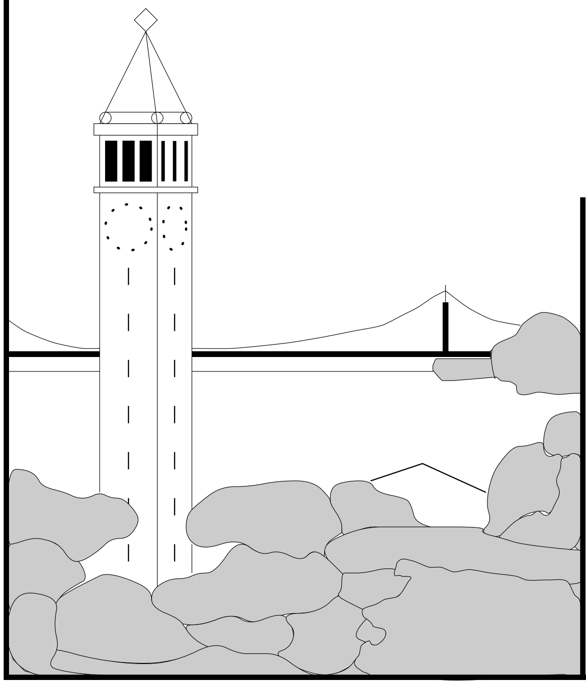
Flow-Insensitive Points-to Analysis with Term and Set Constraints

Jeffrey S. Foster *Manuel Fähndrich* *Alexander Aiken*



Report No. UCB/CSD-97-964

August 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Flow-Insensitive Points-to Analysis with Term and Set Constraints*

Jeffrey S. Foster†‡ Manuel Fähndrich‡ Alexander Aiken‡

EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
{jfoster, manuel, aiken}@cs.berkeley.edu

August 1997

Abstract

We describe new type systems for two kinds of flow-insensitive points-to analyses, one based on Andersen's algorithm and one based on Steensgaard's. The type systems are formulated using a mixed constraint framework. These systems can be seen as a straightforward axiomatization of an informal description of the algorithms, and we show this formally by proving soundness with respect to an operational semantics. Further, we show that these two systems are nearly identical, except that one uses subset constraints and one uses unification. We discuss an implementation of these systems and describe experiments that demonstrate that our general framework achieves running times within a small constant factor of a hand-coded solution. We conclude that a mixed constraint system provides a useful, practical framework for static semantic analyses.

1 Introduction

For each pointer-valued expression in a program, points-to analysis computes the set of memory locations to which that expression could point. The *may-alias* information computed by points-to analysis enables many optimizations in pointer-based languages such as C and C++ [ASU88]. Several *flow-insensitive* points-to analyses for C have been proposed [And94, Ste96a, Ste96b, SH97]. These flow-insensitive analyses achieve good scaling behavior by ignoring the ordering of statements [Hor97].

Shapiro and Horwitz [SH97] compare Andersen's and Steensgaard's flow-insensitive points-to analyses [And94, Ste96b]. Both algorithms were originally presented using non-standard type systems; however, although both algorithms solve the same problem, the solutions look very different. [SH97] informally unifies the two approaches by describing the algorithms' operation on a points-to graph [EGH94].

In this paper, we present new constraint-based type systems for Andersen's and Steensgaard's analyses. The correspondence between these new systems is made clear, formalizing the description in [SH97]. The differences between the two algorithms are expressed in a minimal fashion as a choice between inclusion and equality constraints.

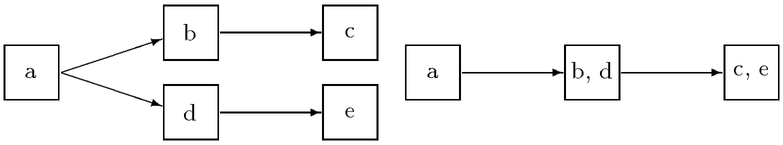
Our type systems are designed using term and set constraints. Set constraints [AW92, AW93, FA96, HJ90, Hei92] define inclusion relationships between types; we use set constraints to describe Andersen's analysis. Term constraints define equality relationships between types (*e.g.* ML type inference [Mil78]); we use term equations to describe Steensgaard's analysis. We have implemented these new type systems in a mixed term and set constraint framework [FA97]. The goal of our work is to show that not only

*Revised November, 1997.

†Supported by an NDSEG fellowship.

‡Supported in part by NSF Young Investigator Award CCR-9457812, NSF Grant CCR-9416973, and a gift from Rockwell Corporation.

```
a = &b
b = &c
a = &d
d = &e
```



(a) Andersen's Analysis (b) Steensgaard's Analysis

Figure 1: Points-to graphs generated by Andersen and Steensgaard's algorithms (from [SH97]).

is this framework a natural formalism for expressing two very practical algorithms, but that it also is a useful and productive system in which to implement the analyses.

In Section 1.1, we give an intuitive description of Andersen's and Steensgaard's analyses. Section 2 discusses the original type systems for these analyses. In Section 3 we briefly discuss the term and set constraint framework. We present our type system for Andersen's analysis in Sections 4 and 5, and our system for Steensgaard's analysis in Section 6. We formally show the correspondence between the two analyses and prove a soundness result in Section 7. Finally, Section 8 discusses implementation and performance issues. Section 9 concludes.

1.1 Informal Description

Both Andersen's and Steensgaard's analyses build a *points-to* graph [EGH94]. The nodes of a points-to graph represent memory locations or sets of memory locations, and there must be an edge from x to y if any permutation of the assignment statements in the program may cause location x to contain a pointer to y. In our analysis, memory locations include local and global variables, function addresses, strings, and heap objects.

Figure 1 shows the points-to graphs computed by Andersen's and Steensgaard's analyses for a simple C program. Informally, Andersen's analysis begins begins with some initial points-to relationships and closes the graph under the following rule:

> For an assignment $e_1 = e_2$, anything in the points-to set for e_2 must also be in the points-to set for e_1.

Thus, each node may have n successors, where n is the number of locations in the program. For example, applying this rule to the assignment statements in Figure 1 yields the points-to graph in Figure 1a. The statement `a = &b` adds b to the points-to set for a. The next statement `b = &c` adds c to the points-to set for b. The last two statements add d to the points-to set for a and e to the points-to set for d.

Steensgaard's analysis uses the same idea, except each node is allowed at most one successor. Rather than representing individual variables, each node in the points-to graph is an equivalence class $[\cdot]$ of variables. The algorithm joins equivalence classes to satisfy the one-sucessor constraint. Thus, there is an edge from $[x]$ to $[y]$ if any node in $[x]$ may contain a pointer to a node in $[y]$. Informally, we apply the rule

> For the assignment $e_1 = e_2$, the points-to set for e_2 must be equal to the points-to set for e_1.

For example, in Figure 1b, the first two assignments act as in Andersen's algorithm: a points to b and b points to c. But then the assignment `a = &d` causes b and d to be unified, making d also point to c. The last assignment `d = &e` causes c and e to be unified.

Intuitively, Steensgaard's analysis trades precision for efficiency. In Figure 1b, Steensgaard's analysis conservatively computes that c and e may be aliased, when in fact they cannot.

$$
\begin{array}{rcl}
\cup & : & \texttt{Set Set} \rightarrow \texttt{Set} \\
\cap & : & \texttt{Set Set} \rightarrow \texttt{Set} \\
0 & : & \texttt{Set} \\
1 & : & \texttt{Set}
\end{array}
$$

Figure 2: Operations in the sort `Set`.

2 Related Work

Andersen proposes his algorithm as part of a larger thesis about the analysis of C programs [And94]. His type system requires a specialized constraint to model updatable references. In contrast, we show that inclusion constraints on updatable references can be modeled naturally using standard notions of covariance and contravariance. We discuss this issue in Section 4.

Steensgaard proposed his analysis as an efficient alternative to Andersen's [Ste96b]. Andersen's analysis runs in $O(n^3)$ time, where n is the number of locations in the program. By using unification instead of inclusion constraints, Steensgaard reduces this to $O(n\alpha(n,n))$, where α is the inverse Ackerman's function.

Steensgaard's system is surprisingly difficult to understand. First, [Ste96b] gives a system for type checking rather than type inference: Statements are type checked but the system does not actually assign types to program variables. Second, Steensgaard's type system only includes rules for certain forms of statements and not for arbitrary expressions, which complicates the rules by mixing addressing, dereferencing, and assignment expressions together in the same rule. Finally, his system uses a non-standard notion of type equivalence. These three points combined to make formal reasoning about the system very difficult.

Our work was inspired by Shapiro and Horwitz's paper examining the precision-efficiency tradeoff between the two analyses [SH97]. They describe the analyses as being at two ends of a spectrum: Andersen's, the more precise, allows an arbitrary set of successors for a node, while Steensgaard's, which is less precise, allows only one. They measure the performance and precision of both analyses. To make these comparisons, Shapiro and Horwitz implemented their algorithms in C. In Section 8, we describe our own implementation and compare its performance to their hand-coded C version.

3 The Analysis Framework

We have developed type systems for Andersen's and Steensgaard's points-to analyses in the context of the mixed term and set constraint framework described in [FA97]. Each analysis generates a system of constraints, and we compute the points-to graph for a program by solving the generated constraints. In this section, we briefly discuss the framework. We describe the constraint language and give resolution rules for solving a system of constraints.

From the user's point of view, our framework consists of a number of sorts of expressions together with resolution rules for constraints over those expressions. We discuss only the two sorts `Term` and `Set` used in our points-to analyses.

Each sort s is parameterized by user-defined constructor signatures. If S is the set of sorts, each n-ary constructor c is given a signature

$$c : \iota_1 \ldots \iota_n \rightarrow S$$

where ι_i is s or $\overline{s}$ for some $s \in S$. Overlined sorts mark contravariant arguments of c; the rest are covariant arguments. Sort `Term` is a set of constructors $\Sigma_{\texttt{Term}}$ and variables $V_{\texttt{Term}}$. Terms over $\Sigma_{\texttt{Term}}$ and $V_{\texttt{Term}}$ are defined by giving constructor signatures

$$c : \underbrace{\texttt{Term} \ldots \texttt{Term}}_{arity(c)} \rightarrow \texttt{Term} \quad c \in \Sigma_{\texttt{Term}}$$

Sorts may also have operations; sort `Set` includes the set operations in Figure 2 (the set operations plus least and greatest sets). Set expressions are defined by the signatures

$$c : \underbrace{\texttt{Set} \ldots \texttt{Set}}_{arity(c)} \rightarrow \texttt{Set} \quad c \in \Sigma_{\texttt{Set}}$$

$$
\begin{array}{rcll}
S \wedge f(T_1, \ldots, T_n) =_{\mathtt{t}} f(T'_1, \ldots, T'_n) & \equiv & S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge T'_1 \subseteq_{\iota_1} T_1 \wedge \cdots \wedge & \text{if } f : \iota_1 \cdots \iota_n \to \mathtt{Term} \\
 & & T_n \subseteq_{\iota_n} T'_n \wedge T'_n \subseteq_{\iota_n} T_n & \\
S \wedge f(\ldots) =_{\mathtt{t}} g(\ldots) & \equiv & \text{inconsistent} & \text{if } f \neq g \\
S \wedge f(\ldots) =_{\mathtt{t}} \alpha \wedge \alpha =_{\mathtt{c}} T & \equiv & \alpha =_{\mathtt{t}} T &
\end{array}
$$

(a) Resolution rules for sort `Term`.

$$
\begin{array}{rcll}
S \wedge 0 \subseteq_{\mathtt{s}} T & \equiv & S & \\
S \wedge T \subseteq_{\mathtt{s}} 1 & \equiv & S & \\
S \wedge c(T_1, \ldots, T_n) \subseteq_{\mathtt{s}} c(T'_1, \ldots, T'_n) & \equiv & S \wedge T_1 \subseteq_{\iota_1} T'_1 \wedge \cdots \wedge T_n \subseteq_{\iota_n} T'_n & \text{if } c : \iota_1 \cdots \iota_n \to \mathtt{Set} \\
S \wedge c(\ldots) \subseteq_{\mathtt{s}} d(\ldots) & \equiv & \text{inconsistent} & \text{if } c \neq d \\
S \wedge T_1 \cup T_2 \subseteq_{\mathtt{s}} T & \equiv & S \wedge T_1 \subseteq_{\mathtt{s}} T \wedge T_2 \subseteq_{\mathtt{s}} T & \\
S \wedge T \subseteq_{\mathtt{s}} T_1 \cap T_2 & \equiv & S \wedge T \subseteq_{\mathtt{s}} T_1 \wedge T \subseteq_{\mathtt{s}} T_2 & \\
S \wedge \alpha \subseteq_{\mathtt{s}} \alpha & \equiv & S & \\
S \wedge \alpha \cap T \subseteq_{\mathtt{s}} \alpha & \equiv & S &
\end{array}
$$

(b) Resolution rules for sort `Set`.

$$
\begin{array}{rcl}
S \wedge X \subseteq_{\iota} \alpha \wedge \alpha \subseteq_{\iota} Y & \equiv & S \wedge X \subseteq_{\iota} \alpha \wedge \alpha \subseteq_{\iota} Y \wedge X \subseteq_{\iota} Y \\
S \wedge T_1 \subseteq_{\overline{\iota}} T_2 & \equiv & S \wedge T_2 \subseteq_{\iota} T_1
\end{array}
$$

(c) General rules.

Figure 3: Resolution rules for constraints.

Each sort s has a constraint relation $\subseteq_s$ and resolution rules. Figure 3 gives the rules for sorts `Term` and `Set`. Constraints and resolution rules preserve sorts, so that $X \subseteq_s Y$ implies X and Y are s-expressions.

For the `Term` sort, there are two constraint relations. $\subseteq_{\mathtt{Term}}$ is equality, and the first two resolution rules in Figure 3a implement term unification for constructors with signatures `Term`...`Term` $\to$ `Term`. For clarity we write the constraint relation as " $=_{\mathtt{t}}$ " instead of $\subseteq_{\mathtt{Term}}$. The second constraint relation is conditional equality, written $\subseteq_{\mathtt{c}}$ or $=_{\mathtt{c}}$. The last rule in Figure 3a turns conditional equality $\alpha =_{\mathtt{c}} T$ into unconditional equality $\alpha =_{\mathtt{t}} T$ when α is instantiated with a constructed term. We use conditional equality for Steensgaard's analysis; see Section 6.

Figure 3b shows the rules for the `Set` sort. We give only the standard rules [AW93] used for Andersen's analysis. We have omitted rules for negations, rules for simplifying intersections, and some restrictions on the form of solvable constraints. The details may be found in [AW93, FA97].

Figure 3c gives two general rules that apply to all sorts. The first rule expresses that $\subseteq_\iota$ is transitive. The second flips constraints that arise from contravariant constructor arguments.

4 Andersen's Analysis: A First Cut

We begin by examining why standard set-based analysis [Hei92] seems insufficient for expressing Andersen's algorithm. Consider the C fragment

$$e ::= \mathtt{x} \mid *e \mid \&e \mid e_1 = e_2 \mid e_1, e_2$$

consisting of variables, pointer dereferencing, the address-of operator, assignment, and sequencing. Our goal is to assign a type and a series of constraints to each expression such that the solution to the constraints yields the Andersen points-to graph of the program.

With each variable `x` we associate a type variable x representing the contents of `x`. We shall maintain this font distinction, writing the source language in `typewriter` and writing types in *italics*. To express pointer types, we define a parameterized type constructor $ref(\alpha)$, whose informal meaning is *pointer to* α. Thus, our initial grammar for types τ is

$$\tau ::= \alpha \mid ref(\tau)$$

We ignore the type of function values for the moment. To compute the points-to graph, we give a series of inference rules with constraints as side conditions. A derivation is valid only if the constraints in the derivation are satisfiable.

Definition 4.1 Let CS be a set of constraints, and let c be an arbitrary constraint. Define $CS \vdash c$ if c holds in any interpretation of CS consistent with the resolution rules in Figure 3.

For the type systems presented here, $CS \vdash c$ can be determined by applying the resolution rules until closure and then checking whether c is in the final system of constraints. We extract the points-to graph from the constraints by defining the points-to relation P_{Prelim}:

Definition 4.2 Let L be the set of memory locations, $L = \{\mathtt{x} \mid \mathtt{x} \text{ is a program variable}\}$. Define the points-to relation $P : L \to 2^L$ as $\mathtt{y} \in P_{Prelim}(\mathtt{x})$ if

$$CS \vdash \mathit{ref}(y) \subseteq_{\mathsf{s}} x$$

Thus $\mathtt{x}$ points to $\mathtt{y}$ if $\mathtt{y} \in P_{Prelim}(\mathtt{x})$.

The first inference rule is the standard rule for sequencing:

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{e_1, e_2 : \tau_2} \qquad (\text{Seq}_{\text{Prelim}})$$

The rule for variables is

$$\frac{}{\mathtt{x} : \mathit{ref}(x)} \qquad (\text{Var}_{\text{Prelim}})$$

Notice that we have lifted $\mathtt{x}$ to have the type *pointer to x*. By treating a variable as its l-value (its address) we avoid separate rules for l- and r-values.

The address-of operator takes an expression of type τ and returns a pointer to τ:

$$\frac{e : \tau}{\&e : \mathit{ref}(\tau)} \qquad (\text{Addr}_{\text{Prelim}})$$

For the assignment $e_1 = e_2$, the points-to set for e_2 should be a subset of the points-to set for e_1. We express this requirement using an inclusion constraint:

$$\frac{\begin{array}{c} e_1 : \tau_1 \quad e_2 : \tau_2 \\ \tau_2 \subseteq_{\mathsf{s}} \tau_1 \end{array}}{e_1 = e_2 : \tau_2} \qquad (\text{Asst}_{\text{Prelim}})$$

Here the type of the assignment is the type of the right-hand side, which is more precise. The types τ_1 and τ_2 describe the types that e_1 and e_2 point to. Assigning e_2 to e_1 means that e_1 could point to anything e_2 points to, hence the constraint $\tau_2 \subseteq_{\mathsf{s}} \tau_1$.

The previous rules work correctly within the fragment of the language they cover. However, a problem arises when we try to write down the rule for dereferencing. What the rule should do is clear: It should be the inverse of ($\text{Addr}_{\text{Prelim}}$). If the expression e contains a pointer to some type τ, then $*e$ has type τ. To find the type e points to, we project out a ref from the type of e:[1]

$$\frac{e : \tau \quad \tau \subseteq_{\mathsf{s}} \mathit{ref}(\alpha)}{*e : \alpha} \qquad (\text{Deref}_{\text{Prelim}})$$

Unfortunately, with this rule we have broken an implicit invariant of the other four rules. Previously, the type of an expression was exactly a constructed type $\mathit{ref}(\alpha)$. But in the rule ($\text{Deref}_{\text{Prelim}}$), if e has type $\mathit{ref}(\beta)$ then the type of $*e$ is only an upper bound on β. Thus, rule ($\text{Asst}_{\text{Prelim}}$) no longer works. In that rule, now τ_1 may only be an upper bound on the type we want to constrain, and so the constraint $\tau_2 \subseteq_{\mathsf{s}} \tau_1$ may have no effect.

For example, consider the statement `*x = y`. Applying the type rules yields

$$\frac{\dfrac{\dfrac{}{\mathtt{x} : \mathit{ref}(x)} \quad \mathit{ref}(x) \subseteq_{\mathsf{s}} \mathit{ref}(\alpha)}{*\mathtt{x} : \alpha} \qquad \dfrac{}{\mathtt{y} : \mathit{ref}(y)} \qquad \mathit{ref}(y) \subseteq_{\mathsf{s}} \alpha}{*\mathtt{x} = \mathtt{y} : \mathit{ref}(y)}$$

[1] In set-based analysis [Hei92], this would be written with an explicit projector ref^{-1}.

def	::=	$\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \rightarrow \mathtt{y} = e$	Function definition
	|	$def_1\, def_2$	Sequencing
e	::=	n	Constant integer n
	|	$\mathtt{x}$	Variable
	|	$*e$	Pointer dereference
	|	$\&e$	Address of e
	|	$\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$	Conditional
	|	$e_1 = e_2$	Assignment
	|	$\mathtt{f}(\mathtt{z}_1, \ldots, \mathtt{z}_m)$	Function application
	|	*"string"*	String constant
	|	$\mathtt{malloc}(e)$	Heap allocation
	|	$e_1\, op\ e_2$	Scalar operation ($e.g.$ +, --, *)
	|	e_1, e_2	Sequencing
	|	$(type)\ e$	Type cast
	|	$e.id$	Field access
	|	$e \rightarrow id$	Field indirection ($= (*e).id$)
	|	$e_1[e_2]$	Array access ($\approx *(e_1 + e_2)$)

Figure 4: Syntax for the source language

The points-to set for y flows from y to α, but the connection to x is broken because α is an upper bound on x.

This direct approach fails because of a well-known problem with updatable references. References of type α can be viewed as an abstract data type with two operations $get : unit \rightarrow \alpha$ and $set : \alpha \rightarrow unit$. Notice that α appears both covariantly and contravariantly, which suggests that correctly modeling references requires covariant and contravariant components. Our naive type rules model references as covariant, which fails. Set-based analysis [Hei92] does not directly model contravariance, and so without specialized constraints as in [And94], it cannot express Andersen's analysis.

5 Andersen-Style Analysis

5.1 Source language

Figure 4 shows the fragment of C for which we present our type rules. Because the analyses are flow-insensitive, many constructs of the language have no effect. A program consists of a sequence of function definitions. Note that we give a name to the return value from a function (written as y in the grammar). Return statements are treated as assignments to this special return value. For example, within the function foo, the statement `return a` is interpreted as `@foo_return = a`.

We assume that the source program satisfies C's type rules. For example, we make no effort to disallow strings on the left-hand side of an assignment in the abstract syntax, although of course this could not happen in a legal program.

5.2 Types

Our solution to the problem outlined in Section 4 is to reflect the *get* and *set* operations of updatable references directly in the type rules. We split what was one field in Section 4 into two fields, one of which is covariant and the other of which is contravariant. In the cases when these fields are identical, we have an invariant field. By carefully choosing where these fields are not identical, we can control the flow of information precisely to yield Andersen's analysis.

In this new system, types are described by the following grammar:

$$\begin{array}{lcl} \tau_A & ::= & \alpha \mid ref(p_{get} = \tau_A, p_{set} = \tau_A, f_{get} = \lambda_A, f_{set} = \lambda_A) \\ \lambda_A & ::= & \beta \mid lam_n(\tau_{A_0}, \tau_{A_1}, \ldots, \tau_{A_n}) \end{array}$$

with constructor signatures

And

$$\frac{}{n : 0} \qquad (\mathrm{Const-Int_A})$$

$$\frac{}{\mathtt{x} : \mathit{ref}(p_{get} = p_{set} = x, f_{get} = f_{set} = x')} \qquad (\mathrm{Var_A})$$

$$\frac{e : \tau}{\&e : \mathit{ref}(p_{get} = p_{set} = \tau)} \qquad (\mathrm{Addr_A})$$

$$\frac{e : \tau \quad \tau \subseteq_s \mathit{ref}(p_{get} = \alpha)}{*e : \alpha} \qquad (\mathrm{Deref_A})$$

$$\frac{\begin{array}{cc} e_1 : \tau_1 & e_2 : \tau_2 \\ \tau_1 \subseteq_s \mathit{ref}(p_{set} = \alpha, f_{set} = \alpha') & \tau_2 \subseteq_s \mathit{ref}(p_{get} = \beta, f_{get} = \beta') \\ \beta \subseteq_s \alpha & \beta' \subseteq_s \alpha' \end{array}}{e_1 = e_2 : \tau_2} \qquad (\mathrm{Asst_A})$$

$$\frac{\begin{array}{c} \mathtt{f} : \mathit{ref}(f_{get} = f_{set} = f) \\ \mathtt{y} : \tau_y \quad e : \tau_e \quad \forall 1 \le i \le n \,.\, \mathtt{x}_i : \tau_i \\ \mathit{lam}_n(\tau_y, \tau_1, \ldots, \tau_n) \subseteq_s f \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \rightarrow \mathtt{y} = e : \mathtt{void}} \qquad (\mathrm{Defn_A})$$

$$\frac{\begin{array}{ccc} \mathtt{f} : \tau_f & \mathtt{x}_i : \tau_i & \forall 1 \le i \le n \\ \tau_f \subseteq_s \mathit{ref}(f_{get} = \mathit{lam}_n(\alpha_0, \alpha_1, \ldots, \alpha_n)) & & \\ \alpha_i \subseteq_s \mathit{ref}(p_{set} = \alpha_{i_f}, f_{set} = \alpha'_{i_f}) & \tau_i \subseteq_s \mathit{ref}(p_{get} = \alpha_{i_x}, f_{get} = \alpha'_{i_x}) & \\ \alpha_{i_x} \subseteq_s \alpha_{i_f} & \alpha'_{i_x} \subseteq_s \alpha'_{i_f} & \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) : \alpha_0} \qquad (\mathrm{App_A})$$

Figure 5: Inference rules for Andersen-style analysis.

$$\begin{array}{rl} \mathit{ref} : & (p_{get} = \mathtt{Set}, p_{set} = \overline{\mathtt{Set}}, f_{get} = \mathtt{Set}, f_{set} = \overline{\mathtt{Set}}) \rightarrow \mathtt{Set} \\ \mathit{lam}_n : & \underbrace{\mathtt{Set} \cdots \mathtt{Set}}_{n+1} \rightarrow \mathtt{Set} \end{array}$$

The informal meaning of $\mathit{ref}(p_{get} = p_{set} = \alpha, f_{get} = f_{set} = \beta)$ is *pointer to location* α and *pointer to function* β. The p_{get} field is covariant, the p_{set} field contravariant. There is one lam_n constructor for each function arity, where $\mathit{lam}_n(\alpha_0, \alpha_1, \ldots \alpha_n)$ stands for a function with n arguments whose formal parameters have types $\alpha_1 \ldots \alpha_n$ and whose return value has type α_0. We discuss functions with variable numbers of arguments in Section 8. When we omit a label from a type, it means that field contains a fresh, unconstrained variable.

Figure 5 shows the basic type system **And** for our implementation of Andersen's algorithm. An implicit global environment maps each variable `x` to a type $\mathit{ref}(p_{get} = p_{set} = x, f_{get} = f_{set} = x')$, as expressed in the rule ($\mathrm{Var_A}$). We assume that all variables are distinct; in our implementation, we first α-convert the program before applying the type rules.

The remaining rules **Common** covering the rest of C are shown in Figure 6. Although some of these rules look non-compositional, in fact that is not the case. The non-compositional antecedents of type rules like (Array) merely serve as convenient abbreviations for more complicated rules.

Let

$$\vdash_A \equiv \text{provable in } \textbf{And+Common}$$

We define the points-to graph for Andersen-style analysis:

Definition 5.1 Let L be the set of all memory locations in the program; a memory location is a local or global variable, string, or a syntactic occurrence of `malloc`. Define the points-to relation $P_A : L \rightarrow 2^L$ as $\mathtt{y} \in P_A(\mathtt{x})$ if

$$CS \vdash_A \mathit{ref}(p_{get} = p_{set} = y, f_{get} = f_{set} = y') \subseteq_s x$$

where x and y are from the rule ($\mathrm{Var_A}$).

Common

$$\frac{\&V_l : \tau \qquad V_l \text{ fresh}}{\text{``}string_l'' : \tau} \qquad (\mathrm{Const-Str})$$

$$\frac{\&V_l : \tau \quad e : \tau_e \qquad V_l \text{ fresh}}{malloc_l(e) : \tau} \qquad (\mathrm{Malloc})$$

$$\frac{\begin{array}{c} V_l : \tau \quad e_1 : \tau_1 \quad e_2 : \tau_2 \quad e_3 : \tau_3 \qquad V_l \text{ fresh} \\ V_l = e_2 : \tau_1' \quad V_l = e_3 : \tau_2' \end{array}}{\mathtt{if}_l\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau} \qquad (\mathrm{Cond})$$

$$\frac{\begin{array}{c} V_l : \tau \quad e_1 : \tau_1 \quad e_2 : \tau_2 \qquad V_l \text{ fresh} \\ V_l = e_1 : \tau_1' \quad V_l = e_2 : \tau_2' \end{array}}{e_1\ op_l\ e_2 : \tau} \qquad (\mathrm{Op})$$

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{e_1, e_2 : \tau_2} \qquad (\mathrm{Seq})$$

$$\frac{def_1 : \mathtt{void} \quad def_2 : \mathtt{void}}{def_1 def_2 : \mathtt{void}} \qquad (\mathrm{Def-Seq})$$

$$\frac{e : \tau}{(type)\ e : \tau} \qquad (\mathrm{Cast})$$

$$\frac{e : \tau}{e.id : \tau} \qquad (\mathrm{Field-direct})$$

$$\frac{*e : \tau}{e \to id : \tau} \qquad (\mathrm{Field-indirect})$$

$$\frac{*(e_1 + e_2) : \tau}{e_1[e_2] : \tau} \qquad (\mathrm{Array})$$

Figure 6: Inference rules common to both analyses.

P_A defines the points-to graph: There is an edge from $\mathtt{x}$ to $\mathtt{y}$ if $\mathtt{y} \in P_A(\mathtt{x})$. Note that this interpretation is somewhat different than the typical result of set constraint analysis. In this case, only the relationships between types are important. It would be possible to add constants to the system by adding more fields to ref, but that adds no power.

We assert that **And+Common** is exactly an axiomatization of the informal algorithm described earlier. We shall discuss each type rule of **And** to illustrate this correspondence and prove the claim formally in Section 7.2.

5.3 Discussion of Inference Rules

The first rule of **And**, $(\mathrm{Const-Int_A})$, assigns the empty set to integers. As discussed before, the rule for variables $(\mathrm{Var_A})$ lifts the type of a variable to a pointer type. The rule $(\mathrm{Addr_A})$ says that $\&e$ points to e. The rule for dereferencing $(\mathrm{Deref_A})$ states almost exactly the reverse: $*e$ is an upper bound on the type of whatever e points to. Notice that we use the p_{get} field to *get* the contents of e.

Informally, the rule $(\mathrm{Asst_A})$ should make the points-to set for e_2 a lower bound on the points-to set for e_1. Suppose e_1 points to some type τ_l and e_2 points to some type τ_r. Ignoring the f fields for simplicity, by the definition of P_A, we have

$$\begin{array}{rcl} ref(p_{get} = p_{set} = \tau_l) & \subseteq_s & \tau_1 \\ ref(p_{get} = p_{set} = \tau_r) & \subseteq_s & \tau_2 \end{array}$$

The $\subseteq_s$ is not necessarily an equality because of rule $(\mathrm{Deref_A})$. Appendix A provides justification for having p_{get} and p_{set} the same. For now, simply observe that it is certainly true of the types of variables, and these are the types that propagate through the rest of the system. Since p_{set} is a contravariant field, the constraint

$$\tau_1 \subseteq_s ref(p_{set} = \alpha)$$

makes α a lower bound on the points-to set for τ_l:

$$\alpha \subseteq_s \tau_l$$

Similarly, $\tau_2 \subseteq ref(p_{get} = \beta)$ makes β an upper bound on τ_r, $\beta \subseteq_s \tau_r$. Thus, we get the desired constraints

$$\tau_r \subseteq_s \beta \subseteq_s \alpha \subseteq_s \tau_l$$

and the points-to set for e_2 becomes a lower bound on the points-to set for e_1.

Function pointers originate with the function definition, in the rule (Defn_A). This rule boxes up the (l-value) types for the formal parameters and the return value for the function, making it a lower bound on the points-to set for f. Function definitions have type `void`, *i.e.* they have no type.

When a function is applied, rule (App_A) extracts the types of the actual parameters and assigns them to the formal parameters. In effect, (App_A) rewrites the application $\mathtt{f}(\mathtt{z}_1, \ldots, \mathtt{z}_n)$ as the sequence

$$\mathtt{x}_1 = \mathtt{z}_1, \ldots, \mathtt{x}_n = \mathtt{z}_n, \mathtt{y}$$

where the $\mathtt{x}_i$ are the formal parameters and $\mathtt{y}$ is the return value.

The rules in **Common** are straightforward, so we discuss only the essential ideas. The type of a string or dynamically allocated memory is a pointer to a fresh variable, as in the rules (Const − Str) and (Malloc).

The rules (Cond) and (Op) create a fresh program variable V_l and join the subexpression types by assigning them to V_l. We explain the reason for these slightly convoluted rules in Section 6. For Andersen's analysis, the result type points to the union of the subexpressions' points-to sets. This is overly conservative for **And+Common**, but we chose this rule to match [SH97]. We discuss this issue further in Section 8.

Structures and arrays are atomic; we make no distinction between array elements or between fields of structures. These rules assume that the programmer does not take advantage of the compiler's data layout strategy. Note that, in fact, a common (non-portable) implementation of functions with variable numbers of arguments relies on having parameters contiguous in memory to access the variable-length part. Currently, neither our system nor Shapiro and Horwitz's system supports this.

5.4 Implementation

Our implementation uses type rules that differ slightly from those presented here. In general, only two of the four fields of a particular *ref* are used, because most C programs do not mix function types and pointer types in the same variable. Thus, to reduce the number of type variables generated by the rules, in our implementation we remove the f_{get} and f_{set} fields and place lam_n terms in p_{get} and p_{set}. Thus, in general the p fields may contain a union of terms with *ref* and lam_n head constructors. We use the pattern matching mechanism described in [FA97] to select the correct terms when projecting out of a certain constructor.

6 Steensgaard-Style Analysis

Intuitively, Steensgaard's analysis replaces inclusion constraints with equality constraints. Where Andersen's analysis creates a set, Steensgaard's analysis unifies the members of that set to create an equivalence class. For example, for an assignment $e_1 = e_2$, the rule (Asst_A) adds the members of the points-to set for e_2 to the points-to set for e_1. The analogous rule for Steensgaard's analysis instead unifies the equivalence classes for the points-to sets of e_1 and e_2.

Figure 7 shows the rules for our implementation of Steensgaard's Algorithm, **Ste**. As before, **Ste+Common** is the complete type system. The type language is a small modification of the previous system:

$$\begin{array}{rcl} \tau_S & ::= & \alpha \mid ref(p = \tau_S, f = \lambda_S, t = \tau_S) \\ \lambda_S & ::= & \beta \mid lam_n(\tau_{S_0}, \tau_{S_1}, \ldots, \tau_{S_n}) \end{array}$$

Ste

$$\frac{}{n : \_} \qquad (\mathrm{Const}-\mathrm{Int}_S)$$

$$\frac{}{\mathtt{x} : \mathit{ref}(p = x, f = x', t = x_t)} \qquad (\mathrm{Var}_S)$$

$$\frac{e : \tau}{\&e : \mathit{ref}(p =_{\mathtt{c}} \tau)} \qquad (\mathrm{Addr}_S)$$

$$\frac{e : \tau \quad \tau =_{\mathtt{c}} \mathit{ref}(p = \alpha)}{*e : \alpha} \qquad (\mathrm{Deref}_S)$$

$$\frac{\begin{array}{cc} e_1 : \tau_1 & e_2 : \tau_2 \\ \tau_1 =_{\mathtt{c}} \mathit{ref}(p = \alpha, f = \alpha') & \tau_2 =_{\mathtt{c}} \mathit{ref}(p = \beta, f = \beta') \\ \beta =_{\mathtt{c}} \alpha & \beta' =_{\mathtt{c}} \alpha' \end{array}}{e_1 = e_2 : \tau_2} \qquad (\mathrm{Asst}_S)$$

$$\frac{\begin{array}{c} \mathtt{f} : \mathit{ref}(p = f, f = f', t = f'') \\ \mathtt{y} : \tau_y \quad e : \tau_e \quad \forall 1 \le i \le n \;.\; \mathtt{x}_i : \tau_i \\ \mathit{lam}_n(\tau_y, \tau_1, \ldots, \tau_n) =_{\mathtt{c}} f' \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \rightarrow \mathtt{y} = e : \mathtt{void}} \qquad (\mathrm{Defn}_S)$$

$$\frac{\begin{array}{ccc} \mathtt{f} : \tau_f & \mathtt{x}_i : \tau_i & \forall 1 \le i \le n \\ \multicolumn{3}{l}{\tau_f =_{\mathtt{c}} \mathit{ref}(f = \mathit{lam}_n(\alpha_0, \alpha_1, \ldots, \alpha_n))} \\ \alpha_i =_{\mathtt{c}} \mathit{ref}(p = \alpha_{i_1}, f = \alpha_{i_2}) & \tau_i =_{\mathtt{c}} \mathit{ref}(p = \alpha_{i'_1}, f = \alpha_{i'_2}) & \\ \alpha_{i'_1} =_{\mathtt{c}} \alpha_{i_1} & \alpha_{i'_2} =_{\mathtt{c}} \alpha_{i_2} & \end{array}}{\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) : \tau_0} \qquad (\mathrm{App}_S)$$

Figure 7: Type system for Steensgaard's algorithm

with signatures

$$\begin{array}{c} \mathit{ref} : (p = \mathtt{Term}, f = \mathtt{Term}, t = \mathtt{Term}) \rightarrow \mathtt{Term} \\ \mathit{lam}_n : \underbrace{\mathtt{Term} \cdots \mathtt{Term}}_{n+1} \rightarrow \mathtt{Term} \end{array}$$

This system uses `Term` fields instead of `Set` fields and replaces inclusion constraints with unification. The p_{get} and p_{set} fields in the previous type system have coalesced into one field in this system because $=_{\mathtt{t}}$ models invariance. Function types lam_n now appear structurally within *ref* types because unification can only join terms with the same head constructor. The tag field t serves to identify equivalence classes.

Because pure unification is a coarse operation, **Ste** uses a form conditional unification to improve the precision of the analysis, as proposed by Steensgaard [Ste96b]. We use $=_{\mathtt{c}}$ to denote Steensgaard's conditional unification. Figure 3a contains the rewrite rule for $=_{\mathtt{c}}$. Intuitively, conditional unification $x =_{\mathtt{c}} y$ becomes unconditional unification $x =_{\mathtt{t}} y$ if x is instantiated with a constructed term. More formally,

Definition 6.1 Define $x =_{\mathtt{c}} y \equiv (\exists c. CS \vdash c(\ldots) =_{\mathtt{t}} x \implies x =_{\mathtt{t}} y)$.

For an example in which conditional unification is useful, consider the program from [Ste96b]

```
a = 2, x = a, y = a
```

Since `a` is not a pointer, x and y may remain distinct.

Define the relation

$$\vdash_S \equiv \text{provable in } \textbf{Ste+Common}$$

Then we define the points-to relation as before.

Definition 6.2 The points-to relation $P_S : L \to 2^L$ is defined by $\mathtt{y} \in P_S(\mathtt{x})$ if

$$CS \vdash_S \mathit{ref}(p = y, f = y', t = y_t) =_{\mathfrak{t}} x$$

Although the points-to relation is defined in terms of memory locations, the points-to graph is defined in terms of equivalence classes:

Definition 6.3 Let L be the set of all memory locations. The equivalence relation on locations $[\cdot]$ is defined as $[\mathtt{x}] = \{\mathtt{y} \mid CS \vdash_S x_t = y_t\}$.

The nodes of the points-to graph are the equivalence classes defined by $[\cdot]$, and there is an edge from $[\mathtt{x}]$ to $[\mathtt{y}]$ if $\mathtt{y} \in P_S(\mathtt{x})$. Note that $\forall \mathtt{y} \in P_S(\mathtt{x}).[\mathtt{y}] = P_S(\mathtt{x})$. To illustrate the usefulness of the tag field, consider the program `a = &c, b = &c`. Ignoring the f field, this program generates typings and constraints

$$\begin{array}{l} \vdash_S \mathtt{a} : \mathit{ref}(p = a, t = a_t) \\ \vdash_S \mathtt{b} : \mathit{ref}(p = b, t = b_t) \\ \vdash_S \mathtt{c} : \mathit{ref}(p = c, t = c_t) \\ a = \mathit{ref}(p = c, t = c_t) \\ b = \mathit{ref}(p = c, t = c_t) \end{array}$$

If the types of `a` and `b` did not contain the tag field, then we could not tell that they are distinct locations pointing to the same thing, since their p fields are equal.

Note that an equivalence class may point to itself. Thus, circular unification is needed to implement this system.

The rules for **Ste** are similar to those for **And**. The "_" in (Const $-$ Int$_S$) stands for a wild-card, which is a fresh, unconstrained variable. The somewhat mysterious rules (Cond) and (Op) are written so that conditional unification between the subexpressions is symmetric. The remaining rules are straightforward.

7 Properties of the Type Systems

7.1 Correspondence Between the Two Systems

It is clear that the systems are closely related. In fact, it is possible to express both systems with one set of rules, where the only difference is in the signatures for constructors. To do this, we combine the type languages for **Ste** and **And** to yield a *ref* with two p fields, two f fields, and a tag field:

$$\begin{array}{lcl} \tau_{AS} & ::= & \alpha \mid \mathit{ref}(p_{get} = \tau_{AS}, p_{set} = \tau_{AS}, f_{get} = \lambda_{AS}, f_{set} = \lambda_{AS}, t = \tau_{AS}) \\ \lambda_{AS} & ::= & \beta \mid \mathit{lam}_n(\tau_{AS_0}, \tau_{AS_1}, \ldots, \tau_{AS_n}) \end{array}$$

For an Andersen-style analysis, the p_{get} and f_{get} fields of *ref* are covariant, the p_{set} and f_{set} fields contravariant, and the t field is ignored:

$$\begin{array}{c} \mathit{ref} : (p_{get} = \mathtt{Set}, p_{set} = \overline{\mathtt{Set}}, f_{get} = \mathtt{Set}, f_{set} = \overline{\mathtt{Set}}, t = \mathtt{Set}) \to \mathtt{Set} \\ \mathit{lam}_n : \underbrace{\mathtt{Set} \cdots \mathtt{Set}}_{n+1} \to \mathtt{Set} \end{array}$$

For a Steensgaard-style analysis, all the subfields are `Term` fields, and we assure, in our rules, that $p_{get} = p_{set}$ and $f_{get} = f_{set}$:

$$\begin{array}{c} \mathit{ref} : (p_{get} = \mathtt{Term}, p_{set} = \mathtt{Term}, f_{get} = \mathtt{Term}, f_{set} = \mathtt{Term}, t = \mathtt{Term}) \to \mathtt{Term} \\ \mathit{lam}_n : \underbrace{\mathtt{Term} \cdots \mathtt{Term}}_{n+1} \to \mathtt{Term} \end{array}$$

The type rules are similar to those we presented before for an Andersen-style analysis. The interesting rules are shown in Figure 8. If the constraints $\subseteq_\iota$ are subset constraints ($\subseteq_{\iota=\mathfrak{s}}$, constraint system $CS_{\mathfrak{s}}$), then the rules compute an Andersen-style analysis. If we use conditional unification constraints ($\subseteq_{\iota=\mathfrak{c}}$, constraint system $CS_{\mathfrak{c}}$), we get a Steensgaard-style analysis.

Definition 7.1 Let L be the set of memory locations, and define the parameterized points-to relation $Q_\iota : L \to 2^L$ as $\mathtt{y} \in Q_\iota(\mathtt{x})$ if

$$CS_\iota \vdash \mathit{ref}(p_{get} = p_{set} = y, f_{get} = f_{set} = y', t = y_t) \subseteq_\iota x$$

Comb$_\iota$

$$\frac{}{\mathtt{x} : ref(p_{get} = p_{set} = x, f_{get} = f_{set} = x', t = x_t)} \quad (\mathrm{Var}_{\mathrm{Comb}_\iota})$$

$$\frac{e : \tau}{\&e : ref(p_{get} = p_{set} = \tau)} \quad (\mathrm{Addr}_{\mathrm{Comb}_\iota})$$

$$\frac{e : \tau \quad \tau \subseteq_\iota ref(p_{get} = \alpha)}{*e : \alpha} \quad (\mathrm{Deref}_{\mathrm{Comb}_\iota})$$

$$\frac{\begin{array}{cc} e_1 : \tau_1 & e_2 : \tau_2 \\ \tau_1 \subseteq_\iota ref(p_{set} = \alpha', f_{set} = \alpha'') & \tau_2 \subseteq_\iota ref(p_{get} = \beta', f_{get} = \beta'') \\ \beta' \subseteq_\iota \alpha' & \beta'' \subseteq_\iota \alpha'' \end{array}}{e_1 = e_2 : \tau_2} \quad (\mathrm{Asst}_{\mathrm{Comb}_\iota})$$

Figure 8: Type system for combined analysis.

Although we will not prove it formally, the following lemma should be clear:

Lemma 7.2 (a) $\forall \mathtt{x}. Q_{\iota=\mathtt{s}}(\mathtt{x}) = P_A(\mathtt{x})$, and
(b) $\forall \mathtt{x}. Q_{\iota=\mathtt{c}}(\mathtt{x}) = P_S(\mathtt{x})$, and
(c) $\forall \mathtt{x}. [\mathtt{x}] = [\mathtt{x}]'$ where $[\cdot]$ is from **Ste+Common** and $[\cdot]'$ is from $Comb_{\iota=\mathtt{c}}$.

Before proving the correspondence between the systems, we need the following lemma.

Lemma 7.3 (Translation) Let $CS_\mathtt{s}$ be a system of set constraints containing only variables and constructed terms, and let $CS_\mathtt{c}$ be the translated system in which each inclusion constraint $\tau_1 \subseteq_\mathtt{s} \tau_2$ has been replaced by a conditional unification constraint $\tau_1 =_\mathtt{c} \tau_2$, and the constructor signatures have changed from `Set` to `Term`. Then

$$CS_\mathtt{s} \vdash \tau_1' \subseteq_\mathtt{s} \tau_2' \implies CS_\mathtt{c} \vdash \tau_1' =_\mathtt{c} \tau_2'$$

Proof idea: There are two important cases. If $CS_\mathtt{s} \vdash \tau_1' = 0$ then $CS_\mathtt{c} \vdash \tau_1' = 0$, and thus where $\tau_1' \subseteq_\mathtt{s} \tau_2'$ makes no constraint on τ_2', neither does $\tau_1' =_\mathtt{c} \tau_2'$. On the other hand, if $CS_\mathtt{s} \vdash c(\dots) \subseteq_\mathtt{s} \tau_1'$ then $CS_\mathtt{c} \vdash c(\dots) =_\mathtt{c} \tau_1'$, and so $c(\dots) =_\mathtt{c} \tau_1'$ becomes $c(\dots) =_\mathtt{t} \tau_1'$. Thus, $CS_\mathtt{c} \tau_1' =_\mathtt{t} \tau_2'$.

Thus we can state the following:

Theorem 7.4 If $\mathtt{y} \in P_A(\mathtt{x})$ then $\mathtt{y} \in P_S(\mathtt{x})$.

Proof: Suppose $\mathtt{y} \in P_A(\mathtt{x})$. Then by Lemma 7.2, $\mathtt{y} \in Q_{\iota=\mathtt{s}}(\mathtt{x})$. By definition of $Q_{\iota=\mathtt{s}}$,

$$ref(p_{get} = p_{set} = y, f_{get} = f_{set} = y', t = y_t) \subseteq_\mathtt{s} x$$

Therefore by Lemma 7.3,

$$ref(p_{get} = p_{set} = y, f_{get} = f_{set} = y', t = y_t) =_\mathtt{c} x$$

in **Comb**$_{\iota=\mathtt{c}}$. But then, by Lemma 7.2 again, we have $\mathtt{y} \in P_S(\mathtt{x})$.

7.2 Soundness

Figure 9 shows an operational semantics for the C fragment

$$e ::= \mathtt{x} \mid *e \mid \&e \mid e_1 = e_2 \mid e_1, e_2$$

Variable names such as x stand for memory locations. We maintain a global mapping θ from dynamic memory locations *Loc* to syntactic locations *SyntacticLoc*. Recall that the points-to analyses confound all dynamic occurrences of a variable with the same name; θ formalizes this property. For this fragment there is a one-to-one correspondence between semantic (dynamic) locations and syntactic locations, and so

$$\begin{array}{rcl} \theta & = & \{[l_\mathtt{x} \mapsto \mathtt{x}] \mid \mathtt{x} \text{ is a program variable}\} \cup \\ & & \{[l' \mapsto \&_i] \mid \&_i \text{ appears in the program, } l' \text{ is the corresponding location from (Addr)}\} \end{array}$$

Domains: $Loc = \{l_{\mathtt{x}}, l_{\mathtt{y}}, \ldots\}$
$SyntacticLoc = \{\mathtt{x}, \mathtt{y}, \ldots \,\&_i, \ldots\}$
$Expr = SyntacticLoc \mid *Expr \mid \&_i Expr \mid Expr_1 = Expr_2 \mid Expr_1, Expr_2$
$Store : Loc \to Loc$
$\cdot \to \cdot : < Expr, Store > \to < Loc, Store >$

$$\frac{}{< \mathtt{x}, \sigma > \to < l_{\mathtt{x}}, \sigma >} \quad \text{(Var)}$$

$$\frac{< e_1, \sigma > \to < l_1, \sigma_1 > \quad < e_2, \sigma_1 > \to < l_2, \sigma_2 >}{< (e_1, e_2), \sigma > \to < l_2, \sigma_2 >} \quad \text{(Seq)}$$

$$\frac{< e, \sigma > \to < l, \sigma' > \qquad l' \text{ fresh}}{< \&_i e, \sigma > \to < l', \sigma'[l' \mapsto l] >} \quad \text{(Addr)}$$

$$\frac{< e, \sigma > \to < l, \sigma' >}{< *e, \sigma > \to < \sigma'(l), \sigma' >} \quad \text{(Deref)}$$

$$\frac{< e_1, \sigma > \to < l_1, \sigma_1 > \quad < e_2, \sigma_1 > \to < l_2, \sigma_2 >}{< e_1 = e_2, \sigma > \to < l_2, \sigma_2[l_1 \mapsto \sigma_2(l_2)] >} \quad \text{(Asst)}$$

Figure 9: Operational semantics for fragment of the source language.

We explain the second set of mappings below. For the full source language this mapping is in general a many-to-one mapping, and it can be modeled by threading an environment through the semantics.

The model of the store maps locations to their contents, which can only be other locations. We write $\emptyset$ for the empty store. The semantics maps a pair $< Expr, Store >$ to a pair $< Loc, Store >$ containing the location the expression corresponds to (its l-value) and a possibly modified store.

We briefly discuss the semantics. The result of evaluating a variable in (Var) returns the l-value of the variable and does not change the store. The rule (Seq) evaluates a sequence of expressions, executing them in order from left to right and returning the value of the right expression.

Because our semantics evaluates expressions to l-values, we need a location for the value of intermediate expressions to handle the address-of operator correctly. Accordingly, we add a new program variable $\&_i$ for each occurrence of an address-of operation in the program. The rule (Addr) operating on $\&_i e$ creates a fresh location and stores the l-value of e in it; the subscript i serves to identify the syntactic location of the operation. The second set of mappings in θ maps the fresh locations to the program variables $\&_i$. This semantics, which is more general than C, allows expressions such as $\&_i\&_j\mathtt{x}$.

To prove soundness, we must extend the type system (which uses the C semantics) slightly:

$$\frac{e : \tau}{\&_i e : ref(p_{set} = p_{get} = \tau) \quad \&_i : ref(p_{get} = p_{set} = \tau)} \quad (\text{Addr}'_{\text{A}})$$

Notice that the variable $\&_i$ corresponds to the value of $\&e$; thus, the type of $\&_i e$ is a pointer to e, and the semantic location l' is the l-value of $\&_i$.

The rule (Deref) returns the value stored at the location corresponding to expression e. Finally, the assignment rule (Asst) operating on $e_1 = e_2$ stores the contents of e_2 in e_1. Here we can clearly see an effect that C-like imperative languages most often hide: In evaluating an assignment, there is an implicit dereference of the right-hand side.

Given this semantics, we want to prove that for any program, if variable $\mathtt{x}$ ever contains the location of $\mathtt{y}$, then $\mathtt{y} \in P_A(\mathtt{x})$. In order to do so, we must prove something slightly stronger. We also need to show that the l-values assigned by the operational semantics correspond to the types given by **And**. In addition, we need to assume that the initial state corresponds to the points-to sets computed by **And**; otherwise, we cannot possibly show that **And** is sound with respect to the final state.

Let $\xrightarrow{*}$ be the reflexive transitive closure of $\to$. For simplicity, we omit the f_{set} and f_{get} fields, since there are no functions in this fragment.

Definition 7.5 For each syntactic location $\mathtt{x}$, let $\phi(\mathtt{x})$ be the type τ such that $\vdash_A \mathtt{x} : ref(p_{get} = p_{set} = \tau)$. For a program variable, $\phi(\mathtt{x}) = x$.

Let P_A be the points-to relation for the input program, and let θ be the global mapping described above.

Theorem 7.6 For any program e, if $< e, \sigma > \stackrel{*}{\rightarrow} < l, \sigma' >$ and $\forall x \in Dom(\sigma).\theta(\sigma(x)) \in P_A(\theta(x))$, then

$$\begin{array}{ll} (a) & \vdash_A e : \tau \text{ where } ref(p_{get} = p_{set} = \phi(\theta(l))) \subseteq \tau, \text{ and} \\ (b) & \forall x \in Dom(\sigma').\theta(\sigma'(x)) \in P_A(\theta(x)) \end{array}$$

Proof: By induction on the derivation of $< e, \sigma > \stackrel{*}{\rightarrow} < l, \sigma' >$.

Base case (Var)**:** $< \mathtt{x}, \sigma > \rightarrow < l, \sigma' >$. Then clearly (b) holds, and (a) holds by the definition of P_A, since $\vdash_A \mathtt{x} : ref(p_{get} = p_{set} = x)$, and $x = \phi(\mathtt{x}) = \phi(\theta(l_{\mathtt{x}}))$.

Induction:

Case 1 (Seq)**:** $< (e_1, e_2), \sigma > \rightarrow < l_2, \sigma_2 >$. Then by (Seq) we have (1) $< e_1, \sigma > \rightarrow < l_1, \sigma_1 >$ and (2) $< e_2, \sigma_1 > \rightarrow < l_2, \sigma_2 >$. To show (b), by induction on (1) we have $\forall x \in Dom(\sigma_1).\theta(\sigma_1(x)) \in P_A(\theta(x))$ and therefore by induction on (2) we have $\forall x \in Dom(\sigma_2).\theta(\sigma_2(x)) \in P_A(\theta(x))$. Similarly, (a) holds by induction on (1) and (2): $\vdash_A e_2 : \tau$ where $ref(p_{get} = p_{set} = \phi(\theta(l_2))) \subseteq \tau$.

Case 2 (Addr)**:** $< \&_i e, \sigma > \rightarrow < l', \sigma'[l' \mapsto l] >$. Then by (Addr) we have $(*)$ $< e, \sigma > \rightarrow < l, \sigma' >$. By induction on $(*)$, (1) $\vdash_A e : \tau$ where $ref(p_{get} = p_{set} = \phi(\theta(l))) \subseteq \tau$ and (2) $\forall x \in Dom(\sigma').\theta(\sigma'(x)) \in P_A(\theta(x))$. By definition of θ, $\theta(l') = \&_i$. Then by ($\mathrm{Addr}'_{\mathrm{A}}$), (a) holds, since program variable $\&_i$ has type $ref(p_{get} = p_{set} = \tau)$, thus $\phi(\&_i) = \tau$. To show (b), by (2) we need only show that $\theta(\sigma'[l' \mapsto l](l')) \in P_A(\theta(l'))$, *i.e.* $\theta(l) \in P_A(\&_i)$. By (1) we have $ref(p_{get} = p_{set} = \phi(\theta(l))) \subseteq \phi(\&_i)$, thus $\theta(l) \in P_A(\&_i)$.

Case 3 (Deref)**:** $< *e, \sigma > \rightarrow < \sigma'(l), \sigma' >$. Then by (Deref) we have $(*)$ $< e, \sigma > \rightarrow < l, \sigma' >$. (b) holds by induction on $(*)$, since $\forall x \in Dom(\sigma').\theta(\sigma'(x)) \in P_A(\theta(x))$. Also by induction on $(*)$, $\vdash_A e : \tau$ where $ref(p_{get} = p_{set} = \phi(\theta(l))) \subseteq \tau$. But by ($\mathrm{Deref}_{\mathrm{A}}$) $\vdash_A *e : \alpha$ where $\tau \subseteq ref(p_{get} = \alpha)$. Thus, we have $\phi(\theta(l)) \subseteq \alpha$, and since $\theta(\sigma'(l)) \in P_A(\theta(l))$ by (b), we have (a) $ref(p_{get} = p_{set} = \phi(\theta(\sigma'(l)))) \subseteq \phi(\theta(l)) \subseteq \alpha$.

Case 4 (Asst)**:** $< e_1 = e_2, \sigma > \rightarrow < l_2, \sigma_2[l_1 \mapsto \sigma_2(l_2)] >$. By (Asst), (1) $< e_1, \sigma > \rightarrow < l_1, \sigma_1 >$ and (2) $< e_2, \sigma_1 > \rightarrow < l_2, \sigma_2 >$. By induction on (1) and (2), (1a) $\vdash_A$: $e_1 : \tau_1$ where $ref(p_{get} = p_{set} = \phi(\theta(l_1))) \subseteq \tau_1$ and (2a) $\vdash_A$: $e_2 : \tau_2$ where $ref(p_{get} = p_{set} = \phi(\theta(l_2))) \subseteq \tau_2$ Thus (a) holds, since it's just (2a). By the same argument as in Case 1, $(*)$ $\forall x \in Dom(\sigma_2).\theta(\sigma_2(x)) \in P_A(\theta(x))$, so for (b) we need only show that $\theta(\sigma_2[l_1 \mapsto \sigma_2(l_2)](l_1)) = \theta(\sigma_2(l_2))$ is in $P_A(\theta(l_1))$.

If $l_1 = l_2$, then $\sigma_2[l_1 \mapsto \sigma_2(l_2)] = \sigma_2[l_1 \mapsto \sigma_2(l_1)] = \sigma_2$, and so (b) is just $(*)$.

Otherwise, suppose $l_1 \neq l_2$. By the constraints in ($\mathrm{Asst}_{\mathrm{A}}$), we have $\tau_1 \subseteq ref(p_{set} = \alpha)$, $\tau_2 \subseteq ref(p_{get} = \beta)$, and $\beta \subseteq \alpha$. Putting this together with (1a) and (2a) yields

$$\begin{array}{c} ref(p_{get} = p_{set} = \phi(\theta(l_1))) \subseteq \tau_1 \subseteq ref(p_{set} = \alpha) \\ ref(p_{get} = p_{set} = \phi(\theta(l_2))) \subseteq \tau_2 \subseteq ref(p_{get} = \beta) \\ \beta \subseteq \alpha \end{array}$$

And thus $\phi(\theta(l_2)) \subseteq \beta \subseteq \alpha \subseteq \phi(\theta(l_1))$ By induction, $\theta(\sigma_2(l_2)) \in P_A(\theta(l_2))$, and therefore since $\phi(\theta(l_2)) \subseteq \phi(\theta(l_1)) \implies P_A(\theta(l_2)) \subseteq P_A(\theta(l_1))$, we have $\theta(\sigma_2(l_2)) \in P_A(\theta(l_1))$.

Corollary 7.7 (Soundness$_A$) For any program e, if $< e, \emptyset > \stackrel{*}{\rightarrow} < l, \sigma >$, then $\forall x \in Dom(\sigma).\theta(\sigma(x)) \in P_A(\theta(x))$.

Corollary 7.8 (Soundness$_S$) For any program e, if $< e, \emptyset > \stackrel{*}{\rightarrow} < l, \sigma >$, then $\forall x \in Dom(\sigma).\theta(\sigma(x)) \in P_S(\theta(x))$.
Proof: By Corollary 7.7 and Theorem 7.4.

8 Implementation

We have implemented both type systems for C using the mixed term and set constraint solver described in Section 3. The type systems and constraint solver are coded in ML. The analysis traverses the source program's parse tree and applies inference rules at the appropriate points. Once the constraints have been generated, the framework solves the constraints, and the definitions of P_A and P_S are used to extract the points-to sets.

Only the code that implements the inference rules calls the constraint solver. The rules are packaged as an ML structure, one for **And+Common** and one for **Ste+Common**, and they parameterize a functor for the traversal code. The traversal calls the inference rule module in exactly eight cases to assert the appropriate constraints.

Shapiro and Horwitz (SH) have implemented these analysis in C [SH97]. SH uses the same general technique, although, as their implementation is in C, they do not have a built-in module system. Their rules separate l- and r-types, which doubles the number of cases. They use Andersen's specialized constraints, which is appropriate as their implementation is custom-designed for points-to analysis.

8.1 Difficulties with C

C is difficult to analyze because of its weak semantics and numerous details. Several features in particular complicate the implementation.

Library functions are the largest problem. Standard C includes, as does any practical language, many library functions, the sources of which are not directly available. Thus, the analyses cannot be applied to these routines. Our solution is to assume, in general, that any undefined function has no effect on the analysis. For those functions that have an effect, we write a short, simple code routine that models the effect. For example, the `strcpy(char* s1, char* s2)` function copies the string `s2` into the string `s1` and returns `s1`. To model this, we simply write a function that takes two arguments and returns the first. By preprocessing a file of these definitions, we can correctly model the points-to effects of library functions.

A second problematical aspect of C is functions that take variable numbers of arguments (varargs). The most common example is `printf` and its brethren, and by simply ignoring calls to these functions (which do not affect points-to sets) we solve most of the problem. Unfortunately, in at least one (non-portable) implementation of varags, the address of the last argument is used as a pointer to any subsequent arguments. For example, in the declaration `void foo(int a, ...)`, the address of `a` is used to retrieve any arguments that follow. A general solution to the problem requires pre-computing which functions are varargs and treating calls to them accordingly. To our knowledge, this problem has not be addressed in the literature, and neither our implementation nor SH handles this correctly.

For our current set of test programs, we have manually modified the few varags functions to take a fixed number of arguments, as our type systems require, padding out calls to those functions with `0`'s, and adding in some assignment statements to simulate the fetching of extra arguments. This process could also be automated by preprocessing the input program.

Handling arrays is the last difficulty. In C, defining an array `int a[5]` allocates space for the array contents and makes variable `a` point to it. To handle this situation correctly, we make a new variable, in this case `a[]`, and add it to the points-to set for the array name `a`. Defining a multidimensional array allocates a contiguous piece of memory rather than a series of pointer arrays. Consider the following piece of code:

```
int *a[2][2], **b, c, d;

b = (int**) a;
a[0][0] = &c;
b[0] = &d;
```

Here `b[0]` and `a[0][0]` are aliased. Solving this problem requires treating the expression $e_1[e_2]$ differently depending on the C type of e_1. If e_1 is a one-dimensional array (or a pointer), then $e_1[e_2]$ is exactly $*(e_1 + ke_2)$ for some constant k. If e_1 has dimension two or higher, then $e_1[e_2]$ can be interpreted as e_1+ke_2, with no dereference. Solving these problems requires knowing the full C types of each expression, which complicates the implementation.

Andersen-style Analysis								
		And+Common			SH			
Name	Lines	Time (s)	Size	Sets	Time (s)	Size	Sets	$\frac{\text{SH Time}}{\text{Framework}}$
diff.diffh	293	0.13	42	19	0.06	42	19	0.45
genetic	324	0.12	34	16	0.07	34	16	0.62
anagram	344	0.11	33	25	0.07	34	26	0.60
allroots	428	0.04	11	7	0.04	11	7	0.90
ul	445	0.16	10	10	0.08	10	10	0.53
ks	574	0.26	190	55	0.17	222	62	0.66
compress	652	0.25	15	15	0.10	15	15	0.40
ft	1179	0.40	140	64	0.15	140	64	0.37
ratfor	1540	1.11	642	111	0.42	618	113	0.38
compiler	1895	0.50	406	29	0.40	406	29	0.80
eqntott	2316	1.72	506	159	0.47	296	158	0.27
assembler	2987	1.48	596	179	0.66	522	183	0.45
simulator	4230	6.51	14721	288	8.79	14377	289	1.35
ML-typecheck	4903	3.20	10070	254	2.03	9973	252	0.63
li	5798	473.72	809489	1314	3880.47	809834	1316	8.19
flex-2.4.7	9358	13.59	3929	373	45.28	3661	371	3.33
less-177	12108	8.43	33497	492	11.25	32794	504	1.33
make-3.72.1	15214	190.67	221937	1141	384.60	222147	1144	2.02
espresso	21583	713.16	249965	1932	343.56	243521	1928	0.48

Steensgaard-style Analysis								
		Ste+Common			SH			
Name	Lines	Time (s)	Size	Sets	Time (s)	Size	Sets	$\frac{\text{SH Time}}{\text{Framework}}$
diff.diffh	293	0.13	192	19	0.04	192	19	0.33
genetic	324	0.10	92	16	0.05	92	16	0.53
anagram	344	0.10	151	30	0.04	220	31	0.41
allroots	428	0.03	14	7	0.03	14	7	0.88
ul	445	0.13	10	10	0.08	11	11	0.61
ks	574	0.11	537	57	0.08	607	64	0.71
compress	652	0.15	15	15	0.10	15	15	0.63
ft	1179	0.13	223	64	0.10	259	73	0.76
ratfor	1540	0.70	13567	177	0.42	13154	181	0.60
compiler	1895	0.58	1080	49	0.18	1080	49	0.32
eqntott	2316	0.93	1495	160	0.30	1116	160	0.32
assembler	2987	0.92	4069	227	0.38	3201	229	0.41
simulator	4230	2.45	24012	304	1.38	25320	325	0.56
ML-typecheck	4903	1.29	13685	262	0.53	13556	261	0.41
li	5798	5.67	1054796	1409	100.03	1152974	1417	17.64
flex-2.4.7	9358	10.03	23775	415	2.04	25854	452	0.20
less-177	12108	2.73	147795	646	5.15	151839	656	1.89
make-3.72.1	15214	7.10	927441	1609	73.72	938203	1634	10.38
espresso	21583	12.51	315372	1935	27.23	316999	1970	2.18

Figure 10: Comparison of **And+Common** and **Ste+Common** with Shapiro and Horwitz [SH97].

8.2 Measurements

Figure 10 compares our implementation with SH. All times were measured on a Sun UltraSPARC; hence the SH times differ from [SH97]. The results are summarized by two metrics. *Sets* counts the number of non-empty points-to sets computed. *Total size* is the sum of the cardinalities of all points-to sets. SH has been improved since [SH97], and so the reported sizes do not match. The sizes we report do not include temporary variables introduced by SH during program transformations.

In order to correspond more closely with SH, we made two modifications to our system. First, we temporarily disabled the file of library routine stubs, since although SH has the same sort of facility, that code is currently disabled. Second, we modified the rule (Op) slightly. Recall that this rule gives $a\ op\ b$ the type of a fresh variable and assigns a and b to the fresh variable. This results in either a union (**And+Common**) or a symmetric conditional unification (**Ste+Common**). Depending on the operation, this may be overly conservative; for example, it is possible to recover neither a nor b from a relational operation, which returns a boolean. To match SH, we only apply (Op) to addition, subtraction, and exclusive-or. We give other operations the type 0 for **And** or a fresh variable for **Ste**.

Even with these adjustments, the set sizes differ slightly because the systems still disagree in some details. We are confident that the performance comparison is accurate, because in most cases the set sizes differ by only a few percent, and in almost all remaining cases our implementation is the more conservative, and making it less conservative can only make it faster.

The remaining difference that we know about concerns conditional unification, though clearly there are also differences we do not understand. Recall from the definition that a conditional unification $x =_c y$ becomes a regular unification if x is instantiated with a constructed term. In SH, the return values for undefined functions trigger regular unification, but in our system they do not.

The reported times show that our implementation is often faster than SH for large programs. One program in particular stands out: `li`. In this case, the Andersen analysis in **And+Common** is over eight times faster than SH, and the Steensgaard analysis is seventeen times faster.[2] For small programs, our implementation is at most three times slower (except one program, `eqntott`, which is 3.7 times slower), though the absolute running times are also small. We attribute these results to the effort spent making our general framework scale to large programs [FA96], something that is difficult to do for one-time use systems like SH. There are also two anomalous results: Our system takes twice as long for Andersen-style analysis of `espresso` and five times as long for Steensgaard-style analysis of `flex`. We currently have no explanation for these apparent outliers.

Our framework is designed to make analyses easy to write, and so it is useful to subjectively compare the two code bases. The code for **And**, **Ste**, and **Common** comprises about 2,000 lines of ML, not counting the parser. Around 500 of these lines are for constraint generation and points-to set extraction, with the traversal making up the rest of the code. The reusable constraint framework is another 12,000 lines of code, not all of which is used by these analyses.

Compare this to SH, which is approximately 13,000 lines of C, not counting the parser. While line counts are only a proxy for programming effort, a factor of six is significant. Additionally, our framework lends itself to experimentation, which is often difficult in a custom implementation. For example, we are currently experimenting with changing from fixed-length argument lists to variable-length argument lists using a row type [Rém89]. The change to the type systems took only an hour. (We have not reported these results because the experiments were not complete at publication time.) The change from regular unification to conditional unification took only a few hours and required no change (except to the kind of constraint) in the type rules.

9 Conclusion

We have discussed two type systems that capture Andersen's algorithm and Steensgaard's analyses. We have shown that, as has been intuitively described in [SH97], these are closely related analyses, and we can account for the difference as a tradeoff between using inclusion constraints and using unification.

We believe these type systems demonstrate that our mixed term and set constraint framework is a natural formalism for expressing this kind of analysis. The translation from type rules to program code is straightforward, and thus we are confident that the soundness proof translates into correct code. Performance is within a small constant factor of a hand-coded implementation, and often better for large

[2] An essential routine in `li` takes a variable number of arguments. This routine uses a non-portable implementation of varags. When we modified this routine to take a fixed number of arguments (so that we could model it correctly), the set sizes increased dramatically from [SH97].

programs. With low implementation cost and competitive performance, our framework provides a useful and productive platform with which to develop program analyses.

10 Acknowledgments

The idea for dealing with library functions came from a discussion with Erik Ruf. Thanks to Susan Horwitz and Marc Shapiro for providing their large suite of test programs and the source code for their analyses. Thanks to Detlef Sodtke and Jens Krinke for an ANSI C parser. Also thanks to Martín Abadi and Gordon Woodhull for commenting on an earlier draft of this paper.

References

[And94] L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Cophenhagen, May 1994.

[ASU88] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1988.

[AW92] A. Aiken and E. Wimmers. Solving Systems of Set Constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 1994.

[FA96] M. Fähndrich and A. Aiken. Making Set-Constraint Based Program Analyses Scale. In *First Workshop on Set Constraints at CP '96*, Cambridge, MA, August 1996. Available as CSD-TR-96-917, University of California at Berkeley.

[FA97] M. Fähndrich and A. Aiken. Program Analysis using Mixed Term and Set Constraints. In *Static Analysis, Fourth International Symposium, SAS'97*, 1997.

[Hei92] N. Heintze. *Set Based Program Analysis.* PhD dissertation, Carnegie Mellon University, Department of Computer Science, October 1992.

[HJ90] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Herbrand Set Constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[Hor97] S. Horwitz. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM TOPLAS*, 19(1):1–6, January 1997.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76, January 1989.

[SH97] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[Ste96a] B. Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and Unions. In *Proceedings of the International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 136–150. Springer-Verlag, April 1996.

[Ste96b] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

A Typings given by And+Common

The discussion in Section 5 arguing that **And+Common** implements Andersen's algorithm requires that the system give expressions types of a certain form. Specifically, although the type language allows types like $ref(p_{get} = \alpha, p_{set} = \beta)$, in **And** it is always the case that $\alpha = \beta$. This is easy to show:

Lemma A.1 Define the property B

$$B(\tau) = (\tau \supseteq ref(p_{get} = \alpha, p_{set} = \beta) \Rightarrow \alpha = \beta \text{ and } B(\alpha))$$

Then $\vdash_A e : \tau \Rightarrow B(\tau)$.

Proof: By induction on the derivation of $\vdash_A e : \tau$. We shall only show the important cases. Clearly the property holds if the only rule used was $(\mathrm{Var_A})$. If the last rule used was $(\mathrm{Addr_A})$, then $B(\tau)$ holds by induction and therefore $B(ref(p_{get} = p_{set} = \tau))$ holds. If we last used $(\mathrm{Deref_A})$, then by induction $B(\tau)$, so $\tau \subseteq ref(p_{get} = \alpha)$ implies $B(\alpha)$.

Finally, if we last used $(\mathrm{Asst_A})$, then by induction $B(\tau_1)$ and $B(\tau_2)$ hold before the rule. Thus, $B(\beta)$ holds, which implies $B(\alpha)$, since $\beta \subseteq_s \alpha$ and α and β do not leave the scope of the rule. Then, since $B(\tau_1)$ holds before we applied this rule and $B(\alpha)$, $B(\tau_1)$ holds after we apply the rule, too, since all the new lower bounds we added satisfy the property B.

This proof can be extended to the rest of **And+Common** easily, and a similar result holds for the f fields. Note that it makes no claim that all of the types used in the constraints have this special property. Specifically, it is essential in $(\mathrm{Deref_A})$ and $(\mathrm{Asst_A})$ that in the ref constructors p_{get} and p_{set} are distinct.

B * and & as inverses

In both type systems, we can basically prove that `*&` and `&*` pairs cancel each other out. Let e be an arbitrary expression.

Lemma B.1 If $\vdash_A e : \tau$ and $\vdash_A *\&e : \tau'$, then $\tau \subseteq \tau'$.

Proof:

$$\dfrac{\dfrac{\vdash_A e : \tau}{\vdash_A \&e : ref(p_{get} = p_{set} = \tau)} \quad ref(p_{get} = p_{set} = \tau) \subseteq_s ref(p_{get} = \tau')}{\vdash_A *\&\mathsf{e} : \tau'}$$

and since $ref(p_{get} = p_{set} = \tau) \subseteq_s ref(p_{get} = \tau')$, we have $\tau \subseteq \tau'$.

We can prove something slightly weaker for `&*`:

Lemma B.2 If $\vdash_A e : ref(p_{get} = \tau)$ and $\vdash_A \&*e : ref(p_{get} = p_{set} = \tau')$, then $\tau \subseteq \tau'$.

Proof:

$$\dfrac{\dfrac{\vdash_A e : ref(p_{get} = \tau) \quad ref(p_{get} = \tau) \subseteq_s ref(p_{get} = \tau')}{\vdash_A *e : \tau'}}{\vdash_A \&*e : ref(p_{get} = p_{set} = \tau')}$$

We claim that these lemmas imply that `*` and `&` are inverses, in the sense of points-to sets. In order to formalize this fully, we need to show that these upper bounds do not effect the points-to sets. We shall state without proof the corresponding lemmas for Steensgaard's system:

Lemma B.3 If $\vdash_S e : \tau$ and $\vdash_S *\&e : \tau'$, then $\tau = \tau'$.

Lemma B.4 If $\vdash_S e : ref(p = \tau)$ and $\vdash_S \&*e : ref(p = \tau')$, then $\tau = \tau'$.