Points-to Analysis in Almost Linear Time

Bjarne Steensgaard

Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

rusa@research.microsoft.com

Abstract

We present an interprocedural flow-insensitive points-to analysis based on type inference methods with an almost linear time cost complexity. To our knowledge, this is the asymptotically fastest non-trivial interprocedural points-to analysis algorithm yet described. The algorithm is based on a non-standard type system. The type inferred for any variable represents a set of locations and includes a type which in turn represents a set of locations possibly pointed to by the variable. The type inferred for a function variable represents a set of functions it may point to and includes a type signature for these functions. The results are equivalent to those of a flow-insensitive alias analysis (and control flow analysis) that assumes alias relations are reflexive and transitive.

This work makes three contributions. The first is a type system for describing a universally valid storage shape graph for a program in linear space. The second is a constraint system which often leads to better results than the "obvious" constraint system for the given type system. The third is an almost linear time algorithm for points-to analysis by solving a constraint system.

1 Introduction

Modern optimizing compilers and program understanding and browsing tools for pointer languages such as C [KR88] are dependent on semantic information obtained by either an alias analysis or a points-to analysis. Alias analyses compute pairs of expressions (or access paths) that may be aliased (*e.g.*, [LR92, LRZ93]). Points-to analyses compute a store model using abstract locations (*e.g.*, [CWZ90, EGH94, WL95, Ruf95]).

Most current compilers and programming tools use only intraprocedural analyses, as the polynomial time and space complexity of the common data-flow based analyses prevents the use of interprocedural analyses for large programs. Interprocedural analysis is becoming increasingly important, as it is necessary to support whole-program optimization and various program understanding tools. Previously published interprocedural analysis algorithms have not been reported to have been successfully applied to programs around 100,000 lines of C code (the published results are practically all for less than 10,000 lines of C code).

We present a flow-insensitive interprocedural points-to analysis algorithm that has a desirable linear space and almost linear time complexity and is also very fast in practice. The algorithm is easily applicable to programs with many hundreds of thousands of lines of code. The analysis results are often not as accurate as those obtained by flow-sensitive analyses. However, the results are roughly comparable to those of, *e.g.*, the cubic time complexity flow insensitive analysis of [Wei80].

The algorithm, which is inspired by Henglein's binding time analysis by type inference [Hen91], uses a non-standard type system to describe the store usage at runtime by using types to construct a storage shape graph [CWZ90]. While we describe the principles behind the algorithm in terms of types and typing rules, we also provide a detailed description of the algorithm which can be used almost directly to implement the algorithm in a compiler.

In Section 2 we state the source language for which we describe the algorithm. The language captures the essential parts of a language like C. In Section 3 we define the non-standard set of types we use to model the storage use, and in Section 4 we state a set of typing rules for programs. The typing rules impose constraints on the relationships of types of program variables. Finding a typing of the program that obeys these constraints amounts to performing a points-to analysis. In Section 5 we show how to efficiently infer the minimal typing that obeys the constraints. In Section 6 we report on practical experience with the algorithm in a C programming environment. In Section 7 we describe related work, and in Section 8 we present our conclusions and point out directions for future work.

2 The source language

We describe the points-to analysis for a small imperative pointer language which captures the important properties of languages like C [KR88]. The language includes pointers to locations, pointers to functions, dynamic allocation, and computing addresses of variables. Since the analysis is flow insensitive, the control structures of the language are irrelevant. The abstract syntax of the relevant statements of the language is shown in Figure 1.

The syntax for computing the addresses of variables and for pointer indirection is borrowed from the C programming language [KR88]. All variables are assumed to have unique names. The $\mathsf{op}(\ldots)$ expression form is used to describe primitive computations such as arithmetic operations and computing offsets into aggregate objects. The $\mathsf{allocate}(\mathsf{y})$ expression dynamically allocates a block of memory of size y.

Functions are constant values described by the $\mathsf{fun}(\ldots)\rightarrow(\ldots)S^*$ expression form[1]. The f_i variables are formal parameters (some-

[1] We have generalized function definitions to allow functions with multiple return values; a feature not found in C.

$$
\begin{array}{rcl}
S & ::= & \mathsf{x = y} \\
 & | & \mathsf{x = \&y} \\
 & | & \mathsf{x = *y} \\
 & | & \mathsf{x = op(y_1 \ldots y_n)} \\
 & | & \mathsf{x = allocate(y)} \\
 & | & \mathsf{*x = y} \\
 & | & \mathsf{x = fun(f_1 \ldots f_n) \rightarrow (r_1 \ldots r_m)}\ S^* \\
 & | & \mathsf{x_1 \ldots x_m = p(y_1 \ldots y_n)}
\end{array}
$$

Figure 1: Abstract syntax of the relevant statements, *S*, of the source language. x, y, f, r, and p range over the (unbounded) set of variable names and constants. op ranges over the set of primitive operator names. S^* denotes a sequence of statements. The control structures of the language are irrelevant for the purposes of this paper.

```
fact = fun(x)→(r)
  if lessthan(x 1) then
    r = 1
  else
    xminusone = subtract(x 1)
    nextfac = fact(xminusone)
    r = multiply(x nextfac)
  fi

result = fact(10)
```

Figure 2: A program in the source language that computes factorial(10).

times called *in parameters*), and the r_i variables are return parameters (sometimes called *out parameters*). Function calls have call-by-value semantics [ASU86]. Both formal and return parameter variables may appear in left-hand-side position in statements in the function body. Formal and return parameter variables as well as local variables may not occur in the body of another function; this is always true for C programs, which are not allowed to contain nested function definitions.

Figure 2 shows an implementation of the factorial function (and a call of same) in the abstract syntax of the source language.

We assume that programs are as well-behaved as (mostly) portable C programs. The analysis tracks flow of pointer values, so the analysis algorithm may produce wrong results for programs that construct pointers from scratch (*e.g.*, by bitwise duplication of pointers) and for non-portable programs (*e.g.*, programs that rely on how a specific compiler allocates variables relative to each other). However, the analysis algorithm as presented below will deal with, *e.g.*, exclusive-or operations on pointer values, where there is a real flow of values.

3 Types

For the purposes of performing the points-to analysis, we define a non-standard set of types describing the store. The types have nothing to do with the types normally used in typed imperative languages (*e.g.*, `integer`, `float`, `pointer`, `struct`).

We use types to model how storage is used in a program at runtime (a storage model). Locations of variables and locations created by dynamic allocation are all described by types. Each type describes a set of locations as well as the possible runtime contents of those locations.

A type can be viewed as a node in a storage shape graph [CWZ90]. Each node may have edges to other nodes, which is modelled in the type system by letting types have type components. The storage shape graph may be cyclic for some programs, so the types may also be recursive.

The set of types inferred for the variables of a program represents a storage shape graph which is valid at all program points. The storage shape graph conservatively models all the points-to relations that may hold at runtime. Alias relations can also be extracted from the storage shape graph [EGH94].

Our goal is a points-to analysis with an almost linear time cost complexity. The size of the storage shape graph represented by types must therefore be linear in the size of the input program. Consequently, the maximum number of graph nodes must be linear in the size of the input program. Additionally, each graph node may not have more than a fixed number of out-going edges, meaning that each type may only have a fixed number of component types.

We describe the locations pointed to by a pointer variable by a single type. For composite objects (such as `struct` objects in C), we also describe all the elements of the object by a single type.

Describing each element in a composite object by separate types would, for most imperative languages, imply that the size of the storage shape graph could potentially be exponential in the size of the input program (*e.g.*, by extreme use of `typedef` and `struct` in C). Describing the elements of composite objects by separate types may still be desirable, as the sum of sizes of variables is unlikely to be exponential in the size of the input program. Extending the type system to do so is not addressed in the present paper.

The source language allows pointers to functions. Function pointer values are described by signature types describing the types of the argument and result values.

Values may be (or include) pointers to locations and pointers to functions. The type of a value must therefore accommodate both types of pointers. In our type system, a value type is therefore a pair including a location type and a function signature type.

The non-standard set of types used by our points-to analysis can be described by the following productions:

$$
\begin{array}{rcl}
\alpha & ::= & \tau \times \lambda \\
\tau & ::= & \perp \mid \mathbf{ref}(\alpha) \\
\lambda & ::= & \perp \mid \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})
\end{array}
$$

The α types describe values, the τ types describe locations (or pointers to locations), and the λ types describe functions (or pointers to functions).

Types may be recursive, and it may therefore be impossible to write out the types directly. The types can be written out by using type variables. Two types are not equal unless they either both are $\perp$ or are described by the same type variable. Note that this is different from the usual structural equality criterion on types. We could use the structural equality criterion if we added a tag to the τ and λ types.

4 Typing rules

In this section we define a set of typing rules based on the set of non-standard types defined in the previous section. The typing rules specify when a program is well-typed. A well-typed program is one for which the static storage shape graph indicated by the types is a safe (conservative) description of all possible dynamic (runtime) storage configurations. Before stating the typing rules, we argue for using inequalities rather than equalities in the typing rules and argue for the way we have defined the typing rule for statements with primitive operations.

Each location in the program is described by a single type. A pointer to a location is described by the type of the location pointed to. If several locations may contain a pointer to the same location, then the types of these locations must all have the same location type component. This requirement must be reflected in the typing rules.

Consider a simple assignment statement, x = y. Assume that x has type τ_x (meaning that the location allocated to hold the value of x has type τ_x) and that y has type τ_y. If a location pointer value may be assigned to x when executing the statement, then the location component of both τ_x and τ_y must be τ_p, where τ_p is the type describing the pointer value being assigned to x. If a function pointer value may be assigned, then τ_x and τ_y must have the same function signature component.

The "obvious" typing rule for simple assignment statements would be:

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha) \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

This rule states that this part of the program is well-typed under type environment A if the contents of variables x and y are described by the same type(s). Previous work has used this typing rule for simple assignment [Ste95a].

The above typing rule is, however, too strict. This is illustrated by the following sequence of statements:

```
a = 4
x = a
y = a
```

Using the above rule, the content components of the types for a, x, and y must all be the same. That is not strictly necessary, as no pointer value is ever assigned. If x and y are used in other parts of the program to hold pointers to disjoint locations, the above statements would unnecessarily force all the pointed-to locations to be described by the same type. Furthermore, if x is used in another part of the program to hold a pointer value, the analysis results will indicate that both y and a may hold the same pointer value, even if they are only assigned integer values in the program.

Given an assignment statement x = y, the content component types for x and y need only be the same if y may contain a pointer. In order to state this requirement in a typing rule, we introduce a partial order on types defined as follows:

$$t_1 \trianglelefteq t_2 \Leftrightarrow (t_1 = \bot) \vee (t_1 = t_2)$$

$$(t_1 \times t_2) \trianglelefteq (t_3 \times t_4) \Leftrightarrow (t_1 \trianglelefteq t_3) \wedge (t_2 \trianglelefteq t_4).$$

Given that non-pointers are represented by type $\bot$, the requirement can now by expressed by the following typing rule:

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

The rule states that each component type of α_2 must be either $\bot$ or equal to the corresponding component type of α_1.

In statements of the form x = op($\mathsf{y}_1 \ldots \mathsf{y}_n$), the op operation may be a comparison, a bit-wise operation, an addition, etc. Consider a subtraction of two pointer values. The result is not a pointer value, but either of the two operand pointer values can be reconstituted from the result (given the other pointer value). The result must therefore be described by the same type as the two input pointer values.

There are operations from which argument pointer values cannot be reconstituted from the result (*e.g.*, comparisons: $<, \neq$, etc.). For such operations, the result is not required to be described by the same type as any input pointer values. For the purposes of this paper, we will treat all primitive operations identically.

In Figure 3 we state the typing rules for the relevant parts of the source language. A program is well-typed under typing environment A if all the statements of the program are well-typed under A. The typing environment A associates all variables with a type.

The typing rule for dynamic allocation implies that a location type is required to describe the value stored in the variable assigned to. The type used to describe the allocated location need not be the type of any variable in the program. The type of the allocated location is then only indirectly available through the type of the variable assigned to. All locations allocated in the same statement will have the same type, but locations allocated by different allocation statements may have different types.

Figure 4 contains an example program and a typing of all the variables occurring in the program that obeys the typing rules of Figure 3. Variables x and z must be described by the same type variable, as a single type variable must represent the locations pointed to by all the pointers possible stored in the location for variable y.

5 Efficient type inference

The task of performing a points-to analysis has now been reduced to the task of inferring a typing environment under which a program is well-typed. More precisely, the typing environment we seek is the minimal solution to the well-typedness problem, *i.e.*, each location type variable in the typing environment describes as few locations as possible. In this section we state how to compute such a minimal solution with an almost linear time complexity.

The basic principle of the algorithm is that we start with the assumption that all variables are described by different types (type variables) and then proceed to merge types as necessary to ensure well-typedness of different parts of the program. Merging two types means replacing the two type variables with a single type variable throughout the typing environment. Joining is made fast by using fast union/find data structures. We first describe the initialization and our assumptions about how the program is represented. Then we describe how to deal with equalities and inequalities in the typing rules in a manner ensuring that we only have to process each statement in the program exactly once. Finally we argue that the algorithm has linear space complexity and almost linear time complexity.

5.1 Algorithm stages

In the first stage of the algorithm, we provide a typing environment where all variables are described by different type variables. A type variable consists of a fast union/find structure (an equivalence class representative (ECR)) with associated type information. The type of each of the type variables in the typing environment is initially $\mathbf{ref}(\bot \times \bot)$. We assume that the program is represented in some program representation where name resolution has occurred, so we can encode the typing environment in the program representation and get constant time access to the type variable associated with a variable name.

In the next stage of the algorithm, we process each statement exactly once. Type variables are joined as necessary to ensure well-typedness of each statement (as described below). When joining two type variables, the associated type information is unified by computing the least upper bound of the two types, joining component type variables as necessary. Joining two types will never make a statement that was well-typed no longer be well-typed. When all program statements are well-typed, the program is well-typed.

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\tau \times \_) \\ A \vdash \mathsf{y} : \tau \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \&\mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha_1) \\ A \vdash \mathsf{y} : \mathbf{ref}(\mathbf{ref}(\alpha_2) \times \_) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = *\mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\alpha) \\ A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha_i) \\ \forall i \in [1 \ldots n] : \alpha_i \trianglelefteq \alpha \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

$$\frac{A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\_) \times \_)}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{allocate}(\mathsf{y}))}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\mathbf{ref}(\alpha_1) \times \_) \\ A \vdash \mathsf{y} : \mathbf{ref}(\alpha_2) \\ \alpha_2 \trianglelefteq \alpha_1 \end{array}}{A \vdash \mathit{welltyped}(*\mathsf{x} = \mathsf{y})}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{f}_i : \mathbf{ref}(\alpha_i) \\ A \vdash \mathsf{r}_j : \mathbf{ref}(\alpha_{n+j}) \\ \forall s \in S^* : A \vdash \mathit{welltyped}(s) \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} = \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^*)}$$

$$\frac{\begin{array}{c} A \vdash \mathsf{x}_j : \mathbf{ref}(\alpha'_{n+j}) \\ A \vdash \mathsf{p} : \mathbf{ref}(\_ \times \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})) \\ A \vdash \mathsf{y}_i : \mathbf{ref}(\alpha'_i) \\ \forall i \in [1 \ldots n] : \alpha'_i \trianglelefteq \alpha_i \\ \forall j \in [1 \ldots m] : \alpha_{n+j} \trianglelefteq \alpha'_{n+j} \end{array}}{A \vdash \mathit{welltyped}(\mathsf{x}_1 \ldots \mathsf{x}_m = \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

Figure 3: Type rules for the relevant statement types of the source language. All variables are assumed to have been associated with a type in the type environment A. (Distinct variables are assumed to have distinct names, so the type environment can describe all variables in all scopes simultaneously.) "_" is a wild-card value in the rules, imposing no restrictions on the type component it represents.

```
a = &x
b = &y
if p then
  y = &z;
else
  y = &x
fi
c = &y
```

a: $\tau_1 = \mathbf{ref}(\tau_4 \times \bot)$
b: $\tau_2 = \mathbf{ref}(\tau_5 \times \bot)$
c: $\tau_3 = \mathbf{ref}(\tau_5 \times \bot)$
x: $\tau_4 = \mathbf{ref}(\bot \times \bot)$
y: $\tau_5 = \mathbf{ref}(\tau_4 \times \bot)$
z: τ_4
p: $\tau_6 = \mathbf{ref}(\bot \times \bot)$

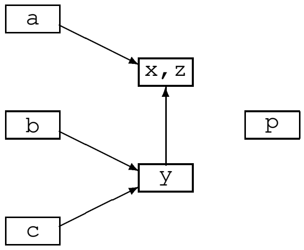

Figure 4: Example program, a typing of same that obeys the typing rules, and graphical representation of the corresponding storage shape graph. Note that variables x and z are described by the same type. Even though types τ_1 and τ_5 are structurally equivalent (as are τ_2 and τ_3, and τ_4 and τ_6), they are not considered the same types.

If type variables are only joined when necessary to ensure well-typedness, the final solution will be the minimal solution we are seeking.

5.2 Processing constraints

If the typing rules for a statement impose the constraint that two types are identical, then the corresponding type variables must be joined to obey the constraint.

An inequality constraint ($\trianglelefteq$) between two types is slightly more difficult as it may not always be possible to determine, at the time of processing a statement, whether the two types should be joined. If the left hand side type variable is associated with a type other than $\bot$, then the two type variables must be joined to meet the constraint. Assume that the left hand side type variable is associated with the type $\bot$ at the time a statement is processed. At this point, there is no need to join the two type variables. The typing rule for another statement may subsequently force a change of the type associated with the type variable implying that the type variable should be joined with the type variable on the right hand side of the current constraint.

To deal with this, we associate each type variable with type $\bot$ with a set of other type variables to join with, should the type ever become anything other than $\bot$. If an inequality relation must hold between two type variables, then we perform a *conditional join* of the two. If the left hand side type variable has type $\bot$, then we add the right hand side type variable to the set associated with the left hand side type variable. If the left hand side type variable has a type other than $\bot$, then a real join of the two type variables is performed. Whenever the type associated with a type variable changes from $\bot$, either because of a typing rule or because of unification, the type variable must be joined with the type variables in the associated set.

The precise rules for processing each statement of the program are given in Figure 5. The details of the join and unification operations are given in Figure 6.

5.3 Complexity

We argue that the algorithm has a linear space and almost linear time complexity in the size of the input program.

The space cost of the algorithm is determined by the total number of ECRs created and the number of join operations performed. The initial number of ECRs is proportional to the number of variables in the program. The number of ECRs created during the processing of a single statement is either bounded by a small constant or, in the case of a procedure call, at worst proportional to the number

x = y:
let $\mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_1, \lambda_2)$

x = &y:
let $\mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\tau_2 = \mathbf{ecr}(\mathsf{y})$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$

x = $*$y:
let $\mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}))$ in
if $\mathbf{type}(\tau_2) = \bot$ then
$\mathbf{settype}(\tau_2, \mathbf{ref}(\tau_1 \times \lambda_1))$
else
let $\mathbf{ref}(\tau_3 \times \lambda_3) = \mathbf{type}(\tau_2)$ in
if $\tau_1 \neq \tau_3$ then $\mathbf{cjoin}(\tau_1, \tau_3)$
if $\lambda_1 \neq \lambda_3$ then $\mathbf{cjoin}(\lambda_1, \lambda_3)$

x = op(y$_1$ ... y$_n$):
for $i \in [1 \ldots n]$ do
let $\mathbf{ref}(\tau_1 \times \lambda_1) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_1, \lambda_2)$

x = allocate(y):
let $\mathbf{ref}(\tau \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$ in
if $\mathbf{type}(\tau) = \bot$ then
let $[e_1, e_2] = \mathbf{MakeECR}(2)$ in
$\mathbf{settype}(\tau, \mathbf{ref}(e_1 \times e_2))$

$*$x = y:
let $\mathbf{ref}(\tau_1 \times \_) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}))$
if $\mathbf{type}(\tau_1) = \bot$ then
$\mathbf{settype}(\tau_1, \mathbf{ref}(\tau_2 \times \lambda_2))$
else
let $\mathbf{ref}(\tau_3 \times \lambda_3) = \mathbf{type}(\tau_1)$ in
if $\tau_2 \neq \tau_3$ then $\mathbf{cjoin}(\tau_3, \tau_2)$
if $\lambda_2 \neq \lambda_3$ then $\mathbf{cjoin}(\lambda_3, \lambda_2)$

x = fun(f$_1$... f$_n$)$\rightarrow$(r$_1$... r$_m$) S^*:
let $\mathbf{ref}(\_ \times \lambda) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}))$
if $\mathbf{type}(\lambda) = \bot$ then
$\mathbf{settype}(\lambda, \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))$
where
$\mathbf{ref}(\alpha_i) = \mathbf{type}(\mathbf{ecr}(\mathsf{f}_i))$, for $i \leq n$
$\mathbf{ref}(\alpha_i) = \mathbf{type}(\mathbf{ecr}(\mathsf{r}_{i-n}))$, for $i > n$
else
let $\mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathbf{type}(\lambda)$ in
for $i \in [1 \ldots n]$ do
let $\tau_1 \times \lambda_1 = \alpha_i$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{f}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_2, \tau_1)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_2, \lambda_1)$
for $i \in [1 \ldots m]$ do
let $\tau_1 \times \lambda_1 = \alpha_{n+i}$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{r}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{join}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{join}(\lambda_1, \lambda_2)$

x$_1$... x$_m$ = p(y$_1$... y$_n$):
let $\mathbf{ref}(\_ \times \lambda) = \mathbf{type}(\mathbf{ecr}(\mathsf{p}))$ in
if $\mathbf{type}(\lambda) = \bot$ then
$\mathbf{settype}(\lambda, \mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}))$
where
$\alpha_i = \tau_i \times \lambda_i$
$[\tau_i, \lambda_i] = \mathbf{MakeECR}(2)$
let $\mathbf{lam}(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m}) = \mathbf{type}(\lambda)$ in
for $i \in [1 \ldots n]$ do
let $\tau_1 \times \lambda_1 = \alpha_i$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{y}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_1, \tau_2)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_1, \lambda_2)$
for $i \in [1 \ldots m]$ do
let $\tau_1 \times \lambda_1 = \alpha_{n+i}$
$\mathbf{ref}(\tau_2 \times \lambda_2) = \mathbf{type}(\mathbf{ecr}(\mathsf{x}_i))$ in
if $\tau_1 \neq \tau_2$ then $\mathbf{cjoin}(\tau_2, \tau_1)$
if $\lambda_1 \neq \lambda_2$ then $\mathbf{cjoin}(\lambda_2, \lambda_1)$

Figure 5: Inference rules corresponding to the typing rules given in Figure 3. $\mathbf{ecr}(\mathsf{x})$ is the ECR representing the type of variable x, and $\mathbf{type}(E)$ is the type associated with the ECR E. $\mathbf{cjoin}(x, y)$ performs the conditional join of ECRs x and y, and $\mathbf{settype}(E, X)$ associates ECR E with type X and forces the conditional joins with E. $\mathbf{MakeECR}(x)$ constructs a list of x new ECRs, each associated with the bottom type, $\bot$.

settype(e, t):
  type$(e) \leftarrow t$
  for $x \in$ **pending**(e) do **join**(e, x)

cjoin(e_1, e_2):
  if **type**$(e_2) = \perp$ then
    pending$(e_2) \leftarrow \{e_1\} \cup$ **pending**(e_2)
  else
    join(e_1, e_2)

unify(**ref**$(\tau_1 \times \lambda_1)$, **ref**$(\tau_2 \times \lambda_2)$):
  if $\tau_1 \neq \tau_2$ then **join**(τ_1, τ_2)
  if $\lambda_1 \neq \lambda_2$ then **join**(λ_1, λ_2)

unify(**lam**$(\alpha_1 \ldots \alpha_n)(\alpha_{n+1} \ldots \alpha_{n+m})$,
    lam$(\alpha'_1 \ldots \alpha'_n)(\alpha'_{n+1} \ldots \alpha'_{n+m})$)
  for $i \in [1 \ldots (n+m)]$ do
  let $\tau_1 \times \lambda_1 = \alpha_i$
    $\tau_2 \times \lambda_2 = \alpha'_i$ in
   if $\tau_1 \neq \tau_2$ then **join**(τ_1, τ_2)
   if $\lambda_1 \neq \lambda_2$ then **join**(λ_1, λ_2)

join(e_1, e_2):
  let $t_1 =$ **type**(e_1)
    $t_2 =$ **type**(e_2)
    $e =$ ecr-union(e_1, e_2) in
  if $t_1 = \perp$ then
    type$(e) \leftarrow t_2$
    if $t_2 = \perp$ then
      pending$(e) \leftarrow$**pending**$(e_1) \cup$
          pending(e_2)
    else
      for $x \in$ **pending**(e_1) do **join**(e, x)
  else
    type$(e) \leftarrow t_1$
    if $t_2 = \perp$ then
      for $x \in$ **pending**(e_2) do **join**(e, x)
    else
      unify(t_1, t_2)

Figure 6: Rules for unification of two types represented by ECRs. We assume that ecr-union performs a (fast union/find) join operation on its ECR arguments and returns the value of a subsequent find operation on one of them.

of variables occuring in the statement. The number of ECRs is consequently proportional to the size of the input program. The number of join operations is bounded by the total number of ECRs. The space cost of a join operation amounts to the (constant) cost of the ecr-union operation. The cost of unifying/joining component type ECRs can be attributed to those joins. The cost of performing a conditional join or a join of two type variables with type $\perp$ is constant if we use a binary tree structure to represent the "pending" sets.

The time cost of the algorithm is determined by the cost of traversing the statements of the program, the cost of creating ECRs and types, the cost of performing join operations, and the cost of (fast union/find) "find" operations on ECRs. The cost of traversal and creation of ECRs and types is clearly proportional to the size of the input program. The cost of performing join operations is a constant plus the cost of ECR "find" operations. The average cost of N ECR "find" operations are $O(N\alpha(N, N))$, where α is a (very slowly increasing) inverse Ackermann's function [Tar83]. The time cost complexity of the algorithm is consequently $O(N\alpha(N, N))$, where N is the size of the input program (almost linear in the size of the input program).

6 Experience

We have implemented a slightly improved version of the above algorithm in our prototype programming system based on the Value Dependence Graph [WCES94] and implemented in the programming language Scheme [CR91]. The implementation uses a weaker typing rule than presented above for primitive operations returning boolean values and uses predetermined transfer functions for direct calls of library functions (the algorithm is thus context-sensitive/polymorhpic for calls to library functions). The analysis algorithm is routinely applied to the C programs processed by the system.

Two implementations of an earlier type inference based points-to analysis algorithm [Ste95a] have been performed at University of California, San Diego; one in C [Mor95] and one in Scheme [Gri95]. Both implementations have been augmented to model slots of structured objects independently. Our earlier algorithm was based on the same non-standard type system as used in the present algorithm but used stricter typing rules, implying that the results are more conservative than they need be.

Our implementation demonstrates that running time of the algorithm is roughly linear in the size of the input program on our test-suite of around 50 programs. Using our own implementation, we have performed points-to analysis of programs up to 75,000 lines of code[2] (an internal Microsoft tool). The running time for the algorithm on the 75,000 line C program is approximately 27 seconds (15 seconds process time) on an SGI Indigo2 workstation, or roughly 4 times the cost of traversing all nodes in the program representation. For a 25,000 line C program (LambdaMOO available from Xerox PARC) the running time is approximately 8 seconds (5.5 seconds process time). The analysis is performed as a separate stage after the program representation has been built.

Morgenthaler's implementation of our previous algorithm performs the processing of statements during parsing of the program. He found the parse time to increase by approximately 50% by adding points-to analysis to the parser. Counting only the extra time for performing the analysis, emacs (127,000 non-empty lines of code) could be analyzed in approximately 50 seconds, and FElt (273,000 non-empty lines of code) could be analyzed in approximately 82 seconds on a SparcStation 10 [Mor95]. The present algorithm can also easily be implemented to process the statements during parsing. The running times of the previous and the present algorithm are roughly the same (only minor fluctuations).

Table 1, Table 2, and Table 3 illustrate the distribution of program variables per type variable for a number of benchmark programs. The programs are from Bill Landi's and Todd Austin's benchmark suites for their analyses [LRZ93, ABS94] as well as the SPEC'92 benchmark suite. LambdaMOO is a large C program available from Xerox PARC (we used version 1.7.1).

Table 1 gives the raw distribution for the total analysis solution when performed on an (almost) unoptimized version of the program representation. Most of the type variables describe the location

[2] At the time of writing, this is the largest program represented using the VDG program representation.

of variables whose addresses are never taken. The type variables describing zero program variables represent non-$\perp$ types describing user functions and locations allocated by the runtime system (*e.g.*, the locations for the `argv` and `argc` arguments to `main`).

Table 2 gives the distribution for those type variables that occur as location components of other types in the solution of the analysis performed on an (almost) unoptimized version of the program representation. These type variables represent program variables that are pointed to by other variables. They do not necessarily represent all the program variables that are pointed to in the program, as minor optimizations are performed on the VDG program representation as it is being built; some of these optimizations eliminate storing values in variables if this is trivial to avoid, as described in [WCES94]. The number of type variables describing more than one program location is reduced relative to Table 1. The reduction is mostly caused by eliminating type variables for values passed to functions but never pointed to by a pointer. The values would not have been grouped in the first place, if a polymorphic analysis had been used.

Table 3 gives the distribution for the location component type variables in the solution of the analysis performed on an optimized version of the program representation. The optimizations performed on the program representation include a local transformation eliminating all local variables whose address is never taken. These type variables describe the program variables that are the hardest to get good analysis results for. The program variables are all pointed to by other program variables which cannot be eliminated by local transformations. Many of the program variables described by a type variable representing no other program variables are candidates for global optimizations such as being represented by a register rather than a memory location.

The distributions shown in the tables demonstrate that there are a considerable number of type variables describing only a single program variable, even for those type variables describing pointed to program variables. Most other type variables describe a small number of program variables. There are a couple of major exceptions; type variables describing several hundred program locations. However, for most of the programs, the locations described by these exceptional type variables are all global constant strings. For example, for the LambdaMOO program, the program locations described by the "largest" single type variable are all strings passed as argument to user defined logging and tracing procedures. Any context-insensitive analysis is bound to show a similar number of possible pointer values for the formal parameters of these logging and tracing procedures.

Our subjective evaluation of the quality of the analysis results is that they are pretty good, given that the contents of all the slots of structured variables are represented by a single value type. However, many programs use data structures such as trees and lists as central data structures. For these programs the inability to distinguish between structure elements is a serious loss.

7 Related work

Henglein used type inference to perform a binding time analysis in almost linear time [Hen91]. His types represent binding time values. He presents a set of typing rules, extract constraints from the typing rules, and finally solve the constraints by using fast union/find data structures. Our points-to analysis algorithm was inspired by Henglein's type inference algorithm.

The points-to analysis that closest resembles our analysis is Weihl's [Wei80]. His analysis is also flow-insensitive, interprocedural, and deals with pointers to functions. His algorithm does not assume that alias relations are reflexive and transitive, and will

# of vars.	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6 ...	30	31	32	33 ...	44	45 ...	52 ...	74 ...	78 ...	83 ...	113 ...	120 ...	285 ...	613 ...	624
landi:allroots	18	67																																								
landi:assembler	157	446	3		3	1					1																				1								1			
landi:loader	90	211	1	1			1	1						1																												
landi:compiler	116	166																					1										1									
landi:simulator	232	464	3	1	2	1			2				2		1								1	2										2			1					
landi:lex315	52	91																						1																		
landi:football	214	570	15	14	1					1																1																
austin:anagram	54	65					1		1																																	
austin:backprop	43	69																																								
austin:bc	297	551	2													2			2																	1						
austin:ft	61	150																																								
austin:ks	68	158	2			1																																				
austin:yacr2	260	474	3	1																																						
spec:compress	88	113	1																																							
spec:eqntott	228	437	2				1	1	1																																	
spec:espresso	1155	2556	14	3	2	1	2	1								1	1						1							1		1			1							
spec:li	449	877	1	2	1		2																					1														1
spec:sc	557	1000	26	4	3			1	2						1										1													1		1		
spec:alvinn	43	73																																								
spec:ear	192	532	3		2			1																			1															
LambdaMOO	1369	2580	9	2	3		1	2																	1								1								1	

Table 1: Number of type variables describing a given number of program variables for the unoptimized program representation. For example, for landi:allroots, there are 67 type variables each describing the location of a single program variable.

# of vars.	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
landi:allroots	0	1																		
landi:assembler	10	8	3		3	1					1									
landi:loader	9	6	1	1			1	1					1							
landi:compiler	0	1																		
landi:simulator	2	4	3	1	2	1			2				2		1					
landi:lex315	2																			
landi:football	5		1	1					1											
austin:anagram	3	3					1		1											
austin:backprop	1	9																		
austin:bc	5	5	2													2			2	
austin:ft	4	2																		
austin:ks	4	1	2			1														
austin:yacr2	29	1	3	1																
spec:compress	2	4	1																	
spec:eqntott	5	8	2				1	1	1											
spec:espresso	14	19	12	2	2	1	2	1								1	1			
spec:li	2	4	1	2	1		2													
spec:sc	7	10	5	4	2			1	1						1					
spec:alvinn	1	9																		
spec:ear	15	23	3		2			1												
LambdaMOO	8	15	8	2	3		1	2		1										

# of vars.	0	1	2	3	4	5	6	...	30	31	32	33	...	44	45	...	52	...	74	...	78	...	83	...	113	...	120	...	285	...	613	...	624
landi:allroots																																	
landi:assembler												1															1						
landi:loader																																	
landi:compiler			1												1																		
landi:simulator			1	2													2						1										
landi:lex315				1																													
landi:football						1																											
austin:anagram																																	
austin:backprop																																	
austin:bc																					1												
austin:ft																																	
austin:ks																																	
austin:yacr2																																	
spec:compress																																	
spec:eqntott																																	
spec:espresso			1								1			1					1														
spec:li									1																								1
spec:sc					1																				1				1				
spec:alvinn																																	
spec:ear							1																										
LambdaMOO					1										1																1		

Table 2: Number of type variables describing a given number of pointed to program variables for the unoptimized program representation.

# of vars.	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
landi:allroots																					
landi:assembler	2																				
landi:loader	1													1							
landi:compiler																					
landi:simulator	1	1																			
landi:lex315	1																				
landi:football	3																				
austin:anagram	2	1																			
austin:backprop																					
austin:bc	2																				
austin:ft	1	1																			
austin:ks	1					1															
austin:yacr2	26																				
spec:compress			1																		
spec:eqntott	1	2					1	1													
spec:espresso	2	1					1									1	1				
spec:li																					
spec:sc	1	2																			
spec:alvinn																					
spec:ear	8	12						1													
LambdaMOO	5	1	1		2																

# of vars.	1	2	3	4	5	6	...	9	30	31	32	33	...	44	45	...	52	...	74	...	78	...	83	...	113	...	120	...	285	...	613	...	624
landi:allroots																																	
landi:assembler												1															1						
landi:loader																																	
landi:compiler		1																															
landi:simulator		1																					1										
landi:lex315			1																														
landi:football																																	
austin:anagram																																	
austin:backprop																																	
austin:bc																					1												
austin:ft																																	
austin:ks																																	
austin:yacr2																																	
spec:compress																																	
spec:eqntott																																	
spec:espresso		1						1											1														
spec:li																																	1
spec:sc																									1								
spec:alvinn																																	
spec:ear						1																											
LambdaMOO				1																											1		

Table 3: Number of type variables describing a given number of pointed to program variables for the optimized program representation.

therefore in some cases produce better results than our algorithm. On the other hand, his algorithm does not distinguish between one or several levels of pointer indirection. Additionally, his algorithm works best if a call graph is available, and it does not deal elegantly with recursive functions. His algorithm has a time cost complexity that is cubic in the size of the input program whereas our algorithm has an almost linear time cost complexity.

More precise points-to analysis exist, *e.g.*, [CWZ90, EGH94, WL95, Ruf95]. These analyses are all flow-sensitive interprocedural data flow analyses. Both Chase's algorithm [CWZ90] and Ruf's algorithm [Ruf95] are context-insensitive and have polynomial time complexity. The two other algorithms are context-sensitive, meaning that the algorithm distinguishes between effects of different calls of the same function instead of computing just one effect that is valid for all calls of the function[3]. The algorithm by Emami, et. al., [EGH94] has a exponential time complexity, as it performs a virtual unfolding of all non-recursive calls. The algorithm by Wilson and Lam [WL95] also has exponential time complexity but is likely to exhibit polynomial time complexity in practice as it uses partial transfer functions to summarize the behavior of already analyzed functions and procedures.

Whereas a points-to analysis builds and maintains a model of the store during analysis, an alias analysis builds and maintains a list of access path expressions that may evaluate to the same location (in other words: they are aliased). The most relevant alias analysis algorithms are [LR92, LRZ93]. The length of access-paths are k-limited, using a relatively simple truncation mechanism to eliminate extra path elements.

Deutsch presents an alias analysis for an imperative subset of ML [Deu92]. Access paths are defined in terms of monomial relations (a kind of multi-variable polynomial expression with structure accessors as the variables). The analysis is therefore only relevant for strongly typed languages such as ML and strongly typable programs written in weakly typed languages such as C (as shown in [Deu94]). Access paths are combined by unification.

A higher order (context-sensitive) points-to analysis by type inference has been developed by Tofte and Talpin for the purposes of creating an ML interpreter without a garbage collector [TT94]. The analysis is based on polymorphic type inference over a non-standard set of types. They assume a runtime model that makes allocation regions explicit, where allocation regions resemble the storage shape graph nodes of our algorithm. Their algorithm does not deal with objects that may be updated after being assigned an initial value (as is normal for imperative programs). Whether their work can be generalized to work for general imperative programs is an open question.

Andersen defines context-sensitive and context-insensitive analyses that are flow-insensitive[4] points-to analysis in terms of constraints and constraint solving [And94]. The context-sensitive algorithm distinguishes between immediate calling contexts in a 1-limited version of the static program call graph, effectively taking two layers of context into consideration. The values being constrained are sets of abstract locations. Andersen's algorithm allows an abstract location to be a member of non-identical sets. Our algorithm only allows an abstract location to be described by one type representing a set of abstract locations. The size of the solution of his context-insensitive algorithm is $O(A^2)$, and the size of the solution of his context-sensitive algorithm is $O(A^4)$, where A is the number of abstract locations, which in turn is $O(\exp N)$, where N is the size of the program[5]. In contrast, the size of the solution of our algorithm is $O(N)$.

Choi *et al.* present both flow-sensitive and flow-insensitive analyses [CBC93]. The flow-insensitive analysis algorithm is described in more detail in [BCCH95]. Their algorithm computes alias information rather than points-to information but uses a representation that shares many properties with the storage shape graph. The representation allows abstract locations to be members of non-identical sets. Their algorithm is based on iterated processing of the program statements and it thus likely to be slower than a similar constraint based algorithm (such as Andersen's context-sensitive algorithm but only considering one level of calling context).

The algorithm presented in this paper is an extension of another almost linear points-to analysis algorithm [Ste95a]. Bill Landi has independently arrived at the same earlier algorithm [Lan95]. Barbara Ryder and Sean Zhang are also working on a version of the earlier algorithm with the extension that elements of composite objects are represented by separate type components [Zha95].

8 Conclusion and Future Work

We have presented a flow-insensitive, interprocedural, context-insensitive points-to analysis based on type inference methods with an almost linear time complexity. The algorithm has been implemented and shown to be very efficient in practice, and we have found the results to be much better than the results of intraprocedural analyses.

A problem with the analysis as presented is that it does not disambiguate information for different elements of structured objects. The type system can be extended to do so, but the resulting analysis algorithm will not have an almost linear time complexity. The algorithm will still be asymptotically faster than other existing algorithms that does distinguish between different elements of structured objects.

Our main interest has been developing efficient interprocedural points-to analysis algorithms for large programs. We would like to develop efficient algorithms yielding greater precision than the algorithm presented in this paper. Given the algorithm presented in this paper, there are two possible directions to investigate.

One way to obtain improved results is to develop an efficient flow-sensitive algorithm. The results from the algorithm presented in the present paper can be used to prime a data flow analysis algorithm or otherwise reduce the amount of work to be done by the algorithm. One possible method is splitting of functional stores as suggested in [Ste95b].

Another way to obtain improved results is to develop an efficient flow-insensitive, context-sensitive algorithm. This can be done using types to represent sets of locations, as in the almost linear time algorithm, but using polymorphic instead of monomorphic type inference methods.

We are currently pursuing both directions of research.

Acknowledgements

Roger Crew, Michael Ernst, Erik Ruf, Ellen Spertus, and Daniel Weise of the Analysts group at Microsoft Research co-developed the VDG-based programming environment without which this work would not have come into existence. Members of the Analysts group also commented on and proofread versions of this paper. The author also enjoyed interesting discussions with David Morgenthaler, William Griswold, Barbara Ryder, Sean Zhang, and Bill Landi on

[3] Our analysis is context-insensitive because the type system is monomorphic. If we had used a polymorphic type system and polymorphic type inference, the algorithm would have been context-sensitive.

[4] Andersen uses the term "intra-procedural" to mean "context-insensitive" and the term "inter-procedural" to mean "context-sensitive".

[5] To be fair, A is probably proportional to N in practice.

various points-to analysis algorithms with almost linear time complexity. We would like to thank Bill Landi and Todd Austin for sharing their benchmark suites with us.

References

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN'94: Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Department of Computer Science, University of Copenhagen, May 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCCH95] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings from the* 7th International Workshop on Languages and Compilers for Parallel Computing, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer-Verlag, 1995. Extended version published as Research Report RC 19546, IBM T.J. Watson Research Center, September 1994.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[CR91] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme, November 1991.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *International Conference on Computer Languages*, pages 2–13. IEEE, April 1992.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN'94: Conference on Programming Language Design and Implementation*, pages 230–241, June 20-24 1994.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN'94: Conference on Programming Language Design and Implementation*, pages 242–256, June 20-24 1994.

[Gri95] William G. Griswold. Use of algorithm from [Ste95a] in a program restructuring tool. Personal communication at PLDI'95, June 1995.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*, pages 448–472, 1991.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second edition*. Prentice Hall, 1988.

[Lan95] William Landi. Almost linear time points-to analyses. Personal communication at POPL'95, January 1995.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[LRZ93] William A. Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[Mor95] David Morgenthaler. Poster presentation at PLDI'95, June 1995.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[Ste95a] Bjarne Steensgaard. Points-to analysis in almost linear time. Technical Report MSR-TR-95-08, Microsoft Research, March 1995.

[Ste95b] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 62–70, San Francisco, CA, January 22 1995. Proceedings appear as March 1995 issue of SIGPLAN Notices.

[Tar83] Robert E. Tarjan. Data structures and network flow algorithms. In *Regional Conference Series in Applied Mathematics*, volume CMBS 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings 21st SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings 21st SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, January 1994.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[Zha95] Sean Zhang. Poster presentation at PLDI'95, June 1995.