graph again shows the *userinfo* state, as in Figure 3. The lower graph shows the *userinfo* state when invoked with UNIX user name *zimmerth*; the common subgraph of the two graphs is outlined. One can clearly see the remaining differences.

If we knew, for instance, that the first run works fine, but the second does not, we know that the cause for the failure must be somewhere in the difference between the program state. Comparing memory graphs gives us this ability.

Comparing Memory Graphs

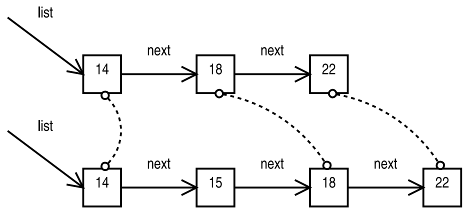
Since they abstract from concrete locations, memory graphs allow comparing program states on a *structural level*. As an example, consider these two memory graphs. What has changed?



As a human, you can quickly see that the element 15 has been inserted into the list. To detect this automatically, though, requires some graph operations. The basic idea is to compute a *maximum common subgraph* of G_1 and G_2 and to flag all the vertices that do not occur in both G_1 and G_2.
How does one compute a maximum common subgraph? Barrow and Burstall [1] first observed that a maximum subgraph can be obtained by using a *correspondence graph*. In our notation, their algorithm looks like this:

1. Create the set of all pairs of vertices (v_1, v_2) with the same value and the same type, one from each graph. Formally, $v_1 \in V_1$, $v_2 \in V_2$ and $val_1 = val_2 \wedge tp_1 = tp_2$ holds where $(val_1, tp_1, addr_1) = v_1$ and $(val_2, tp_2, addr_2) = v_2$.
2. Form the *correspondence graph* C whose nodes are the pairs from (1). Any two vertex pairs $v = (v_1, v_2)$ and $v' = (v'_1, v'_2)$ in C are connected if
   – the operations of the edges (v_1, v'_1, op_1) in G_1 and (v_2, v'_2, op_2) in G_2 are the same, i.e. $op_1 = op_2$, or
   – neither (v_1, v'_1, op_1) nor (v_2, v'_2, op_2) exist.
3. The maximal common subgraph then corresponds to the *maximum clique* in C—that is, a complete subgraph of C that is not contained in any other complete subgraph. This maximum clique can efficiently be computed using the algorithm of Bron and Kerbosch [2].

For our purposes, the resulting maximum clique (i.e. the set of corresponding vertices) already suffices: Any vertex that is not in the clique indicates a difference between G_1 and G_2.
The following figure shows the pairs obtained in (1). Since this is pretty unambiguous, finding the maximum clique is trivial—it is simply the one set of pairs. But it is plain to see that the element 15 in G_2 has no counterpart in G_1.



By highlighting inserted or deleted vertices this way, future debugging tools can quickly compare program states and identify what has changed between two states of a program run.

Fig. 10. Detecting differences

5 Querying Memory Graphs

As a last memory graph, consider Figure 11. This memory graph was obtained from the GNU compiler as it compiled the C statement

```
z[i] = z[i] * (z[0] + 1.0);
```

The graph shows the statement as a *register transfer language* (RTL) tree, the internal representation of the intermediate language used by the GNU compiler. (The GNU compiler first converts its input into a syntax tree, which is transformed into RTL, which, after a series of optimizations, is then finalized into assembler language.)

This graph shows only a subset of the full GNU compiler state, whose memory graph at this time has about 40,000 vertices. However, even this subset is already close to the limits of visualization: if the RTL expression were any larger, we would no longer be able to depict it.

Nonetheless, we can use this graph to debug programs. It turns out that GCC crashes when its internal RTL expression takes this form. This is so because this RTL tree is not a tree; it contains a cycle in the lower right edge. This cycle causes an endless recursion in the GNU compiler, eventually eating up all available heap space.

We do not assume that programmers can spot cycles immediately from the visualization in Figure 11. However, we can imagine traditional graph properties (such as the graph being complete, cycle-free, its spanning tree having a the maximum depth and so forth) being computed for memory graphs, for instance in a debugging environment. A click on a button could identify the cycle and thus immediately point the programmer to the failure cause.

6 Drawing Memory Graphs

The figures in this paper were drawn in a straight-forward way using the DOT graph layouter from AT&T's *graphviz* package [3]. While these layouts are nice, they do not scale to large memory graphs (with 1,000 vertices and more).

More promising are *interactive* graph renderings that allow the user to navigate along the graph. We are currently experimenting with the *h3viewer* program that creates hyperbolic visualizations of very large graphs [4].

Figure 12 shows a screenshot of *h3viewer*.[3] The actual program is interactive: clicking on any vertex brings it to front, showing detailed information. By dragging and rotating the view, the programmer can quickly follow and examine data structures. If future successors to DDD will have an interactive graph drawing interface, it may look close to this.

Another idea to be explored for presentation is *summarizing* parts of the graph. For instance, rather than showing all *n* elements of a linked list, it may suffice to present only the basic *shape* of the list—in the style of *shape analysis* [5], for instance.

Finally, there are several pragmatic means to reduce the graph size: for instance, one can prune the graph at a certain depth, or, simpler still, restrict the view to a particular module or variable.

[3] Colors have been altered to fit printing needs.

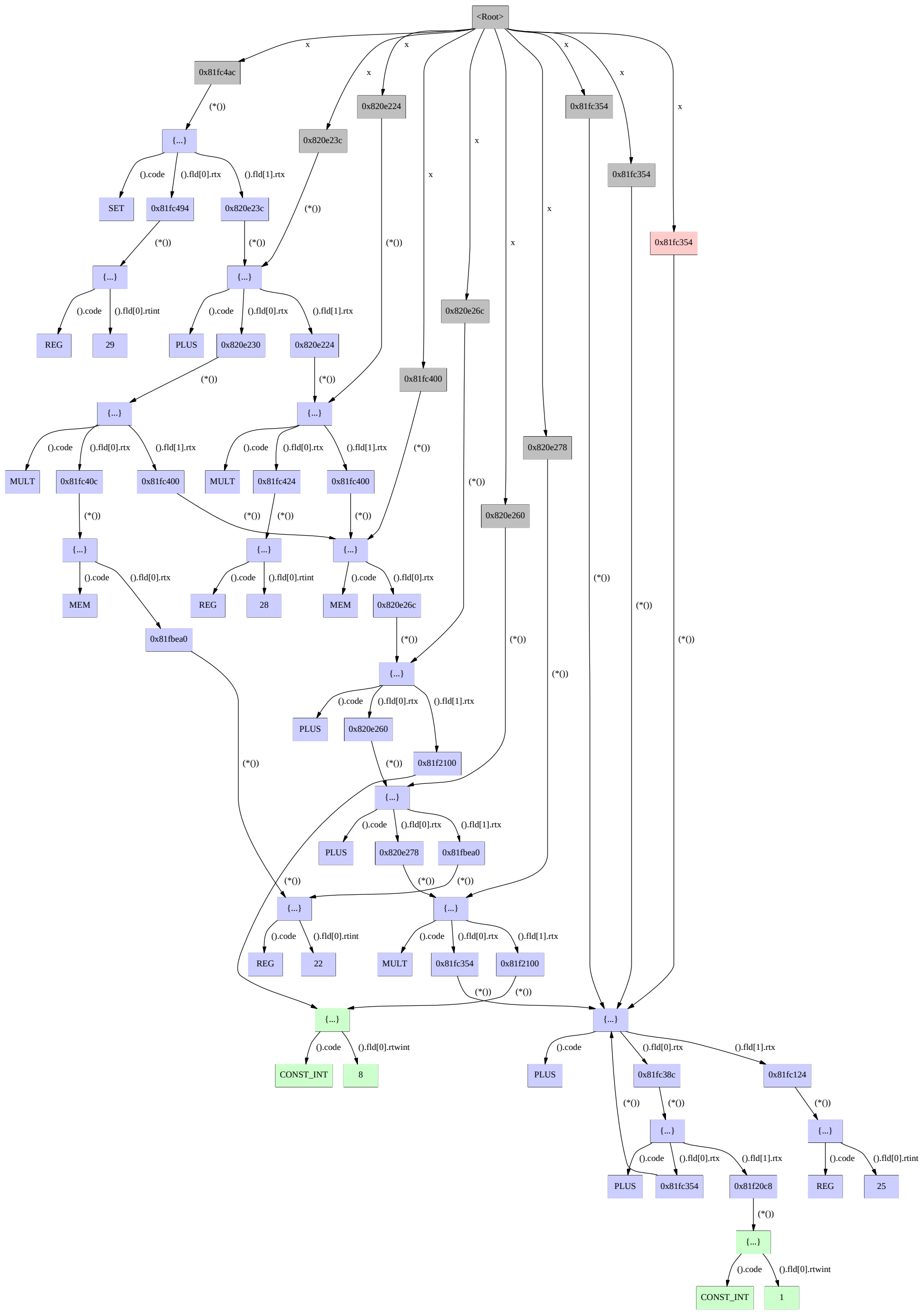


Fig. 11. An RTL tree in the GNU compiler. Can you spot the cycle?

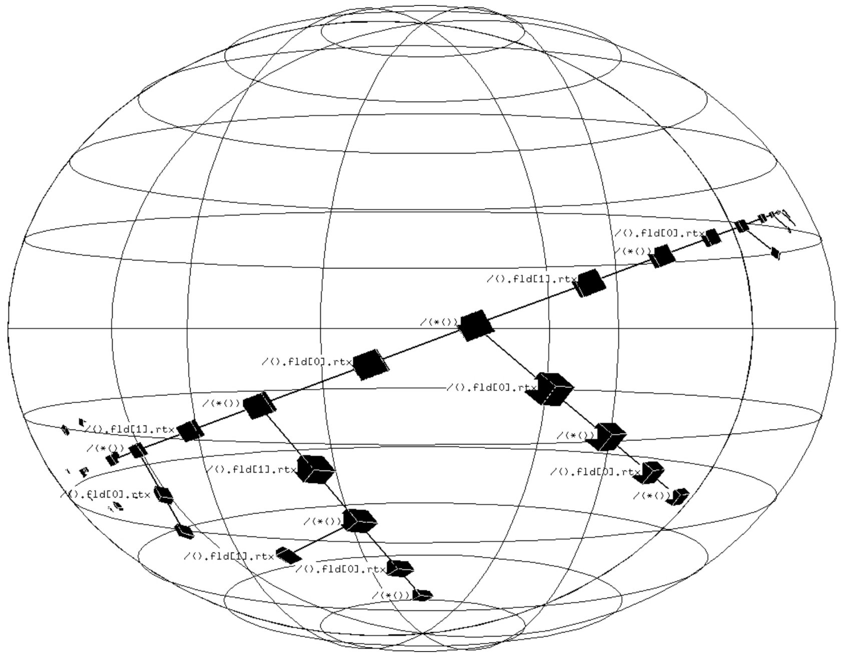


Fig. 12. The RTL tree from Figure 11 as visualized by *h3viewer*

7 Conclusion

Capturing memory states into a graph is new, and so are the applications on these graphs. Realizing appropriate navigation tools, efficient analysis and extraction methods and useful visual representations are challenges not without reward.

More information on memory graphs can be found at

`http://www.st.cs.uni-sb.de/memgraphs/`

References

1. H. G. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, 1976.
2. C. Bron and J. Kerbosch. Algorithm 457—finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.
3. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.
4. T. Munzner. Drawing large graphs with h3viewer and site manager. In *Graph Drawing '98*, volume 1547 of *Lecture Notes in Computer Science*, pages 384–393, Montreal, Canada, Aug. 1998. Springer-Verlag.
5. R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. of CC 2000: 9th International Conference on Compiler Construction*, Berlin, Germany, Mar. 2000.
6. A. Zeller and D. Lütkehaus. DDD—A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, Jan. 1996.