Optimizing Compilers

Alias Analysis

Jens Knoop, Markus Schordan, Gergö Barany

Institut für Computersprachen
Technische Universität Wien

Aliasing Everywhere

Answers to the question "What is an alias?" in different areas:

- A short, easy to remember name created for use in place of a longer, more complicated name; commonly used in e-mail applications. Also referred to as a "nickname".
- A hostname that replaces another hostname, such as an alias which is another name for the same Internet address. For example, www.company.com could be an alias for server03.company.com.
- A feature of UNIX shells that enables users to define program names (and parameters) and commands with abbreviations. (e.g. alias ls `ls -l`)
- In MGI (Mouse Genome Informatics), an alternative symbol or name for part of the sequence of a known gene that resembles names for other anonymous DNA segments. For example, D6Mit236 is an alias for Cftr.

Aliasing in Programs

In programs aliasing occurs when there exists more than one ac path to a storage location.

An access path is the l-value of an expression that is constructe variables, pointer dereference operators, and structure field op operators.

Java (References)	C++ (References)

`A a,b;`
`a = new A();`
`b = a;`
`b.val = 0;`
	`A& a = *new A();`
	`A& b = a;`
	`b.val = 0;`

C++ (Pointers)	C (Pointers)

`A* a; A* b;`
`a = new A();`
`b = a;`
`b->val = 0;`
	`A *a, *b;`
	`a = (A*)malloc(sizeof(A`
	`b = a;`
	`b->val = 0;`

Examples of Different Forms of Aliasing

- Pascal, Modula 2/3, Java:
  - Variable of a reference type is restricted to have either the value nil/null or to refer to objects of a particular specified type.
  - An object may be accessible through several references at once, but it cannot both have its own variable name *and* be accessible through a pointer.
- C:
  - The union type specifier allows to create static aliases. A union type may have several fields declared, all of which overlap in (= share) storage.
  - It is legal to compute the address of an object with the & operator (statically, automatically, or dynamically allocated).
  - Allows arithmetic on pointers and considers it equivalent to array indexing

Relevance of Alias Analysis to Optimization

Alias analysis refers to the determination of storage locations that may be accessed in two or more ways.

- Ambiguous memory references interfere with an optimizer's ability to improve code.
- One major source of ambiguity is the use of pointer-based values.

Goal: determine for each pointer the set of memory locations to which it may refer.

Without alias analysis the compiler must assume that each pointer can refer to any addressable value, including

- any space allocated in the run-time heap
- any variable whose address is explicitly taken
- any variable passed as a call-by-reference parameter

Characterization of Aliasing

Flow-insensitive information: Binary relation on the variables in a procedure, $alias \in Var \times Var$ such that x $alias$ y if and only if

- may possibly at different times refer to the same memory location.
- must throughout the execution of the procedure refer to same memory location.

Flow-sensitive information: A function from program points and variables to sets of abstract storage locations. $alias(p, v) = Loc$ means that at program point p variable v

- may refer to any of the locations in Loc.
- must refer to the location $l \in Loc$ with $|Loc| \leq 1$.

Representation of Alias Information

Representation of aliasing with pairs:

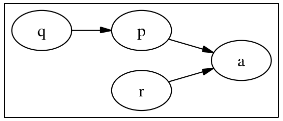
	`q=&p; p=&a; r=&a;`

complete alias pairs	`<*q,p>, <*p,a>, <*r,a>,<**q,*p>,`
	`<**q,a>,<*p,*r>,<**q,*r>`

compact alias pairs	`<*q,p>, <*p,a>, <*r,a>`
points-to relations	`(q,p),(p,a),(r,a)`

Representation of alias information and the shapes of data structures:

- graphs
- regular expressions
- 3-valued logic



Questions about Heap Contents (1)

Let *execution state* mean the set of cells in the heap, the connections between them (via pointer components of heap cells) and the values of pointer variables in the store.

NULL pointers. Does a pointer variable or a pointer component of a heap cell contain NULL at the entry to a statement that dereferences the pointer or component?

- Yes (for every state). Issue an error message
- No (for every state). Eliminate a check for NULL.
- Maybe. Warn about the potential NULL dereference.

Memory leak. Does a procedure or a program leave behind unreachable heap cells when it returns?

- Yes (in some state). Issue a warning.

Questions about Heap Contents (2)

Aliasing. Do two pointer expressions reference the same heap

- Yes (for every state).
  - trigger a prefetch to improve cache performance
  - predict a cache hit to improve cache behavior predic
  - increase the sets of uses and definitions for an improve liveness analysis
- No (for every state). Disambiguate memory references c improve program dependence information.

Sharing. Is a heap cell shared? (within the heap)

- Yes (for some state). Warn about explicit deallocation, b the memory manager may run into an inconsistent state
- No (for every state). Explicitly deallocate the heap cell w the last pointer to ceases to exist.

Questions about Heap Contents (3)

Reachability. Is a heap cell reachable from a specific variable or from any pointer variable?

- Yes (for every state). Use this information for program verification.
- No (for every state). Insert code at compile time that collects unreachable cells at run-time.

Disjointness. Do two data structures pointed to by two distinct pointer variables ever have common elements?

- No (for every state). Distribute disjoint data structures and their computations to different processors.

Cyclicity. Is a heap cell part of a cycle?

- No (for every state). Perform garbage collection of data structures by reference counting. Process all elements in an acyclic linked list in a doall-parallel fashion.

Shape Analysis

The aim of shape analysis is to determine a finite representation of heap allocated data structures which can grow arbitrarily large.

It can determine the possible shapes data structures may take such as:

- lists
- trees
- directed acyclic graphs
- arbitrary graphs
- properties such as whether a data structure is or may be cyclic

As example we shall discuss a precise shape analysis (from PoPA Ch 2.6) that performs strong update and uses shape graphs to represent heap allocated data structures. It emphasises the analysis of list like data structures.

Strong Update

Here "strong" means that an update or nullification of a pointe expression allows one to *remove* (kill) the existing binding befor adding a new one (gen).

We shall study a powerful analysis that achieves

- Strong nullification
- Strong update

for destructive updates that destroy (overwrite) existing values i pointer variables and in heap allocated data structures in gene

Examples:

- $[x := nil]^\ell$
- $[x.sel_1 := y.sel_2]^\ell$

Extending the WHILE Language

We extend the WHILE-language syntax with constructs that allow to create cells in the heap.

- the cells are structured and may contain values as well as pointers to other cells
- the data stored in cells is accessed via selectors; we assume that a finite and non-empty set Sel of selector names is given:

$$sel \in \mathsf{Sel} \qquad \text{selector names}$$

- we add a new syntactic category

$$p \in \mathsf{PExp} \qquad \text{pointer expressions}$$

- op_r is extended to allow for testing of equality of pointers
- unary operations op_p on pointers (e.g. is-null) are added

Abstract Syntax of Pointer Language

The syntax of the while language is extended to have:

$$\begin{array}{lcl}
p & ::= & x \mid x.sel \mid \text{null} \\
a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\
b & ::= & \text{true} \mid \text{false} \mid \text{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \\
S & ::= & [\text{p:=a}]^\ell \mid [\text{skip}]^\ell \\
 & & \mid \text{if}\ [b]^\ell\ \text{then}\ S_1\ \text{else}\ S_2 \\
 & & \mid \text{while}[b]^\ell\ \text{do}\ S\ \text{od} \\
 & & \mid [\text{new}\ (p)]^\ell \\
 & & \mid S_1; S_2
\end{array}$$

In the case where p contains a selector we have a destructive update of the heap. Statement new creates a new cell pointed to by p.

Shape Graphs

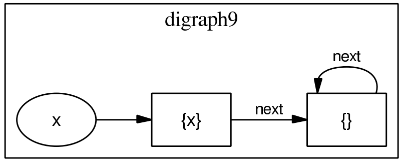
We shall introduce a method for combining the locations of the semantics into a finite number of *abstract locations*.

The analysis operates on shape graphs (S, H, is) consisting of:

- an abstract state, S (mapping variables to abstract locatio
- an abstract heap, H (specifying links between abstract loca
- sharing information, is, for the abstract locations.

The last component allows us to recover some of the imprecisio introduced by combining many locations into one abstract loc

Example



$g_9 = (\mathsf{S}, \mathsf{H}, \mathsf{is})$ where

$\mathsf{S} = \{(\mathtt{x}, n_{\{\mathtt{x}\}})\}$

$\mathsf{H} = \{(n_{\{\mathtt{x}\}}, \mathtt{next}, n_\emptyset), (n_\emptyset, \mathtt{next}, n_\emptyset)\}$

$\mathsf{is} = \emptyset$

Abstract Locations

The *abstract locations* have the form n_X where X is a subset of the variables of $\mathsf{Var}_\star$:

$$\mathsf{ALoc} = \{n_X \mid X \subseteq \mathsf{Var}_\star\}$$

A shape graph contains a subset of the abstract locations of ALoc

The abstract location $n_\emptyset$ is called the *abstract summary location* and represents all the locations that cannot be reached directly from the state without consulting the heap.

Clearly n_X and $n_\emptyset$ represent disjoint sets of locations when $X \neq \emptyset$.

Invariant 1: If two abstract locations n_X and n_Y occur in the same shape graph then either $X = Y$ or $X \cap Y = \emptyset$. (i.e. two distinct abstract locations n_X and n_Y always represent disjoint sets of locations)

Abstract State

The abstract state, S, maps variables to abstract locations. To maintain the naming convention for abstract locations we sh ensure that:

Invariant 2: If x is mapped to n_X by the abstract state then $x \in$

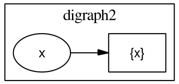
From Invariant 1 it follows that there will be at most one abstrac location in the (same) shape graph containing a given variable

We shall only be interested in the shape of heap so we shall not distinguish between integer values, nil-pointers, and uninitialized hence we can view the abstract state as an element of

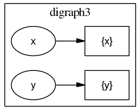
$$\mathsf{S} \in \mathsf{AState} = \mathcal{P}(\mathsf{Var}_\star \times \mathsf{ALoc})$$

Example: Creating Linked Data Structures

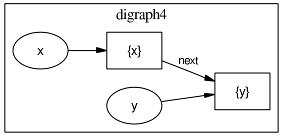
$[new(x)]^2$



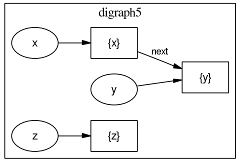
$[new(y)]^3$



[x.next := y]^4



$[new(z)]^5$



Abstract Heap

The abstract heap, H, specifies the links between the abstract locations.

The links will be specified by triples (n_V, sel, n_W) and formally we take the abstract heap as an element of

$$\text{H} \in \text{AHeap} = \mathcal{P}(\text{ALoc} \times \text{Sel} \times \text{ALoc})$$

where we again not distinguish between integers, nil-pointers and uninitialized fields.

Invariant 3: Whenever (n_V, sel, n_W) and (n_V, sel, n'_W) are in the abstract heap then either $V = \emptyset$ or $W = W'$.

Thus the target of a selector field will be uniquely determined by the source unless the source is the abstract summary location $n_\emptyset$.

Sharing Information

The idea is to specify a subset, is, of the abstract locations that represents locations that are shared due to pointers *in* the heap

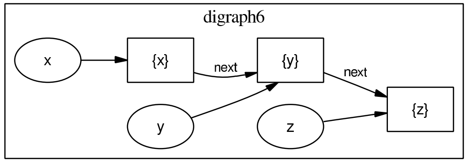
- an abstract location n_X will be included in is if it represents a location that is the target of more than one pointer in the h

In the case of the abstract summary location, $n_\emptyset$, the explicit sh information clearly gives extra information:

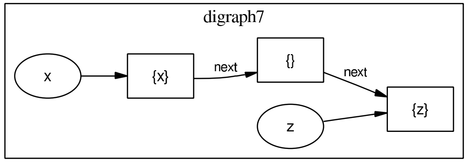
- if $n_\emptyset \in$ is then there might be a location represented by $n_\emptyset$ th the target of two or more heap pointers.
- if $n_\emptyset \notin$ is then all the locations of represented by $n_\emptyset$ will be th target of at most one heap pointer.

Maintaining Sharing Information

[y.next := z]^6

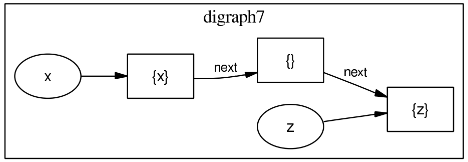


[y := null]^7

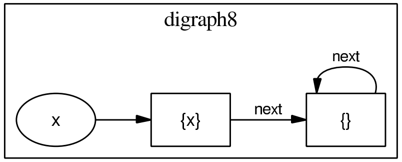


Maintaining Sharing Information

[y := null]^7



[z := null]^8



Sharing Information Invariants (1)

We shall impose two invariants to ensure that information in the component is also reflected in the abstract heap.

The first ensures that information in the sharing component is als reflected in the abstract heap:

Invariant 4: If $n_X \in$ is then either

a) $(n_\emptyset, sel, n_X)$ is in the abstract heap for some sel, or

b) there exist two distinct triples (n_V, sel_1, n_X) and $(n_W, sel_2, n$ the abstract heap (that is either $sel_1 \neq sel_2$ or $V \neq W$).

- case 4a) means that there might be several locations repre by $n_\emptyset$ that point to n_X
- case 4b) means that two distinct pointers (with different sou different selectors) point to n_X.

Sharing Information Invariants (2)

The second invariant ensures that sharing information present in the abstract heap is also reflected in the sharing component:

Invariant 5: Whenever there are two distinct triples (n_V, sel_1, n_X) and (n_W, sel_2, n_X) in the abstract heap and $n_X \neq n_\emptyset$ then $n_X \in$ is.

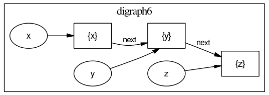
This invariant takes care of the situation where n_X represents a single location being the target of two or more heap pointers.

Note that invariant 5 is the "inverse" of invariant 4(b).

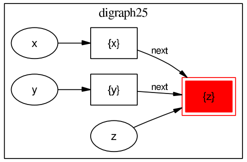
We have no "inverse" of invariant 4(a) - the presence of a pointer from $n_\emptyset$ to n_X gives no information about sharing properties of n_X that are represented in is.

Sharing Component Example 1

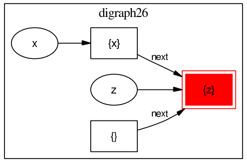
$[\text{y.next} := \text{z}]^6$



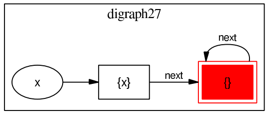
$[\text{x.next} := \text{z}]^{7'}$



$[\text{y} := \text{null}]^{8'}$

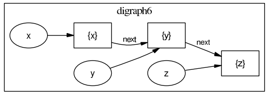


$[\text{z} := \text{null}]^{9'}$

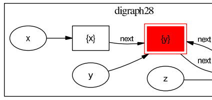


Sharing Component Example 2

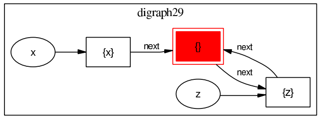
$[\text{y.next} := \text{z}]^6$



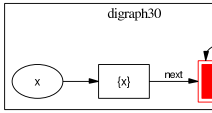
$[\text{z.next} := \text{y}]^{7''}$



$[\text{y} := \text{null}]^{8''}$



$[\text{z} := \text{null}]^{9''}$



Compatible Shape Graphs

A shape graph is a triple (S, H, is):

$$\begin{aligned} \text{S} \in \text{AState} &= \mathcal{P}(\text{Var}_\star \times \text{ALoc}) \\ \text{H} \in \text{AHeap} &= \mathcal{P}(\text{ALoc} \times \text{Sel} \times \text{ALoc}) \\ \text{is} \in \text{IsShared} &= \mathcal{P}(\text{ALoc}) \end{aligned}$$

where ALoc $= \{n_X \mid X \subseteq \text{Var}_\star\}$.

A shape graph is a compatible shape graph if it fulfills the five invariants, 1-5, presented above. The set of compatible shape graphs is denoted

$$\text{SG} = \{(\text{S}, \text{H}, \text{is}) \mid (\text{S}, \text{H}, \text{is}) \text{ is compatible}\}$$

Complete Lattice of Shape Graphs

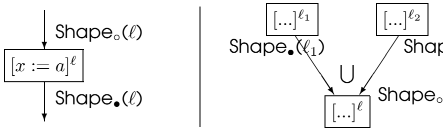
The analysis, to be called *Shape*, will operate over *sets* of compatible shape graphs, i.e. elements of $\mathcal{P}$(SG).
Since $\mathcal{P}$(SG) is a power set it is trivially a complete lattice with

- ordering relation $\sqsubseteq$ being $\subseteq$
- combination operator $\sqcup$ being $\cup$ (may analysis)

$\mathcal{P}$(SG) is finite because SG $\subseteq$ AState $\times$ AHeap $\times$ IsShared and all of AState, AHeap, IsShared are finite.

The analysis will be specified as an instance of a Monotone Framework with the complete lattice of properties being $\mathcal{P}$(SG), and as a forward analysis.

Analysis



$$\begin{aligned} \text{Shape}_\circ(\ell) &= \begin{cases} \iota & : \text{ if } \ell = \text{ini} \\ \bigcup\{\text{Shape}_\bullet(\ell') \mid (\ell', \ell) \in \text{flow}(S_\star)\} & : \text{ otherwi} \end{cases} \\ \text{Shape}_\bullet(\ell) &= f_\ell^{SA}(\text{Shape}_\circ(\ell)) \end{aligned}$$

where $\iota \in \mathcal{P}(SG)$ is the extremal value holding at entry to $S_\star$.

Transfer Functions

The transfer function $f_\ell^{\mathsf{SA}} : \mathcal{P}(\mathsf{SG}) \rightarrow \mathcal{P}(\mathsf{SG})$ has the form

$$f_\ell^{SA}(\mathsf{SG}) = \bigcup \{ \phi_\ell^{\mathsf{SA}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \mid (\mathsf{S}, \mathsf{H}, \mathsf{is}) \in \mathsf{SG} \}$$

where ϕ_ℓ^{SA} specifies how a *single* shape graph (in $\mathsf{Shape}_\circ(\ell)$) may be transformed into a *set* of shape graphs (in $\mathsf{Shape}_\bullet(\ell)$).

The functions ϕ_ℓ^{SA} for the statements

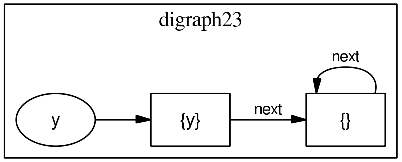
x := a	x := y	x := y.sel
x.sel := a	x.sel := y	x.sel := y.sel

(illustrated by example)

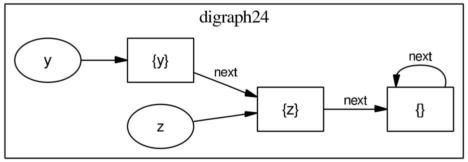
transform a shape graph into a set of different shape graphs.

The transfer functions for other statements and expressions are specified by the identity function.

Example: Materialization



[z := y.next]^7

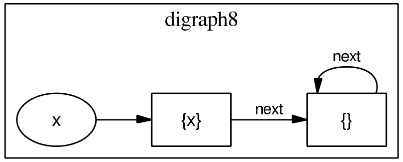


Example: Reverse List

```
[y := null]^1;
while [not isnull(x)]^2 do
  [t := y]^3;
  [y := x]^4;
  [x := x.next]^5;
  [y.next := t]^6;
od
[t := null]^7
```

The program reverses the list pointed to by x and leaves the resu

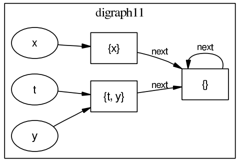
Reverse List: Extremal Value



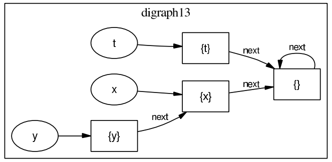
The extremal value ι is a set of graphs. The above graph is an element of this set for our example analysis of the list reversal program.

Shape Graphs in $\mathsf{Shape}_\bullet(\ell)$

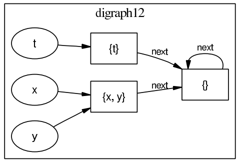
[t := y]^3



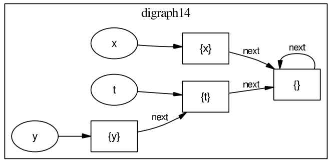
[x := x.next]^5



[y := x]^4

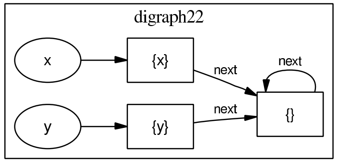


[y.next := t]^6

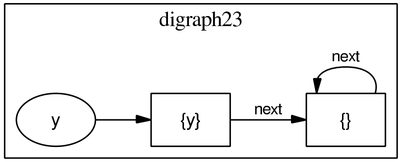


Shape Graphs in $\mathsf{Shape}_\bullet(\ell)$

[t := null]^7



[x := null]^7



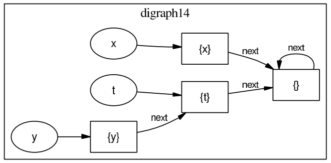
Reverse List: Established Properties

For the list reversal program shape analysis can detect that at the beginning of each iteration of the loop the following properties hold:

Invariant 1: Variable x points to an unshared, acyclic, singly linked list.

Invariant 2: Variable y points to an unshared, acyclic, singly linked list, and variable t may point to the second element of the y-list (if such an element exists).

Invariant 3: The lists pointed to by x and y are disjoint.



Drawbacks and Improvements

An improved version, on which the discussed analysis is based on, can be found in (SRW'98):

- Operates on a single shape graph instead of sets of shape graphs
- Merges sets of compatible shape graphs in one summary shape graph
- Uses various mechanisms for extracting parts of individual compatible shape graphs
- Avoids the exponential factor in the cost of the discussed analysis

The sharing component of the shape graphs is designed to detect list-like properties:

- It can be replaced by other components detecting other shape properties (SRW'02, CDH Ch 5)

References

- Material for this 6th lecture

  `www.complang.tuwien.ac.at/knoop/oue185187_ws0910.html`

- Book

  Flemming Nielson, Hanne Riis Nielson, Chris Hankin:

  Principles of Program Analysis.

  Springer, (450 pages, ISBN 3-540-65410-0), 1999.
  - Chapter 2.6 (Shape Analysis)

- Book

  Steven S. Muchnick:

  Advanced Compiler Design and Implementation, Morgan Kaufmann; (856 pages, ISBN: 1558603204), 1997.
  - Chapter 10 (Alias Analysis)

References

- Book

  Y. N. Srikant, Priti Shankar:

  CDH: The Compiler Design Handbook: Optimizations & Machine Code Generation

  CRC Press; 1st edition, (928 pages, ISBN: 084931240X), 2002.
  - Chapter 5 (Shape Analysis and Applications)

- Journal publication

  SRW'98: Sagiv, M., Reps, T., and Wilhelm, R.

  Solving shape-analysis problems in languages with destructive updating. TOPLAS, 20:1 (January 1998), 1-50.

- Journal publication

  SRW'02: Sagiv M., Reps T., Wilhelm R.

  Parametric shape analysis via 3-valued logic TOPLAS, 24:3 (2002)