Points-to Analysis by Type Inference of Programs with Structures and Unions

Bjarne Steensgaard

Microsoft Research*

Abstract

We present an interprocedural flow-insensitive points-to analysis algorithm based on monomorphic type inference. The source language model the important features of C including pointers, pointer arithmetic, pointers to functions, structured objects, and unions. The algorithm is based on a non-standard type system where types represent nodes and edges in a storage shape graph.

This work is an extension of previous work on performing points-to analysis of C programs in almost linear time. This work makes three new contributions. The first is an extension of a type system for describing storage shape graphs to include objects with internal structure. The second is a constraint system that can deal with arbitrary use of pointers and which incorporates a two-tier domain of pointer offsets to improve the results of the analysis. The third is an efficient inference algorithm for the constraint system, leading to an algorithm that has close to linear time and space performance in practice.

Keywords: interprocedural program analysis, points-to analysis, C programs, non-standard types, constraint solving.

1 Introduction

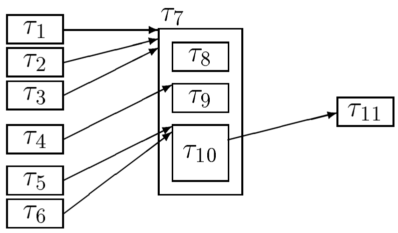
Modern optimizing compilers and program understanding and browsing tools for pointer languages like C [Ame89, KR88] are dependent on semantic information obtained by either an alias analysis or a points-to analysis. Alias analyses compute pairs of expressions (or access paths) that may be aliased (*e.g.*, [LR92, LRZ93]). Points-to analyses compute a store model using abstract locations (*e.g.*, [CWZ90, EGH94, WL95, Ruf95]). Points-to analysis results serve no purpose in themselves, but they are a prerequisite for most other analyses and transformations for imperative programs (*e.g.*, computing use-def relations, permitted code motion, and detection of use of uninitialized variables).

Most current compilers and programming tools use only intraprocedural points-to analyses, as the polynomial time and space complexity of the common data-flow based points-to analyses prevents the use of interprocedural analyses for large programs. Interprocedural analysis is becoming increasingly important, as it is a prerequisite for whole-program optimization and various program understanding tools.

*Author's address: Microsoft Corporation, One Microsoft Way, Redmond, WA, USA. E-mail: rusa@research.microsoft.com

```
int i1, *i2, **i3, **i4;
float f1, **f2;
struct {int a, *b, *c;} s1, *s2;
struct {int d, *e; float f, *g;} s3, *s4;

s2 = &s1;
s4 = &s3;
f2 = &s4->g;
*f2 = &f1;
i3 = &s2->b;
i4 = &s2->c;
*i4 = &i1;
i2 = (int*) s2;
i2 = (int*) s4;
```



i1:	τ_{11}	i2:	τ_1
i3:	τ_4	i4:	τ_6
f1:	τ_{11}	f2:	τ_5
s1:	τ_7	s2:	τ_2
s3:	τ_7	s4:	τ_3

Figure 1: A small C program fragment, the storage shape graph that our algorithm builds for it out of types, and a typing of the program variables. Type τ_7 represents both structured variables in the program. The third type component, τ_{10}, of τ_7 represents structure elements `s1.c`, `s2.f` and `s2.g`.

We extend our previous work on flow-insensitive interprocedural points-to analysis of C programs by type inference methods [Ste96, Ste95] by enabling the algorithm to distinguish between components of structured objects, thereby increasing the precision of the analysis in the presence of structures and unions in the program to be analyzed. Other members of our research group have found this extension crucial to the value (accuracy) of some subsequent analyses (*e.g.*, detection of use of uninitialized variables). The extended algorithm does not have the almost linear time complexity of the original algorithms, but it is exhibiting close to linear time complexity in practice (Sect. 5.3 discusses complexity).

The algorithm is based on type inference over a domain of types that can model a storage shape graph [CWZ90]. The inferred types describe the *use* of memory locations. The *declarations* of locations are irrelevant. The algorithm computes a valid typing even when memory locations are used in inconsistent ways, in contrast to ML type inference which will fail to compute a typing in that case. An example illustrating types modeling a storage shape graph for a program is shown in Fig. 1.

The computed solution is a storage shape graph that is a conservative description of the dynamic storage shape graphs for all program points simultaneously. If programmers use locations in a consistent manner throughout their programs the loss in precision by not computing separate solutions for each program point is typically small. Computing only one storage shape graph permits the algorithm to be fast for even very large programs.

We proceed by stating the source language (Sect. 2), which captures the essential parts of the C programming language, the non-standard set of types we use to model the storage use (Sect. 3), and a set of typing rules for programs (Sect. 4). Finding a typing of the program that obeys the constraints imposed by the typing rules amounts to performing a points-to analysis. We then show how to efficiently deduce the minimal typing that obeys the constraints (Sect. 5) and report on practical experience with the algorithm (Sect. 6). Finally we describe related work (Sect. 7) and present our conclusions and point out directions for future work (Sect. 8).

$$
\begin{array}{rcl}
S & ::= & \mathsf{x} =_s \mathsf{y} \\
 & | & \mathsf{x} =_s \&\mathsf{y} \\
 & | & \mathsf{x} =_s *\mathsf{y} \\
 & | & \mathsf{x} =_s \mathsf{allocate(y)} \\
 & | & *\mathsf{x} =_s \mathsf{y} \\
 & | & \mathsf{x} =_s \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n) \\
 & | & \mathsf{x} =_s \&\mathsf{y}\text{->}\mathsf{n} \\
 & | & \mathsf{x} =_s \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^* \\
 & | & \mathsf{x}_1 \ldots \mathsf{x}_m =_{s_1 \ldots s_m} \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)
\end{array}
$$

Figure 2: 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. n ranges over the (unbounded) set of structure element names. op ranges over the set of primitive operator names. S^* denotes a sequence of statements. The assignment operator, =, is annotated with a size, s, indicating the size of the representation of the value being assigned. The control structures of the language are irrelevant.

2 The Source Language

We describe the points-to analysis for a pointer language with structures and unions that captures the important properties of the C programming language [Ame89, KR88]. Since the analysis is flow insensitive, the control structures of the language are irrelevant. An important feature of the language is that any memory object may be accessed as a unit or as a structured object. Type casts and variable declarations as found in C are irrelevant; the source language permits inconsistent use of locations as well as the use of any memory object as a structured object without such constructs. Unions are implicit in the use of memory objects. Figure 2 shows the abstract syntax of the relevant parts of the language.

The syntax for pointer operations borrows from the C programming language. All variables are assumed to have unique names. The op(...) expression form is used to describe primitive computations like arithmetic operations. The allocate(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 (sometimes called *in parameters*), and the r_i variables are return values (sometimes called *out parameters*). Function calls have call-by-value semantics [ASU86]. Both formal and return parameter variables may appear in left- and right-hand-side position in statements in the function body.

We assume that programs are as well-behaved as (mostly) portable C programs. We allow assignment of a structured value to a location supposed to hold only pointer values, and vice versa, provided the representation of the assigned value fits within the size of the representation of the object being modified. The analysis algorithm may produce wrong (unsafe) results for programs that construct pointers from scratch (*e.g.*, by bitwise duplication of pointers by control flow rather than data flow) and non-portable programs (*e.g.*, programs that rely on how a specific compiler allocates variables relative to each other). All previously described analyses suffer from the same problem. However, the analysis algorithm as presented below will deal with, *e.g.*, exclusive-or operations on pointer values, where there is a flow of values.

[1]We allow functions with multiple return values; a feature not found in C.

3 Types

For the purpose of performing the points-to analysis, we define a non-standard set of types to describe the store. The types are unrelated to the types normally used in C (*e.g.*, `integer`, `float`, `pointer`, `struct`). The types are used to model how storage is used in a program at runtime (a storage model). Locations of program 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.

The types must be able to model both simple locations, which are only ever accessed as a whole (e.g., integer variables), structured locations, and locations that are accessed in inconsistent ways. We want to accommodate inconsistent accesses of locations with minimal information loss. We use four different kinds of types: **blank** describes locations with no access pattern, **simple** describes locations only accessed as a whole, **struct** describes locations only accessed as structured objects, and **object** describes locations accessed in ways not covered by the other three kinds of types.

Structured objects may be accessed in inconsistent ways. We want the **struct** types to be able to describe commonalities in the accesses anyway. We assume structures with a "common prefix" share layout of the common prefix elements. The **struct** types have component types describing distinguishable components of a location, where "distinguishable" means that any access of part of the memory object only accesses a single component. For the program fragment in Fig. 1, one distinguishable component describes the first element of the structured objects, another the second element, and a third the remaining components.

The size of an access is important. For example, if a pointer value may point to an integer component of a structured object and an access through the pointer is "larger" than the size of an integer, other components of the structure pointed into may be modified or retrieved. For example, if the program fragment shown in Fig. 1 were extended with a reference of "`*(long*)i3`" then the structures would only have two distinguishable components.

The type of a memory object must also describe the contents of the object. Only pointer values are relevant. We describe pointers to locations by the type representing the object(s) it pointed to or into and an offset, which may be either **zero** or **unknown**. If a pointer with an **unknown** offset is used in an indirect assignment (*e.g.*, $*\mathtt{x} =_s \mathtt{y}$) then we don't know what part of the referenced object is being modified, and the object must be described by an **object** type.

Functions, or rather function pointer values, are described by signature types describing the locations of the argument and result values.

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

$$
\begin{array}{lcll}
\tau & ::= & \bot \mid \mathbf{simple}(\alpha, \lambda, s, p) \mid \mathbf{struct}(m, s, p) \mid & \\
 & & \mathbf{object}(\alpha, \lambda, s, p) \mid \mathbf{blank}(s, p) & \text{(Objects)} \\
\alpha & ::= & (\tau \times o) & \text{(Pointers)} \\
o & ::= & \mathbf{zero} \mid \mathbf{unknown} & \text{(Offsets)} \\
\lambda & ::= & \bot \mid \mathbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m}) & \text{(Functions)} \\
s & ::= & \text{SIZE} \mid \top & \text{(Sizes)} \\
p & ::= & \mathcal{P}(\tau) \mid \top & \text{(Parents)} \\
m & ::= & (\text{element} \mapsto \tau) \text{ mapping} & \text{(Elements)}
\end{array}
$$

i1:	$\tau_{11} = \mathbf{simple}(\perp, \perp, \top, \emptyset)$
i2:	$\tau_1 = \mathbf{simple}(\tau_7, \perp, <\texttt{ptr}>, \emptyset)$
i3:	$\tau_4 = \mathbf{simple}(\tau_9, \perp, <\texttt{ptr}>, \emptyset)$
i4:	$\tau_6 = \mathbf{simple}(\tau_{10}, \perp, <\texttt{ptr}>, \emptyset)$
f1:	τ_{11}
f2:	$\tau_5 = \mathbf{simple}(\tau_{10}, \perp, <\texttt{ptr}>, \emptyset)$
s1:	$\tau_7 = \mathbf{struct}([<\texttt{int}> \mapsto \tau_8, <\texttt{int}, \texttt{int}> \mapsto \tau_9, {}^* \mapsto \tau_{10}], \top, \emptyset)$
s2:	$\tau_2 = \mathbf{simple}(\tau_9, \perp, <\texttt{ptr}>, \emptyset)$
s3:	τ_7
s4:	$\tau_3 = \mathbf{simple}(\tau_9, \perp, <\texttt{ptr}>, \emptyset)$
	$\tau_8 = \mathbf{simple}(\perp, \perp, <\texttt{int}>, \{\tau_7\})$
	$\tau_9 = \mathbf{simple}(\perp, \perp, <\texttt{int}>, \{\tau_7\})$
	$\tau_{10} = \mathbf{object}(\tau_{11}, \perp, \top, \{\tau_7\})$

Figure 3: Typing of the program fragment of Fig. 1 in terms of the types of our analysis algorithm.

The τ types describe objects or object components, the α types describe pointers to locations, the λ types describe pointers to functions. The m type components are mappings from structure element specifiers to component types. The element specifiers can be either symbolic or numeric.

The s type components describe object or object component sizes. The sizes can be either numeric or symbolic. The $\top$ size indicates the rest of a memory object and is used in types describing objects of different sizes. The p type components describe the set of **struct** types (parents) of which a given type is a component. The $\top$ value means "no parents" and is introduced to enable a requirement that a type has no parents while allowing use of least-upper-bound operators in the inference algorithm. We assume the programmer is denied knowledge of the activation record layout and therefore do not consider parents of λ type.

Types may be recursive (the type graph may be cyclic). The types may be written out using type identifiers (type variables). Two types are equal when they are either both $\perp$ or are described by the same type identifier. 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, o, and λ types.

Figure 3 shows the typing of the variables of the program fragment of Fig. 1.

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 and which also safely describes the use of the storage.

There are three kinds of use of storage in the source language. Use via pointer indirection uses the pointer location as a whole. Computing the address of a structure element is a use of a location as a structured object. These two uses force the location being addressed to be described by at least a **simple** or **struct** type respectively in the $\trianglelefteq_s$ partial order described below. The third kind of use is by assignment of entire objects. For example, if we assign a structured value to a location that is otherwise used only as a whole the contents of the location assigned to is used in inconsistent ways. The assigned-to location must therefore be described by an **object** type, while the assigned-from location may still be described by a **struct** type.

We use the partial order $a \trianglelefteq_s b$ to describe the relationship between the type, b, of the assigned-to location, and the type, a, of the assigned-from location. The partial order is parameterized by a size, s, as the size of the representation of the assigned value must be smaller than that of (the types of) the assigned-to and the assigned-from location to avoid problems with unmodeled capture of adjacent elements in a structured object. The size constraints are trivially fulfilled if the types describe entire objects or the entire rest of objects ($s = \top$).

The $\trianglelefteq_s$ partial order uses the following hierarchy among the kinds of types:

$$\begin{array}{ccc} & \textbf{object} & \\ \diagup & & \diagdown \\ \textbf{simple} & & \textbf{struct} \\ \diagdown & & \diagup \\ & \textbf{blank} & \end{array}$$

where a necessary (but not sufficient) requirement for $a \trianglelefteq_s b$ to hold is that a and b are either of the same kind or the kind of b appears above the kind of a in the hierarchy.

If the offset component of either a or b is **unknown** then we have to assume the worst about the usage of the described memory location and the memory object component should be of the **object** kind.

Since there is a flow of data from the assigned-from location to the assigned-to location, any pointer content of the assigned-from location should also be described by the content components of the assigned-to location. We describe the relationship between the assigned-from and assigned-to location contents by the $\sqsubseteq_s$ and $\sqsubseteq$ partial order between memory and function pointer component types respectively defined as follows:

$$\begin{array}{rcl} (\tau_1 \times o_1) \sqsubseteq_s (\tau_2 \times o_2) & \Leftrightarrow & (\tau_1 = \bot) \vee ((\tau_1 \sqsubseteq_s \tau_2) \wedge (o_1 \sqsubseteq o_2)) \\ \tau_1 \sqsubseteq_s \tau_2 & \Leftrightarrow & (\tau_1 = \tau_2) \wedge (s \sqsubseteq \mathrm{sizeof}(\tau_1)) \\ o_1 \sqsubseteq o_2 & \Leftrightarrow & (o_1 = \mathbf{zero}) \vee (o_1 = o_2) \\ s_1 \sqsubseteq s_2 & \Leftrightarrow & (s_1 = s_2) \vee (s_2 = \top) \\ \lambda_1 \sqsubseteq \lambda_2 & \Leftrightarrow & (\lambda_1 = \bot) \vee (\lambda_1 = \lambda_2), \end{array}$$

where "sizeof(τ)" denotes the size component of whatever kind of type τ is. For example, a necessary requirement for $\mathbf{simple}(\alpha_1, \lambda_1, s_1, p_1) \trianglelefteq_s \mathbf{simple}(\alpha_2, \lambda_2, s_2, p_2)$ to hold is that $\alpha_1 \sqsubseteq_s \alpha_2$ and $\lambda_1 \sqsubseteq \lambda_2$ both hold.

We could have used equality (=) instead of $\sqsubseteq$ ordering. The primary reason for not doing so is discussed in [Ste96]. Of particular importance to the type system used in the present paper is that use of $\sqsubseteq$ rather than = permits non-pointer content of components of **struct** mappings when a value in a **struct** location is assigned to an **object** location.

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \tau_1 \qquad A \vdash \mathsf{y} : \tau_2 \\ (\tau_2 \times \textbf{zero}) \trianglelefteq_s (\tau_1 \times \textbf{zero})\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{y})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \textbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) \\ A \vdash \mathsf{y} : \tau_2 \qquad s \sqsubseteq s_1 \\ (\tau_2 \times \textbf{zero}) \sqsubseteq_s \alpha_1\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{\&y})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{y} : \textbf{sim/obj}(\alpha_2, \lambda_2, s_2, p_2) \\ A \vdash \mathsf{x} : \tau_1 \qquad \alpha_2 \trianglelefteq_s (\tau_1 \times \textbf{zero})\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{*y})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \textbf{sim/obj}((\tau_1 \times o_1), \lambda_1, s_1, p_1) \\ \tau_1 \neq \bot \qquad s \sqsubseteq s_1\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{allocate(y)})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \textbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) \\ A \vdash \mathsf{y} : \tau_2 \qquad (\tau_2 \times \textbf{zero}) \trianglelefteq_s \alpha_1\end{array}}{A \vdash \mathit{welltyped}(\mathsf{*x} =_s \mathsf{y})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \tau \qquad A \vdash \mathsf{y}_i : \tau_i \\ \forall i \in [1 \ldots n] : (\tau_i \times \textbf{zero}) \trianglelefteq_s (\tau \times \textbf{zero}) \\ \tau = \textbf{sim/obj}((\tau' \times \textbf{unknown}), \lambda, s', p)\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \textbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) \\ A \vdash \mathsf{y} : \textbf{sim/obj}((\tau_2 \times o_2), \lambda_2, s_2, p_2) \\ \tau_2 = \textbf{object}(\alpha_3, \lambda_3, \top, \top) \\ s \sqsubseteq s_1 \qquad (\tau_2 \times o_2) \sqsubseteq_s \alpha_1\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{\&y\text{->}n})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \textbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) \\ A \vdash \mathsf{y} : \textbf{simple}((\tau_2 \times \textbf{zero}), \lambda_2, s_2, p_2) \\ \tau_2 = \textbf{struct}(m_3, s_3, p_3) \\ \text{compatible}(\mathsf{n}, m_3) \\ s \sqsubseteq s_1 \qquad (m_3(\mathsf{n}) \times \textbf{zero}) \sqsubseteq_s \alpha_1\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{\&y\text{->}n})}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \textbf{sim/obj}(\alpha_0, \lambda_0, s_0, p_0) \\ \lambda_0 = \textbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m}) \\ A \vdash \mathsf{f}_i : \tau'_i \qquad A \vdash \mathsf{r}_j : \tau'_{n+j} \\ s_i = \text{sizeof}(\mathsf{f}_i) \qquad s_{n+j} = \text{sizeof}(\mathsf{r}_j) \qquad s \sqsubseteq s_0 \\ \forall i \in [1 \ldots n] : (\tau_i \times \textbf{zero}) \trianglelefteq_{s_i} (\tau'_i \times \textbf{zero}) \\ \forall j \in [1 \ldots m] : (\tau'_{n+j} \times \textbf{zero}) \trianglelefteq_{s_{n+j}} (\tau_{n+j} \times \textbf{zero}) \\ \forall x \in S^* : A \vdash \mathit{welltyped}(x)\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \!\rightarrow\! (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^*)}$$

$$\dfrac{\begin{array}{c}A \vdash \mathsf{p} : \textbf{sim/obj}(\alpha_0, \lambda_0, s_0, p_0) \\ \lambda_0 = \textbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m}) \\ A \vdash \mathsf{x}_j : \tau'_{n+j} \qquad A \vdash \mathsf{y}_i : \tau'_i \qquad s_i = \text{sizeof}(\mathsf{y}_i) \\ \forall i \in [1 \ldots n] : (\tau'_i \times \textbf{zero}) \trianglelefteq_{s_i} (\tau_i \times \textbf{zero}) \\ \forall j \in [1 \ldots m] : (\tau_{n+j} \times \textbf{zero}) \trianglelefteq_{s_{j+n}} (\tau'_{n+j} \times \textbf{zero})\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x}_1 \ldots \mathsf{x}_m =_{s_{n+1} \ldots s_{n+m}} \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n))}$$

Figure 4: Type rules for the relevant statement types of the source language. The **sim/obj** pattern matches both **simple** and **object** types. All variables are assumed to have been associated with a type in the type environment, A.

Given the $\trianglelefteq_s$ partial order, well-typedness of a simple assignment statement can be expressed as follows:

$$\dfrac{\begin{array}{c}A \vdash \mathsf{x} : \tau_1 \qquad A \vdash \mathsf{y} : \tau_2 \\ (\tau_2 \times \textbf{zero}) \trianglelefteq_s (\tau_1 \times \textbf{zero})\end{array}}{A \vdash \mathit{welltyped}(\mathsf{x} =_s \mathsf{y})}$$

In Fig. 4 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. A typing environment associates all variables with a type.

In statements of the form $\mathsf{x} = \mathsf{op}(\mathsf{y}_1 \ \ldots \ \mathsf{y}_n)$, the op operation may be a comparison, a bit-wise operation, an addition, etc. Consider a subtraction (or bitwise exclusive or) of two pointer values. The result is not a pointer value, but either of the two pointer values can be reconstituted from the result given the other pointer value[2]. The result must therefore be described by the same location type as the two input pointer values and an **unknown** offset. There are operations from which operand pointer values cannot be reconstituted from the result (*e.g.*, comparisons: $<$, $\neq$, etc.). For such

[2] This is true for most implementations of C even though subtraction of pointers to different objects is implementation dependent according to the ANSI C specification [Ame89].

operations, the result is not required to be described by the same type as any input pointer value. We treat all primitive operations identically.

The typing rule for dynamic allocation states that some pointer value is being assigned. The type that describes 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 by the same statement will have the same type, but locations allocated by different allocation statements may have different types.

The typing rule for computing the address of a structure element makes use of a predicate, compatible($\mathtt{n}$,m). The details of the predicate is dependent on the choice of representation of element specifiers, but the predicate should capture that the mapping describes a structure whose prefix matches that of the structure being accessed up to and including the element `n`.

We have defined the typing rules under the assumption that the number of formal and actual parameters (and results) always match up. The rules are trivially extendible to handle programs where this is not the case and to handle programs with variable arguments (*e.g.*, using `<stdarg.h>` in C).

5 Efficient Type Inference

Performing a points-to analysis amounts to inferring a typing environment under which a program is well-typed. The typing environment we seek is the minimal solution to the well-typedness problem, *i.e.*, each location type describes as few locations as possible, and each function type describes as few functions as possible. In this section we state how to efficiently compute such a minimal solution.

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 unifying and merging 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. When all parts of the program has been processed, the program is well-typed.

5.1 Algorithm Stages

In the first stage of the algorithm we provide a typing environment where all program 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. ECRs allows us to replace two type variables with a single type variables by a constant time "union" operation. The initial type of each program variable is $\mathbf{blank}(s, \emptyset)$, where s is the size of the representation of the variable. We assume that name resolution has been performed and that 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 second stage of the algorithm we process each statement of the program exactly once. Type variables are joined as necessary to ensure well-typedness of each statement (as described in the next section). 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 once well-typed statement no longer be well-typed. If type variables are only joined when necessary to ensure well-typedness, the final type graph is the minimal solution we seek.

5.2 Processing Constraints

When processing a statement, we must ensure that the constraints imposed by the $\sqsubseteq_s$ and $\trianglelefteq_s$ partial orders are obeyed. This can be achieved by joining type variables and by "upgrading" **simple** and **struct** types to **object** types and **blank** types to **simple**, **struct**, or **object** types.

It may happen that the effects of a constraint cannot be determined at the time of processing the statement introducing the constraint. The algorithm uses latent constraints by annotating type variables with actions that are to be invoked if the "value" of the type variable should change.

For example, consider a partial order constraint between two function types, $\lambda_1 \sqsubseteq \lambda_2$. If λ_1 is anything other than $\perp$, then λ_1 and λ_2 must be joined to meet the constraint. However, we may not know at the time of processing the statement with the constraint whether λ_1 will be $\perp$ or something else in the final solution. Joining the two type variables will be safe, but it may be too conservative, and the final result may not be the minimal solution we seek. If λ_1 is $\perp$ at the time we encounter the constraint, we add to the set of latent actions associated with λ_1 that it should be joined with λ_2 if it ever changes value.

Figure 5 provides the precise set of rules for processing the relevant kinds of statements of a program. The processing rules follow immediately from the well-typedness rules and are straightforward to implement. Figure 6 provides the details of the join operations.

5.3 Complexity

We argue that the space and time complexity is exponential in the size of the input program using a theoretically correct (but practically meaningless) metric, is quadratic in the size of the program using a more reasonable metric, and is likely to be close to linear in the size of the program in practice.

The number of distinguishable memory locations in a program is $O(\exp N)$, where N is the size of the program. This is achievable by building a structure in the shape of a binary tree. A size N program could also populate all the "left" leaves of such a binary tree with pointers to the root of the tree. The points-to solution for such a program would be of size $O(\exp N)$. The runtime complexity of any points-to algorithm computing such a solution must therefore be exponential or worse.

While theoretically correct, expressing the algorithm complexity in terms of N is a practically meaningless metric of the complexity of the algorithm. We know of no related work using this metric; although several specify complexity in terms of N they are really using a different metric. A more reasonable metric measures the complexity of the algorithm in terms of the combined size, S, of all variables of the program.

The number of type variables created during the stages of our algorithm is $O(S)$. Any constraints not involving **struct** types can be processed in linear space and almost linear time complexity in terms of the number of type variables joined. For programs that do not use structured variables, the algorithm has a $O(S)$ space and

$\mathsf{x} =_s \mathsf{y}$
  let $\tau_1 = \mathbf{ecr}(\mathsf{x}),\ \tau_2 = \mathbf{ecr}(\mathsf{y})$ in
    $\mathbf{cjoin}(s, \tau_2, \tau_1)$

$\mathsf{x} =_s \&\mathsf{y}$
  let $\tau_1 = \mathbf{ecr}(\mathsf{x}),\ \tau_2 = \mathbf{ecr}(\mathsf{y})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau_1, s)$
    let $\mathbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) = \mathbf{type}(\tau_1)$ in
      if $s \not\sqsubseteq s_1$ then $\mathbf{expand}(\tau_1)$
      $\mathbf{join}((\tau_2 \times \mathbf{zero}), \alpha_1)$

$\mathsf{x} =_s *\mathsf{y}$
  let $\tau_1 = \mathbf{ecr}(\mathsf{x}),\ \tau_2 = \mathbf{ecr}(\mathsf{y})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau_2, s)$
    let $\mathbf{sim/obj}(\alpha_2, \lambda_2, s_2, p_2) = \mathbf{type}(\tau_2)$ in
      let $(\tau_3 \times o_3) = \alpha_2$ in $\mathbf{unless\text{-}zero}(o_3, \tau_3)$
      $\mathbf{cjoin}(s, \tau_3, \tau_1)$

$\mathsf{x} =_s \mathsf{allocate}(\mathsf{y})$
  let $\tau = \mathbf{ecr}(\mathsf{x})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau, s)$
    let $\mathbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) = \mathbf{type}(\tau)$ in
      if $s \not\sqsubseteq s_1$ then $\mathbf{expand}(\tau)$
      let $(\tau_1 \times o_1) = \alpha_1$ in
        if $\mathbf{type}(\tau_1) = \perp$ then
          $\mathbf{settype}(\tau_1, \mathbf{blank}(\top, \emptyset))$

$*\mathsf{x} =_s \mathsf{y}$
  let $\tau_1 = \mathbf{ecr}(\mathsf{x}),\ \tau_2 = \mathbf{ecr}(\mathsf{y})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau_1, s)$
    let $\mathbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) = \mathbf{type}(\tau_1)$ in
      let $(\tau_3 \times o_3) = \alpha_1$ in $\mathbf{unless\text{-}zero}(o_3, \tau_3)$
      $\mathbf{cjoin}(s, \tau_2, \tau_3)$

$\mathsf{x} =_s \mathsf{fun}(\mathsf{f}_1 \ldots \mathsf{f}_n) \rightarrow (\mathsf{r}_1 \ldots \mathsf{r}_m)\ S^*$
  let $\tau_0 = \mathbf{ecr}(\mathsf{x})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau_0, s)$
    let $\mathbf{sim/obj}(\alpha_0, \lambda_0, s_0, p_0) = \mathbf{type}(\tau_0)$ in
      if $s \not\sqsubseteq s_0$ then $\mathbf{expand}(\tau_0)$
      if $\mathbf{type}(\lambda_0) = \perp$ then
        let $[\tau_1 \ldots \tau_{n+m}] =$
            $\mathbf{MakeECR}(n+m)$ in
          let $t = \mathbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m})$ in
            $\mathbf{settype}(\lambda_0, t)$,
      let $\mathbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m}) = \lambda_0$ in
        for $i \in [1 \ldots n]$ do
          let $s_i = \text{sizeof}(\mathsf{f}_i),\ \tau_i' = \mathbf{ecr}(\mathsf{f}_i)$ in
            $\mathbf{cjoin}(s_i, \tau_i, \tau_i')$
        for $j \in [1 \ldots m]$ do
          let $s_{n+j} = \text{sizeof}(\mathsf{r}_j),\ \tau_{n+j}' = \mathbf{ecr}(\mathsf{r}_j)$ in
            $\mathbf{cjoin}(s_{n+j}, \tau_{n+j}', \tau_{n+j})$

$\mathsf{x} =_s \mathsf{op}(\mathsf{y}_1 \ldots \mathsf{y}_n)$
  let $\tau = \mathbf{ecr}(\mathsf{x})$ in
    for $i \in [1 \ldots n]$ do
      let $\tau_i = \mathbf{ecr}(\mathsf{y}_i)$ in $\mathbf{cjoin}(s, \tau_i, \tau)$
    $\mathbf{ensure\text{-}sim/obj}(\tau, s)$
    let $\mathbf{sim/obj}(\alpha', \lambda', s', p') = \mathbf{type}(\tau)$ in
      let $(\tau' \times o') = \alpha'$ in
        if $\mathbf{type}(o') = \mathbf{zero}$ then
          $\mathbf{make\text{-}unknown}(o')$

$\mathsf{x} =_s \&\mathsf{y\text{-}\!>n}$
  let $\tau_1 = \mathbf{ecr}(\mathsf{x}),\ \tau_0 = \mathbf{ecr}(\mathsf{y})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau_1, s)$
    $\mathbf{ensure\text{-}sim/obj}(\tau_0, \text{sizeof}(\mathsf{y}))$
    let $\mathbf{sim/obj}(\alpha_1, \lambda_1, s_1, p_1) = \mathbf{type}(\tau_1)$
        $\mathbf{sim/obj}(\alpha_2, \lambda_2, s_2, p_2) = \mathbf{type}(\tau_0)$ in
      if $s \not\sqsubseteq s_1$ then $\mathbf{expand}(\tau_1)$
      let $(\tau_2 \times o_2) = \alpha_2$ in
        if $\mathbf{type}(o_2) = \mathbf{unknown}$ then
          $\mathbf{collapse}(\tau_2)$, $\mathbf{join}(\alpha_2, \alpha_1)$
        else
          $\mathbf{unless\text{-}zero}(o_2, \tau_2)$
          if $\mathbf{type}(\tau_2) = \mathbf{blank}(s_3, p_3)$ then
            $m_3 = [\,]$
            $\mathbf{settype}(\tau_2, \mathbf{struct}(m_3, s_3, p_3))$
            $\mathbf{make\text{-}compatible}(\mathsf{n}, m_3)$
            $\mathbf{join}((m_3(\mathsf{n}) \times \mathbf{zero}), \alpha_1)$
          elseif $\mathbf{type}(\tau_2) = \mathbf{struct}(m_3, s_3, p_3)$ then
            $\mathbf{make\text{-}compatible}(\mathsf{n}, m_3)$
            $\mathbf{join}((m_3(\mathsf{n}) \times \mathbf{zero}), \alpha_1)$
          else
            $\mathbf{promote}(\tau_2, \text{sizeof}(*\text{y}))$, $\mathbf{join}(\alpha_2, \alpha_1)$

$\mathsf{x}_1 \ldots \mathsf{x}_m =_{s_{n+1} \ldots s_{n+m}} \mathsf{p}(\mathsf{y}_1 \ldots \mathsf{y}_n)$
  let $\tau_0 = \mathbf{ecr}(\mathsf{p})$ in
    $\mathbf{ensure\text{-}sim/obj}(\tau_0, \text{sizeof}(\mathsf{p}))$
    let $\mathbf{sim/obj}(\alpha_0, \lambda_0, s_0, p_0) = \mathbf{type}(\tau_0)$ in
      if $\mathbf{type}(\lambda_0) = \perp$ then
        let $[\tau_1 \ldots \tau_{n+m}] = \mathbf{MakeECR}(n+m)$ in
          let $t = \mathbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m})$ in
            $\mathbf{settype}(\lambda_0, t)$
      let $\mathbf{lam}(\tau_1 \ldots \tau_n)(\tau_{n+1} \ldots \tau_{n+m}) = \lambda_0$ in
        for $i \in [1 \ldots n]$ do
          let $s_i = \text{sizeof}(\mathsf{y}_i),\ \tau_i' = \mathbf{ecr}(\mathsf{y}_i)$ in
            $\mathbf{cjoin}(s_i, \tau_i', \tau_i)$
        for $j \in [1 \ldots m]$ do
          let $\tau_{n+j}' = \mathbf{ecr}(\mathsf{x}_j)$ in
            $\mathbf{cjoin}(s_{n+j}, \tau_{n+j}, \tau_{n+j}')$

Figure 5: Inference rules corresponding to the typing rules given in Fig. 4. $\mathbf{make\text{-}compatible}(\mathsf{n}, m)$ is a side-effecting predicate that modifies mapping m to be compatible with access of structure element n (if possible and necessary) and returns a boolean value indicating the success of this modification. $\mathbf{MakeECR}(x)$ constructs a list of x new ECRs, each associated with the bottom type, $\perp$. Figure 6 provides details of the other functions used in the above rules.

```
join((τ_1 × o_1), (τ_2 × o_2)):
  if type(o_1) = zero then
    pending(o_1) ←pending(o_2) ∪
      {<makeunknown,o_2>}
  elseif type(o_2) = zero then
    make-unknown(o_2)
  join(τ_1, τ_2)

join(e_1, e_2):
  if type(e_1) = ⊥ then
    pending(e_1) ←pending(e_1) ∪
      {<join,e_1,e_2>}
  else
    let e = ecr-union(e_1, e_2) in
      pending(e) ←
        pending(e_1) ∪ pending(e_2)
      type(e) ← type(e_1)
      settype(e, unify(e_1, e_2))

settype(e, t):
  type(e) ← t
  for a ∈ pending(e) do
    case a of
      [<join,e_1,e_2>]: join(e_1, e_2)
      [<cjoin,s,e_1,e_2>]: cjoin(s, e_1, e_2)

ensure-sim/obj(τ, s):
  case type(τ) of
    [⊥]: settype(τ, simple(⊥, ⊥, s, ∅))
    [blank(s', p)]:
      settype(τ, simple(⊥, ⊥, s', p))
      if s ⋢ s' then expand(τ)
    [simple(α, λ, s', p)]:
      if s ⋢ s' then expand(τ)
    [struct(m, s', p)]: promote(τ, s')

expand(e):
  let τ = blank(⊤, ∅) in
    settype(e, unify(type(e), τ))

promote(e, s):
  let τ = object(⊥, ⊥, s, ∅) in
    settype(e, unify(type(e), τ))

collapse(e):
  let τ = object(⊥, ⊥, ⊤, ⊤) in
    settype(e, unify(type(e), τ))

make-unknown(o):
  type(o) ← unknown
  for a ∈ pending(o) do
    case a of
      [<collapse,τ>]: collapse(τ)
      [<makeunknown,o'>]:
        make-unknown(o')

unless-zero(o, τ):
  if type(o) = zero then
    pending(o) ← {<collapse,τ>} ∪ pending(o)
  else collapse(τ)

cjoin(s, e_1, e_2):
  pending(e_1) ←{<cjoin,s,e_1,e_2>} ∪ pending(e_1)
  case type(e_1) of
    [⊥]: /* nothing */
    [blank(s_1, p_1)]:
      if s ⋢ s_1 then expand(e_1)
      elseif type(e_2) = ⊥ then
        settype(e_2, blank(s, ∅))
      elseif s ⋢ sizeof(type(e_2)) then
        expand(e_2)
    [simple(α_1, λ_1, s_1, p_1)]:
      if s ⋢ s_1 then expand(e_1)
      else
        case type(e_2) of
          [⊥]: settype(e_2, simple(α_1, λ_1, s, ∅))
          [blank(s_2, p_2)]:
            settype(e_2, simple(α_1, λ_1, s_2, p_2))
            if s ⋢ s_2 then expand(e_2)
          [simple(α_2, λ_2, s_2, p_2)]:
            join(α_1, α_2), join(λ_1, λ_2)
            if s ⋢ s_2 then expand(e_2)
          [struct(m_2, s_2, p_2)]: promote(e_2, s_2)
          [object(α_2, λ_2, ⊤, ∅)]:
            join(α_1, α_2), join(λ_1, λ_2)
    [struct(m_1, s_1, p_1)]:
      if s ⋢ s_1 then expand(e_1)
      else
        case type(e_2) of
          [⊥]: settype(e_2, struct(m_1, s, ∅))
          [blank(s_2, p_2)]:
            settype(e_2, struct(m_1, s_2, p_2))
            if s ⋢ s_2 then expand(e_2)
          [simple(α_2, λ_2, s_2, p_2)]: promote(e_2, s_2)
          [struct(m_2, s_2, p_2)]:
            if s ⊑ s_2 ∧
              ∀x ∈ Dom(m_1) :
                make-compatible(x, m_2) then
              for x ∈ Dom(m_1) do
                cjoin(sizeof(x), m_1(x), m_2(x))
            else expand(e_2)
          [object(α_2, λ_2, ⊤, ∅)]:
            for x ∈ Dom(m_1) do
              cjoin(sizeof(x), m_1(x), e_2)
    [object(α_1, λ_1, ⊤, ∅)]:
      if type(e_2) = object(α_2, λ_2, ⊤, ∅) then
        join(α_1, α_2), join(λ_1, λ_2)
      else promote(e_2, s)
```

Figure 6: Implementation details for the function used in the inference rules in Figure 5. **ecr**(x) is the ECR representing the type of variable x, and **type**(E) is the type associated with the ECR E. **join**(x, y) performs the conditional $\sqsubseteq_s$ join and **cjoin**(s, x, y) performs the conditional $\trianglelefteq_s$ join of ECRs x and y. 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.

$O(S\alpha(S,S))$ time complexity, where α is the inverse Ackerman's function [Tar83]. The $\alpha(S,S)$ component of the time complexity is due to the use of fast union/find data structures. This complexity result is equal to that of our previous algorithm [Ste96].

Constraints involving **struct** types may require processing all the element types in addition to any joins being performed. If all structures have R or fewer elements, the algorithm has an $O(S)$ space and $O(RS\alpha(S,S))$ time complexity. While this means that the algorithm has a quadratic worst-case running time complexity in terms of S, the actual running time complexity is likely to be close to linear as R is typically a fairly small number. While R *does* grow with program size, the growth is controlled by the tendency of programmers to group structure elements in substructures when the number of elements grows large.

6 Experience

We have implemented a slightly improved version of the above algorithm in our prototype programming system based on the Value Dependence Graph (VDG) [WCES94]. The implementation is performed in the Scheme programming language [CR91]. The implementation uses a weaker typing rule for primitive operations returning boolean values (thus leading to better results). It also uses predetermined transfer functions for calls of library functions, effectively making the type inference algorithm be polymorphic (context-sensitive) for all direct calls of library functions.

Our implementation demonstrates that the running time of the algorithm is roughly linear in the size of the input program on our test-suite of around 50 C programs. We have performed points-to analysis of programs up to 75,000 lines of code[3]. The experience with the algorithm is very encouraging; we are considering doing an implementation that allows piecewise analysis of programs, thus permitting analysis of programs of a million lines of code or more.

In Table 1 we present empirical data on the performance of the algorithm on the unoptimized representation of a number of programs. The programs are a subset of the programs in William Landi's test suite, Todd Austin's test suite, the SPEC92 benchmarks, and LambdaMOO (version 1.7.1) from Xerox PARC. These programs are the same we presented results for in our previous paper [Ste96]. We have also included information on analysis of a Microsoft tool of 75,000 lines of C code.

The first column indicates running time for our implementation of the algorithm. The time is the result of a single measurement. The time includes initial setup and type inference. The runtime measurements are not directly comparable with the runtime measurements presented in [Ste96] as the old implementation was able to use a trick to reduce the number of initial type variables by 50%. The second column indicates the number of extra distinguishable elements of structured objects compared with our previous algorithm [Ste96]. An object with two distinguishable elements will thus contribute a count of one to this number. These numbers are very significant as they in most cases represent separation of distinguishable elements in central data structures. The separation has significant second-order effects on the results, but space limitations prevent us from providing details.

[3]This is the largest program we have represented in the VDG program representation.

Benchmark name	running time	struct count
landi:allroots	0.23/0.21s	0
landi:assembler	2.47/2.38s	10
landi:loader	0.99/0.96s	6
landi:compiler	1.17/1.16s	5
landi:simulator	2.81/2.62s	8
landi:lex315	0.50/0.49s	0
landi:football	4.34/3.51s	1
austin:anagram	0.44/0.37s	2
austin:backprop	0.30/0.28s	0
austin:bc	5.03/4.19s	11
austin:ft	0.73/0.65s	12

Benchmark name	running time	struct count
austin:ks	0.76/0.70s	4
austin:yacr2	3.40/2.45s	0
spec:compress	1.12/0.80s	0
spec:eqntott	3.05/2.30s	1
spec:espresso	30.0/22.2s	121
spec:li	8.96/6.47s	41
spec:sc	10.8/8.08s	12
spec:alvinn	0.28/0.27s	0
spec:ear	2.40/2.12s	6
LambdaMOO	25.3/19.5s	147
MS tool	95.4/58.7s	1747

Table 1: Running time (wall time and process time on a 150MHz Indigo2 running Chez Scheme) and number of extra distinguishable structure components relative to our previous algorithm [Ste96].

7 Related Work

The algorithm presented in this paper is an extension of two almost-linear points-to analysis algorithms that did not distinguish between components of structured objects [Ste96, Ste95]. William Landi independently arrived at the earliest of these algorithms [Lan95]. Barbara Ryder and Sean Zhang have independently developed an similar algorithm that distinguishes components of structured objects [Zha95]. They use a type system without a $\perp$ element, substituting the $\sqsubseteq$ operator by the $=$ operator, thus not being as precise as our algorithm. David Morgenthaler extended our earliest algorithm to distinguish components of structured objects [Mor95]. His algorithm also uses a type system without a $\perp$ element and does not incorporate pointer offsets in the constraint system. Furthermore, his implementation is not meant to deal correctly with unions. His analysis is performed during parsing of the program.

Henglein used type inference to perform a binding time analysis in almost linear time [Hen91]. His types represent binding time values. Our points-to analysis algorithms have been inspired by Henglein's type inference algorithm.

Choi, *et al.*, developed a flow-insensitive points-to analysis based on data flow methods [CBC93]. Their algorithm was only developed for a language with pair structures (like cons cells in Lisp). Their algorithm has worse time and space complexity than our algorithm. Burke, *et al.*, describes an improvement of the algorithm [BCCH95]. The improved algorithm does not deal with pointers into structured objects and has worse time and space complexity than our algorithm. Both algorithms are potentially more accurate than our algorithm, as their analysis results permit a location representative to have pointers to multiple other location representatives.

Andersen defined a flow-insensitive, context-sensitive[4] points-to analysis in terms of constraints and constraint solving [And94]. The values being constrained are sets of abstract locations, the analysis being more conventional than the analysis presented in the present paper. His algorithm assumes source programs to be strictly conforming to ANSI C and may generate unsafe results for the large class of programs written by

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

programmers who make "creative" assumptions about the language implementation. A context-insensitive version of Andersen's algorithm would compute results very similar to those of [BCCH95] but is likely to be faster since it is based on constraint solving rather than data flow analysis.

O'Callahan and Jackson convert C programs to ML programs and use ML type inference to compute the equivalent of points-to results [OJ95]. Not all C programs can be converted to ML by their techniques, and even then their algorithm may compute unsafe results due to type casts in the source program.

There exist many interprocedural flow-sensitive data flow analyses, *e.g.*, [CWZ90, EGH94, WL95, Ruf95]. Both the algorithm by Chase, *et al.*, [CWZ90] and Ruf's algorithm [Ruf95] are context-insensitive and have polynomial time complexity. The two other algorithms are context-sensitive. The algorithm by Emami, *et al.*, [EGH94] has a exponential time complexity (in theory and in practice), 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.

8 Conclusion and Future Work

We have presented a flow-insensitive, interprocedural, context-insensitive points-to analysis based on type inference methods. The algorithm is being implemented. We will have empirical evidence that the algorithm is very efficient in practice before the final version of the paper is due.

This work is part of an effort to construct very efficient points-to analysis algorithms for large programs. We have found type inference methods a very useful tool for doing so. The algorithms presented in this paper and in previous papers [Ste96, Ste95] are based on monomorphic type inference methods. We have also investigated extending the algorithm of [Ste96] to use polymorphic type inference methods. We have yet to combine the extensions to generate an context-sensitive (polymorphic) points-to algorithm that can distinguish between elements of structured objects.

Acknowledgments

Roger Crew, Michael Ernst, Erik Ruf, 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. Daniel Weise and the reviewers provided helpful comments. The author also enjoyed interesting discussions with David Morgenthaler, William Griswold, Barbara Ryder, Sean Zhang, and Bill Landi on performing points-to analysis by type inference methods.

References

[Ame89] American National Standards Institute, Inc. Programming language — C, December 1989.

[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, 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.

[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.

[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.

[OJ95] Robert O'Callahan and Daniel Jackson. Detecting shared representations using type inference. Technical Report CMU-CS-95-202, School of Computer Science, Carnegie Mellon University, September 1995.

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

[Ste95] Bjarne Steensgaard. Points-to analysis in almost linear time. Technical Report MSR-TR-95-08, Microsoft Research, March 1995.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings 23rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

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

[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.

[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.