An Interactive System for Data Structure Development

Jibin Ou
ETH Zürich
jou@student.ethz.ch

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Otmar Hilliges
ETH Zürich
otmar.hilliges@inf.ethz.ch

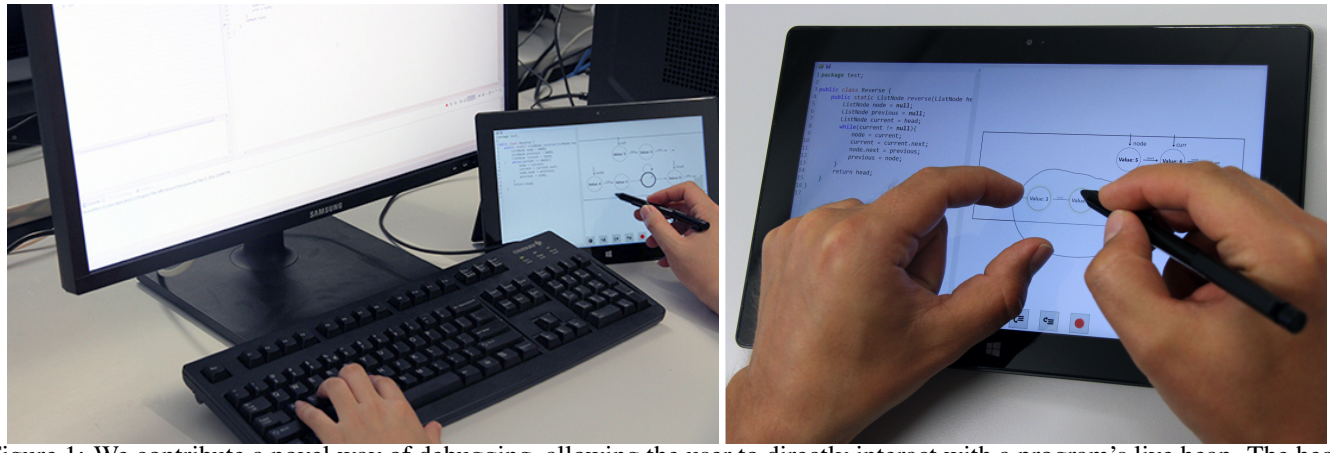


Figure 1: We contribute a novel way of debugging, allowing the user to directly interact with a program's live heap. The heap graph is abstracted using a method for online parametric heap abstraction which allows us to only visualize the algorithm's essential operation. The user can change and refine these automatic abstractions interactively. Our system can be used during regular development (left) and in a classroom setting using a pen+touch user interface (right).

ABSTRACT

Data structure algorithms are of fundamental importance in teaching and software development, yet are difficult to understand. We propose a new approach for understanding, debugging and developing heap manipulating data structures.

The key technical idea of our work is to combine deep parametric abstraction techniques emerging from the area of static analysis with interactive abstraction manipulation. Our approach bridges program analysis with HCI and enables new capabilities not possible before: i) online automatic visualization of the data structure in a way which captures its *essential operation*, thus enabling powerful local reasoning, and ii) fine grained pen and touch gestures allowing for interactive control of the abstraction – at any point the developer can pause the program, graphically interact with the data, and continue program execution. These features address some of the most pressing challenges in developing data structures.

We implemented our approach in a Java-based system called *FluidEdt* and evaluated it with 27 developers. The results indicate that *FluidEdt* is more effective in helping developers find data structure errors than existing state of the art IDEs (e.g. Eclipse) or pure visualization based approaches.

Author Keywords

debugging; program analysis; software development

ACM Classification Keywords

D.2.6 Programming Environments; H.5.2 User Interfaces

INTRODUCTION

Data structures and algorithms are at the core of modern software systems and are a staple of computer science curricula worldwide. It is via data structures and algorithms that students are first exposed to many of the core concepts in computer science. However, despite decades of research and teaching, understanding the underlying operation of even small data structure algorithms remains as difficult and as cognitively demanding as ever. Even shorter programs can perform complex and difficult to understand manipulations on the program's heap. Thus data structure manipulating algorithms are an interesting case for HCI research as they are well understood *in principle* yet are very hard to master *in practice*. To crystalize some of the issues associated with their development, we conducted a series of interviews with colleagues involved with the data structures class at our University as well as a thorough analysis of hundreds of online posts related to this topic (found on Stack Overflow*). The results of our study clearly illustrate a diverse set of errors and misunderstandings involving linked lists, arrays, trees and other structures – the complete catalog can be accessed via supplementary materials.

CHI 2015, April 18 - 23 2015, Seoul, Republic of Korea
Copyright ©2015 ACM 978-1-4503-3145-6/15/04... $15.00
http://dx.doi.org/10.1145/2702123.270231

*`http://stackoverflow.com`

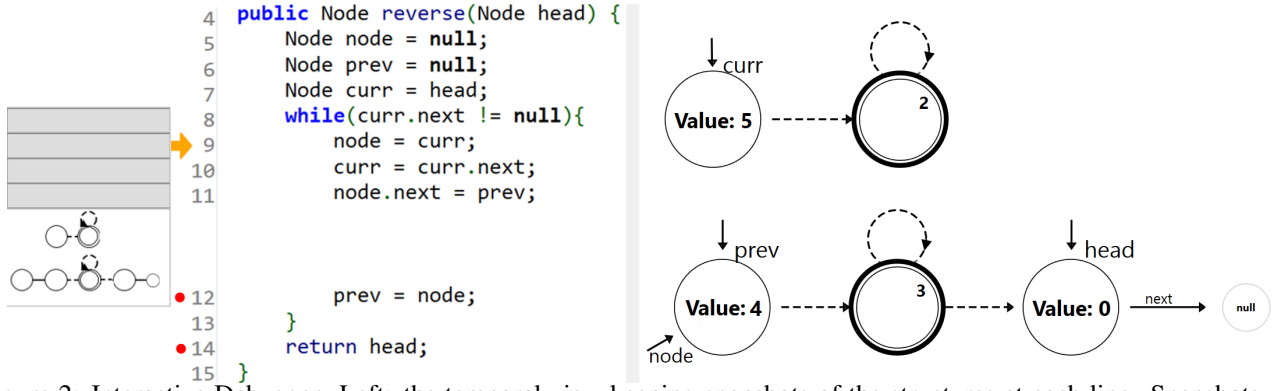


Figure 2: Interactive Debugger. Left: the temporal view keeping snapshots of the structures at each line. Snapshots can be collapsed and expanded on demand and are aligned with lines of code. Middle: the code (in this case, an attempt at reversing a linked list). Right: the abstracted heap view. Red dots indicate break points, yellow arrow indicates the current line.

Our study firmly reinforces that complex pointer manipulations are a key source of confusion in understanding existing algorithms, in discovering and repairing errors, and even in explaining how algorithms work. Unfortunately, despite the universal importance of these algorithms, there is still little tool support for effectively understanding their operation.

In this paper we introduce a new approach and a complementary tool for interactive debugging and development of data structure manipulating algorithms. Our approach bridges the areas of HCI and program analysis. The key technical idea of our work is based on a novel combination of parametric predicate abstraction techniques emerging from the area of static analysis with *interactive* abstraction manipulation. Our tool has two benefits. First, it automatically visualizes the data structure in a way which *captures its essential operation* while abstracting the rest. This is an important feature which enables *local reasoning*, meaning that even though the data structure may contain a large number of nodes, at any given time only the (small number of) nodes relevant to the current program context are kept concrete. Second, our tool provides the developer with fine grained controls over *which data is visualized* – the developer can pause the program, interact with the data structure (e.g. abstract, concretize, modify), and continue program execution with the new structure. This enables one to try various hypotheses for the algorithm operation.

We envision our approach being useful in a wide range of settings including the classroom but also in the increasingly common search-and-modify code development setting. For instance, in this setting, a developer will usually i) search online for the particular functionality (e.g., an AVL tree), ii) try to understand what the suggested program does [16], and iii) modify the program according to their needs. Proliferation of online source code repositories such as GitHub and sites such as Stack Overflow, along with other web resources have made step i) almost instant. Further, the HCI research community has come up with a number of tools to better integrate the usage of web resources and the development process [5, 9, 22, 28]. Step ii) is most time consuming and challenging as the developer is not familiar with the code and even short algorithms can admit many different implementations. Worse, step iii) cannot be accomplished until step ii) is thoroughly completed. We believe that our tool can be very effective in reducing the cost of the search-and-modify loop by drastically reducing the time for step ii).

In our experimental user study, we show that our system makes it easier and faster to detect data structure errors than modern debuggers found in IDEs such as Eclipse or Visual Studio.

Main contributions:

- A method for online and automatic parametric heap abstraction, enabling *local reasoning* by focusing developer attention on the essential parts of the data structure.
- A fine grained interface for *interactive heap abstraction manipulation* involving visual representation of (abstract) objects and gestures for navigation and modification.
- An implementation of our approach in a system called *FluidEdt* and an experimental results indicating increased efficiency in finding errors over existing IDEs.

SYSTEM OVERVIEW

The user interface consists of i) a temporal view containing (easily accessible) structures recorded at each program point (left), ii) the code view in the middle (here showing a program attempting a list reversal), and iii) the main drawing canvas where the (abstract) heap is rendered on the right.

Temporal View

The temporal view captures a visual history of (abstract) shapes. At every point this history keeps a snapshot of the current heap. To prevent visual clutter, the snapshots are initially collapsed (similar to [18]) and the user can expand a thumbnail for any snapshot, or drag it onto the *heap view* to restore it. This feature enables relational reasoning by comparing heap structures at different execution points. We currently keep the most recent snapshot at a program point, usually effective for common programming scenarios (if needed, the system can record more snapshots). We note that static analyzers have also successfully used forms of recency abstractions [2].

Code View

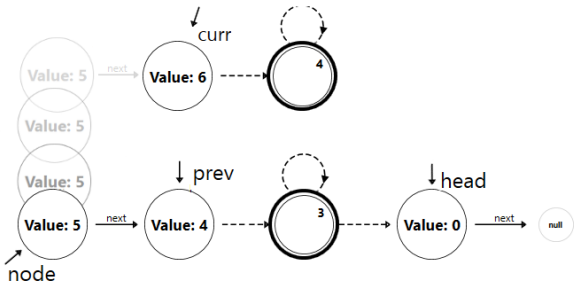
The code view shows the part of the program currently being inspected. As with traditional IDEs, red dots indicate break points while a yellow arrow indicates the current line (in this

case line 9). Beyond giving the user a simultaneous view onto code and data, this view also links together temporal and heap views. Here, snapshots are aligned with the line of code they belong to. The code shows a short but challenging example: (a faulty) algorithm which attempts to perform linked list reversal. A typical Java implementation of this algorithm only takes about 10 lines of code, yet despite its simplicity, developers struggle with this problem – there are hundreds of threads on this topic on StackOverflow. Many of these are lengthy discussions with many answers (which themselves often need to be corrected). Almost all threads include some form of graphical depiction illustrating the algorithm operation.

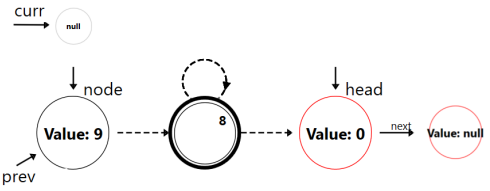
Abstracted Heap View

This view renders an abstraction of the data structure and allows for interactive manipulation. It aims to distill the *essential* parts of the structure while abstracting the rest, enabling *local reasoning*. Here the view shows an abstraction of the concrete structure arising at line 9 – abstract nodes are shown with thick bold edges, while abstract edges are denoted by dashed lines. An abstract node can contain a number of concrete nodes (indicated by its label). Similarly, an abstract edge may contain a number of concrete edges. In this example we have two abstract nodes, one with 2 concrete nodes and another with 3. Note how the abstract shape already conveys much useful information: we immediately see that the list is disconnected into two parts and that `head` points to the last element of one of the structures. Importantly, regardless of the size of the concrete list, it will still be collapsed into this shape *automatically*, greatly reducing irrelevant details (e.g. the size of the list).

Suppose the programmer inspects the structure and finds that the structure captures her intuition of how the algorithm should work. Naturally, she may then set a breakpoint, say at line 12, run the program, and inspect the new structure arising at that line. To allow programmers to better understand the changes between program lines (e.g. lines 9 to 12), we smoothly animate these transitions in the displayed graph. In particular, nodes that were concrete in one frame and will be abstracted next, are moved smoothly into the new abstract node and vice versa. This feature is crucial for users to understand the temporal effects on the heap during program execution. Further, as the developer steps through the code (or reaches breakpoints), our system will automatically update the abstraction. The abstract structure at line 12 then is:



The animation indicates the movement of a node to the front of the second structure while `curr` advanced towards the abstract node, necessitating concretization. The rest of the structure remains the same. The programmer now inspects the structure and assuming it conforms to their intuition, sets a breakpoint at line 14 resulting in the following structure:



We now clearly see the algorithmic error: `head` points to the end rather than the beginning of the list and hence the algorithm returns an incorrect result! Note how it is unnecessary to know what is in the middle of the list (the abstracted part) to know that an error has occurred.

Interactions and Usage Scenarios

We envision our tool being useful in at least two separate but related scenarios. The first scenario is regular development of new code. This scenario usually happens on a regular desktop PC with a large monitor, mouse and keyboard. The main form of interaction with *FluidEdt* in this scenario is likely to be focused on stepping through code, comparing (abstract) snapshots and specifying the visual abstraction levels.

The second scenario focuses on code understanding rather than writing new code from scratch. In particular, it applies to the classroom where code snippets are provided by the instructor for students to experiment with the program on their own devices. Today almost all students own personal touch screen devices and bring them to class. Towards this, we designed a set of touch and pen gestures that allow one to directly interact with the data of a program. Similar approaches have been used successfully to foster understanding of complex issues such as mathematical equations [30] and multivariate datasets [23], suggesting that direct, physical manipulation of complex and abstract data aids understanding.

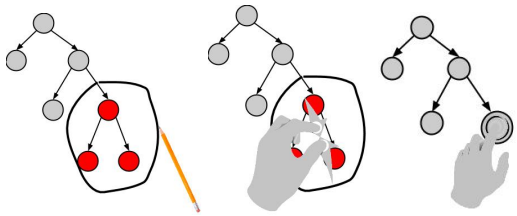
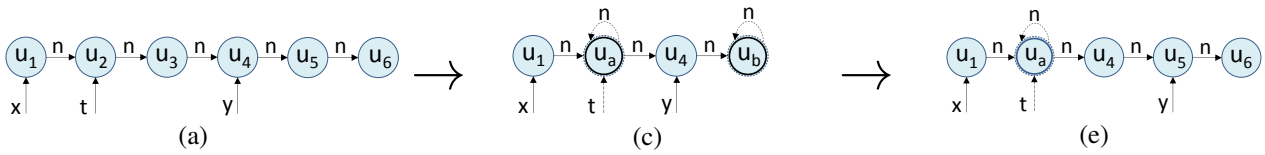


Figure 3: Pen+Touch gestures for heap manipulation

We provide a succinct set of gestures allowing users to interact with the heap. Fig. 3 illustrates how different tasks are allocated to pen and touch respectively. Touch is mainly used to pan and zoom the canvas and to arrange the nodes spatially, in case the automatic layout does not suit the user. The user can also select abstract nodes in order to inspect the values encapsulated by the node. The stylus is primarily used to select multiple nodes simultaneously by circling them (Fig. 3, left). Selected nodes can either be moved together, or be abstracted by a pinch gesture performed directly on the selection (Fig. 3, middle). Abstracted nodes can be concretized by double tapping the node (Fig. 3, middle).

Concrete Heap | Abstracted Heap | Abstracted Heap after: `y := y.n`



	u_1	u_2	u_3	u_4	u_5	u_6
$x(u)$	1	0	0	0	0	0
$y(u)$	0	0	0	1	0	0
$r_{n,x}(u)$	1	1	1	1	1	1
$r_{n,y}(u)$	0	0	0	1	1	1
$t(u)$	0	1	0	0	0	0

(b)

	u_1	u_a	u_4	u_b
$x(u)$	1	0	0	0
$y(u)$	0	0	1	0
$r_{n,x}(u)$	1	1	1	1
$r_{n,y}(u)$	0	0	1	1
$t(u)$	0	1/2	0	0

(d)

	u_1	u_a	u_4	u_5	u_6
$x(u)$	1	0	0	0	0
$y(u)$	0	0	0	1	0
$r_{n,x}(u)$	1	1	1	1	1
$r_{n,y}(u)$	0	0	0	1	1
$t(u)$	0	1/2	0	0	0

(f)

Table 4: The figure shows how the concrete linked list in (a) is abstracted into the shape in (c) via the abstraction predicates x, y, $r_{n,x}$, $r_{n,y}$ and how the abstraction in (c) is updated when performing the statement `y := y.n` resulting in (e).

ABSTRACTION BY EXAMPLE

We now present an intuitive understanding of the core techniques underlying our approach (formal explanations are provided later). The three core features of our approach are: i) how to perform abstraction of a concrete graph (used to perform the abstraction of the heap upon starting the program), ii) how the abstraction is updated when stepping through the program, and iii) how we support interactive user driven abstraction manipulation. Our machinery is inspired by techniques developed in the static analysis community [24], but departs from these in a number of important ways.

A key ingredient of a parametric abstraction are predicates – logical formulas which given a (concrete) heap evaluate to 1 (i.e. $true$) or 0 (i.e. $false$). These predicates capture information which we would like to preserve in the abstracted graph. Predicates are the knobs for controlling what information should be abstracted away. Roughly speaking, we abstract the heap in a way that relevant predicates (abstraction predicates) are kept concrete. We now illustrate how to abstract the concrete list in Table 4(a) into the abstract list sin Table 4(c).

Step 1: select predicates

We first select the relevant predicates for our problem. By default, we use local pointer variables as *points-to* predicates – these have proven useful in enabling local reasoning [24], a key to understanding intricate data structure behavior. We usually instantiate new *points-to* predicates corresponding to the local pointer variables in the stack frame as well as few other predicates we found generally applicable (based on our experimental study). Of course, the user can switch-off predicates at any point if they do not find them useful. For our example, we used five unary predicates: $x(u)$, $y(u)$, $t(u)$, $r_{n,x}(u)$ and $r_{n,y}(u)$. That is, we use points-to predicates x, y, t and reachability predicates (as we may be interested in keeping precise information of what is reachable from pointers x and y). Out of these five, all are abstraction predicates except $t(u)$ meaning that t's value need not be kept concrete. We selected $t(u)$ on purpose to illustrate how one can obtain $1/2$ values (edges carrying $1/2$ information are dashed in the pictures). For our list reversal in Fig. 2, the automatically selected abstraction predicates will be `curr`, `prev`, `node` and `head`.

Step 2: predicate evaluation

Next, we evaluate each of the selected predicates on each concrete node. The result of this evaluation is shown in Table 4(b). In the general case where the predicates can be arbitrary logical formulas, the evaluation process involves evaluating logical formulas (as done in [24]). However, in our case, the predicates have a pre-defined meaning (that is, they have a fixed logical formula) and hence we can evaluate them with graph algorithms specific to their particular meaning.

Step 3: direct abstraction

After predicate evaluation, the next step is to perform abstraction. Using our abstraction approach, we obtain the predicate values shown in Table 4(d). The basic idea of abstraction is conceptually simple: all nodes which share the same set of predicate evaluations for the *abstraction predicates* are collapsed into an abstract node. For example, nodes u_2 and u_3 are collapsed into the same abstract node u_a because they both share the exact same valuations for all four of the abstraction predicates. Similarly, nodes u_5 and u_6 are collapsed into the same abstract node u_b. However, node u_4 is *not* abstracted into u_a or u_b because $y(u_3) \neq y(u_4)$. It is important to note that all of the abstraction predicates remain with *concrete* values, that is, they still evaluate to 0 or 1 for the abstracted node u_a and u_b. However, the values of the non-abstraction predicates such as $t(u)$ need not be kept concrete, and as we can see, indeed, it now evaluates to $1/2$ for u_a (because there were two concrete nodes u_2 and u_3 for which t evaluates to 1 and 0 respectively).

Step 4: visualize

Once the new matrix is obtained, we can now visualize that matrix – the result is shown in Table 4(c). Here, if for a given abstract node, the predicate evaluates to 1, we show the node, otherwise (if its $1/2$ or 0) it is omitted from the picture. To reduce clutter, in our figure we do not show $r_{n,x}$ or $r_{n,y}$.

On-demand abstraction updates

Now suppose that starting from the shape shown in Table 4(c), the programmer performs one step by executing `y := y.n`. If we proceed naively, y would end up pointing to the abstract node u_b and it may be confusing to the programmer what exactly is inside u_b. To avoid this, we automatically concretize node u_b on-demand – the system will observe that the abstraction predicate y now evaluates to $1/2$ and will concretize the node, splitting it into two abstract nodes where y will evaluate to 0 on one of the abstract nodes and to 1 on the other node (in our example, there are only two concrete nodes hence we fully concretize u_b but this need not be the case with all other unary predicates and shapes). Note that node u_a still stays abstracted as it is not relevant to the current "local view". The result of this step is shown in Table 4(f) and the visualization in Table 4(e). We argue that this gradual update of the abstraction *on demand* helps developers in understanding the algorithm as it focuses on the local changes to the data while abstracting the uninformative complexity of the heap.

Stepping into functions

When we step into a function, the previous predicates are still used to maintain the abstract shape, but we identify the new stack pointers and add the corresponding points-to predicates corresponding to these pointers as abstraction predicates.

Manual abstraction interactions

We allow the user to control the abstraction by selecting nodes which are to be abstracted and to concretize existing nodes. This enables users to customize the heap view to the particular intuition they have about the algorithm at a given point. The machinery for these manual interactions is similar to what is discussed above and is elaborated in the next section.

A note on predicates

In general, predicates can be arbitrary logical formulas. However, in our work, we decided against supporting arbitrarily defined predicates for two reasons: i) we do not expect developers to write complex logical formulas, and ii) perhaps more importantly, automatically abstracting arbitrary logical formulas is computationally expensive [24] and not suitable for an interactive environment. Therefore, in our approach we still use the abstraction semantics as described above (same as in [24]), but we work with a set of predefined predicates that can be instantiated. We provide the full set of built-in predicates in the Appendix. Our set of built-in predicates was derived from our detailed experimental evaluation and inspecting various programming forums, in which we found it to naturally captures programmer's intentions. We do note however that there is nothing inherent in our parametric approach that precludes us from adding more predicates. We note that the user can extend this list with other predicates without requiring any change to our (parametric) system.

ABSTRACTION FORMALLY

In this section, we present a more formal discussion of how our abstractions work: the abstraction representation, the initial abstraction of the program, the update of the abstraction when a piece of code is executed and how we combine predicate abstraction with user level manipulations of the shape.

Abstraction representation

To capture the abstraction and the concrete nodes that are connected with it, our system maintains the four maps described below. These four maps are continually updated during program navigation:

- $\alpha_N(v) : CN \rightarrow AN$: the map maintains a mapping from a concrete to an abstract node.
- $\alpha_E(e) : CE \rightarrow AE$: the map maintains a mapping from a concrete to an abstract edge.
- $\gamma_N(v) : AN \rightarrow \mathcal{P}(CN)$: reverse map from an abstract node to a *set* of concrete nodes.
- $\gamma_E(e) : AE \rightarrow \mathcal{P}(CE)$: reverse map from an abstract edge to a *set* of concrete edges.

Here, we use CN to denote the set of concrete heap nodes, AN to denote the set of abstract nodes, CE to denote the set of concrete edges and AE to denote the set of abstract edges. We note that such maps are *not* naturally maintained by static tools such as TVLA (Three-Valued Logic Analyzer, a tool implementing the approach in [24]) because the concrete nodes are not available (the entire purpose of static analysis is to soundly approximate these concrete nodes with abstract nodes). Maintaining such maps is unique to our setting.

In a static analysis setting (e.g. TVLA), abstraction is done entirely based on predicates without the option of interactively modifying the abstract shapes. In our case however, modification of the abstract shape is a first class operation and therefore we need to address the challenge of combining such modifications with the standard predicate abstraction – a challenge which simply does not arise in the pure static analysis setting.

Computing the initial abstraction

Initially, when the program starts, we have a concrete heap G and a current set of pointer variables. A natural problem then is how to abstract this initial shape (as the shape may be too large due to the input test case, etc). Algorithm 1 shows how to perform the initial abstraction. Here, we use $Preds$ to denote the set of predicates that we use for abstraction. First, at step 1, the algorithm starts by evaluating each predicate on each node (object) in the concrete graph (we already illustrated this step on an example). Because the graph is concrete, the resulting value of the evaluation can only be 0 ($false$) or 1 ($true$). Then, in step 2, it groups all nodes which have the same value for *all* of the predicates. Finally, at step 3, we update the node and the edge abstraction and concretization maps accordingly – this is done via the auxiliary functions described in Algorithm 2. For readability purposes, we slightly abuse notation and write $f(a) = b$ for $f' = f[a \rightarrow b]$ followed by $f = f'$.

Updated abstract shapes

Once an abstract shape is computed, it is important to define what happens to it as the user steps through the program. In particular, we need to define how such shapes are transformed during program execution. For instance, if the user steps over a large function (which results in some concrete heap), we need to define how the new abstract shape is computed. Note that in practice, the function over which the user has stepped over may not even be subject to instrumentation (e.g a native method in Java). Therefore, we cannot adopt the approach of TVLA

```
// create temporary maps
NodePredVals = new (CN → Preds → {0,1})
PredValNodes = new (Preds → {0,1} → P(CN))
// step 1: compute predicate values for each node
foreach node ∈ G.nodes do
    foreach pred ∈ Preds do
        NodePredVals(node)(pred) = pred(node)
    end
end
// step 2: build equivalence classes
foreach node ∈ G.nodes do
    PredVals = NodePredVals(node)
    PredValNodes(PredVals) ∪= {node}
end
// step 3: create the abstract maps: nodes and edges
foreach (_, nodes) ∈ PredValNodes do
    abstractNodes(nodes)
end
abstractEdges(G.edges)
```

Algorithm 1: Creating an abstraction.

```
abstractNodes(nodes) {
absNode = createUniqueAbstractNode()
foreach n ∈ nodes do
    αV(n) = absNode
    γV(absNode) ∪= {n}
end
}
abstractEdges(edges) {
foreach e ∈ edges do
    absEdge = ⟨αV(e.source), αV(e.target)⟩
    αE(e) = absEdge
    γE(absEdge) ∪= {e}
end
}
```

Algorithm 2: Auxiliary functions.

```
foreach an ∈ range(αV) do
    foreach pred ∈ Preds do
        if pred(an) = 1/2 then
            N1 = {cn | cn ∈ γV(an) ∧ pred(cn) = 1}
            N0 = {cn | cn ∈ γV(an) ∧ pred(cn) = 0}
            abstractNodes(N0)
            abstractNodes(N1)
            abstractEdges(dom(αE))
        end
    end
end
```

Algorithm 3: Focusing / Refining the abstraction.

Statement	Updated Map
$new\ edge\langle e\rangle$	$\alpha'_E = \alpha_E \cup (e,e), \gamma'_E = \gamma_E \cup (e,\{e\})$
$del\ edge\langle e\rangle$	$dom(\alpha'_E) = dom(\alpha_E) \setminus \{e\}$ $\gamma'_E = \gamma_E[(\alpha_E(e)) \to \gamma_E(\alpha_E(e)) \setminus \{e\}]$
$new\ node\langle v\rangle$	$\alpha'_V = \alpha_V \cup (v,v), \gamma'_V = \gamma_V \cup (v,\{v\})$
$del\ node\langle v\rangle$	$dom(\alpha'_V) = dom(\alpha_V) \setminus \{v\}$ $\gamma'_V = \gamma_V[(\alpha_V(v)) \to \gamma_V(\alpha_V(v)) \setminus \{v\}]$

Table 5: Updating the abstraction and concretization maps.

which defines the effects of a statement on the abstract shape (as we may not have the statement available). We next describe a 3-step process for computing this new abstract shape.

Step 1: compute shape differences

Given the initial graph C and the new graph C' resulting from stepping over statements, we compute their heap difference. That is, we compute the quadruple $\langle newEdges, delEdges, newNodes, delNodes\rangle$ denoting the set of new edges in C' not in C, the set of deleted edges from C not in C' (and similarly for nodes).

Step 2: update maps

We next update the abstraction and concretization maps defined earlier: we iterate over the sets and update the map accordingly. The rules for updating the maps for each of the four cases (added edge, deleted edge, new node, deleted node) are described in Table 5. For instance, if a new edge is created, we add that edge to the abstraction and concretization maps.

Step 3: focus maps

Simply updating the maps in step 2 is not enough. The reason is that a useful unary predicate may suddenly evaluate to $1/2$ in the new abstract graph and we would like these predicates to be concrete (an example of that was shown in the overview section). A predicate evaluates to $1/2$ if the abstract node contains two concrete nodes where the predicate evaluates to 0 for one node and to 1 for the other. In Algorithm 3, we show how to focus the values of these predicates. Here, if a node is evaluated to $1/2$, we concretize the node into two sets of nodes and then re-abstract these sets of concrete nodes as before using the auxiliary functions (and update the maps). As a result, we end up with a more intuitive abstract graph.

Interactive Abstraction Manipulation

Manual abstraction manipulations (e.g. via pen+touch) involve selection of nodes to be abstracted or concretized. Both of these operations update the maps as discussed so far – for instance, the interactive abstraction works via the two functions shown in Algorithm 2. Note that these operations may change the values of the selected abstraction predicates as they will now be evaluated on the new (possibly abstract) structure.

IMPLEMENTATION

We implemented our approach described so far in a prototype system called *FluidEdt* targeting Java (we built an Eclipse plug-in to query the heap information). The plug-in sends the heap information to the front-end which processes and renders the graph. The front-end is built as a stand alone Windows app.

For the graph rendering component, we used a customized version of the GraphX framework[†]. AvalonEdit[‡] is used to format the code view. We implemented a communication protocol to initially transfer the graph data from back-end to front-end and to incrementally transfer heap modifications (i.e. the delta between two shapes) at each program step.

In general, when we step in or out of a function, we simply copy the abstract maps – nothing else needs to be changed. By default, all predicates in the new function are disabled, but the user is free to enable them as she wishes. Because stepping in or out of a function does not change the maps, this entails that the only time we compute differences between heap graphs is when the graphs belong to the same function.

In addition to the abstraction and concretization, we support data *allocation* and *modification*. For *data modification*, one can change a node's primitive value or connect two nodes together. For *data allocation*, we allow the user to create new nodes on-demand without changing the code. This is useful in a debugging scenario where the programmer wishes to check if certain data configurations behave well (i.e. do not exhibit bugs) when starting from particular program points. Our implementation automatically updates the current program state in order to reflect these graphical changes allowing the program to continue running with the new heap state. Currently we do not persist these changes but leave this for future work.

EXPERIMENTAL EVALUATION

Our literature review and analysis of online forums has revealed that the two main sources of confusion – and hence errors – are complexity of data structures and complex pointer manipulations. In this section we report findings from a controlled experiment assessing the benefits of our design in respect to these difficulties. Our hypotheses are:

H1: Visualizing the heap of a running program improves code understanding and makes it *faster for developers to find errors* in the program code, compared to a *standard debugger*.

H2: Automatic and user driven abstraction of the heap graph will further improve code understanding. Hence, developers will be *faster to find errors* with these features than when using either a *standard debugger* or a *pure visualization*.

Experiment

To evaluate the hypothesis, we conducted a controlled experiment comparing our interface (***FluidEdt***) with i) the standard debugger of Eclipse IDE[§] (**Eclipse**), and ii) a reduced version of *FluidEdt* that does visualize the heap but has no abstraction & concretization functionality (**Viz**). This variant also forbids the user to interact with the heap graph.

To test our hypothesis we asked experienced developers to try and find errors in programs. To avoid learning effects we chose an across-groups design i.e., each participant worked with each of the three **interfaces** but performed a different task in each condition. The experiment was conducted on a desktop PC (Core i7 CPU 3.4GHz) running Windows 8.1, equipped with a 19" monitor and an additional Wacom Cyntiq HD 24 tablet providing touch and pen input to drive the touch enabled UI. The resolution was set to 1920×1200.

[†] `https://github.com/panthernet/GraphX`

[‡] `http://avalonedit.net/`

[§] `https://www.eclipse.org/`

Participants

We recruited 27 participants (3 female, 24 male) from our institution, mostly students (2 undergrad, 20 grad) and 5 postdocs from the computer science (CS) department. By selecting CS students only we aim to control the variance in programming skills (inline with prior experiments [3, 14]). Participants' ages ranged from 22 to 35 years ($M = 24$) and their self reported programming experience ranged from 2 to 7+ years ($M = 4.3$) with weekly programming activities ranging from 3 to 40 hours ($M = 17.3$). All participants reported 2 or more years of experience using some IDE.

Tasks

Each participant had to complete three tasks in total. Each of the tasks was performed with a different *interface* and to avoid learning effects each participant worked with different programs for each task. Participants were given a description of the expected program behavior and were told that the program contained at least one error. We designed the tasks such that they fall into three levels of difficulty from easy to hard.

T1: 10 lines of Java code reversing a singly-linked list. The code contained two bugs. First, it returns the tail of the reversed list (not the head). Second, after reversal the last node is missing from the list due to early-loop termination.

T2: 30 lines of Java code which reverse a linked-list from index m to n (not the whole list). The sole program error introduces a cycle into the list.

T3: 200 lines of Java code implementing an AVL-tree including insertion of new values and tree re-balancing. This code contained four errors but all of them were very similar so that once one is found all can be fixed. The errors cause entire subtrees to be detached from the tree during re-balancing. However, the error is not in the re-balance method but caused by wrong assignment of the method's return value (the sub-tree).

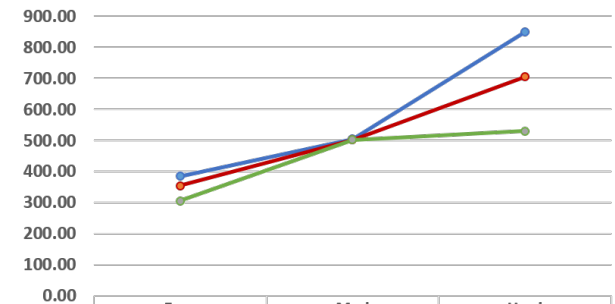
A description of the data-structures was read to the participants and we provided them with a set of sensible inputs to test the code. All participants managed to complete the tasks within the allotted 20 minutes.

Procedure

We used a $3 \times 3 \times 1$ between-subjects design with **tct** the dependent variable and with **interface** and **task** as the independent variables. Presentation order of conditions was counterbalanced using a Latin Squares design. Participants were split into three groups so that each task was only performed once per participant (and per interface). Resulting in a total of nine participants per *interface* $\times$ *task* combination and a total of 81 debug sessions (lasting 20 minutes each). Before each trial the participants were given written instructions on how to use the interface, were allowed to ask questions and to practice. During this warm-up phase they used different code, unrelated

to the programs used in the actual experiments. As metric for **tct** we measured the time before the participants localized the error in the code and provided an appropriate fix for the error. We also encouraged participants to think out loud and performed an exit interview to gather qualitative feedback.

Results



	Easy	Med	Hard
Eclipse	385.55	504.89	848.44
Viz	354.55	501.89	705.11
Ours	305.33	502.00	529.89

Figure 7: Task completion time in seconds as function of the task difficulty. The two interfaces with heap visualization perform significantly better than the standard debugger. Our system *FluidEdt* (referred to as Ours in the figure) is significantly faster than the other two in the hardest task.

We performed a two-way ANOVA which yielded stat. significant main effects for *interface* ($F_2 = 9.2, p = .0002$) *task* ($F_2 = 51.79, p = .0001$) and for the interaction *interface* $\times$ *task* ($F_4 = 5.8, p = .0004$). This suggests that the task difficulties are indeed different, that the interface has a significant effect on **tct** and that the difference between the means depends on the interaction between interface and task difficulty. Fig. 7 summarizes the mean **tct** across all tasks and interfaces. Indicating that both interfaces with heap graph visualization are faster than the standard debugger and that our tool performs better with increasing task difficulty. Note that the definition of 'difficulty' used here is not rigorous and the three tasks can not be placed equidistant along this dimension. Informally, the 3rd task (tree) seems more challenging than the others for several reasons: i) the program contains more code (>200 LOC), ii) the data structure is richer and inherently recursive, and iii) the program makes frequent use of nested operations.

Interestingly, there is evidence from the static analysis literature that trees are more difficult to reason about than lists, for two reasons: i) the invariants for trees (i.e. the predicates needed for verification) are more complicated to state and update, and ii) the number of states of a tree is much larger than a list. We believe our approach is particularly helpful for these more complex tasks due to its ability to abstract away the complexity of the data structure to allow for temporal comparison.

To further analyze these effects we performed post-hoc pairwise comparisons. These reveal that ***FluidEdt*** and **Viz** are statistically significantly faster than **Eclipse** ($p = .0036$ and $p = .0001$ respectively), allowing us to accept **H1**. While ***FluidEdt*** ($M = 452$ sec) was faster than **Viz** ($M = 487$ sec), these differences are not stat. significant ($p = .15$), and does not allow us to accept **H2**. However, the graph in Fig. 7 suggests a clear benefit in using our tool with increasing difficulty. This is also supported by the main effect on the *interface* $\times$ *task* interaction. To crystalize where these differences came from we performed two 2×2 ANOVAs, comparing *T1*vs.*T2* and *T2* vs.*T3* respectively. This time only the second ANOVA (T2$\times$T3) revealed a main effect for *interface* ($F_2 = 6.583$), showing that ***FluidEdt*** ($M = 529$ sec) is stat. significantly faster than **Viz** ($M = 705$ sec $p = .003$) for the hard task.

In summary the experimental data suggests that visualizing the heap graph of a running program does benefit debugging efficiency (**H1**) and that our mechanisms for automatic abstraction and concretization alongside the possibility to directly interact with the heap graph are indeed useful. In particular, our tool outperforms the baseline interfaces in the hard task suggesting that the utility of our mechanism increases with task difficulty.

Qualitative Results

Here we briefly summarize insights that we gathered from think aloud commentary during the debugging sessions and from the exit-interviews. Overall users reacted positively to our interface. All participants uniformly found that the visualization is helpful in understanding the data structure saying e.g., *"It helps me to form an [mental] image of the data structure"* or simply *"It helped me to find the problem in the data structure, without even looking at the code"*. Furthermore, participants commented (without a direct experimenter question) that they think this would be helpful to learn/teach basic algorithms in introductory CS classes. A total of 16 participants also commented that the tool is particularly *"useful for complex data structures"*. One participant commented that he has to deal a lot with data structures when working on OS kernel code and often finds himself drawing depictions of data structure state on paper during this work, further validating our approach beyond introductory CS education.

In terms of automatic and user driven abstraction all participants but one preferred the interface with these capabilities over the "pure" visualization. The single outlier commented that he simply wants to see all the data and found it too work intensive to concretize abstract nodes by hand. 28 out of 30 participants ended up refining the initial abstraction using the pen & touch interface with the remaining two using the mouse and keyboard only. Participants found the possibility to interact "directly" with the data compelling and thought that it allows them to more quickly reconcile their mental model of the structure with the rendering on-screen. In particular, participants often simple changed the visual layout of the graph in order to better understand the program state. While participants found the automatic abstraction useful, almost all of them would manually concretize at least one of the abstract nodes fully at the beginning of each session. When asked why they had done this they said they simply wanted to verify that the initial data structure was populated correctly with the input data. This visual check is currently laborious and points to the need for a fast way to preview the content of an abstract node without having to concretize the whole structure one-by-one.

Finally, the participants would have liked to see a tighter integration between visualization and code views. For example,

most users found that selecting an edge in the graph view should also highlight the variable in the code view. A further area for improvement that got commented on frequently was the limited code editing and navigation capabilities of our tool. In fact many participants preferred our tool to find the bug but would switch back to Eclipse to fix the bug. This suggests that there are opportunities in either integrating our approach into a regular IDE or to combine it with recent more visual approaches for code navigation (e.g., [3, 7, 14]).

Implications for information vizualisation

To the best of our knowledge, there is currently no graph layout algorithm that meets the many requirements that arise from visualizing the dynamics of an evolving data structure. Most layout algorithms have no means to enforce temporal consistency as the graph changes over time. In our case these changes can be dramatic, e.g., if a large part of the heap gets abstracted or concretized from one timestep to another. Occasionally this is not reconcilable with the automated layout algorithm and temporal consistency is broken with the algorithm rendering an entirely new graph view. This is clearly not ideal and users commented on this frequently. It is paramount to keep the parts of the structure that the user was (or is) inspecting in the same location and to preserve the relative spatial arrangement over time. This requirement necessitates entirely new graph layout algorithms that incorporate temporal consistency into their formulation. User modifications are another difficult topic. Currently we cannot guarantee that manually positioned nodes will stay in the same place when stepping through the program – an interesting item for future work.

RELATED WORK

Our work spans the two areas of HCI (developer support tools) and programming languages (e.g. program analysis). Here we briefly survey some of the most related work.

Development Tools

Both the software development process [8, 21] as well as development tools have been of long standing interest to HCI. In particular, debugging and bug isolation strategies have seen significant attention [15, 27]. In particular, Gilmore highlights the importance of understanding debugging as an essential part of program development [10].

There is now a lot of evidence in the literature that code development does not happen in isolation and programmers often encounter problems that they can only overcome with external resources [17]. Hence, recent work has focused on supporting developers in code understanding related activities where several challenges arise: i) developers need to find appropriate code examples (e.g., [13, 19]), and ii) then they need to reconcile program output and their understanding of its operation [16], and finally iii) adapt the examples to their own needs [29]. Tools have been developed that allow users to work backwards from program output in order to understand code behavior and to isolate bugs in the code [16].

Furthermore, many approaches attempt to more tightly integrate resources found on the web with programming related activities [13, 19, 28]. In particular, example driven development [4, 12] has received much attention recently. Others have tried to auto-generate useful code-snippets [9]. While in spirit these works are similar to ours, they tend to focus solely on the program code while we treat both code and data as first class citizens. This is especially critical in the context of data structure development, a fundamental area which has not received much research attention in the HCI community.

Heap Analysis and Visualization

Over the years, there has been substantial amount of work on static and dynamic program analysis with the goal of better program understanding, bug finding or verification. Representative works here are deep static analysis abstractions popularized by the TVLA approach [24] and runtime techniques for visualizing and/or abstracting the heap [1, 20].

In general, static verification is a very challenging problem often requiring significant manual annotations. And while our setting is quite different (i.e. we allow user manipulation of the abstract shape not possible in TVLA), we draw inspiration from the predicate based parameterized approach used in their work. However, existing works are typically not interactive, that is, the user cannot control the abstraction or interact with it in meaningful ways. Further, these approaches do not provide practical evidence that the particular visualization and/or hardcoded abstraction is actually useful.

We note that there are many variants on the above works found in the research literature, however from our perspective, all of these variants share the same limitations described above.

Program visualization for educational purpose

There exist a number of approaches to generate or compose fixed, mostly pre-computed animations of algorithms. These are aimed at teaching (e.g., [6, 11, 26]) and supporting code understanding purposes. Sorva et al. [25] provide an excellent overview which we refer the interested reader to.

Many of these tools have been evaluated quantitatively and qualitatively and results indicate that visualizing program behavior is beneficial for code understanding and learning. Furthermore, there seems to be evidence that interactivity and being able to use and edit own code is beneficial [25]. In our work we build on these findings.

However, our system differs from prior work in that we allow not only interact with the code but also enable complex interactions with the data (e.g. mechanisms for automatic abstraction of large data structures). This makes our approach scale to complex real-world settings.

DISCUSSION AND FUTURE WORK

Our approach bridges program analysis techniques with HCI and leverages online automatic abstraction and visualization of the data structure in a way which captures its *essential operation*, thus enabling powerful local reasoning. Local reasoning is key to understanding the core data structure algorithm operation as it strips away irrelevant details. In addition, we provide a fine grained mechanism for manually modifying the abstracted view using pen and touch gestures – this enables the developer to pause the program, carefully interact with its data, and then resume program execution, thus allowing them to try out various hypotheses.

Our experimental evaluation has shown this to be a very effective mechanism particularly for more complex structures and in combination with our user interface that gives developers interactive control over the abstraction.

Based on the quantitative and qualitative feedback we elicited we believe this is an exciting first step in the right direction. However, there are also many limitations and plentiful areas left for future research. Perhaps the most interesting and promising idea would be to extend our approach to generic and more complex programs. We believe this would require a combination of our user interface with an approach that is better suited for the debugging of control flow and program logic. Furthermore, we discovered that current graph layout algorithms are not well suited for our purposes as they do not allow for incremental and user edit preserving layouts. We believe this to be another interesting area for future research.

CONCLUSION

In this work we proposed a new approach for understanding, debugging and developing heap manipulating data structures. We implemented our approach in a tool called *FluidEdt* and evaluated it against modern debuggers. Our tool begins to address some of the most pressing challenges in developing data structures – we obtained a categorization of these after performing a thorough experimental study spanning online forums and interviews with students and staff involved with teaching data structures at our university.

Our work bridges the areas of program analysis and HCI by combining deep parametric abstraction techniques arising from the area of static analysis with interactive fine grained abstraction manipulation via pen and touch gestures. Based on our experimental user study, we believe this to be a fruitful direction towards reducing the difficulties that developers – beginners and experts alike – encounter when dealing with heap manipulating algorithms.

REFERENCES

1. Aftandilian, E. E. et al. Heapviz: interactive heap visualization for program understanding and debugging. In *Proc. SOFTVIS '10*, 2010, p.53–62.
2. Balakrishnan, G., and T. Reps. Recency-abstraction for heap-allocated storage. In *Proc. SAS'06*.
3. Bragdon, A. et al. Code bubbles: A working set-based interface for code understanding and maintenance. In *Proc. CHI '10*.
4. Brandt, J. et al. Example-centric programming: integrating web search into the development environment. In *Proc. CHI '10*.
5. Brandt, J. et al. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proc. CHI '09*.
6. Brown, M. H., and M. A. Najork. Algorithm animation using 3d interactive graphics. In *Proc. UIST '93*.
7. DeLine, R., and K. Rowan. Code canvas: Zooming towards better development environments. In *Proc. ICSE '10*.
8. Détienne, F. *Software Design–Cognitive Aspect*. Springer, 2002.
9. Galenson, J. et al. Codehint: Dynamic and interactive synthesis of code snippets. In *Proc. ICSE '14*.
10. Gilmore, D. J. Models of debugging. *Acta Psychologica 78*, 13 (1991), 151 – 172.
11. Guo, P. J. Online python tutor: Embeddable web-based program visualization for cs education. In *Proc. SIGCSE '13*.
12. Hartmann, B., M. Dhillon, and M. K. Chan. Hypersource: Bridging the gap between source and code-related web sites. In *Proc. CHI '11*.
13. Hoffmann, R., J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proc. UIST' 07*.
14. Karrer, T. et al. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proc. UIST '11*.
15. Katz, I. R., and J. R. Anderson. Debugging: An analysis of bug-location strategies. *SIGCHI Bull. 21*, 1 (Aug. 1989).
16. Ko, A. J., and B. A. Myers. Extracting and answering why and why not questions about java program output. *ACM Trans. Softw. Eng. Methodol. 20*, 2 (Sept. 2010), 4:1–4:36.
17. Ko, A. J., B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Proc. VL/HCC '04*.
18. Lieber, T., J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proc. CHI' 14*.
19. Mamykina, L. et al. Design Lessons from the Fastest Q&A site in the west. In *Proc. CHI '11*.
20. Marron, M. et al. Abstracting runtime heaps for program understanding. *Trans. Softw. Eng.*.
21. Mayer, R. E. The psychology of how novices learn computer programming. *ACM Computing Surveys (CSUR) 13*, 1 (1981), 121–141.
22. Oney, S., and J. Brandt. Codelets: Linking interactive documentation and example code in the editor. In *Proc. CHI '12*.
23. Rzeszotarski, J. M., and A. Kittur. Kinetica: Naturalistic multi-touch data visualization. In *Proc. CHI' 14*.
24. Sagiv, M., T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*.
25. Sorva, J., V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *Trans. Comput. Educ.*.
26. Stasko, J. Tango: a framework and system for algorithm animation. *Computer 23*, 9 (Sept 1990), 27–39.
27. Weiser, M. Programmers use slices when debugging. *Commun. ACM 25*, 7 (July 1982), 446–452.
28. Wightman, D. et al. Snipmatch: Using source code context to enhance snippet retrieval and parameterization. In *Proc. UIST '12*.
29. Yeh, R. B., A. Paepcke, and S. R. Klemmer. Iterative design and evaluation of an event architecture for pen-and-paper interfaces. In *Proc. UIST '08*.
30. Zeleznik, R. et al. Hands-on Math: A Page-based Multi-touch and Pen Desktop for Technical Work and Problem Solving. In *Proc. UIST '10*.

Predicate	Meaning
$p(u)$	is u pointed to by pointer p?
$r_x(u)$	is u reachable from variable x?
$r_{n,x}(u)$	is u reachable from x via field n?
$onCycle(u)$	does u belong on a cycle?
$isAncestor_x(u)$	is x an ancestor of u?
$isTreeRoot(u)$	is u the root of a tree?
$isNull(u)$	is u null?
$isList_n(u)$	does u belong on a linked list with field n?

Table 6: Pre-defined structural predicates

APPENDIX

BUILT-IN STRUCTURAL PREDICATES

In Table 6 we list the set of unary predicates used by our system. These predicate templates can be instantiated with the appropriate names used in the program. For instance, predicate p can be instantiated to predicates x and y if the program uses pointer variables x and y. Similarly for the other predicates.