Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis

Electronic Appendix

PRITAM M. GHARAT and UDAY P. KHEDKER, Indian Institute of Technology Bombay, India
ALAN MYCROFT, University of Cambridge, UK

This electronic appendix describes handling of heap and handling of function pointers.

A AUGMENTED GPU REDUCTION ALGORITHM FOR COMPUTING QUEUED GPUS

Calculating the set of GPUs Queued for dead GPU elimination can be performed parallely with edge reduction. We thus define a new algorithm given in Definition A.1 for GPU reduction which augments the method of computing set Queued.

A GPU $\boldsymbol{p} \in$ Queued could belong to any of the following categories:

- Composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ may be postponed because $\boldsymbol{p}$ may be a GPU blocked by the presence of a barrier. It is possible that the barrier which may be simplified after Δ is inlined in a caller and may not block $\boldsymbol{p}$ any more enabling its composition with $\boldsymbol{c}$.
- Composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ may be *undesirable*. It is possible that $\boldsymbol{p}$ may be simplified after Δ is inlined in a caller making the composition *desirable*.

In the first case, a GPU composition is *admissible* when RGIn is used for GPU reduction but with the GPU $\boldsymbol{p}$ being blocked ($\boldsymbol{p} \in \overline{\text{RGIn}}$), the composition is postponed. These conditions are checked at line numbers 19 and 20 in Definition A.1 and accordingly the flag *postpone* is set.

In the second case, we identify a *valid* but *undesirable* GPU composition using the predicate Undes_comp which checks that a pivot exists ($v = x$ for *TS* and $u = x$ for *SS*) and the composition is *undesirable* ($l > k$). This check is performed at line number 26 in Definition A.1 and accordingly the flag *postpone* is set.

$$\text{Undes\_comp}\left(\tau, u \xrightarrow[t]{i|j} v, x \xrightarrow[s]{k|l} y\right) = \begin{cases} \textbf{true} & (\tau = \text{ts}) \wedge (v = x) \wedge (l > k) \\ \textbf{true} & (\tau = \text{ss}) \wedge (u = x) \wedge (l > k) \\ \textbf{false} & \text{otherwise} \end{cases} \tag{1}$$

Example A.1. The set of GPUs reaching statement 04 is $\{p \xrightarrow[02]{1|0} a, x \xrightarrow[03]{2|0} b\}$. The barrier GPU (in this case $x \xrightarrow[03]{2|0} b$) blocks the GPU $p \xrightarrow[02]{1|0} a$ and hence and hence $\overline{\text{RGIn}}_{04} = \{x \xrightarrow[03]{2|0} b\}$. Thus, the composition between GPUs $\boldsymbol{c} : q \xrightarrow[04]{1|1} p$ and $\boldsymbol{p} : p \xrightarrow[02]{1|0} a$ is *admissible* if RGIn$_3$ is used for GPU reduction and the condition on line 19 of Definition A.1 is **true**. However since $\boldsymbol{p} \in \overline{\text{RGIn}_3}$, the condition on line 20 is **false** and the flag *postpone* is set to **true** indicating that a composition is postponed and $\boldsymbol{p}$ should be added to Queued as seen on line numbers 9 and 10 of the algorithm.

```
01   void Q()
02   {  p = &a;
03      *x = &b;
04      q = p;
05   }
```

```
Input:    $\boldsymbol{c}$            // The consumer GPU to be simplified
          $\mathcal{R}$            // The set of GPUs using which $\boldsymbol{c}$ is to be simplified
          $\overline{\mathcal{R}}$            // The set of GPUs that have been blocked by a barrier
Output: Red         // The set of simplified GPUs equivalent to $\boldsymbol{c}$
          Queued     // The set of GPUs which may be used later
Augmented_GPU_reduction ($\boldsymbol{c}, \mathcal{R}, \overline{\mathcal{R}}$)
{  Red = Queued = $\emptyset$
   $W = \{\boldsymbol{c}\}$
   while ($W \neq \emptyset$)
   {  extract $w$ from $W$
      for each $\gamma \in \mathcal{R}$
      {  $\langle W, tscomp, tspost \rangle$ = Compose_GPUs($ts, w, \gamma, W, \overline{\mathcal{R}}$)
         $\langle W, sscomp, sspost \rangle$ = Compose_GPUs($ss, w, \gamma, W, \overline{\mathcal{R}}$)
         if ($tspost$ or $sspost$)
            Queued = Queued $\cup \{\gamma\}$
      }
      if ($\neg$ ($tscomp$ or $sscomp$))
         Red = Red $\cup \{w\}$
   }
   return (Red , Queued)
}
Compose_GPUs($\tau, w, \gamma, W, blocked$)
{  $composed = postpone =$ false
   if ($r = w \circ \gamma$) $succeeds$
   {  if ($\gamma \notin blocked$)
      {  $W = W \cup \{r\}$
         $composed =$ true
      }
      else $postpone =$ true
   }
   else if (Undes_comp($\tau, w, \gamma$))
      $postpone =$ true
   return $\langle W, composed, postpone \rangle$
}
```

Fig. A.1. Definition of Augmented edge reduction algorithm for computing Queued GPUs

Example A.2. The composition between GPUs $\boldsymbol{c} : p \xrightarrow[3]{1|2} y$ and $\boldsymbol{p} : y \xrightarrow[4]{1|2} x$ is *undesirable* because the result of composition is a GPU $p \xrightarrow[3]{1|3} x$ whose *indlev* exceeds that of $\boldsymbol{c}$. This composition will be performed once the $\boldsymbol{p}$ is simplified. The predicate Undes_comp returns **true** because $(l > k)$

(in this case $l = 2$ and $k = 1$) indicating that the composition is *undesirable* and adds $\boldsymbol{p}$ to the set Queued.

B HANDLING HEAP FOR POINTS-TO ANALYSIS USING GPGS

So far we have created the concept of GPGs for pointers to scalars allocated on the stack or in the static area. This section extends the concepts to data *structures* containing named fields created using C style **struct** or **union** and possibly allocated on the heap (as well as on the stack or in static memory). For clarity, in this section, we show only the set of GPUs reaching a given statement and do not show the complete GPG of a procedure.

Extending GPGs to handle structures and heap-allocated data requires the following changes:

- The concept of *indlev*s is generalized to indirection lists (*indlist*s) to handle structures and heap accesses field sensitively.
- Heap locations are abstracted using allocation sites. In this abstraction, all locations allocated at a particular allocation site are treated alike. This approximation allows us to handle the unbounded nature of heap as if it were bounded [1]. Hence only weak updates can be performed on heap locations.[B.1]
- When the GPG of a procedure is being constructed, the allocation sites may appear in a caller procedure and hence may not be known. We deal with this by an additional summarization based on k-limiting to bound the accesses in a loop. Both these summarization techniques are required to create a decidable version of our method of constructing procedure summaries in the form of GPGs. The resulting points-to analysis is a precise flow-sensitive, field-sensitive, and context-sensitive analysis (relative to these two summarization techniques).[B.2]
- Introduction of *indlist*s and k-limiting summarization requires extending the concept of GPU composition to handle them.
- The allocation-site-based abstraction and k-limiting summarization may create cycles in GPUs; a simple extension to GPU reduction handles them naturally.

The optimizations performed on GPGs and the required analyses remain the same. Hence, the discussion in these sections is driven mainly by examples that illustrate how the theory developed earlier is adapted to handle structures (typically, but not necessarily, heap-allocated).

B.1 Extending GPU Composition to Indirection Lists

The *indlev* "$i|j$" of a GPU $x \xrightarrow[s]{i|j} y$ represents i dereferences of x and j dereferences of y using the dereference operator $*$. We can also view the *indlev* "$i|j$" as lists (also referred to as indirection list or *indlist*) containing i and j occurrences of $*$. This representation naturally allows field-sensitive handling of structures by using indirection lists containing field dereferences. Consider the statements $x = *y$ and $x = y \rightarrow n$ involving pointer dereferences. Since $x = y \rightarrow n$ is equivalent to $x = (*y).n$, we can represent the two statements by GPUs as shown below:

[B.1] We also perform weak updates for address-escaped variables (Section 10.1) because they share many similarities with heap locations. Like heap locations, address-escaped variables could outlive the lifetime of the procedures that create them. They potentially represent multiple concrete locations because of multiple calls to the procedure. Further, this number could be unbounded in case of recursive calls.

[B.2] In a top-down analysis, k-limiting is not required because allocation sites are propagated from callers to callees. While the use of k-limiting in a bottom-up approach seems like an additional restriction, unless the locations involved in a pointer chain are allocated by $m > k$ distinct allocation sites, there is no loss of precision compared to a top-down approach.

Pointer assignment	GPU	Remark
$x = malloc(\ldots)$	$x \xrightarrow{[*]\|[\,]} h_i$	The allocation site name is i
$x =$ NULL	$x \xrightarrow{[*]\|[\,]}$ NULL	NULL is distinguished location
$x = y.n$	$x \xrightarrow{[*]\|[n]} y$	
$x.n = y$	$x \xrightarrow{[n]\|[*]} y$	
$x = y \rightarrow n$	$x \xrightarrow{[*]\|[*,n]} y$	
$x \rightarrow n = y$	$x \xrightarrow{[*,n]\|[*]} y$	

Fig. B.1. GPUs with indirection lists (*indlist*) for basic pointer assignments in C for structures.

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\|[*,*]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{1\|2} y$
$x = y \rightarrow n$	$x \xrightarrow{[*]\|[*,n]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{[*]\|[*,n]} y$

We achieve field sensitivity by enumerating field names. Having a field-insensitive representation which does not distinguish between different fields, makes no difference for a statement $x = *y$, but loses precision for a statement $x = y \rightarrow n$. Figure B.1 illustrates the GPUs corresponding to the basic pointer assignments involving structures.

The dereference in the pointer expression $y \rightarrow n$ is represented by an *indlist* written as $[*, n]$ associated with pointer variable y. It means that, first the address in y is read and then the address in field n is read. On the other hand, the access $y.n$ as shown in the third row of Figure B.1 can be mapped to location by adding the offset of field n to the virtual address of y at compile time. Hence, it can be treated as a separate variable which is represented by a node $y.n$ with an *indlist* $[*]$. We can also represent $y.n$ with a node y and an *indlist* $[n]$. For our implementation, we chose the former representation. However, the latter representation is more convenient for explaining the GPU compositions and hence we use it in the rest of the paper. For structures,we ensure field sensitivity by maintaining *indlist* in terms of field names. We choose to handle unions field-insensitively to capture aliasing between its fields.

Recall that a GPU composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ involves balancing the *indlev* of the pivot in $\boldsymbol{c}$ and $\boldsymbol{p}$ (Section 4.2). With *indlist* replacing *indlev*, the operations remain similar in spirit, although now they become operations on lists rather than operations on numbers. To motivate the operations on *indlist*s, let us recall the operations on *indlev*s: GPU composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ requires balancing *indlev*s of the pivot which involves computing the difference between the *indlev* of the pivot in $\boldsymbol{c}$ and $\boldsymbol{p}$. This difference is then added to the *indlev* of the non-pivot node in $\boldsymbol{p}$. Recall that a GPU composition is *valid* (Section 4.2.2) only when the *indlev* of the pivot in $\boldsymbol{c}$ is greater than or equal to the *indlev* of the pivot in $\boldsymbol{p}$. For convenience, we illustrate it again in the following example.

Example B.1. Consider $\boldsymbol{p}\!:\!y \xrightarrow{1|0} x$ and $\boldsymbol{c}\!:\!w \xrightarrow{1|2} y$ where y is the pivot. Then a *TS* composition $\boldsymbol{c} \circ^{\text{ts}} \boldsymbol{p}$ is *valid* because *indlev* of y in $\boldsymbol{c}$ (which is 2) is greater than *indlev* of y in $\boldsymbol{p}$ (which is 1). The difference $(2 - 1)$ is added to the *indlev* of x (which then becomes 1) resulting in a reduced GPU $\boldsymbol{r}\!:\!w \xrightarrow{1|(2-1+0)} x$, i.e. $\boldsymbol{r}\!:\!w \xrightarrow{1|1} x$.

$$\left(z \xrightarrow[t]{il_1|il_2} x\right) \circ^{\text{ts}} \left(v \xrightarrow[u]{il_3|il_4} y\right) := \begin{cases} z \xrightarrow[t]{il_1|il_5} y & (v = x) \wedge (il_2 = il_3@il_6) \wedge (il_5 = il_4@il_6) \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\left(x \xrightarrow[t]{il_1|il_2} z\right) \circ^{\text{ss}} \left(v \xrightarrow[u]{il_3|il_4} y\right) := \begin{cases} y \xrightarrow[t]{il_5|il_2} z & (v = x) \wedge (il_1 = il_3@il_6) \wedge (il_5 = il_4@il_6) \\ & \wedge\ il_6 \neq [\,] \\ \text{fail} & \text{otherwise} \end{cases}$$

Definition B.1. *GPU Composition* $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ *using indlists*

We define similar operations for *indlist*s. A GPU composition is *valid* if the *indlist* of the pivot in GPU $\boldsymbol{p}$ is a prefix of the *indlist* of the pivot in GPU $\boldsymbol{c}$. For example, the *indlist* "$[*]$" is a prefix of the *indlist* "$[*, n]$". The addition (+) of the difference (−) in the *indlev*s of the pivot to the *indlev* of one of the other two nodes is replaced by the list-append operation denoted @.

Similarly computing the difference (−) in the *indlev* of the pivot is replaced by the 'list-difference' or 'list-remainder' operation, Remainder : *indlist* × *indlist* ⟶ *indlist*; this takes two *indlist*s as its arguments where the first is a prefix of the second and returns the suffix of the second *indlist* that remains after removing the first *indlist* from it. Given $il_2 = il_1$ @ il_3, Remainder(il_1, il_2) = il_3. When $il_1 = il_2$, the remainder il_3 is an empty *indlist* (denoted [ ]). A GPU composition is *valid* only when il_1 is a prefix of il_2; Remainder(il_1, il_2) is computed only for *valid* GPU compositions. This is again a natural generalization of the integer *indlev* formulation earlier.

Example B.2. Consider the statement sequence $y = x; w = y \rightarrow n$;. In order to compose the corresponding GPUs $\boldsymbol{p}\!: y \xrightarrow{[*]|[*]} x$ and $\boldsymbol{c}\!: w \xrightarrow{[*]|[*,n]} y$ we find the list remainder of the *indlist*s of y in the two GPUs. This operation (Remainder($[*], [*, n]$) returns $[n]$ which is appended to the *indlist* of node x (which is $[*]$) resulting in a new *indlist* $[*]$ @ $[n]$ = $[*, n]$ and thus, we get a reduced GPU $w \xrightarrow{[*]|[*,n]} x$ representing $w = x \rightarrow n$.

The formal definition of GPU composition using *indlist*s is similar to that using *indlev*s (Definition 3) and is given in Definition B.1. Note that for *TS* and *SS* compositions in the equations, the pivot is x. Besides, for *SS* composition, the condition $il_6 \neq [\,]$ (generalizing the strict inequality '<' in Definition 3) ensures that the consumer GPU does not redefine the location defined by the producer GPU. Unlike the case of pointers to scalars, *TS* and *SS* compositions are not mutually exclusive for pointers to structures. For example, an assignment $x \rightarrow n = x$ could have both *TS* and *SS* compositions with a GPU $\boldsymbol{p}$ defining x. The two compositions are independent because *SS* composition resolves the source of a GPU whereas *TS* composition resolves the target of the GPU. Hence, they can be performed in any order.

A GPU composition is *desirable* if the *indlev* of $\boldsymbol{r}$ does not exceed that of $\boldsymbol{c}$. Similarly, in the case of *indlist*s, a GPU composition is *desirable* if *indlist*s of $\boldsymbol{r}$ (say $il_1|il_2$) does not exceed that of $\boldsymbol{c}$ (say $il_1'|il_2'$), i.e. $|il_1| \leq |il_1'| \ \wedge \ |il_2| \leq |il_2'|$ where $|il|$ denotes the length of *indlist* il. Note that, for *desirability*, we only need a smaller length and not a prefix relation between *indlist*s. In fact, the *indlist* in $\boldsymbol{r}$ is always a suffix of the *indlist* in $\boldsymbol{c}$ as illustrated by the following example.

Example B.3. Consider the code snippet on right. The effect of statement 22 in the context of

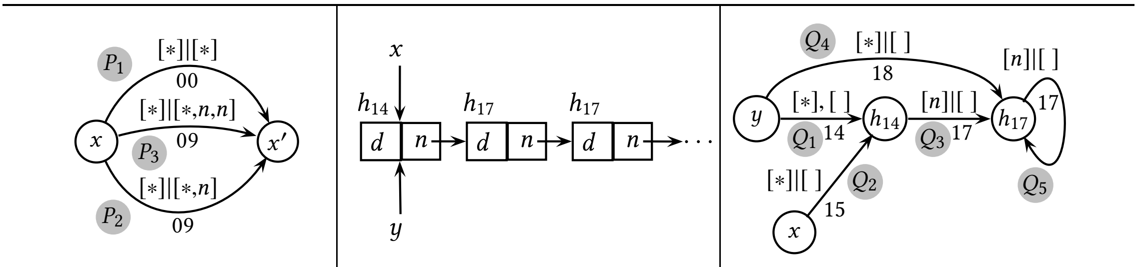
```
struct node ∗ x;

01  struct node {
02      struct node ∗ n;
03      int d;
04  };

05  void P() {
06      struct node ∗ y;
07      while (...) {
08          print x → d;
09          x = x → n;
10      }
11  }

12  void Q() {
13      struct node ∗ y;
14      y = malloc(. . .);
15      x = y;
16      while (...) {
17          y → n = malloc(. . .);
18          y = y → n;
19      }
20      P();
21  }
```

(a) A program for creating a linked list and traversing it. We have omitted the null assignment for the last node of the list and the associated GPUs



(b) $\overline{\text{RGOut}}_{11}$ (GPUs reaching the End of g for $k = 3$)

(c) Linked list created by procedure f

(d) $\overline{\text{RGIn}}_{20}$ (GPUs reaching the call to g on line 20)

Fig. B.2. An example demonstrating the need of k-limiting summarization technique in addition to allocation-site-based abstraction for the heap. h_{14} and h_{17} are the heap nodes allocated on lines 14 and 17 respectively.

statement 21 can be seen as an assignment $z = y.n$. The composition of GPUs $\boldsymbol{c}\!:\!z \xrightarrow[22]{[*]\|[*,n]} x$ and $\boldsymbol{p}\!:\!x \xrightarrow[21]{[*]\|[\,]} y$ results in the GPU $\boldsymbol{r}\!:\!z \xrightarrow[22]{[*]\|[n]} y$. The *indlist* of the target (y) of $\boldsymbol{r}$ is not a prefix of that of target (x) of $\boldsymbol{c}$ but is a suffix.

```
21 :  x  =  &y;
22 :  z  =  x → n;
```

B.2 Summarization Using Allocation Sites

Under the allocation-site-based abstraction for the heap, the objects created by an allocation statement are collectively named by the allocation site and undergo weak update. Thus, a statement $x = malloc(\ldots)$ is represented by a GPU $x \xrightarrow[i]{[*]\|[\,]} \text{h}_i$ where h_i is the heap location created at the allocation site i. Note that we have created allocation sites flow-insensitively. More precision can be obtained by cloning the allocation sites based on the context. We have left it as future work.

The example below illustrates how this bounds an unbounded heap in a GPG. For convenience, we identify GPUs using procedure names i.e., $P_i,\ 1 \le i \le 4$, denote the GPUs of procedure P whereas $Q_i,\ 1 \le i \le 9$, denote the GPUs of procedure Q.

Example B.4. For procedure f shown in Figure B.2 we create heap objects h_{14} and h_{17} allocated at line numbers 14 and 17. The GPU set $\overline{\mathsf{RGIn}}_{20}$ in procedure f represents a linked list with x as its head pointer (Figure B.2(d)) and h_{14} as its first node. The remaining nodes in the list are represented by the heap location h_{17} and are summarized by a self-loop over the node. This set of GPUs is computed as follows: The GPU $Q_1 : y \xrightarrow[14]{[*]\|[\,]} h_{14}$ is created for allocation-site 14. The GPU $x \xrightarrow[15]{[*]\|[*]} y$ composes with Q_1 (under *TS* composition) to create a new GPU $Q_2 : x \xrightarrow[15]{[*]\|[\,]} h_{14}$. When statement 17 is processed for the first time, GPU $y \xrightarrow[17]{[*,n]\|[\,]} h_{17}$ composes with Q_1 (under *SS* composition) to create a GPU $Q_3 : h_{14} \xrightarrow[17]{[n]\|[\,]} h_{17}$. When statement 18 is processed for the first time, the GPU $y \xrightarrow[18]{[*]\|[*,n]} y$ composes with Q_1 (under *TS* composition) to create a GPU $y \xrightarrow[18]{[*]\|[n]} h_{14}$ which is further composed with Q_3 (under *TS* composition) to create a GPU $Q_4 : y \xrightarrow[18]{[*]\|[\,]} h_{17}$. GPU Q_4 kills GPU Q_1 because y is redefined by statement 18. This completes the first iteration of the loop and the set of GPUs $\overline{\mathsf{RGOut}}_{19}$ is $\{Q_2, Q_3, Q_4\}$ representing the following information:

- Q_2 indicates that x points to the head of the linked list.
- Q_3 indicates that the field n of heap location h_{14} points to heap location h_{17}.
- Q_4 indicates that y points to heap location h_{17}.

In the second iteration of the reaching GPUs analysis over the loop, $\overline{\mathsf{RGOut}}_{15}$ and $\overline{\mathsf{RGOut}}_{19}$ are merged to compute $\overline{\mathsf{RGIn}}_{16}$ as $\{Q_1, Q_2, Q_3, Q_4\}$. When statement 17 is processed for the second time, the GPU $y \xrightarrow[17]{[*,n]\|[\,]} h_{17}$ composes with

- Q_1 (under *SS* composition) to create Q_3, and with
- Q_4 (under *SS* composition) to create $Q_5 : h_{17} \xrightarrow[17]{[n]\|[\,]} h_{17}$.

When statement 18 is processed for the second time, Q_4 is recreated killing Q_1. This completes the second iteration of the loop and the set of GPUs $\overline{\mathsf{RGIn}}_{20}$ is $\{Q_1, Q_2, Q_3, Q_4, Q_5\}$. The new GPU Q_5 implies that the field n of heap location h_{17} holds the address of heap location h_{17}. The self loop represents an unbounded list $\left(h_{17} \xrightarrow{n} h_{17} \xrightarrow{n} h_{17} \xrightarrow{n} h_{17} \ldots\right)$ under the allocation-site-based abstraction. The third iteration of reaching GPUs analysis over the loop does not add any new information and reaching GPUs analysis reaches a fixed point.

The following example discusses the absence of blocking in the procedures in Figure B.2.

Example B.5. The GPUs in $\overline{\mathsf{RGIn}}_{14}$ reach statement 17 unblocked because there is no barrier. Since the pointee of y is available, the set $\overline{\mathsf{RGGen}}_{14}$ does not contain any indirect GPUs and hence do not contribute to the blocking of any GPUs. If the allocation site at statement 14 was not available, then the GPU for statement 17 would not have been reduced and hence the set $\overline{\mathsf{RGGen}}_{17}$ would contain an indirect GPU $y \xrightarrow[17]{[*,n]\|[\,]} h_{17}$. This GPU would block all GPUs in $\overline{\mathsf{RGIn}}_{18}$ and in turn would be blocked by the GPUs in $\overline{\mathsf{RGGen}}_{18}$ so that it cannot be used for reduction of any successive GPUs.

B.3 Summarization Using k-Limiting

This section shows why allocation-site-based abstraction is not sufficient for a bottom-up points-to analysis although it serves the purpose well in a top-down analysis.

B.3.1 The Need for k-Limiting. In some cases, the allocation site may not be available during the construction of the GPG of a procedure. For our example in Figure B.2, when the GPG is constructed for procedure g, we do not know the allocation site because the accesses to heap in procedure g refer to the data-structure created in procedure f. Thus allocation-site-based abstraction is not applicable for procedure g and the indirection lists grow without bound.

In a top-down analysis, k-limiting is not required because allocation sites are propagated from callers to callees.

Example B.6. When the GPG for procedure g in Figure B.2 is constructed, we have a boundary definition $P_1 : x \xrightarrow[00]{[*]\|[*]} x'$ at the start of the procedure. In the first iteration of the analysis over the loop, the GPU $x \xrightarrow[09]{[*]\|[*,n]} x$ composes with P_1 (under *TS* composition) creating a reduced GPU $P_2 : x \xrightarrow[09]{[*]\|[*,n]} x'$. The GPU P_2 kills GPU P_1 because x is redefined by statement at 09. However, the merge at the top of the loop reintroduces it. In the second iteration, the GPU $x \xrightarrow[09]{[*]\|[*,n]} x$ composes with P_1 to recreate P_2, and with P_2 to create $P_3 : x \xrightarrow[09]{[*]\|[*,n,n]} x'$. In the third iteration, we get an additional GPU $P_4 : x \xrightarrow[09]{[*]\|[*,n,n,n]} x'$ apart from P_2 and P_3. This continues and the indirection lists of the GPUs between x and x' grow without bound leading to non-termination.

There are two ways of handling traversals of data structures created in some other procedure.

- As the above example illustrates, we perform compositions involving upwards-exposed variables in spite of these compositions being *valid* but *undesirable*.
- Alternatively, we can postpone these compositions (as suggested before) until call inlining enables their reduction.

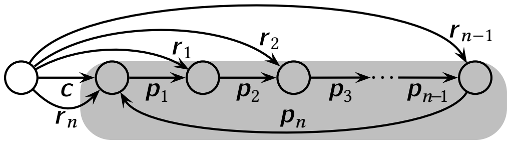
We use the first approach and bound the length of indirection lists using k-limiting. This limits the participation of the GPUs in the fixed-point computation for the procedures containing them. The second approach requires the GPUs to participate in the fixed-point computations for the callers as well. This could cause inefficiency.

While the use of k-limiting in a bottom-up approach seems like an additional restriction, unless the locations involved in a pointer chain are allocated by $m > k$ distinct allocation sites, there is no loss of precision compared to a top-down approach.

B.3.2 Incorporating k-Limiting. We limit the length of *indlist*s to k such that the *indlist* is exact up to $k - 1$ dereferences and approximate for k or more dereferences in terms of an unbounded number of dereferences. Besides, the dereferences are field-insensitive beyond k. This summarization is implemented by redefining the list concatenation operator @ such that for il_1 @ il_2, the result is a k-limited prefix of the concatenation of il_1 and il_2.

Example B.7. The set of GPUs $\overline{\text{RGOut}_{11}}$ reaching the End of procedure g of Figure B.2, for $k = 3$ is given in the Figure B.2(b). A GPU between x and x' has an *indlist* $[*, n]$ of length 2 and all *indlist*s of length ≥ 3 are approximated by $[*, n, n]$.

GPU $P_1 : x \xrightarrow[00]{[*][*]} x'$ in the GPG for procedure g represents the effect of **while** loop not executed even once. GPU $P_2 : x \xrightarrow[09]{[*]\|[*,n]} x'$ represents the effect of the first iteration of the **while** loop. The GPU $P_3 : x \xrightarrow[09]{[*]\|[*,n,n]} x'$ represents the combined effect of the second and all subsequent iterations



- The shaded part shows the GPUs in $\overline{\text{RGIn}}$.
- Let $\boldsymbol{r}_0 = \boldsymbol{c}$. Then $\boldsymbol{r}_i = \boldsymbol{r}_{i-1} \circ^{\tau} \boldsymbol{p}_i$, $i > 0$.
- For simplicity, the directions chosen in the GPUs illustrate only *TS* compositions.

Fig. B.3. Series of compositions and its consequence when the graph induced by the GPUs in $\overline{\text{RGIn}}$ (shown by the shaded part) has a cycle. The compositions may happen more than the required number of times, resulting in a points-to edge.

of the **while** loop. The GPG of procedure g (Δ_g) contains a single GPB which in turn contains a set of GPUs $\{P_2, P_3\}$.

Note that an explicit summarization is required only for heap locations and address-escaped stack locations in recursive procedures because the *indlist*s can grow without bound only in these cases (see Footnote B.1).

The GPU composition defined in Section B.1 (Definition B.1) is extended to handle k-limited *indlist*s in the following manner: The removal of a prefix from a k-limited *indlist* in the Remainder operation is over-approximated by suffixing special field-insensitive dereferences denoted by "†" where † represents any field. For an operation Remainder(il_1, il_2), il_1 must be a prefix of il_2 as explained in Section B.1. Let $il_2 = il_1 \mathbin{@} il_3$ for Remainder(il_1, il_2). We define a summarized list-remainder operation sRemainder : *indlist* × *indlist* ⟶ 2^{indlist} which takes two *indlist*s as its arguments and computes a set of *indlist*s as shown below:

$$\text{sRemainder}(il_1, il_2) = \begin{cases} \{il_3 \mid il_2 = il_1 \mathbin{@} il_3\} & |il_2| < k \\ \{il_3 \mathbin{@} \sigma \mid il_2 = il_1 \mathbin{@} il_3, \sigma \text{ is a sequence of } \dagger, 0 \le |\sigma| \le |il_1|\} & \text{otherwise} \end{cases}$$

Observe that sRemainder is a generalization of Remainder defined in Section B.1 because it computes a set of *indlist*s when its second argument is a k-limited *indlist*; for non k-limited *indlist*, sRemainder returns a singleton set. The longest *indlist* in the set computed by sRemainder represents a summary whereas the other *indlist*s are exact in length but approximate in terms of fields because of field insensitivity introduced by †.[B.3] This is illustrated in the example below.

Example B.8. For $k = 3$, some examples of the sets of *indlist*s computed by the sRemainder operation are shown below:

$$\text{sRemainder}([*], [*, n, n]) = \{[n, n], [n, n, \dagger]\}$$
$$\text{sRemainder}([*, n], [*, n, n]) = \{[n], [n, \dagger], [n, \dagger, \dagger]\}$$
$$\text{sRemainder}([*, n, n], [*, n, n]) = \{[\ ], [\dagger], [\dagger, \dagger], [\dagger, \dagger, \dagger]\}$$

For the last case, the sRemainder operation can be viewed as an operation that creates an intermediate set $S = \{[*, n, n], [*, n, n, \dagger], [*, n, n, \dagger, \dagger], [*, n, n, \dagger, \dagger, \dagger]\}$ obtained by adding upto 3 occurrences of † (because $k = 3$). The sRemainder operation can then be viewed as a collection of Remainder($[*, n, n], \sigma$) for each σ in this set:

$$\text{sRemainder}([*, n, n], [*, n, n]) = \{\text{Remainder}([*, n, n], \sigma) \mid \sigma \in S\}$$

The first two cases in this example can also be explained in a similar manner.

[B.3] This is somewhat similar to materialization [3] which extracts copies out of summary representation of an object to create some exact objects.

```
Input: $\boldsymbol{c}$              // The consumer GPU to be simplified
       $\mathcal{R}$              // The context (set of GPUs) in which $\boldsymbol{c}$ is to be simplified
       Used           // The set of GPUs used for GPU reduction for a GPU
Output: Red          // The set of simplified GPUs equivalent to $\boldsymbol{c}$
    GPU_reduction ($\boldsymbol{c}, \mathcal{R}$, Used)
    {  Red = $\emptyset$
       composed = false
       for each $\gamma \in (\mathcal{R} -$ Used)
       {  for each $r \in (\boldsymbol{c} \circ^{ts} \gamma)$
          {  Red = Red $\cup$ GPU_reduction ($r, \mathcal{R}$, Used $\cup$ $\{\gamma\}$)
             composed = true
          }
          for each $r \in (\boldsymbol{c} \circ^{ss} \gamma)$
          {  Red = Red $\cup$ GPU_reduction ($r, \mathcal{R}$, Used $\cup$ $\{\gamma\}$)
             composed = true
          }
       }
       if ($\neg$ composed )
          Red = Red $\cup$ $\{\boldsymbol{c}\}$
       return Red
    }
```

Definition B.2. *GPU Reduction $\boldsymbol{c} \circ \mathcal{R}$ for Handling Heap*

GPU composition using *indlevs* (Section 4.2.2) or using *indlists* (Section B.1) is a partial operation defined to compute a single GPU as its result when it succeeds. Since we do not have a representation for an "invalid" GPU, we model failure by defining GPU composition as a partial function for GPUs containing *indlevs* or non-k-limited *indlists*. However, when *indlists* are summarized using k-limiting, sRemainder naturally computes a set of *indlists* (unlike Remainder which computes a single *indlist*). This allows us to define GPU composition as a total function, since we can express the previous partiality simply by returning an empty set.

B.4 Extending GPU Reduction to Handle Cycles in GPUs

In the presence of a heap, the graph induced by the set of GPUs reaching a GPB can contain cycles of the following two kinds:

- Cycles arising out of creation of a recursive data structure in a procedure under allocation-site-based abstraction. This manifests itself in the form of a cycle involving heap nodes h_i as illustrated in Example B.4 in Section B.2. These cycles are closed form representations of acyclic unbounded paths in the memory.
- Cycles arising out of cyclic data structures. These cycles represent cycles in the memory.

Both these cases of cycles are handled by GPU composition using sRemainder operation over indirection lists. Definition B.2 extends the algorithm for GPU reduction to use the new definition of GPU composition which computes a set of GPUs instead of a single GPU.

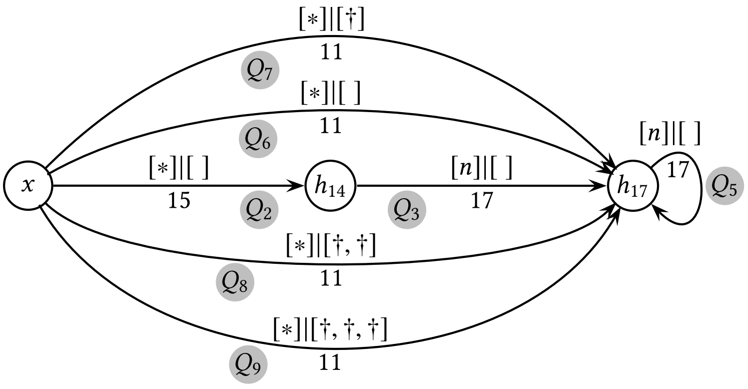


Fig. B.4. The set of GPUs $\overline{\text{RGOut}}_{20}$ after the call to procedure g in procedure f of Figure B.2. Local variable y has been eliminated.

For GPU reduction $\boldsymbol{c} \circ \mathcal{R}$, an *admissible* composition $\boldsymbol{r}_1 = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}_1$ (where $\boldsymbol{p}_1 \in \overline{\text{RGIn}}$) may lead to another composition $\boldsymbol{r}_2 = \boldsymbol{r}_1 \circ^{\tau} \boldsymbol{p}_2$ (where $\boldsymbol{p}_2 \in \overline{\text{RGIn}}$). This in turn may lead to another composition thereby creating a chain of compositions. If the graph induced by the reaching GPUs (i.e. GPUs in $\overline{\text{RGIn}}$) has a cycle (as illustrated in Example B.4 in Section B.2), some $\boldsymbol{p}_m$ must be adjacent to $\boldsymbol{p}_1$ with the length of the cycle being $m + 1$ as illustrated in Figure B.3. The lengths of *indlist*s in $\boldsymbol{r}_i$ would be smaller than (or equal to) those in $\boldsymbol{r}_{i-1}$ because of *admissibility*. If the length of an *indlist* in $\boldsymbol{c}$ exceeds m, the series of compositions would resume with $\boldsymbol{p}_1$ after the composition with $\boldsymbol{p}_m$. In other words, after computing $\boldsymbol{r}_{m-1}$ using the composition $\boldsymbol{r}_{m-2} \circ \boldsymbol{p}_m$, the next GPU $\boldsymbol{r}_m$ would be computed using the composition $\boldsymbol{r}_{m-1} \circ \boldsymbol{p}_1$ and the process will continue until some $\boldsymbol{r}_j$, $j \geq m$ is a points-to edge.[B.4] Thus, we will have more compositions than required and the result of GPU reduction may not represent the updates of locations that are updated by the original GPU $\boldsymbol{c}$. In order to prohibit this, we allow a GPU $\boldsymbol{p}$ to be used only once in a chain of compositions.

Hence, the new definition of GPU reduction (Definition B.2) uses an additional argument, Used, which maintains a set of GPUs that have been used in a chain of GPU compositions. For the top level non-recursive call to GPU_reduction, Used = $\emptyset$. In the case of pointers to scalars, a graph induced by a set of GPUs cannot have a cycle, hence a GPU $\boldsymbol{p}$ cannot be used multiple times in a series of GPU compositions. Therefore, we did not need set Used for defining GPU reduction in the case of pointers to scalars (Definition 4).

Example B.9. This example illustrates GPU reduction with 3-limited *indlist*s using GPU P_3 of Δ_g shown in Figure B.2(b). At the call site 20 in procedure f of Figure B.2(a), the upwards-exposed variable x' in Δ_g is substituted by x in Δ_f (see Section 7). All GPU compositions for this examples are *TS* compositions. The GPUs in $\overline{\text{RGIn}}_{20}$ (Figure B.2(d)) are used for composition. The set $\overline{\text{RGOut}}_{20}$ is same as $\overline{\text{RGOut}}_{21}$ shown in Figure B.4 except that $\overline{\text{RGOut}}_{20}$ also contains the GPUs involving y which is a local variable of f and is not in the scope of the caller procedures.

[B.4] Note that this happens for reducing a single GPU $\boldsymbol{c}$ in the context of $\overline{\text{RGIn}}$ and does not require a cycle in the GPG.

The GPU composition $P_2 \circ Q_2$ for $Q_2 : x \xrightarrow[15]{[*]|[\,]} h_{14}$ and $P_2 : x \xrightarrow[11]{[*]|[*,n]} x$ (with x substituting for x') creates a reduced GPU $x \xrightarrow[11]{[*]|[n]} h_{14}$ which is further composed with $Q_3 : h_{14} \xrightarrow[17]{[n]|[\,]} h_{17}$ to create a reduced GPU $Q_6 : x \xrightarrow[11]{[*]|[\,]} h_{17}$ (Figure B.4).

Now GPU P_3 must be composed with Q_2, Q_3 and Q_5. The composition $P_3 \circ Q_2$ for $P_3 : x \xrightarrow[11]{[*]|[*,n,n]} x$ creates two GPUs $x \xrightarrow[11]{[*]|[n,n]} h_{14}$ and $x \xrightarrow[11]{[*]|[n,n,\dagger]} h_{14}$. The newly created GPU $x \xrightarrow[11]{[*]|[n,n]} h_{14}$ is further composed with Q_3 to create GPU $x \xrightarrow[11]{[*]|[n]} h_{17}$ which is further composed with Q_5 to recreate GPU $Q_6 : x \xrightarrow[11]{[*]|[\,]} h_{17}$. The GPU composition between the other newly created GPU $x \xrightarrow[11]{[*]|[n,n,\dagger]} h_{14}$ and Q_3 creates GPUs $x \xrightarrow[11]{[*]|[n,\dagger]} h_{17}$ and $x \xrightarrow[11]{[*]|[n,\dagger,\dagger]} h_{17}$. The GPU $x \xrightarrow[11]{[*]|[n,\dagger]} h_{17}$ further composes with Q_5 creating a GPU $Q_7 : x \xrightarrow[11]{[*]|[\dagger]} h_{17}$ while the composition between GPUs $x \xrightarrow[11]{[*]|[n,\dagger,\dagger]} h_{17}$ and Q_5 creates two reduced GPUs $Q_8 : x \xrightarrow[11]{[*]|[\dagger,\dagger]} h_{17}$ and $Q_9 : x \xrightarrow[11]{[*]|[\dagger,\dagger,\dagger]} h_{17}$.

Note that GPU Q_5 is used only once in a series of compositions (Example B.10 explains this).

The final reduced GPUs Q_6, Q_7, Q_8, and, Q_9 are members of the set $\overline{\mathsf{RGOut}}_{21}$ containing the GPUs reaching the End of procedure f (as shown in Figure B.4). These reduced GPUs represent the following information:

- Q_6 implies that x now points-to heap location h_{17}.
- Q_7 imply that x points-to heap locations that are one dereference away from h_{17}.
- Q_8 imply that x points-to heap locations that are two dereferences away from h_{17}.
- Q_9 imply that x points-to heap locations that are beyond two dereferences from h_{17}.

Thus, x points to every node in the linked list.

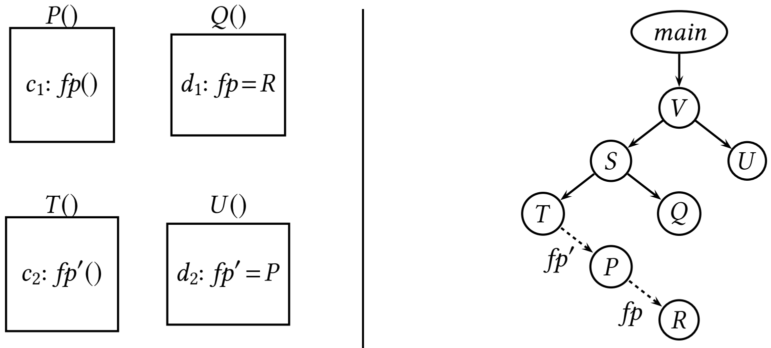
Example B.10. To see why GPU reduction in Definition B.2 excludes a GPU used for composition once, observe that GPUs Q_7, Q_8 and Q_9 can be further composed with GPU Q_5. The composition of Q_7 with Q_5 creates GPU Q_6. Similarly, repetitive compositions of Q_8 with Q_5 also creates GPU Q_6. This indicates that x points to only h_{17} and misses out the fact that x points to every location in the linked list which is represented by h_{17} and is represented by GPUs Q_7, Q_8 and Q_9.

A cycle in a graph induced by a set of GPUs could also occur because of a cyclic data structure.

Example B.11. Let an assignment $y \to n = x$ be inserted in procedure f after line 19 in Figure B.2. This creates a circular linked list instead of a simple linked list. This will cause inclusion of the GPU $h_{17} \xrightarrow{[n]|[\,]} h_{14}$ in Figure B.2(d), thereby creating a cycle between the nodes h_{14} and h_{17}.

C HANDLING CALLS THROUGH FUNCTION POINTERS

Recall that in the case of recursion, we may have incomplete GPGs because the GPGs of the callees are incomplete. Similarly, in the presence of a call through a function pointer, we have incomplete GPGs for a different reason—the callee procedure of such a call is not known. We model a call through function pointer (say *fp*) at call site s as a use statement with a GPU $\boldsymbol{u} \xrightarrow[s]{1|1} fp$ (Section 8).



P has an indirect call on a function pointer defined in Q and T has an indirect call on a function pointer defined in U. S is the closest common ancestor of P and Q whereas V is the closest common ancestor of T and U.

The fact that fp' points to R becomes known only after Δ_T and Δ_Q are inline in Δ_S, which in turn is inlined in Δ_V along with Δ_U. Then, Δ_P is inlined in Δ_V. Since Δ_V also contains Δ_Q, fp becomes known and Δ_R is then inlined in Δ_V. In the second round, Δ_R is inlined in Δ_P which is then inlined in Δ_T.

Fig. C.1. Function pointer resolution. The dashed edges indicate the calls through function pointers that are added in the call graph during the analysis. Initially, Δ_P cannot be inlined in Δ_T because fp' is not known. Similarly Δ_R cannot be inlined in Δ_P.

Our goal is to convert a call through a function pointer into a direct call for every pointee of the function pointer. Let procedure P contain an indirect call $c_1{:}\mathit{fp}()$ through function pointer fp and let procedure Q contain a definition $d_1{:}\mathit{fp} = R$ of fp such that d_1 reaches c_1 in P. Call c_1 is represented by GPU $\boldsymbol{u}\xrightarrow[s]{1|1}\mathit{fp}$ where $\boldsymbol{u}$ is a use node (Section 8) and definition d_1 is represented by gpu of the form $\mathit{fp}\xrightarrow[s]{1|0}R$ where R is the callee procedure. Call c_1 is resolved through GPU composition between the two GPUs by inlining Δ_P and Δ_Q into procedure S that is a common ancestor of both P and Q. As a special case, S could be same as either P or Q. Until c_1 is resolved, the corresponding GPU $\boldsymbol{u}\xrightarrow[s]{1|1}\mathit{fp}$ acts as a barrier that postpones the composition between the GPUs across it. After c_1 is resolved, the indirect call converts to a set of direct calls which are handled as explained in Section 7. In other words, inlining of the GPG of a callee of an indirect call occurs only in the GPG of a procedure where the call is resolved through GPU reduction and not necessarily in its immediate caller procedure.

Note that this differs from the call inlining for direct calls in that the GPG of a callee through a direct call is inlined first into its (immediate) caller procedures and is only then inlined into an ancestor procedure as a part of the GPG of the caller procedure. If we were to use the same strategy of inlining indirect calls into immediate caller procedures first, function pointer resolution may need as many rounds of bottom-up GPG construction as the maximum number of indirect calls in any call chain. However, since we allow inlining in an ancestor, we can resolve all indirect calls in a call chain in a single round beginning with the indirect call closest to *main*.

Example C.1. In Figure C.1, GPGs of procedures T and Q are inlined in the GPG of procedure S thereby introducing the definition d_1 of function pointer fp and an indirect call at call site c_2 through function pointer fp'. The indirect call remains unresolved because the pointee of

fp' is not available in S. When the GPGs of procedures S and and U are inlined in the GPG of procedure V, the definition d_2 of function pointer fp' and its use (indirect call at call site c_2) are reachable in V. Strength reduction resolves the indirect call to a direct call to procedure P which gets inlined in the GPG of procedure V. This inlining leads to the indirect call through function pointer fp to be hoisted to V where the definition d_1 of fp is already hoisted along the call chain $main \rightarrow V \rightarrow S \rightarrow Q$. This GPU reduction resolves the indirect call to a direct call to procedure R. Hence, the GPG of procedure R is inlined in procedure V.

Note that points-to information for the use GPU corresponding to the indirect call is recorded (Section 8) when the indirect call is resolved. This points-to information is then used in the second round to resolve the indirect calls that were not resolved in the first round of GPG construction.

Example C.2. In Figure C.1, GPGs Δ_P, Δ_T, and Δ_S contain unresolved indirect calls through function pointers fp, fp', and fp' respectively thereby leaving their GPGs incomplete in the first round of GPG construction. The indirect call through fp' is resolved when incomplete GPG of S and GPG of U is inlined in V and is converted to a direct call to P. Thus, the points-to information $fp' \xrightarrow[c_2]{1|0} P$ is recorded. Similarly, when incomplete GPG of P is inlined in V and through strength reduction, the indirect call through fp is converted to a direct call to R, the points-to information $fp \xrightarrow[c_1]{1|0} R$ is recorded. This points-to information is used in the second round of GPG construction to resolve all the indirect calls that were unresolved in the first round. Thus, in the incomplete GPG of procedure P, the GPG of R is inlined. Similarly, the GPGs of procedures T and S has procedure P and in turn R inlined in them.

REFERENCES

[1] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. https://doi.org/10.1145/2931098

[2] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings.* https://doi.org/10.1007/978-3-540-27864-1_14

[3] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1998. Solving Shape-analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 20, 1 (Jan. 1998), 1–50. https://doi.org/10.1145/271510.271517