Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Orlando, June 1994, pp. 230–241.

Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting

Alain Deutsch
INRIA Rocquencourt
78153 Le Chesnay Cedex, France
Alain.Deutsch@inria.fr

Abstract

Existing methods for alias analysis of recursive pointer data structures are based on two approximation techniques: *k-limiting*, which blurs distinction between sub-objects below depth k; and *store-based* (or equivalently location or region-based) approximations, which blur distinction between elements of recursive data structures. Although notable progress in interprocedural alias analysis has been recently accomplished, very little progress in the precision of analysis of recursive pointer data structures has been seen since the inception of these approximation techniques by Jones and Muchnick a decade ago. As a result, optimizing, verifying and parallelizing programs with pointers has remained difficult.

We present a new parametric framework for analyzing recursive pointer data structures which can express a new natural class of alias information not accessible to existing methods. The key idea is to represent alias information by pairs of *symbolic access paths* which are qualified by symbolic descriptions of the positions for which the alias pair holds.

Based on this result, we present an algorithm for interprocedural may-alias analysis with pointers which on numerous examples that occur in practice is much more precise than recently published algorithms [CWZ90, He90, LR92, CBC93].

1 Introduction and related work

Alias analysis: definition and applications. Aliasing occurs when two distinct names (data access paths) denote the same run-time location. It is introduced by reference parameters and pointers. The aim of existential alias analysis algorithms is to determine for each program point l an upper approximation of the exact set of possible pairs of access paths that may be aliased when l is reached. Existential alias analysis is also called may-alias analysis.

Compile-time alias information is important for scalar optimizations such as code motion; compile-time garbage-collection; program verification and debugging; dependence analysis; parallelisation and improving code generation for instruction-level parallelism [Wa91, HG92, RF93].

Static determination of aliases for reference parameters and single-level pointers is now a well understood problem for which there exists accurate polynomial intraprocedural [SF+90] and interprocedural [LR91] algorithms. However, determining aliases for recursive pointer datatypes is a much harder problem [La92b]. Intuitively, this is because alias sets become potentially infinite, and because transfer functions are not distributive as with single-level pointers.

Existing methods. Approximate existential alias analysis methods for pointers can be classified into: *store-based methods* and *access-paths based* methods. These methods use either *finite graphs* (or abstract stores) to represent potential run-time stores [JM81, JM82, NPD87, RM88, LH88a, Ha89, HPR89, De90, Sh91, De92a, St92] possibly augmented with reference count information [Hu86, He88, CWZ90], *sets of pairs of access paths* to represent aliasing [CC77b, We80, ASU86, He90, SF+90, La92a, LR92] or a combination of the two [CBC93]. Data flow values are kept finite by either *k-limiting* [JM81] or by using a finite number of graph nodes (abstract locations) [Jo81, JM82] determined by the allocation context.

All these methods partition an infinite number of run-time objects (or access paths) into a finite number of equivalence classes. As a consequence, store-based methods will typically fail to distinguish between elements of recursive pointer data structures. This is because a finite number of graph nodes have to be used during the analysis to represent all the elements of those potentially unbounded structures. This introduces false cycles and precludes, for instance, distinguishing either between a linear and a cyclic list, or between a tree and a graph. Similarly, approximation methods based on k-limiting fail to distinguish between elements of recursive pointer data structures that are below depth k. They can distinguish a tree from a general directed graph. But as soon as a sub-object below depth k becomes aliased, aliasing erroneously propagates to all other sub-objects below depth k, contaminating even objects of different types.

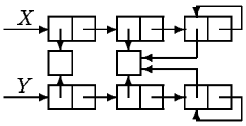
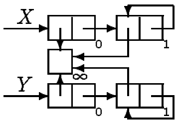
[De92b] presents a theoretical framework for alias analysis. The formalism used is based on Eilenberg's unitary-prefix monomial decomposition [Ei74], on Parikh's commutative decomposition [Pa66] and on a storeless semantic model of aliasing properties based on right-regular equivalence relations. The main result of that paper is the lattice of unitary-prefix monomial relations on subsets of a regular language, which is shown to be an abstract interpretation of the lattice of right-regular equivalence relations. The present paper provides a practical application to imperative languages of the general theory of [De92b].

[He90] cannot handle cyclic data: as noted in [HHN92], this is a serious obstacle to its use in languages with pointers. [CWZ90] and [He90] can distinguish to some extent

```
struct List {
      char *hd;
      struct List *tl;
   };

struct List *
Copy(struct List *L) {
   struct List *p, *t1;
   c1: if (L == null)
         c2: return(L);
   c3: p = malloc(···);
   c4: t1 = L->tl;
   c5: P->tl = Copy(t1);
   c6: p->hd = L->hd;
   c7: return(p);
}

     /*  X is an unaliased list  */
L1: t2 = X;
    Y = Copy(t2);
L2: X = null;
L3:
```

Algorithm	Result at point L2	Spurious aliases
[LH88a]		$\{(X{\to}tl{\to}tl, X{\to}tl{\to}tl{\to}tl),$ $(Y{\to}tl{\to}tl, Y{\to}tl{\to}tl{\to}tl),$ $(Y{\to}tl{\to}hd, Y{\to}tl{\to}tl{\to}hd),$ $\ldots\}$
[CWZ90]		$\{(X{\to}hd, X{\to}tl{\to}hd),$ $(Y{\to}hd, Y{\to}tl{\to}hd),$ $(Y{\to}tl{\to}tl{\to}hd, X{\to}hd),$ $\ldots\}$
[LR92]	$\{(X{\to}hd, Y{\to}hd),$ $(X{\to}tl{\to}hd, Y{\to}tl{\to}hd),$ $(X{\to}tl{\to}tl, Y{\to}tl{\to}tl)\}$	$\{(X{\to}tl{\to}tl, Y{\to}tl{\to}tl),$ $(X{\to}tl{\to}tl{\to}hd, Y{\to}tl{\to}tl{\to}hd),$ $(X{\to}tl{\to}tl, Y{\to}tl{\to}tl{\to}tl),$ $\ldots\}$
[CBC93]	$\{(X, S1), (Y, S2),$ $(S1{\to}hd, S3), (S2{\to}hd, S3),$ $(S1{\to}tl, S1), (S2{\to}tl, S2)\}$	$\{(X{\to}tl, X{\to}tl{\to}tl),$ $(Y{\to}tl, Y{\to}tl{\to}tl),$ $(Y{\to}tl{\to}hd, Y{\to}tl{\to}tl{\to}hd),$ $\ldots\}$
Deutsch	$\{(X{\to}(tl{\to})^i hd, Y{\to}(tl{\to})^j hd) \mid i = j\}$	none

Program property discovered	[LH88a]	[CWZ90]	[LR92]	[CBC93]	Deutsch
P_1 : X and Y are acyclic		yes	yes		yes
P_2 : two successive heads of Y don't alias			yes		yes
P_3 : X and Y tails don't alias	yes	yes		yes	yes
P_4 : heads of X and Y are aliased only pairwise					yes
P_5 : at point L_3, heads and tails of Y are completely unaliased			yes		yes

Figure 1: Precision of alias analysis algorithms on a structure-copying program creating two lists whose elements are pairwise aliased (adapted from [LH88b, p. 103])

between trees, dags and graphs. The first one extends store-based methods with reference counting, but is accurate only intraprocedurally in limited cases, as reference counts cannot, in general, be decremented safely.

Contributions.

1. *an expressive and parametric program analysis framework for may-alias analysis with pointers.*
   Our framework embodies an expressive notion of dependency that, for example, allows relationships between positions in a data structure and aliasing to be captured. For instance, the property:

   "the ith element of list X is aliased to element $2i + 1$ of list Y"

   can be represented exactly in our semilattice framework. We believe that such a notion of *position-dependent aliases* is new. Our alias analysis framework is *parametrized* by a lattice framework $\mathcal{V}^\sharp$ whose purpose is to express information about tuples of integers. In essence, this lattice $\mathcal{V}^\sharp$ controls qualitatively the ability to reason about dependencies (typical examples of $\mathcal{V}^\sharp$ are constant propagation [Ki73], linear arithmetic constraints [Ka76], arithmetic intervals [CC77a], simple sections [BK89] etc., and combinations of these).
2. *an algorithm for interprocedural may-alias analysis with pointers.*
   Based on our parametric framework, we define a polynomial-time, flow-sensitive algorithm for may-alias analysis for a call-by-value, imperative language with arbitrary recursion, dynamic allocation, nested recursive structures, pointer variables and pointers to functions, excluding casting and unions. This encompasses a large subset of the C language.

Why is it significant? An example Our method provides a solution to an open problem [LH88a, CWZ90, He90, HHN92]: how to improve the accuracy of alias analysis in the presence of recursive pointer data structures.

The information obtained by our analysis is generally much more precise than that obtained by previous methods. Figure 1 presents the analysis results of several methods[1] on an example due to Larus & Hilfinger [LH88b, p. 103]. The spurious aliases are due to k-limiting (Larus & Hilfinger, Landi & Ryder) or to collapsing together different heap nodes (Chase *et al.*, Choi *et al.*). As can be seen, the analyses are in general not strictly comparable in precision. In Figure 1, we therefore consider specific program properties $P_1, \ldots, P_5$ and examine which analyses can discover each of them.

Section 2 presents our parametric join semilattice for alias analysis and the corresponding intraprocedural transfer functions. The interprocedural framework, based on the notion of generic object names, is presented in Section 3. Section 4 discusses the time complexity of our method, and our prototype implementation is discussed in Section 5. Finally, we assess the precision of the alias information discovered by our framework in Section 6.

[1] Node labels computed by the method [LH88a] are not shown. In the entry [CWZ90], graph nodes are annotated with approximate reference counts. For the method [CBC93], the $S1, S2, \ldots$ are heap names (allocation sites). Alias pairs are shown for readability without the external level of dereferencing. For instance the pair $(X{\to}hd, Y{\to}hd)$ is really $(*(X{\to}hd), *(Y{\to}hd))$.

2 The intraprocedural framework

2.1 The join semilattice

2.1.1 Symbolic access paths

An *access path* is a string of *selectors* connected by the field component operator ".". Selectors include structure field names, variable names and the dereferencing operator "*". Σ is the set of all selectors for a given program. Access paths are internally represented in postfix notation. For instance the C language pointer expression `X->left`, which is by definition equivalent to `(*X).left`, is represented in postfix notation as the access path `X*.left`. The algorithms we present below operate on access paths in postfix form. For readability, however, we will write access paths using the ordinary C language representation.

A *symbolic access path* (SAP for short) is an access path possibly containing symbolic expressions of the form B^k, where B is a set of access paths called a *basis*, and k is a variable. B^0 denotes the empty access path ϵ, and B^{n+1} is the set $B.B^n$. Consider for instance the SAP[2] $f = X{\to}(tl{\to})^i hd$. If $i = 0$ then f denotes the set $\{X{\to}hd\}$; if $i = 1$ then f denotes the set $\{X{\to}tl{\to}hd\}$ and so on. Consider the SAP $g = T{\to}\{left{\to}, right{\to}\}^j key$. If $j = 0$ then g denotes the set $\{T{\to}key\}$; if $j = 1$ then g denotes the set $\{T{\to}left{\to}key, T{\to}right{\to}key\}$ and so on. What kind of symbolic expressions can occur in a symbolic access path?

The basis of a recursive pointer type t is a set of access paths $B = Basis(t)$ such that: (1) if an access path π yields an object of type t when applied to an object of type t then $\pi \in B^*$; (2) B is minimal. Figure 2 presents recursive types and their associated bases. The function *Basis* maps a recursive pointer type name to its corresponding basis. Bases can be represented by deterministic finite automata (DFA) over the alphabet of accessors Σ, and states of the DFA are just type names. Bases for mutually recursive types are defined similarly, see Appendix.

Definition 2.1 (Symbolic access paths) *A symbolic access path is a string of the form $e_1.e_2.\ldots.e_n$, where for each $1 \le i \le n$, e_i is either:*

1. *a selector $s \in \Sigma$;*
2. *an expression of the form B^k, where k is a coefficient variable and $B = Basis(t)$ is the basis of some recursive type t.*

We write $fv(f)$ to denote the set of coefficient variables occurring in the SAP f. In practice[3] a basis can be represented uniquely by its corresponding type name, and a global table can be maintained by mapping type names to bases represented by DFA.

2.1.2 The numeric lattice

Our lattice for alias analysis is parametrised by a numeric lattice $\mathcal{V}^\sharp$. $\mathcal{V}^\sharp$ determines which class of relations between positions in aliased data structures can be captured. Independent (mono-dimensional) numeric lattices include: *constant propagation* [Ki73], *arithmetic intervals* [CC77a, Mo84] and *arithmetic congruences* [Gr89]. Relational (multidimensional) numeric lattices include: *linear equalities* [Ka76], *linear inequalities* [CH78], *simple sections* [BK89, CC92], *linear congruence equalities* [Gr91] and *congruential trapezoids* [Ma91]. We will now list our assumptions about the numeric lattice and its associated operations, so as to keep our construction parametric by avoiding dependence on a particular numeric lattice.

[2] The internal, postfix representation of the symbolic access path f is $X{*}.(tl{*})^i.hd$.

[3] The theoretically inclined reader is encouraged to consult [De92b] and [De92a, §3] for a theoretical account of the connection between SAPs, the unitary-prefix monomials of Eilenberg's treatise [Ei74] and Parikh's commutative decomposition [Pa66].

```
struct List { char *hd; struct List *tl; }
struct List2 { struct List *hd; struct List2 *tl;}
struct Tree { char *key; struct Tree *left,*right;}
```

$Basis(struct\ List) = \{tl{\to}\}$
$Basis(struct\ List2) = \{tl{\to}\}$
$Basis(struct\ Tree) = \{left{\to}, right{\to}\}$

Figure 2: Recursive pointer types and their corresponding bases

Hypothesis 2.1 (Properties of the numeric lattice) *Given a finite set of variables V, the numerical lattice $\mathcal{V}^\sharp(V)$ is equipped with the following abstract operators which are upper approximations of their exact counterparts in $\mathbb{P}(V \to \mathbb{N})$ (sets of maps from variables to integers):*

1. *the binary operations $\wedge$ (meet) and $\vee$ (join) upper approximate intersection and union on sets; $\perp$ represents exactly the empty set;*
2. projection*: if $U \subseteq V$ then $Project^\sharp(K, U) \in \mathcal{V}^\sharp(U)$ is the projection of $K \in \mathcal{V}^\sharp(V)$ onto U;*
3. extension*: if $K \in \mathcal{V}^\sharp(V)$ then $K \uparrow U \in \mathcal{V}^\sharp(U \cup V)$ is the extension of K to $U \cup V$;*
4. resolution of a linear system*: if S is a system of linear equations over V then $\mathcal{S}^\sharp(S)$ is an upper approximation in $\mathcal{V}^\sharp(V)$ of the set of integer solutions to S;*
5. intersection with a linear system*: if S is a system of linear equations over V and $K \in \mathcal{V}^\sharp(U)$ then $\mathcal{C}^\sharp(K, S)$ is an upper approximation in $\mathcal{V}^\sharp(U \cup V)$ of the set of integer solutions to S that are also in K;*
6. meet of spaces of different dimensions*: if $K_1 \in \mathcal{V}^\sharp(U)$ and $K_2 \in \mathcal{V}^\sharp(V)$, $K_1 \wedge_h K_2 \stackrel{\text{def}}{=} (K_1 \uparrow V) \wedge (K_2 \uparrow U)$.*

In addition we define $\top = \mathcal{S}^\sharp(\emptyset)$. Finally, if $\mathcal{V}^\sharp(V)$ has infinite height then it is equipped with a widening operator *∇ [CC77a] to ensure termination of fixpoint computations.*

The relational lattices enumerated above satisfy this hypothesis (except the lattice of simple sections [BK89] which must be extended as explained in [CC92, §9.1]). Each independent numeric lattice L can also be accommodated by defining $\mathcal{V}^\sharp$ as the n-fold (smash) product of L. $\mathcal{V}^\sharp$ can also be defined using one of the systematic methods for combining analysis frameworks proposed in [CC79, §10]. By abuse of notation, we will write $\mathcal{V}^\sharp$ instead of $\mathcal{V}^\sharp(V)$ when V is clear from the context. If $K \in \mathcal{V}^\sharp(V)$ then $Dom(K) = V$ ($Dom(K)$ is the domain of K).

2.1.3 The parametric semilattice UR($\mathcal{V}^\sharp$)

Definition 2.2 (Symbolic alias pairs) *A* symbolic alias pair *is of the form $(\langle f_1, f_2\rangle, K)$, where f_1 and f_2 are symbolic access paths; $K \in \mathcal{V}^\sharp$; $Dom(K) = fv(f_1) \cup fv(f_2)$ and the coefficient variables of f_1 and f_2 are disjoint.*

In addition we say that a pair $(\langle f_1, f_2\rangle, K)$ is *named canonically* if the sequence of coefficient variables occurring in left to right order in the SAPs f_1 and f_2 is a sequence of

```
Algorithm ⊔ (Join of symbolic alias relations)
Input: two symbolic alias relations ϱ1, ϱ2 ∈ UR(V♯)
Output: their join ϱ1 ⊔ ϱ2
Method:
 ϱ := ϱ1 ∪ ϱ2;
 exhaustively apply the following on ϱ:
   if (⟨f1, f2⟩, K) ∈ ϱ and (⟨f1, f2⟩, K') ∈ ϱ then
     replace these two pairs by (⟨f1, f2⟩, K ∨ K');
 return ϱ
```

Example:
$\varrho_1 = \{(\langle X\to(tl\to)^{k_1}hd,\ Y\to(tl\to)^{k_2}hd\rangle, \mathcal{S}^\sharp\{k_1{=}0, k_2{=}0\})\}$
$\varrho_2 = \{(\langle X\to(tl\to)^{k_1}hd,\ Y\to(tl\to)^{k_2}hd\rangle, \mathcal{S}^\sharp\{k_1{=}1, k_2{=}1\})\}$
if $\mathcal{V}^\sharp$ is the lattice of arithmetic intervals [CC77a]:
$\varrho_1 \sqcup \varrho_2 = \{(\langle X\to(tl\to)^{k_1}hd,\ Y\to(tl\to)^{k_2}hd\rangle, \mathcal{S}^\sharp\{0{\leq}k_1, k_2{\leq}1\})\}$
if $\mathcal{V}^\sharp$ is Karr's lattice [Ka76]:
$\varrho_1 \sqcup \varrho_2 = \{(\langle X\to(tl\to)^{k_1}hd,\ Y\to(tl\to)^{k_2}hd\rangle, \mathcal{S}^\sharp\{k_1{=}k_2\})\}$

Figure 3: The join operator

predefined variables, say $k_1, k_2, \ldots$ The operator *Rename* maps a symbolic alias pair $(\langle f_1, f_2\rangle, K)$ to its canonically named counterpart, and extends componentwise to sets of symbolic alias pairs. For instance:

$$(\langle X(\to tl)^{k_1}\to hd(\to tl)^{k_2}\to hd, Y(\to tl)^{k_3}\to hd\rangle, \mathcal{S}^\sharp\{k_3 = k_1+k_2\})$$

is a symbolic alias pair which is canonically named.

Proposition 2.3 (Symbolic alias relations) *The set* $\mathrm{UR}(\mathcal{V}^\sharp)$ *of symbolic alias relations over* $\mathcal{V}^\sharp$ *is a semilattice with least element* $\bot = \emptyset$ *and join* $\sqcup$ *where:*

1. *a symbolic alias relation* ϱ *over* $\mathcal{V}^\sharp$ *is a set of canonically named symbolic alias pairs over* $\mathcal{V}^\sharp$ *such that if* $(\langle f_1, f_2\rangle, K) \in \varrho$ *and* $(\langle f_1, f_2\rangle, K') \in \varrho$ *then* $K = K'$*;*
2. *the join operator is computed pointwise, see Figure 3.*

2.1.4 Termination of fixpoint computations

The parametric semilattice $\mathrm{UR}(\mathcal{V}^\sharp)$ we have just defined has infinite chains: (1) because the number of possible SAPs is not bounded; and (2) because $\mathcal{V}^\sharp$ may have infinite chains (for instance the lattice of intervals [CC77a]).

We define the normalisation operation *Factor* which maps a SAP f to a SAP f' in which potentially unbounded subsequences of f (paths through recursive data structures) have been replaced by bases guarded by new coefficient variables. For instance, *Factor* applied to the SAP $f = X\to tl\to tl\to hd$ would return a pair (f', S) consisting of the SAP $f' = X\to(tl\to)^i hd$ and of the system of equations $S = \{i = 2\}$. The algorithm *Factor* is shown in Figure 15.

We extend *Factor* by overloading: given a symbolic alias relation ϱ, $Factor(\varrho)$ is a symbolic alias relation obtained by normalising (with *Factor* and *Rename*) the symbolic alias pairs contained in ϱ.

The widening $\varrho_1 \nabla \varrho_2$ of two symbolic alias relations is defined as follows: (1) normalise ϱ_1 and ϱ_2 using *Factor*; (2) apply pointwise the numeric widening operator ∇ (this is similar to the join operator)[4]. This operator ∇ is inserted in the data flow equations at points contained in a feedback set W of the dependence graph of the equations. For instance, the feedback set can be defined as the set of intervals headers or loop headers, see Appendix for details.

[4] If the numeric lattice $\mathcal{V}^\sharp$ has no infinite chains, then the widening operator on symbolic alias relations can be defined simply as: $\varrho_1 \nabla \varrho_2 = Factor(\varrho_1) \sqcup Factor(\varrho_2)$.

$Match_\in(X\to tl\to hd, X\to(tl\to)^i) = \{(\{i=1\}, hd)\}$
$Match_\ni(X\to(tl\to)^i hd, X\to tl) = \{(\{i=j+1\}, \to(tl\to)^j hd)\}$

$Compl(\{i=j+1, k=2\}, \{j\}) =$
$\quad\{\mathcal{S}^\sharp\{0 \leq k \leq 1\}, \mathcal{S}^\sharp\{k \geq 3\}, \mathcal{S}^\sharp\{i = 0\}\}$

if $\varrho = \{(\langle X\to(tl\to)^i hd, Y\to(tl\to)^j hd\rangle, \mathcal{S}^\sharp\{i=j\})\}$ then :
$EquivalenceClass^\sharp(Y\to hd, \varrho) = \{(Y\to hd, \top),$
$\quad(X\to(tl\to)^i hd, \mathcal{S}^\sharp\{i=0\})\}$

$StripPrefix^\sharp(*X, \{(*Y, \top), (X\to tl\to hd, \top)\}) = \{(tl\to hd, \top)\}$
$StripPrefix^\sharp(*(X\to tl), \{(*Y, \top), (*(X\to(tl\to)^i hd), \mathcal{S}^\sharp\{i \geq 0\})\}) =$
$\quad\{((tl\to)^k hd\to, \mathcal{S}^\sharp\{k \geq 1\})\}$

$StarClosure^\sharp(\{(tl\to tl\to, \top)\}, struct\ List) =$
$\quad\{((tl\to)^i, \mathcal{S}^\sharp\{i \bmod 2 = 0\})\}$
$StarClosure^\sharp(\{((tl\to)^i tl\to, \mathcal{S}^\sharp\{i \geq 2\})\}, struct\ List) =$
$\quad\{((tl\to)^i, \mathcal{S}^\sharp\{i \geq 0\})\}$

if $P = \{(X\to(tl\to)^i, \mathcal{S}^\sharp\{i \geq 1\})\}$, $Q = \{((tl\to)^i, \mathcal{S}^\sharp\{i \geq 2\})\}$ then :
$P.Q = \{(X\to(tl\to)^i(tl\to)^j, \mathcal{S}^\sharp\{i \geq 1, j \geq 2\})\}$
$P.* = \{(*(X\to(tl\to)^i), \mathcal{S}^\sharp\{i \geq 1\})\}$

Figure 4: Elementary operations: examples

2.2 The function space

We have defined the parametric semilattice $\mathrm{UR}(\mathcal{V}^\sharp)$ of symbolic alias relations, and we equipped it with a join operation $\sqcup$, a least element $\bot$ and widening operator ∇ to ensure the termination of fixpoint iterations. We now define transfer functions that model the effects of individual program statements on symbolic alias relations.

2.2.1 Elementary operations

Transfer functions operate on sets of symbolic alias pairs, which contain symbolic access paths. Therefore, we define operations to manipulate these symbolic representations. Examples are presented in Figure 4 and full definitions appear in the Appendix.

The binary operator *Match* determines if a symbolic access path f can generate (contains) a particular access path, or a prefix of it[5]. More precisely, $Match_\bowtie(M, N)$ takes as parameters: (1) a relational operator $\bowtie$ which must be one of $\{\in, \ni, =\}$; (2) two access paths M and N, one of which at most is symbolic. $Match_\bowtie(M, N)$ returns a set of solutions the form (S, Δ), where the residual Δ is a (possibly symbolic) access path and S is a system of equations. Each (S, Δ) is such that the equation $M \bowtie N.\Delta$ is true for each assignment of the numerical coefficients of M, N and Δ which is a solution of equation system S.

The operator *Compl* takes a system of linear equations S and a set of variables U, and returns $Compl(S, U)$, a set of elements of $\mathcal{V}^\sharp$ whose union upper approximates the *complement* of the system S with respect to the positive integers. Variables of U occurring in the system S are assumed to be arbitrary positive integers which are eliminated.

A *symbolic path set* is a set of pairs (f, K), where f is a symbolic access path, and K an element of $\mathcal{V}^\sharp$. Such sets will be used to represent finitely possibly infinite sets of access paths. For instance $\{(X\to(tl\to)^i hd\to(tl\to)^j hd, \mathcal{S}^\sharp\{i =$

[5] This is necessary because alias relations are right-regular [Jo81]: if π is aliased to π' then for each path δ (such that $\pi.\delta$ exists), $\pi.\delta$ is aliased to $\pi'.\delta$.

`Algorithm` $\mathrm{Kill}^\sharp(\pi)(\varrho)$ (Deletion of a fixed access path π)
`Input:` a symbolic alias relation $\varrho \in \mathrm{UR}(\mathcal{V}^\sharp)$
`Output:` a symbolic alias relation $\mathrm{Kill}^\sharp(\pi)(\varrho)$
`Method:`
 ϱ' := $\emptyset$;
 foreach symbolic alias pair $(\langle f_1, f_2\rangle, K) \in \varrho$ **do**
  K := $KillPath(\pi, f_1, K)$;
  K := $KillPath(\pi, f_2, K)$;
  if $(K \neq \bot)$ **then** ϱ' := $\varrho' \cup \{(\langle f_1, f_2\rangle, K)\}$
 return ϱ'

`Algorithm` $KillPath(\pi, f, K)$
`Input:` an access path π, a SAP f, and an element K of $\mathcal{V}^\sharp$
`Output:` an element $KillPath(\pi, f, K)$ of $\mathcal{V}^\sharp$
`Method:`
 A := $\{Compl(S, fv(\Delta)) \mid (S, \Delta) \in Match_=(\pi, f)\}$
 K := $K \wedge \left(\bigwedge_{B \in A} \bigvee_{K' \in B} (K \wedge_h K') \right)$;
 return K

Example:
$KillPath(X{\to}tl{\to}hd,\ X{\to}(tl{\to})^{k_1} hd,\ \mathcal{S}^\sharp\{k_1 = 1\}) = \bot$
$KillPath(X{\to}tl{\to}hd,\ X{\to}(tl{\to})^{k_1} hd,\ \mathcal{S}^\sharp\{k_1 \geq 1\}) = \mathcal{S}^\sharp\{k_1 \geq 2\}$
$KillPath(X{\to}tl,\ X{\to}(tl{\to})^{k_1} hd,\ \mathcal{S}^\sharp\{k_1 \geq 0\}) = \mathcal{S}^\sharp\{k_1 = 0\}$

Figure 5: The transfer function $\mathrm{Kill}^\sharp(\pi)$

$j\})\}$ denotes an infinite set of access paths which is *context-free* but not regular.

The binary operator $EquivalenceClass^\sharp(\pi, \varrho)$ computes a symbolic path set which represents the set of access paths to which the access path π is aliased in ϱ. Implicit aliases that can be deduced by reflexive, symmetric and right-regular closure are taken into account. This operation is based on $Match_\in$. In addition, symbolic operations (intersection and projection) on the numerical lattice element K have to be performed to extract relevant information.

The operation $StripPrefix^\sharp(\pi, P)$ computes a symbolic path set denoting the set of access paths obtained by stripping the prefix π out of the access paths represented by P. This operation is also based on *Match*.

The operator $StarClosure^\sharp(P, t)$ computes a symbolic path set denoting the star closure of the set of access paths denoted by P, where t is the recursive type to which paths of P can be applied. This is based on *Match* and *Factor*.

The infix operation "." computes a representation of the concatenation of (the access paths denoted by) two symbolic path sets P and Q. Given a symbolic path set P and an access path π, we also write $P.\pi$ for $P.\{(\pi, \top)\}$.

2.2.2 Deletions (kills)

The transfer function $\mathrm{Kill}^\sharp(\pi)$, where π is a fixed access path, deletes all the alias pairs whose left or right component contains π (or an extension of π). Deletions are used to model: (1) when a variable goes out of scope at the exit from a lexical unit (local block or procedure); (2) assignments: `X = Y` first kills $*X$, then generates aliases introduced by the alias pair $(*X, *Y)$ (using the function $\mathrm{Gen}^\sharp$). Assignments to a component also kills aliases, for instance `X->tl = Y` first kills $*(X{\to}tl)$, then generates aliases induced by $(*(X{\to}tl), *Y)$. $\mathrm{Kill}^\sharp(\pi)$ is presented in Figure 5.

`Algorithm` $\mathrm{Gen}^\sharp(lhs.\sigma, rhs)(\varrho)$
`Input:` two access paths $lhs.\sigma$ and rhs s.t $lhs.\sigma \not\leq rhs$ and $lhs.\sigma \not\geq rhs$; a symbolic alias relation ϱ
`Output:` a symbolic alias relation ϱ' which incorporates the aliases generated by the assignment $lhs.\sigma \leftarrow rhs$
`Method:`
 E := $EquivalenceClass^\sharp(lhs, \varrho)$; /* aliases of lhs */
 B := $StripPrefix^\sharp(rhs, E)$;
 C := $StarClosure^\sharp(B.\sigma, Typeof(rhs))$;
 P := $E.\sigma.C$;
 ϱ' := $\varrho \sqcup Rewrite^\sharp(rhs, P)(\varrho)$;
 return ϱ'

Figure 6: The transfer function $\mathrm{Gen}^\sharp$

2.2.3 Alias introduction

What are the effects of the assignment of rhs to $lhs.\sigma$, where σ is simple selector ? For instance, we have $lhs = X{\to}tl$, $\sigma = *$ and $rhs = *Y$ for the assignment `X->tl = Y`. Assume without loss of generality that the access paths rhs and lhs are not comparable by the prefix relation (otherwise first copy rhs: assignments such as `X = X->tl` are decomposed in the two assignments `tmp = X->tl` and `X = tmp`, followed by $\mathrm{Kill}^\sharp(*tmp)$). We describe exactly the aliases introduced:

1. if lhs is not aliased: it generates the pair $\langle lhs.\sigma, rhs\rangle$; in addition, an incoming alias pair $\langle rhs.\pi', \pi''\rangle$ generates the pair $\langle lhs.\sigma.\pi', \pi''\rangle$ (and symmetrically); an incoming pair $\langle rhs.\pi', rhs.\pi''\rangle$ generates $\langle lhs.\sigma.\pi', lhs.\sigma.\pi''\rangle$. In short, we say that the effect of this assignment is $Rewrite(rhs, \{lhs.\sigma\})$;
2. if lhs is aliased to the access paths $E = \{lhs, \pi_1, \pi_2 \ldots\}$ and no π_i contains rhs as a prefix: as an assignment to lhs must also assign to all of the aliases of lhs, the net effect is $Rewrite(rhs, E.\sigma)$;
3. if lhs is aliased to the access paths $E = \{lhs, \pi_1, \pi_2 \ldots\}$, and $B \subseteq E$, with $B = \{rhs.\beta_1, rhs.\beta_2, \ldots\}$ the aliases of lhs containing rhs as a prefix: the effect is $Rewrite(rhs, E.\sigma.(\{\beta_1, \beta_2, \ldots\}.\sigma)^*)$ (this assignment creates cycles).

This case analysis describes precisely the aliases generated by an assignment [Jo81, De92a], in terms of (possibly infinite) sets of alias pairs. We now define an abstract counterpart of this operation which does not operate on sets of alias pairs, but on (finite) symbolic alias relations. The abstract counterpart of *Rewrite*, the operation $Rewrite^\sharp(rhs, P)$, maps a symbolic alias relation ϱ to a symbolic alias relation $\varrho' = Rewrite^\sharp(rhs, P)(\varrho)$. As for *Rewrite*, rhs is an access path, but P is a *symbolic path set*.

The transfer function $\mathrm{Gen}^\sharp$ is shown in Figure 6.

$\varrho_1 = \{(\langle *(Y{\to}(tl{\to})^i hd), *(Z{\to}(tl{\to})^j hd)\rangle, \mathcal{S}^\sharp\{i = j\})\}$

$\boxed{\texttt{X = Y->tl;}}$ f

$\varrho_2 = \{(\langle *X, *(Y{\to}tl)\rangle, \top),$
$(\langle *(Y{\to}(tl{\to})^i hd), *(Z{\to}(tl{\to})^j hd)\rangle, \mathcal{S}^\sharp\{i = j\}),$
$(\langle *(X{\to}(tl{\to})^i hd), *(Z{\to}(tl{\to})^j hd)\rangle, \mathcal{S}^\sharp\{j = i{+}1\})\}$

Transfer function: $f = \mathrm{Gen}^\sharp(*X, *(Y{\to}tl)) \circ \mathrm{Kill}^\sharp(*X)$

Figure 7: Analysis of an assignment

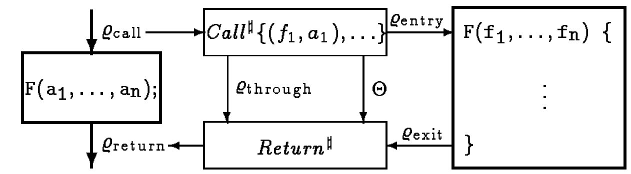


Figure 8: Interprocedural information flow

Example 2.4 *Assume the aliasing just before the assignment shown in Figure 7 is described by the symbolic alias relation ϱ_1, then aliasing after this assignment is $\varrho_2 = f(\varrho_1)$.*

3 The interprocedural framework

Interprocedural methods for non-distributive problems over large semilattices limit information loss by analysing procedures separately [CC77c, SP81, JM82], keyed by some *token* abstracting the call context. However this usually results in information loss for recursive procedures, as each recursive call can generate a new semilattice value. Therefore, following [CC77c, Ha79, MR89, LR92], we perform therefore *generalisation* of data flow values through function calls and *instantiation* through function returns. Consider two alias pairs $\langle x, \pi_1\rangle$ and $\langle x, \pi_2\rangle$ reaching the entry of a function $F(x,y)$. If π_1 and π_2 are not visible in F and in the functions called by F, we can generalise these two alias pairs by a *single* alias pair $\langle x, U\rangle$. U is a generic object name. The key observation, due to [MR89, LR92], is that F operates *uniformly* on all the aliases of x that are not visible. Then F will be analysed with the incoming pair $\langle x, U\rangle$, and the aliasing at the output of F can be propagated back by instantiation. This is done by applying the transformation $[U \mapsto \pi_1, U \mapsto \pi_2]$.

We assume in the rest of the text that no two distinct variables have the same name. This can be achieved, for instance, by prefixing each variable by the name of its static definition point (e.g. procedure name).

3.1 Generic objects

In order to perform instantiation and generalisation on symbolic alias relations, we enrich symbolic access paths with *generic objects* of the form $U[k_1,\ldots,k_n]$, where U is a name. Intuitively, $U[k_1,\ldots,k_n]$ stands for an unknown symbolic access path whose coefficients are $k_1,\ldots,k_n$.

3.2 Function calls

A call S_1: y = F(a1, ..., an); S_2: to a function F with formal parameters $f_1,\ldots,f_n$ is modelled by the transfer function $Call^\sharp$ as follows:

$$(\varrho_{\text{entry}}, \varrho_{\text{through}}, \Theta) = Call^\sharp\{(f_1,a_1),\ldots,(f_n,a_n)\}(\varrho_{\text{call}})$$

ϱ_{call} is the symbolic alias relation describing the aliasing at program point S_1; ϱ_{through} represents the aliases of ϱ_{call} that are not affected by F and that do not affect F, they can be directly propagated to point S_2; ϱ_{entry} represents the aliases of ϱ_{call} plus those induced by the binding of each formal f_i to the corresponding actual a_i, see also Figure 8. The alias relation ϱ_{entry} contains only the symbolic alias pairs relevant to F, and *generalisation* has been performed. The set of symbolic alias pairs Θ represents the substitution to be applied upon return from F. Θ associates generic objects (introduced during the generalisation) with the symbolic access paths they represent. The arguments a1, ..., an are temporaries which are killed by the call: this discipline avoids the introduction of spurious aliases between a formal parameter and a dead argument.

Algorithm $Call^\sharp\{(f_1,a_1),\ldots,(f_n,a_n)\}(\varrho_{\text{call}})$
Input: formals $f_1,\ldots,f_n$, distinct arguments $a_1,\ldots,a_n$, a symbolic alias relation ϱ_{call}
Output: the symbolic alias relations ϱ_{entry}, ϱ_{through} and Θ
Method:
 foreach $i = 1,\ldots,n$ **do**
  ϱ_{call} := $[\text{Kill}^\sharp(a_i.*) \circ \text{Gen}^\sharp(f_i.*, a_i.*)](\varrho_{\text{call}})$
 done;
 support := $\{f_1,\ldots,f_n\} \cup$ *GlobalVariables*;
 ϱ_{entry} := $\emptyset$; ϱ_{through} := $\emptyset$; Θ := $\emptyset$;
 foreach symbolic pair $(\langle g_1,g_2\rangle, K) \in \varrho_{\text{call}}$ **do**
  if g_1 and g_2 are in *support* **then**
   ϱ_{entry} := $\varrho_{\text{entry}} \cup \{(\langle g_1,g_2\rangle, K)\}$;
  if neither g_1 nor g_2 is in *support* **then**
   ϱ_{through} := $\varrho_{\text{through}} \cup \{(\langle g_1,g_2\rangle, K)\}$;
  if g_2 is in *support* and g_1 is not in *support* **then**
   $(\varrho_{\text{entry}}, \Theta)$:= $Generalise^\sharp((\langle g_1,g_2\rangle, K), \varrho_{\text{entry}}, \Theta)$;
  if g_1 is in *support* and g_2 is not in *support* **then**
   $(\varrho_{\text{entry}}, \Theta)$:= $Generalise^\sharp((\langle g_2,g_1\rangle, K), \varrho_{\text{entry}}, \Theta)$
 done;
 return $(\varrho_{\text{entry}}, \varrho_{\text{through}}, \Theta)$

Figure 9: The interprocedural transfer function $Call^\sharp$

The transfer function $Call^\sharp$ is shown in Figure 9. The function $Generalise^\sharp((\langle g_1,g_2\rangle, K), \varrho_{\text{entry}}, \Theta)$ generalises the symbolic pair $(\langle g_1,g_2\rangle, K)$ by replacing g_1 with a generic object $U[k_1,\ldots,k_n]$, where n is the number of coefficients of g_2, and updating accordingly ϱ_{entry} and Θ. The generic object name U is determined uniquely by the factorised form of the symbolic path g_2.

3.3 Function returns

The propagation of aliases back to a calling point is modelled by the transfer function $Return^\sharp$:

$$\varrho_{\text{return}} = Return^\sharp(\varrho_{\text{exit}}, \varrho_{\text{through}}, \Theta)$$

where the symbolic alias relations ϱ_{through} and Θ have been computed by the corresponding $Call^\sharp$, and ϱ_{exit} is the symbolic alias relation describing aliasing at the exit of a function F. The newly computed symbolic alias relation ϱ_{return} describes the aliasing just upon return from F. $Return^\sharp$ essentially *instantiates* each generic object name occurring in ϱ_{exit}. Each generic name is replaced by the symbolic access paths it represents, as described by Θ. Because we consider a call-by-value language, formals need not be replaced by corresponding locals. Finally, the aliases ϱ_{through} are added directly to the result ϱ_{return}.

The transfer function $Return^\sharp$ is shown in Figure 10. Given a variable or a generic object name u, the function $Instantiate^\sharp(u, K, \Theta)$ returns a set of pairs (u', K') obtained by replacing u by the symbolic access paths associated with u in Θ and adjusting accordingly K. For instance if $\Theta = \{(\langle U_1[k], x{\to}(tl{\to})^l hd\rangle, \mathcal{S}^\sharp\{k = 10 - l\})\}$, then $Instantiate^\sharp(U_1[j], \mathcal{S}^\sharp\{j = 2i\}, \Theta) = \{(x{\to}(tl{\to})^l hd, \mathcal{S}^\sharp\{j = 10 - l, j = 2i\})\}$. As in the intraprocedural case, widening operators must be inserted in the interprocedural equations in order to ensure termination of fixpoint iterations, see Appendix.

Example 3.1 *Consider the following program fragment:*

```
void F(struct List *L) {
   F1: result = L->tl;
   F2:
}
```

Algorithm $\mathit{Return}^\sharp(\varrho_{\mathrm{exit}}, \varrho_{\mathrm{through}}, \Theta)$
Input: the symbolic alias relations $\varrho_{\mathrm{exit}}, \varrho_{\mathrm{through}}$ and Θ
Output: the symbolic alias relation $\varrho_{\mathrm{return}}$
Method:

$\varrho_{\mathrm{return}}$:= $\emptyset$;
Θ := $\Theta \cup \{(g,g) \mid g \in \mathit{GlobalVariables}\}$;
foreach symbolic pair $(\langle uM, vN\rangle, K) \in \varrho_{\mathrm{exit}}$
  if u and v are globals or generic names **then**
    foreach $(u', K_1) \in \mathit{Instantiate}^\sharp(u, K, \Theta)$ and
        $(v', K_2) \in \mathit{Instantiate}^\sharp(v, K, \Theta)$
    do
      K' := $K_1 \wedge_h K_2$;
      if $(K' \neq \bot)$ **then**
        $\varrho_{\mathrm{return}}$:= $\varrho_{\mathrm{return}} \sqcup \mathit{Rename}\{(\langle u'M, v'N\rangle, K')\}$
    done
  fi;
$\varrho_{\mathrm{return}}$:= $\varrho_{\mathrm{return}} \sqcup \varrho_{\mathrm{through}}$;
return $\varrho_{\mathrm{return}}$

Figure 10: The interprocedural transfer function $\mathit{Return}^\sharp$

P_1: `F(a);`
P_2:

The corresponding data flow equations are then:

$$\begin{cases} (F_1, \varrho, \Theta) = \mathit{Call}^\sharp\{(L, a)\}(P_1) \\ F_2 = [\mathrm{Gen}^\sharp(*\mathit{result}, *(L\to \mathit{tl})) \circ \mathrm{Kill}^\sharp(*\mathit{result})](F_1) \\ P_2 = \mathit{Return}^\sharp(F_2, \varrho, \Theta) \end{cases}$$

Because the function `F` *is not recursive, widening operators are not needed. Assume that aliasing at point* P_1 *is:*

$$\begin{aligned} P_1 = \{&(\langle *a, *X\rangle, \top), \\ &(\langle *(X\to(\mathit{tl}\to)^{k_1} \mathit{hd}), *(Y\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2+1\}), \\ &(\langle *(a\to(\mathit{tl}\to)^{k_1} \mathit{hd}), *(Y\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2+1\})\} \end{aligned}$$

The solution of the above data flow equations is then:

$$\begin{aligned} F_1 = \{&(\langle U_1, *L\rangle, \top), \\ &(\langle U_2[k_1], *(L\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2\})\} \\ \Theta = \{&(\langle U_1, *X\rangle, \top), \\ &(\langle U_2[k_1], *(Y\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2+1\})\} \\ \varrho = \{&(\langle *(X\to(\mathit{tl}\to)^{k_1} \mathit{hd}), *(Y\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2+1\})\} \\ F_2 = F_1 \cup \{&(\langle *\mathit{result}, *(U_1.\mathit{tl})\rangle, \top), \\ &(\langle *\mathit{result}, *(L\to \mathit{tl})\rangle, \top), \\ &(\langle U_2[k_1], *(\mathit{result}\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2+1\})\} \\ P_2 = \{&(\langle *\mathit{result}, *(X\to \mathit{tl})\rangle, \top), \\ &(\langle *(X\to(\mathit{tl}\to)^{k_1} \mathit{hd}), *(Y\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2+1\}), \\ &(\langle *(\mathit{result}\to(\mathit{tl}\to)^{k_1} \mathit{hd}), *(Y\to(\mathit{tl}\to)^{k_2} \mathit{hd})\rangle, \mathcal{S}^\sharp\{k_1 = k_2\})\} \end{aligned}$$

3.4 Extended interprocedural framework

As with any iterative program analysis, the precision of our basic interprocedural framework can be improved by keeping several symbolic alias relations at each program point of a procedure, each qualified by a different token. This is to avoid non-realizable interprocedural paths. The seminal papers [CC77c, SP81, JM82] present systematic and direct methods to perform this extension. [LR92] uses alias sets of size one as tokens, [CBC93] uses calling points and/or incoming alias sets (which can result in exponential behaviour, see [ML$^+$93]). We clearly separated the basic interprocedural framework from its extensions, unlike [LR92, ML$^+$93]. Our framework can therefore easily be extended to an arbitrary token set. This is an issue orthogonal to the present contribution. *Pointers to functions* can be accomodated using a technique similar to [De90].

```
struct List *
Reverse(struct List *X,*Y) {
    struct List *p, *q;
    if (X == null)
      q = X;
    else {
      p = X->tl;
      X->tl = Y;
      q = Reverse(p, X);
    }
    return(q);
}
G2: l = Reverse(l, null);
G3:
```

Figure 11: A destructive list-reversal function

4 Complexity

We define: n the number of program points, m the maximal length of a normalised symbolic access path, A the number of distinct normalised symbolic access paths, and the parameter β, which varies between 1 (for control flow graphs with fixed outdegree) and 2 (for control flow graphs in which every program point depends on all the others). $h(v)$ is the height of the numerical lattice $\mathcal{V}^\sharp$ on v coefficient variables. In terms of the number of node evaluations, the worst case complexity of our analysis is $O(n^\beta \times A^2 \times h(2m))$. $h(v)$ is $v+1$ for the constant propagation lattice, $4v$ for the lattice of intervals, $v+1$ for the lattice of linear equalities and $8v^2 + 4v$ for the lattice of simple sections. m is the length of the longest access path that traverses each recursive pointer type at most once. In real programs, m is likely to be small and even bounded.

5 Prototype implementation

Our interprocedural program analysis framework has been prototyped in Standard ML, as a parametric module (functor) taking as a parameter a module implementing the numeric framework $\mathcal{V}^\sharp$. Excluding the numerical lattice, which is 2200 lines long, the implementation of the semilattice and its transfer functions requires 6000 lines of Standard ML. Symbolic alias relations are implemented by two-level *tries*: a first trie maps each symbolic access path to a trie mapping symbolic access paths to elements of the numerical lattice $\mathcal{V}^\sharp$. The numerical lattices we have experimented with are: (1) the lattice of arithmetic intervals [CC77a]; (2) the combination of the lattice of intervals and of the lattice of linear equalities [Ka76] (see [CC79] and [Gr92] for an explanation of how to devise an optimal combination). Data flow equations augmented with widening operators are solved using standard iterative techniques. Preliminary experimentation – not yet of statistical value – indicates that the number of iterations was less than 10 and took less than 30 seconds to analyse programs of less than about 50 lines.

6 Precision of the analysis

We have shown in the introduction that our framework can discover *position-dependent* aliasing properties. But how well does our framework perform when the exact dependence between aliased positions cannot be captured? Consider the program fragment shown in Figure 11. `Reverse` destructively reverses the list `X`, without introducing cycles. Exact relationships between initial and final positions in `X` cannot be captured, as it would require information about the length of `X`. Assume that l contains some sharing, for instance that its 10 first elements point to x. Aliasing at point G2 is thus:

$$\varrho_2 = \left\{ \begin{array}{l} (\langle *(l\!\!\to\!(tl\!\!\to)^i hd), *x\rangle, \mathcal{S}^\sharp\{i \leq 9\}), \\ (\langle *(l\!\!\to\!(tl\!\!\to)^i hd), *(l\!\!\to\!(tl\!\!\to)^j hd)\rangle, \mathcal{S}^\sharp\{i, j \leq 9\}) \end{array} \right\}$$

Our analysis discovers in four iterations over Reverse that aliasing at G3 is:

$$\varrho_3 = \left\{ \begin{array}{l} (\langle *(l\!\!\to\!(tl\!\!\to)^i hd), *x\rangle, \mathcal{S}^\sharp\{i \geq 0\}), \\ (\langle *(l\!\!\to\!(tl\!\!\to)^i hd), *(l\!\!\to\!(tl\!\!\to)^j hd)\rangle, \mathcal{S}^\sharp\{i, j \geq 0\}) \end{array} \right\}$$

We have correctly detected that no cycles have been introduced by Reverse. Such information is important for optimisation, for instance to perform software pipelining [HHN92, RF93]. In contrast, [LH88a, HPR89, De90, CWZ90, LR92, St92, CBC93] report that the list l may be cyclic at G3. [He90] would probably detect that l is not cyclic. However, as noted in [HHN92], [He90] is not of general applicability as it cannot handle graph-shaped data. The store-based methods [LH88a, HPR89, De90, St92, CBC93] fail because of their inability to distinguish an unbounded data structure from a cyclic data structure (this is independent of the parameter k of [St92, CBC93]). The addition of reference counting proposed by Chase *et al.* also fails, as discussed in their paper [CWZ90, p.309,§8]. Regardless of the value of the parameter k of their analysis, [LR92] also report a cyclic list as it detects spurious aliases of the form $\langle *(l\!\!\to\! tl\!\!\to\! tl), *(l\!\!\to\! tl\!\!\to\! tl\!\!\to\! tl)\rangle$.

Landi & Ryder's method is based on sets of pairs of k-limited access paths. It will not report aliasing when a data structure is completely unaliased, unlike store-based methods. However, as soon as *one* subcomponent of an object u located at depth $>k$ become aliased, spurious aliasing of *all* the subcomponents of u below depth k will be reported. This is a class of situations in which our method is markedly superior to k-limited methods.

7 Conclusion

Alias analysis for pointers is a long-standing and critical issue in optimising, verifying and parallelising imperative languages. It is becoming even more crucial since the advent of superscalar architectures and massively parallel processing, which place a higher demand on optimising compilers to restructure code.

Virtually every existing alias analysis method is based on two approximation techniques proposed by Jones and Muchnick: store-based approximations and k-limiting. As was pointed out by several researchers, these techniques are not sufficiently accurate to apply optimisation methods to programs with pointers.

Based on our previous theoretical results in semantics, formal language theory and abstract interpretation [De92b, De92a], we have proposed a method for may-alias analysis which radically departs from the currently prevalent store-based and k-limited approximation methods. The key concept is that of *symbolic access paths* qualified by integer coefficients denoting positions in data structures. Using existing numerical approximation techniques developed for scalar and array analysis, we can finitely represent the set of positions for which a given pair of symbolic access paths holds. We obtained thus a practical, flow-sensitive interprocedural analysis framework which can detect a new class of may-alias properties that were out of reach of existing alias analysis methods.

We have implemented a prototype to assess the practical feasibility of our approach. Preliminary experimentation demonstrates that our algorithm is significantly superior, in that it can extract accurate pointer information that other methods fail to detect, even on elementary pointer programs. Although we have not yet experimented with our approach on medium to large size programs, the parametric nature of our method gives us confidence about the scalability of our approach. We are currently undertaking systematic experimentation on real C programs.

We conjecture that many other applications of our original concept of *position-dependent properties* to the determination of properties of dynamically allocated pointer data structures are possible.

Acknowledgements. We would like to thank Keith Cooper, Patrick Cousot, Evelyn Duesterwald, Mooly Sagiv and Linda Torczon for their helpful comments.

References

[POP91] ACM Press. *Eighteenth Annual ACM Symp. on Principles of Programming Languages*, Orlando, FL, Jan. 1991.

[PLD92] ACM Press. *SIGPLAN'92 Conf. on Programming Language Design and Implementation*, volume 27(7) of *SIGPLAN Notices*, San Francisco, June 1992.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *SIGPLAN'89 Conf. on Programming Language Design and Implementation*, volume 24(7) of *SIGPLAN Notices*, pp. 41–53, June 1989.

[Br64] J. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Conf. on Programming Language Design and Implementation*, volume 25(6) of *SIGPLAN Notices*, pp. 296–310, June 1990.

[CBC93] J.D. Choi, M.G. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Twentieth Annual ACM Symp. on Principles of Programming Languages*, pp. 232–245. ACM Press, Jan. 1993.

[CCF91] J.D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse evaluation graphs. In [POP91].

[Co81] P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, pp. 303–342. Prentice-Hall, 1981.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Fourth Annual ACM Symp. on Principles of Programming Languages*, pp. 238–252, Jan. 1977.

[CC77b] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. *SIGPLAN Notices*, 12(3):77–94, Mar. 1977.

[CC77c] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Working Conf. on Formal Description of Programming Concepts*. IFIP WG 2.2, North-Holland, Aug. 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Sixth Annual ACM Symp. on Principles of Programming Languages*, pp. 269–282, 1979.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening-narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, 4th Intl. Symp, PLILP'92*, volume 631 of *Lecture Notes on Computer Science*, pp. 269–295. Springer Verlag, Aug. 1992.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth Annual ACM Symp. on Principles of Programming Languages*, pp. 84–97, Jan. 1978.

[De90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pp. 157–168. ACM Press, Jan. 1990.

[De92a] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data.* PhD thesis, LIX, Ecole Polytechnique, F-91128, Palaiseau, France, 1992.

[De92b] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In [ICC92], pp. 2–13.

[DGS94] E. Duesterwald, R. Gupta, and M.L. Soffa. Reducing the cost of data flow analysis by congruence partitionning. In *International Conference on Compiler Construction*, to appear in the Springer Verlag Lecture Notes in Computer Science, Apr. 1994.

[Ei74] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974.

[Gr89] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[Gr91] P. Granger. Static analysis on linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493 of *Lecture Notes on Computer Science*, pp. 169–192. Springer Verlag, 1991.

[Gr92] P. Granger. Improving the results of static analyses of programs by local decreasing iterations (extended abstract). In *Proc. 12th Conference of Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes on Computer Science, pp. 68–79. Springer Verlag, Dec. 1992.

[Ha79] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* PhD thesis, Université Scientifique et Médicale de Grenoble & Institut National Polytechnique de Grenoble, Grenoble, France, Mar. 1979.

[Ha89] W.L. Harrison. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, Oct. 1989.

[He88] L. Hederman. Compile time garbage collection. Master's thesis, Rice University, Houston, Aug. 1988. Tech. report COMP TR88-75.

[He90] L. Hendren. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Processing*, 1:35–47, Jan. 1990.

[HG92] L.J. Hendren and G.R. Gao. Designing programming languages for analysability: a fresh look at pointer data structures. In [ICC92], pp. 242–251.

[HHN92] L.J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In [PLD92], pp. 249–260.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conf. on Programming Language Design and Implementation*, volume 24(7) of *SIGPLAN Notices*, pp. 28–40, June 1989.

[Hu86] P. Hudak. A semantic model of reference counting and its abstraction. In *Conf. Record of the 1986 ACM Symp. on LISP and Functional Programming*, pp. 351–363, Aug. 1986.

[ICC92] *Proc. of the IEEE 1992 International Conf. on Computer Languages*, San Francisco, Apr. 1992. IEEE Press.

[Jo81] N.D. Jones. Flow analysis of lambda expressions. In *Symp. on Functional Languages and Computer Architecture*, pp. 376–401. Chalmers University of Technology, June 1981.

[JM81] N.D. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pp. 102–131. Prentice-Hall, 1981.

[JM82] N.D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Ninth Annual ACM Symp. on Principles of Programming Languages*, pp. 66–74. ACM Press, 1982.

[Jo81] H.B.M. Jonkers. Abstract storage structures. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pp. 321–343. IFIP, North Holland, 1981.

[Ka76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[Ki73] G. Kildall. A unified approach to global program optimisation. In *ACM Symp. on Principles of Programming Languages*, pp. 194–206, 1973.

[La92a] W. Landi. *Interprocedural Aliasing in the Presence of Pointers.* PhD thesis, Rutgers University, Jan. 1992.

[La92b] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992.

[LR91] W. Landi and B.G. Ryder. Pointer-induced aliasing. In [POP91], pp. 93–103.

[LR92] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In [PLD92], pp. 235–248.

[LH88a] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pp. 21–34, June 1988.

[LH88b] J.R. Larus and P.N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *ACM SIGPLAN'88 Conf. on Parallel Programming: Experiences with Applications, Languages and Systems*, pp. 100–110, June 1988.

[ML+93] T.J. Marlowe, W.G. Landi, B.G. Ryder, J.D. Choi, M.G. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, Sept. 1993.

[MR89] T.J. Marlowe and B.G. Ryder. Hybrid incremental alias algorithms. Tech. report LCSR-TR-129, Rutgers University, Oct. 1989.

[Ma91] F. Masdupuy. Using abstract interpretation to detect array data dependencies. In *Proc. of the International Symp. on Supercomputing*, pp. 19–27. Kyushu University Press, Nov. 1991. ISBN 4-87378-284-8.

[Mo84] E. Morel. Data flow analysis and global optimisation. In B. Lorho, editor, *Methods and Tools for Compiler Construction, an Advanced Course*, pp. 289–315. Cambridge University Press, 1984.

[NPD87] A. Neirynck, P. Panangaden, and A.J. Demers. Computation of aliases and support sets. In *Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pp. 274–283, 1987.

[Pa66] R.J. Parikh. On context-free languages. *J. ACM*, 13:570–581, 1966.

[RF93] B. Rau and J. Fisher. Instruction-level parallel processing. *The Journal of Supercomputing*, 7:9–50, 1993.

[RM88] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Fifteenth Annual ACM Symp. on Principles of Programming Languages*, pp. 285–293. ACM Press, Jan. 1988.

[SF+90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhem. A logic-based approach to data flow analysis. In *Programming Language Implementation and Logic Programming*, volume 456 of *Lecture Notes on Computer Science*, pp. 277–292. Springer Verlag, Aug. 1990.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pp. 189–234. Prentice-Hall, 1981.

[Sh91] O. Shivers. *Control-flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, Pittsburgh, May 1991. CMU-CS-91-145.

[St92] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Information and Computation*, 101(1):70–102, Nov. 1992.

[Wa91] D.W. Wall. Limits of instruction-level parallelism. In *Proc. ASPLOS III*, pp. 176–178, Apr. 1991.

[We80] W.E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Seventh Annual ACM Symp. on Principles of Programming Languages*, pp. 83–94, 1980.

A Appendix

Generating bases for mutually recursive pointer types. Given the mutually recursive types $t_1, \ldots, t_n$, we define $Basis(t_i)$ as the minimal set of access paths B such that B^* generates all paths mapping objects of type t_i to objects of type t_i without traversing objects of type t_j with $j < i$. This ordering condition is necessary to ensure that each path from t_j to itself that traverses t_i has a unique factorisation in $Basis(t_j)$. See Figure 12 for an example.

The algorithm *Match*. $Match_{\bowtie}(M, N)$ is defined by iterative decomposition. The only more complex case occurs when the leading term of M is an iterated basis B^k such no proper prefix p of N consisting only of accessors is properly in B and that some p is in a prefix of B. For instance: $Match_{\bowtie}(B^k M', sons{\rightarrow}tl{\rightarrow}N')$ with $B = sons{\rightarrow}(tl{\rightarrow})^* hd$. In this case we compute the *derivative* of B w.r.t. $sons{\rightarrow}tl{\rightarrow}$

```
struct MTree { char *key; struct TreeList *sons; }
struct TreeList { struct MTree *hd;
                  struct TreeList *tl; }
```

$Basis(struct\ MTree) = sons{\rightarrow}(tl{\rightarrow})^* hd{\rightarrow}$
$Basis(struct\ TreeList) = \{tl{\rightarrow}\}$

Figure 12: Mutually recursive pointer types and their corresponding bases

(see [Br64]), which yields a set $\{B'_1, \ldots, B'_n\}$ of regular expressions in monomial form [Ei74] (no $\cup$ appears in B'_i unless it is contained in a star expression). For our example, we get one regular expression $\{(tl{\rightarrow})^* hd\}$, and the matching proceeds with $Match_{\bowtie}((tl{\rightarrow})^{k'} hd.M', N')$.

The algorithm *Factor* **(Figure 15).** The normalisation of a SAP f is performed in three steps. First, subsequences of (a copy f' of) f beginning with a selector σ and traversing a recursive type t are replaced by a term of the form B^k, where B is the basis of the type t. The system of equations S is augmented with either $k = 0$ (if the traversal is partial) or $k = 1$ (if the subsequence performs a full traversal of t). Second, if the type of f' is a recursive type, a basis B^k is appended to f, and $k = 0$ is recorded in S. The third step simplifies f' by replacing occurences of the form $B^k.B^{k'}$ by $B^{k''}$ and recording the equation $k'' = k + k'$. We then extend *Factor* to symbolic alias relations in Figure 16.

The widening operator ∇ (Figure 14). Given two symbolic alias relations ϱ_1 and ϱ_2, their widening $\varrho_1 \nabla \varrho_2$ is computed by first normalising their SAPs (using *Factor*). The `foreach` loop then applies the widening operator associated to the numeric lattice $\mathcal{V}^\sharp$ to the numeric spaces K and K' of each symbolic alias pair defined in both ϱ_1 and ϱ_2.

Placement of widenings. Widening operators must be inserted in the interprocedural data flow equations [CC77a] as follows: (1) determine a *feedback set* W of the dependence graph of the equations such that any cycle traverses at least a node from W; (2) if the data flow equation $X_i = t(X_j)$ is in the feedback set, with t some transfer function, then replace it by $X_i = X_i \nabla t(X_j)$. W can be defined, for instance, as the set of (interprocedural) loop headers, see [Co81, p.334].

The algorithm *EquivalenceClass*$^\sharp$ **(Figure 17).** The function $EquivalenceClass^\sharp(\pi, \varrho)$ computes a symbolic path set P representing the set of access paths to which π is aliased in ϱ as follows: each pair $(\langle f_1, f_2\rangle, K)$ of ϱ is examined (with $Match_\in$) to check if f_1 (resp. f_2) can generate a prefix of π. In this case, the SAP Δ represents the paths which must be appended to f_1 (resp. f_2) to generate π (this is necessary because of right-regular reduction). The system of numeric equations S describes the values of the coefficients of f_1 (resp. f_2) for which f_1 (resp. f_2) generates a prefix of π, and the corresponding values of the coefficients of Δ. The numeric space K is then intersected with S, and projected onto the coefficients occuring in f_2 (resp. f_1) and Δ, yielding K'. For the example of Figure 17, we have: $S = \{i{=}1\}$ and $\Delta = \{hd\}$, $\mathcal{C}^\sharp(\mathcal{S}^\sharp\{i{=}j{-}1\}, S) = \mathcal{S}^\sharp\{i{=}j{-}1, i{=}1\}$ and $K' = \mathcal{S}^\sharp\{j{=}2\}$. The pair $(f_2.\Delta, K')$ is then added to P. Finally, P is adjusted to take reflexivity into account. The remaining algorithms, see Figures 18, 19, 20 and 21, are based on conceptually similar mechanisms.

Pointers to functions. Among the global variables, we distinguish the set F of function names. Given a particular function name f, the assignment $p = f$ generates (in particular) the alias pair $(\langle *p, *f\rangle, \top)$. To analyse the higher-order function call $\mathtt{y} = (*\mathtt{q})(\mathtt{a_1}, \ldots, \mathtt{a_n})$, where q is of function pointer type, we compute the symbolic path set $Q = EquivalenceClass^\sharp(*q, \varrho_{\text{call}})$. Q then contains directly the set of function names potentially called. This is a technique similar to [De90].

Exploiting sparsity of data flow equation systems. Alias analysis is an interesting candidate for *sparse evaluation graph* techniques [CCF91, DGS94]. These methods simplify data flow equations by eliminating copies and exploiting idempotence of the join (or meet) operator. Oppurtunities for such simplifications occur in data flow equations for alias analysis: copies occur because of statements that do not involve pointers, and joins can be typically eliminated at the end of conditionals that does not involve pointers. We are investigating the incorporation of the new approach [DGS94] in our analyser and plan to evaluate the impact on performance.

Compaction methods. Because aliasing is symmetric, we perform *symmetric reduction* by: (1) defining a total order $\leq$ on SAPs (which ignores coefficient names); (2) enforcing that each symbolic alias pair $(\langle f, g\rangle, K)$ satisfies $f \leq g$. This can divide the number of alias pairs by two. Because aliasing is reflexive, we perform *reflexive reduction*, by discarding a symbolic alias pair $(\langle f, g\rangle, K)$ if it generates only reflexive alias pairs. Because f and g can be symbolic, reflexive reduction is based on the *Match* operation. Not every alias analysis can perform reflexive reduction in general. For instance, an alias pair $\langle u, v\rangle$ where u and v are of length k cannot be safely removed in the analysis of [LR92]. Because aliasing is right-regular (e.g x aliased to y implies $x.\delta$ aliased to $y.\delta$) we also perform *right-regular reduction*. The symbolic alias pairs produced by our framework are generally not right-regularly closed, but there are nevertheless opportunities for right-regular reduction. These three reduction methods should not be applied to the sets of symbolic alias pairs Θ used in $Call^\sharp$ and $Return^\sharp$, as they do not denote symmetric relations. These reductions can however safely be applied to all other symbolic alias relations. Unlike the transitive reduction method proposed in [CBC93], these compaction methods provably do not result in loss of precision.

Generation of data flow equations. We have explained how to handle assignments, function calls and returns. We now illustrate the translation of other statements through the example in Figure 13.

Function return values are handled by assigning the return value to the global variable *result* (see C_{exit_1}, C_{exit_2}).

Because `Copy` is recursive, the dependence graph of the equations in Figure 13 contains two cycles: $C_{entry} \rightarrow C_{in_1} \rightarrow C_5 \rightarrow \cdots \rightarrow C_1 \rightarrow C_{entry}$ and $C_{exit} \rightarrow C_{exit_2} \rightarrow C_7 \rightarrow C_6 \rightarrow C_{6,1} \rightarrow C_{exit}$. Two widening operations have thus been inserted, one at function entry (see C_{entry}) and one at function exit (see C_{exit}).

Conditional branches guarded by pointer comparisons can be taken into account [CC77c, p.271]. For instance, the transfer function corresponding to `if (L == null) then ...` is $\text{Kill}^\sharp(*L)$ (see C_2). Other predicates, such as pointer equality testing can be handled similarly.

$$
\begin{aligned}
C_{entry} &= C_{entry} \nabla (C_{in_1} \sqcup C_{in_2}) \\
C_1 &= C_{entry} \\
C_2 &= \mathrm{Kill}^\sharp(*L)(C_1) \\
C_{exit_1} &= \mathrm{Gen}^\sharp(*result, *L)(C_2) \\
C_3 &= C_1 \\
C_4 &= \mathrm{Kill}^\sharp(*p)(C_3) \\
C_5 &= \mathrm{Gen}^\sharp(*t_1, *(L \to tl))(C_4) \\
(C_{in_1}, C_{through_1}, \Theta_1) &= Call^\sharp\{(L, t_1)\}(C_5) \\
C_{6,1} &= Return^\sharp(C_{exit}, C_{through_1}, \Theta_1) \\
C_{6,2} &= \mathrm{Kill}^\sharp(*(P \to tl))(C_{6,1}) \\
C_{6,3} &= \mathrm{Gen}^\sharp(*(P \to tl), *result)(C_{6,2}) \\
C_6 &= \mathrm{Kill}^\sharp(*result)(C_{6,3}) \\
C_{7,1} &= \mathrm{Kill}^\sharp(*(P \to hd))(C_6) \\
C_7 &= \mathrm{Gen}^\sharp(*(P \to hd), *(L \to hd))(C_{7,1}) \\
C_{exit_2} &= \mathrm{Gen}^\sharp(*result, *p)(C_7) \\
C_{exit} &= C_{exit} \nabla (C_{exit_1} \sqcup C_{exit_2})
\end{aligned}
$$

$$
\begin{aligned}
L_{2,1} &= \mathrm{Gen}^\sharp(*t_2, *X)(L_1) \\
(C_{in_2}, C_{through_2}, \Theta_2) &= Call^\sharp\{(L, t_2)\}(L_{2,1}) \\
L_{2,2} &= Return^\sharp(C_{exit}, C_{through_2}, \Theta_2) \\
L_{2,3} &= \mathrm{Kill}^\sharp(*Y)(L_{2,2}) \\
L_{2,4} &= \mathrm{Gen}^\sharp(*Y, *result)(L_{2,3}) \\
L_2 &= \mathrm{Kill}^\sharp(*result)(L_{2,4}) \\
L_3 &= \mathrm{Kill}^\sharp(*X)(L_2)
\end{aligned}
$$

Figure 13: Data flow equations corresponding to the program of Figure 1

`Algorithm` ∇ (Widening on symbolic alias relations)
`Input:` two symbolic alias relations $\varrho_1, \varrho_2 \in \mathrm{UR}(\mathcal{V}^\sharp)$
`Output:` their widening $\varrho_1 \nabla \varrho_2$
`Method:`
  ϱ `:=` $\emptyset$;
  ϱ_1 `:=` $Factor(\varrho_1)$;
  ϱ_2 `:=` $Factor(\varrho_2)$;
  `foreach symbolic alias pair` $(\langle f_1, f_2\rangle, K) \in \varrho_1$ `do`
    `if there exists a pair` $(\langle f_1, f_2\rangle, K') \in \varrho_2$ `then`
      ϱ `:=` $\varrho \cup \{(\langle f_1, f_2\rangle, K \nabla K')\}$;
      ϱ_2 `:=` $\varrho_2 - \{(\langle f_1, f_2\rangle, K')\}$;
    `else`
      ϱ `:=` $\varrho \cup \{(\langle f_1, f_2\rangle, K)\}$;
  `return` $\varrho \cup \varrho_2$;

Example:
let $\varrho_1 = \{(\langle *(X \to hd), *(Y \to hd)\rangle, \top)\}$
  $\varrho_2 = \{(\langle *(X \to tl \to hd), *(Y \to tl \to hd)\rangle, \top)\} \cup \varrho_1$
if $\mathcal{V}^\sharp$ is the lattice of arithmetic intervals [CC77a]:
  $\varrho_1 \nabla \varrho_2 = \{(\langle *(X \to (tl \to)^i hd), *(Y \to (tl \to)^j hd)\rangle, \mathcal{S}^\sharp\{0 \le i, j \le 1\})\}$
if $\mathcal{V}^\sharp$ is Karr's lattice [Ka76]:
  $\varrho_1 \nabla \varrho_2 = \{(\langle *(X \to (tl \to)^i hd), *(Y \to (tl \to)^j hd)\rangle, \mathcal{S}^\sharp\{i = j\})\}$

Figure 14: The widening operator ∇

`Algorithm` $Factor(f)$
`Input:` a symbolic access path f
`Output:` a normalised symbolic access path f' and a system of linear equations S relating the variables of f and f'
`Method:`
 S `:=` $\emptyset$; f' `:=` f;
 `apply the following to` f' `in left to right order:`
  `let` $f' = e_1 \ldots\ e_i \ldots\ e_n$ `such that:`
   `(1)` e_i `is a selector` $e_i \in \Sigma$ `and`
   `(2)` $Typeof(e_1 \ldots e_{i-1})$ `is a recursive type` t;
  `let` B `:=` $Basis(t)$ `and` k `be a fresh variable;`
  `if there exists a minimal` j *in* $[i+1, n]$
    `such that` $Typeof(e_i \ldots e_j) = t$ `then`
    f' `:=` $e_1 \ldots e_{i-1}.B^k.e_{j+1} \ldots e_n$; S `:=` $S \cup \{k = 1\}$;
  `else`
    f' `:=` $e_1 \ldots e_{i-1}.B^k.e_i \ldots e_n$; S `:=` $S \cup \{k = 0\}$;
  `fi`
 `if` $Typeof(f')$ `is a recursive type` t `then`
  B `:=` $Basis(t)$ `and let` k `be a fresh variable;`
  f' `:=` $f'.B^k$; S `:=` $S \cup \{k = 0\}$;
 `fi`
 `exhaustively apply the following to` f'`:`
  `if` f' `is of the form` $e_1 \ldots e_{i-1}.B_t^k.B_t^{k'}.e_{i+2} \ldots e_n$ `then`
   `let` k'' `be a fresh variable;`
   f' `:=` $e_1 \ldots e_{i-1}.B_t^{k''}.e_{i+2} \ldots e_n$;
   S `:=` $S \cup \{k'' = k + k'\}$
  `fi`
 `return` (f', S)

We also define a similar algorithm, $Factor(g, ty)$, where g is a symbolic access path g, and ty is a type name. g is a partial access path which can be applied to objects of type ty. This is used by $StarClosure^\sharp$.

Example:
 $Factor(X \to tl \to (tl \to)^i hd) = (X \to (tl \to)^j hd, \{j = i + 1\})$

 $Factor(tl \to tl \to hd, struct\ List) = ((tl \to)^j hd,\ \{j = 2\})$

Figure 15: The normalisation algorithm $Factor(f)$

`Algorithm` $Factor(\varrho)$
`Input:` a symbolic alias relation $\varrho \in \mathrm{UR}(\mathcal{V}^\sharp)$
`Output:` a normalised symbolic alias relation $Factor(\varrho)$
`Method:`
 ϱ' `:=` $\emptyset$;
 `foreach symbolic alias pair` $(\langle f_1, f_2\rangle, K) \in \varrho$ `do`
  (g_1, S_1) `:=` $Factor(f_1)$;
  (g_2, S_2) `:=` $Factor(f_2)$;
  K' `:=` $Project^\sharp(\mathcal{C}^\sharp(K, S_1 \cup S_2), fv(g_1) \cup fv(g_2))$
  ϱ' `:=` $\varrho' \sqcup Rename\{(\langle g_1, g_2\rangle, K')\}$
 `done;`
 `return` ϱ'

The algorithm $Factor$ used above is defined in Figure 15.

Example:
 $Factor\{(\langle T \to left \to right \to key, K\rangle, \top)\}$
  $= \{(\langle T \to \{left \to, right \to\}^{k_1} key, K\rangle, \mathcal{S}^\sharp\{k_1 = 2\})\}$

Figure 16: The normalisation algorithm $Factor(\varrho)$

`Algorithm` $EquivalenceClass^\sharp(\pi, \varrho)$
`Input:` an access path π, a symbolic alias relation ϱ
`Output:` a symbolic path set P
`Method:`
  P `:=` $\emptyset$;
  `foreach symbolic alias pair` $(\langle f_1, f_2\rangle, K) \in \varrho$ `do`
    `foreach` $(S, \Delta) \in Match_{\in}(\pi, f_1)$ `do`
      K' `:=` $Project^\sharp(\mathcal{C}^\sharp(K, S), fv(f_2.\Delta))$;
      `if` $(K' \neq \bot)$ `then` P `:=` $P \cup \{(f_2.\Delta, K')\}$;
    `done;`
    `foreach` $(S, \Delta) \in Match_{\in}(\pi, f_2)$ `do`
      K' `:=` $Project^\sharp(\mathcal{C}^\sharp(K, S), fv(f_1.\Delta))$;
      `if` $(K' \neq \bot)$ `then` P `:=` $P \cup \{(f_1.\Delta, K')\}$;
    `done`
  `done;`
  P `:=` $P \cup \{(\pi, \top)\}$;
  `return` P

Example: if $\varrho = \{(\langle Y \to (tl\to)^i, L \to (tl\to)^j\rangle, \mathcal{S}^\sharp\{i = j - 1\})\}$ then:
$EquivalenceClass^\sharp(Y \to tl \to hd, \varrho)$
$= \{(Y \to tl \to hd, \top),\ (L \to (tl\to)^j hd, \mathcal{S}^\sharp\{j = 2\})\}$

Figure 17: Equivalence class of an access path

`Algorithm` $StripPrefix^\sharp(\pi, P)$
`Input:` an access path π, a symbolic path set P
`Output:` a symbolic path set P'
`Method:`
  P' `:=` $\emptyset$;
  `foreach` $(f, K) \in P$
    `foreach` $(S, \Delta) \in Match_{\ni}(f, \pi)$ `do`
      K' `:=` $Project^\sharp(\mathcal{C}^\sharp(K, S), fv(\Delta))$;
      `if` $(K' \neq \bot)$ `then` P `:=` $P \cup \{(\Delta, K')\}$
    `done;`
  `return` P'

Example:
$StripPrefix^\sharp(X \to tl, \{(X \to (tl\to)^{k_1} hd, \mathcal{S}^\sharp\{k_1 \geq 2\})\})$
$= \{(\to (tl\to)^{k_2} hd, \mathcal{S}^\sharp\{k_2 \geq 1\})\}$

Figure 18: Algorithm $StripPrefix^\sharp$

`Algorithm` $P.Q$ (Concatenation of symbolic path sets)
`Input:` two symbolic path sets P, Q
`Output:` a symbolic path set $P.Q$ denoting the concatenation of the access paths denoted by P and Q
`Method:`
  U `:=` $\emptyset$;
  `rename` P `so that the coefficients appearing in` P `and` Q `are distinct;`
  `foreach` $(f_1, K_1) \in P$
    `foreach` $(f_2, K_2) \in P$
      U `:=` $U \cup \{(f_1.f_2, K_1 \wedge_h K_2)\}$
  `return` U

Example: `let` $P = \{(L \to (tl\to)^j hd \to, \mathcal{S}^\sharp\{j = 2\})\}$
`and` $Q = \{((tl\to)^k hd, \mathcal{S}^\sharp\{k \geq 1\})\}$, `then:`
$P.Q = \{(L \to (tl\to)^j hd \to (tl\to)^k hd, \mathcal{S}^\sharp\{j = 2, k \geq 1\})\}$

Figure 19: Concatenation of symbolic path sets

`Algorithm` $Rewrite^\sharp_{left}(\pi, P)(\varrho)$
`Input:` an access path π, a symbolic path set P and a symbolic alias relation ϱ
`Output:` a symbolic alias relation ϱ'
`Method:`
  ϱ' `:=` $\emptyset$;
  `rename` P `s.t. coeffs. of` P `and` ϱ `are disjoint;`
  `foreach symbolic alias pair` $(\langle f_1, f_2\rangle, K) \in \varrho$ `do`
    `foreach` $(S, \Delta) \in Match_{\in}(\pi, f_1)$ `do`
      `foreach` $(g, K') \in P$ `do`
        K'' `:=` $Project^\sharp(\mathcal{C}^\sharp(K \wedge_h K', S), fv(g) \cup fv(f_2.\Delta))$;
        `if` $(K'' \neq \bot)$ `then`
          ϱ' `:=` $\varrho' \sqcup Rename\{(\langle g, f_2.\Delta\rangle, K'')\}$
      `done;`
    `foreach` $(S, \Delta) \in Match_{\ni}(f_1, \pi)$ `do`
      `foreach` $(g, K') \in P$ `do`
        K'' `:=` $Project^\sharp(\mathcal{C}^\sharp(K \wedge_h K', S), fv(g.\Delta) \cup fv(f_2))$;
        `if` $(K'' \neq \bot)$ `then`
          ϱ' `:=` $\varrho' \sqcup Rename\{(\langle g.\Delta, f_2\rangle, K'')\}$
      `done`
  `done;`
  `return` ϱ'

`/*` $Rewrite^\sharp_{right}$ is defined similarly `*/`

`Algorithm` $Rewrite^\sharp(\pi, P)(\varrho)$
`Input:` an access path π, a symbolic path set P and a symbolic alias relation ϱ
`Output:` a symbolic alias relation ϱ'
`Method:`
  ϱ' `:=` $Rewrite^\sharp_{left}(\pi, P)(\varrho)$;
  ϱ' `:=` $\varrho' \sqcup Rewrite^\sharp_{right}(\pi, P)(\varrho \sqcup \varrho')$;
  ϱ' `:=` $\varrho' \sqcup Rename\{(\langle f, \pi\rangle, K) \mid (f, K) \in P\}$;
  `return` ϱ'

Figure 20: The operator $Rewrite^\sharp$

`Algorithm` $Generalise^\sharp((\langle f_1, f_2\rangle, K), \varrho_{\text{entry}}, \Theta)$
`Input:` a symbolic alias pair $(\langle f_1, f_2\rangle, K)$, the symbolic alias relations ϱ_{entry} and Θ
`Output:` the symbolic alias relations ϱ'_{entry} and Θ'
`Method:`
  f'_1 `:=` $MakeGenericName(f_2)$;
  `let` $\langle u_1, \ldots, u_n\rangle = fv(f_2)$ `and` $\langle v_1, \ldots, v_n\rangle = fv(f'_1)$;
  S `:=` $\{u_1 = v_1, \ldots, u_n = v_n\}$;
  ϱ'_{entry} `:=` $\varrho_{\text{entry}} \sqcup Rename\{(\langle f'_1, f_2\rangle, \mathcal{S}^\sharp(S))\}$;
  K' `:=` $Project^\sharp(\mathcal{C}^\sharp(K, S), fv(f'_1) \cup fv(f_1))$;
  Θ' `:=` $\Theta \cup Rename\{(\langle f'_1, f_1\rangle, K')\}$;
  `return`$(\varrho'_{\text{entry}}, \Theta')$

Example:
$Generalise^\sharp((\langle *(l \to tl), *(a \to (tl\to)^i hd)\rangle, \mathcal{S}^\sharp\{i \geq 2\}), \emptyset, \emptyset)$
$= (\varrho_{\text{entry}}, \Theta)$
`with` $\varrho_{\text{entry}} = \{(\langle U_1[j], *(a \to (tl\to)^i hd)\rangle, \mathcal{S}^\sharp\{i = j\})\}$
`and` $\Theta = \{(\langle U_1[k], *(l \to tl)\rangle, \mathcal{S}^\sharp\{k \geq 2\})\}$

The operator $MakeGenericName(f)$ returns a symbolic access path f' consisting of a generic object $U[k_1, \ldots, k_n]$, where n is the number of coefficient variables occurring in f. The name U is determined uniquely from f, ignoring coefficient names and the $k_1, \ldots, k_n$ are fresh variables.

Figure 21: Generalisation of symbolic access paths